causalml Documentation

Someone at Uber

Dec 30, 2021

CONTENTS

Contents:

CHAPTER

ONE

ABOUT CAUSAL ML

`Causal ML` is a Python package that provides a suite of uplift modeling and causal inference methods using machine learning algorithms based on recent research. It provides a standard interface that allows user to estimate the **Conditional Average Treatment Effect** (CATE) or **Individual Treatment Effect** (ITE) from experimental or observational data. Essentially, it estimates the causal impact of intervention **T** on outcome **Y** for users with observed features **X**, without strong assumptions on the model form.

Typical use cases include:

- **Campaign Targeting Optimization**: An important lever to increase ROI in an advertising campaign is to target the ad to the set of customers who will have a favorable response in a given KPI such as engagement or sales. CATE identifies these customers by estimating the effect of the KPI from ad exposure at the individual level from A/B experiment or historical observational data.
- **Personalized Engagement**: Company has multiple options to interact with its customers such as different product choices in up-sell or mess aging channels for communications. One can use CATE to estimate the heterogeneous treatment effect for each customer and treatment option combination for an optimal personalized recommendation system.

The package currently supports the following methods:

- **Tree-based algorithms**
  - *Uplift Random Forests* on KL divergence, Euclidean Distance, and Chi-Square
  - *Uplift Random Forests* on Contextual Treatment Selection
  - *Uplift Random Forests* on delta-delta-p ($\Delta\Delta P$) criterion (only for binary trees and two-class problems)
- **Meta-learner algorithms**
  - *S-Learner*
  - *T-Learner*
  - *X-Learner*
  - *R-Learner*
  - *Doubly Robust (DR) learner*
- **Instrumental variables algorithms**
  - *2-Stage Least Squares (2SLS)*
  - *Doubly Robust Instrumental Variable (DRIV) learner*
- **Neural network based algorithms**
  - CEVAE
  - DragonNet

- **Treatment optimization algorithms**
  - *Counterfactual Unit Selection*
  - *Counterfactual Value Estimator*

CHAPTER

TWO

METHODOLOGY

2.1 Meta-Learner Algorithms

A meta-algorithm (or meta-learner) is a framework to estimate the Conditional Average Treatment Effect (CATE) using any machine learning estimators (called base learners) [15].

A meta-algorithm uses either a single base learner while having the treatment indicator as a feature (e.g. S-learner), or multiple base learners separately for each of the treatment and control groups (e.g. T-learner, X-learner and R-learner).

Confidence intervals of average treatment effect estimates are calculated based on the lower bound formular (7) from [13].

2.1.1 S-Learner

S-learner estimates the treatment effect using a single machine learning model as follows:

Stage 1

Estimate the average outcomes $\mu(x)$ with covariates X and an indicator variable for treatment effect W:

$$\mu(x) = E[Y \mid X = x, W = w]$$

using a machine learning model.

Stage 2

Define the CATE estimate as:

$$\hat{\tau}(x) = \hat{\mu}(x, W = 1) - \hat{\mu}(x, W = 0)$$

Including the propensity score in the model can reduce bias from regularization induced confounding [25].

When the control and treatment groups are very different in covariates, a single linear model is not sufficient to encode the different relevant dimensions and smoothness of features for the control and treatment groups [1].

2.1.2 T-Learner

T-learner [15] consists of two stages as follows:

Stage 1

Estimate the average outcomes $\mu_0(x)$ and $\mu_1(x)$:

$$\mu_0(x) = E[Y(0)|X = x]$$
$$\mu_1(x) = E[Y(1)|X = x]$$

using machine learning models.

Stage 2

Define the CATE estimate as:

$$\hat{\tau}(x) = \hat{\mu}_1(x) - \hat{\mu}_0(x)$$

2.1.3 X-Learner

X-learner [15] is an extension of T-learner, and consists of three stages as follows:

Stage 1

Estimate the average outcomes $\mu_0(x)$ and $\mu_1(x)$:

$$\mu_0(x) = E[Y(0)|X = x]$$
$$\mu_1(x) = E[Y(1)|X = x]$$

using machine learning models.

Stage 2

Impute the user level treatment effects, D_i^1 and D_j^0 for user i in the treatment group based on $\mu_0(x)$, and user j in the control groups based on $\mu_1(x)$:

$$D_i^1 = Y_i^1 - \hat{\mu}_0(X_i^1)$$
$$D_i^0 = \hat{\mu}_1(X_i^0) - Y_i^0$$

then estimate $\tau_1(x) = E[D^1|X = x]$, and $\tau_0(x) = E[D^0|X = x]$ using machine learning models.

Stage 3

Define the CATE estimate by a weighted average of $\tau_1(x)$ and $\tau_0(x)$:

$$\tau(x) = g(x)\tau_0(x) + (1 - g(x))\tau_1(x)$$

where $g \in [0, 1]$. We can use propensity scores for $g(x)$.

2.1.4 R-Learner

R-learner [18] uses the cross-validation out-of-fold estimates of outcomes $\hat{m}^{(-i)}(x_i)$ and propensity scores $\hat{e}^{(-i)}(x_i)$. It consists of two stages as follows:

Stage 1

Fit $\hat{m}(x)$ and $\hat{e}(x)$ with machine learning models using cross-validation.

Stage 2

Estimate treatment effects by minimising the R-loss, $\hat{L}_n(\tau(x))$:

$$\hat{L}_n(\tau(x)) = \frac{1}{n}\sum_{i=1}^{n}\left(\left(Y_i - \hat{m}^{(-i)}(X_i)\right) - \left(W_i - \hat{e}^{(-i)}(X_i)\right)\tau(X_i)\right)^2$$

where $\hat{e}^{(-i)}(X_i)$, etc. denote the out-of-fold held-out predictions made without using the i-th training sample.

2.1.5 Doubly Robust (DR) learner

DR-learner [14] estiamtes the CATE via cross-fitting a doubly-robust score function in two stages as follows. We start by randomly split the data $\{Y, X, W\}$ into 3 partitions $\{Y^i, X^i, W^i\}, i = \{1, 2, 3\}$.

Stage 1

Fit a propensity score model $\hat{e}(x)$ with machine learning using $\{X^1, W^1\}$, and fit outcome regression models $\hat{m}_0(x)$ and $\hat{m}_1(x)$ for treated and untreated users with machine learning using $\{Y^2, X^2, W^2\}$.

Stage 2

Use machine learning to fit the CATE model, $\hat{\tau}(X)$ from the pseudo-outcome

$$\phi = \frac{W - \hat{e}(X)}{\hat{e}(X)(1 - \hat{e}(X))}\left(Y - \hat{m}_W(X)\right) + \hat{m}_1(X) - \hat{m}_0(X)$$

with $\{Y^3, X^3, W^3\}$

Stage 3

Repeat Stage 1 and Stage 2 again twice. First use $\{Y^2, X^2, W^2\}$, $\{Y^3, X^3, W^3\}$, and $\{Y^1, X^1, W^1\}$ for the propensity score model, the outcome models, and the CATE model. Then use $\{Y^3, X^3, W^3\}$, $\{Y^2, X^2, W^2\}$, and $\{Y^1, X^1, W^1\}$ for the propensity score model, the outcome models, and the CATE model. The final CATE model is the average of the 3 CATE models.

2.1.6 Doubly Robust Instrumental Variable (DRIV) learner

We combine the idea from DR-learner [14] with the doubly robust score function for LATE described in [9] to estimate the conditional LATE. Towards that end, we start by randomly split the data $\{Y, X, W, Z\}$ into 3 partitions $\{Y^i, X^i, W^i, Z^i\}, i = \{1, 2, 3\}$.

Stage 1

Fit propensity score models $\hat{e}_0(x)$ and $\hat{e}_1(x)$ for assigned and unassigned users using $\{X^1, W^1, Z^1\}$, and fit outcome regression models $\hat{m}_0(x)$ and $\hat{m}_1(x)$ for assigned and unassigned users with machine learning using $\{Y^2, X^2, Z^2\}$. Assignment probabiliy, p_Z, can either be user provided or come from a simple model, since in most use cases assignment is random by design.

Stage 2

Use machine learning to fit the conditional *LATE* model, $\hat{\tau}(X)$ by minimizing the following loss function

$$L(\hat{\tau}(X)) = \hat{E}\left[\left(\hat{m}_1(X) - \hat{m}_0(X) + \frac{Z(Y - \hat{m}_1(X))}{p_Z} - \frac{(1-Z)(Y - \hat{m}_0(X))}{1 - p_Z}\right.\right.$$
$$\left.\left. -\left(\hat{e}_1(X) - \hat{e}_0(X) + \frac{Z(W - \hat{e}_1(X))}{p_Z} - \frac{(1-Z)(W - \hat{e}_0(X))}{1 - p_Z}\right)\hat{\tau}(X)\right)^2\right]$$

with $\{Y^3, X^3, W^3\}$

Stage 3

Similar to the DR-Learner Repeat Stage 1 and Stage 2 again twice with different permutations of partitions for estimation. The final conditional LATE model is the average of the 3 conditional LATE models.

2.2 Tree-Based Algorithms

2.2.1 Uplift Tree

The Uplift Tree approach consists of a set of methods that use a tree-based algorithm where the splitting criterion is based on differences in uplift. [21] proposed three different ways to quantify the gain in divergence as the result of splitting [10]:

$$D_{gain} = D_{after_{split}}(P^T, P^C) - D_{before_{split}}(P^T, P^C)$$

where D measures the divergence and P^T and P^C refer to the probability distribution of the outcome of interest in the treatment and control groups, respectively. Three different ways to quantify the divergence, KL, ED and Chi, are implemented in the package.

2.2.2 KL

The Kullback-Leibler (KL) divergence is given by:

$$KL(P:Q) = \sum_{k=left,right} p_k log \frac{p_k}{q_k}$$

where p is the sample mean in the treatment group, q is the sample mean in the control group and k indicates the leaf in which p and q are computed [10]

2.2.3 ED

The Euclidean Distance is given by:

$$ED(P:Q) = \sum_{k=left,right} (p_k - q_k)^2$$

where the notation is the same as above.

2.2.4 Chi

Finally, the χ^2-divergence is given by:

$$\chi^2(P:Q) = \sum_{k=left,right} \frac{(p_k - q_k)^2}{q_k}$$

where the notation is again the same as above.

2.2.5 DDP

Another Uplift Tree algorithm that is implemented is the delta-delta-p ($\Delta\Delta P$) approach by [8], where the sample splitting criterion is defined as follows:

$$\Delta\Delta P = |(P^T(y|a_0) - P^C(y|a_0) - (P^T(y|a_1) - P^C(y|a_1)))|$$

where a_0 and a_1 are the outcomes of a Split A, y is the selected class, and P^T and P^C are the response rates of treatment and control group, respectively. In other words, we first calculate the difference in the response rate in each branch (ΔP_{left} and ΔP_{right}), and subsequently, calculate their differences ($\Delta\Delta P = |\Delta P_{left} - \Delta P_{right}|$).

2.2.6 CTS

The final Uplift Tree algorithm that is implemented is the Contextual Treatment Selection (CTS) approach by [23], where the sample splitting criterion is defined as follows:

$$\hat{\Delta}_{\mu}(s) = \hat{p}(\phi_l \mid \phi) \times \max_{t=0,\ldots,K} \hat{y}_t(\phi_l) + \hat{p}(\phi_r \mid \phi) \times \max_{t=0,\ldots,K} \hat{y}_t(\phi_r) - \max_{t=0,\ldots,K} \hat{y}_t(\phi)$$

where ϕ_l and ϕ_r refer to the feature subspaces in the left leaf and the right leaves respectively, $\hat{p}(\phi_j \mid \phi)$ denotes the estimated conditional probability of a subject's being in ϕ_j given ϕ, and $\hat{y}_t(\phi_j)$ is the conditional expected response under treatment t.

2.3 Value optimization methods

The package supports methods for assigning treatment groups when treatments are costly. To understand the problem, it is helpful to divide populations into the following four categories:

- **Compliers**. Those who will have a favourable outcome if and only if they are treated.
- **Always-takers**. Those who will have a favourable outcome whether or not they are treated.
- **Never-takers**. Those who will never have a favourable outcome whether or not they are treated.
- **Defiers**. Those who will have a favourable outcome if and only if they are not treated.

For a more detailed discussion see e.g. [2].

2.3.1 Counterfactual Unit Selection

[17] propose a method for selecting units for treatments using counterfactual logic. Suppose the following benefits for selecting units belonging to the different categories above:

- Compliers: β
- Always-takers: γ
- Never-takers: θ
- Defiers: δ

If X denotes the set of individual's features, the unit selection problem can be formulated as follows:

$$argmax_X \beta P(\text{complier} \mid X) + \gamma P(\text{always-taker} \mid X) + \theta P(\text{never-taker} \mid X) + \delta P(\text{defier} \mid X)$$

The problem can be reformulated using counterfactual logic. Suppose $W = w$ indicates that an individual is treated and $W = w'$ indicates he or she is untreated. Similarly, let $F = f$ denote a favourable outcome for the individual and $F = f'$ an unfavourable outcome. Then the optimization problem becomes:

$$argmax_X \beta P(f_w, f'_{w'} \mid X) + \gamma P(f_w, f_{w'} \mid X) + \theta P(f'_w, f'_{w'} \mid X) + \delta P(f_{w'}, f'_w \mid X)$$

Note that the above simply follows from the definitions of the relevant users segments. [17] then use counterfactual logic ([20]) to solve the above optimization problem under certain conditions.

N.B. The current implementation in the package is highly experimental.

2.3.2 Counterfactual Value Estimator

The counterfactual value estimation method implemented in the package predicts the outcome for a unit under different treatment conditions using a standard machine learning model. The expected value of assigning a unit into a particular treatment is then given by

$$\mathbb{E}[(v - cc_w)Y_w - ic_w]$$

where Y_w is the probability of a favourable event (such as conversion) under a given treatment w, v is the value of the favourable event, cc_w is the cost of the treatment triggered in case of a favourable event, and ic_w is the cost associated with the treatment whether or not the outcome is favourable. This method builds upon the ideas discussed in [24].

2.4 Selected traditional methods

The package supports selected traditional causal inference methods. These are usually used to conduct causal inference with observational (non-experimental) data. In these types of studies, the observed difference between the treatment and the control is in general not equal to the difference between "potential outcomes" $\mathbb{E}[Y(1) - Y(0)]$. Thus, the methods below try to deal with this problem in different ways.

2.4.1 Matching

The general idea in matching is to find treated and non-treated units that are as similar as possible in terms of their relevant characteristics. As such, matching methods can be seen as part of the family of causal inference approaches that try to mimic randomized controlled trials.

While there are a number of different ways to match treated and non-treated units, the most common method is to use the propensity score:

$$e_i(X_i) = P(W_i = 1 \mid X_i)$$

Treated and non-treated units are then matched in terms of $e(X)$ using some criterion of distance, such as $k : 1$ nearest neighbours. Because matching is usually between the treated population and the control, this method estimates the average treatment effect on the treated (ATT):

$$\mathbb{E}[Y(1) \mid W = 1] - \mathbb{E}[Y(0) \mid W = 1]$$

See [22] for a discussion of the strengths and weaknesses of the different matching methods.

2.4.2 Inverse probability of treatment weighting

The inverse probability of treatment weighting (IPTW) approach uses the propensity score e to weigh the treated and non-treated populations by the inverse of the probability of the actual treatment W. For a binary treatment $W \in \{1, 0\}$:

$$\frac{W}{e} + \frac{1 - W}{1 - e}$$

In this way, the IPTW approach can be seen as creating an artificial population in which the treated and non-treated units are similar in terms of their observed features X.

One of the possible benefits of IPTW compared to matching is that less data may be discarded due to lack of overlap between treated and non-treated units. A known problem with the approach is that extreme propensity scores can generate highly variable estimators. Different methods have been proposed for trimming and normalizing the IPT weights ([12]). An overview of the IPTW approach can be found in [7].

2.4.3 2-Stage Least Squares (2SLS)

One of the basic requirements for identifying the treatment effect of W on Y is that W is orthogonal to the potential outcome of Y, conditional on the covariates X. This may be violated if both W and Y are affected by an unobserved variable, the error term after removing the true effect of W from Y, that is not in X. In this case, the instrumental variables approach attempts to estimate the effect of W on Y with the help of a third variable Z that is correlated with W but is uncorrelated with the error term. In other words, the instrument Z is only related with Y through the directed path that goes through W. If these conditions are satisfied, in the case without covariates, the effect of W on Y can be estimated using the sample analog of:

$$\frac{Cov(Y_i, Z_i)}{Cov(W_i, Z_i)}$$

The most common method for instrumental variables estimation is the two-stage least squares (2SLS). In this approach, the cause variable W is first regressed on the instrument Z. Then, in the second stage, the outcome of interest Y is regressed on the predicted value from the first-stage model. Intuitively, the effect of W on Y is estimated by using only the proportion of variation in W due to variation in Z. Specifically, assume that we have the linear model

$$Y = W\alpha + X\beta + u = \Xi\gamma + u$$

Here for convenience we let $\Xi = [W, X]$ and $\gamma = [\alpha', \beta']'$. Assume that we have instrumental variables Z whose number of columns is at least the number of columns of W, let $\Omega = [Z, X]$, 2SLS estimator is as follows

$$\hat{\gamma}_{2SLS} = \left[\Xi'\Omega(\Omega'\Omega)^{-1}\Omega'\Xi\right]^{-1}\left[\Xi'\Omega'(\Omega'\Omega)^{-1}\Omega'Y\right].$$

See [3] for a detailed discussion of the method.

2.4.4 LATE

In many situations the treatment W may depend on user's own choice and cannot be administered directly in an experimental setting. However one can randomly assign users into treatment/control groups so that users in the treatment group can be nudged to take the treatment. This is the case of noncompliance, where users may fail to comply with their assignment status, Z, as to whether to take treatment or not. Similar to the section of Value optimization methods, in general there are 3 types of users in this situation,

- **Compliers** Those who will take the treatment if and only if they are assigned to the treatment group.
- **Always-Taker** Those who will take the treatment regardless which group they are assigned to.
- **Never-Taker** Those who wil not take the treatment regardless which group they are assigned to.

However one assumes that there is no Defier for identification purposes, i.e. those who will only take the treatment if they are assigned to the control group.

In this case one can measure the treatment effect of Compliers,

$$\hat{\tau}_{Complier} = \frac{E[Y|Z=1] - E[Y|Z=0]}{E[W|Z=1] - E[W|Z=0]}$$

This is Local Average Treatment Effect (LATE). The estimator is also equivalent to 2SLS if we take the assignment status, Z, as an instrument.

2.5 Targeted maximum likelihood estimation (TMLE) for ATE

Targeted maximum likelihood estimation (TMLE) [16] provides a doubly robust semiparametric method that "targets" directly on the average treatment effect with the aid from machine learning algorithms. Compared to other methods including outcome regression and inverse probability of treatment weighting, TMLE usually gives better performance especially when dealing with skewed treatment and outliers.

Given binary treatment W, covariates X, and outcome Y, the TMLE for ATE is performed in the following steps

Step 1

Use cross fit to estimate the propensity score $\hat{e}(x)$, the predicted outcome for treated $\hat{m}_1(x)$, and predicted outcome for control $\hat{m}_0(x)$ with machine learning.

Step 2

Scale Y into $\tilde{Y} = \frac{Y - \min Y}{\max Y - \min Y}$ so that $\tilde{Y} \in [0, 1]$. Use the same scale function to transform $\hat{m}_i(x)$ into $\tilde{m}_i(x)$, $i = 0, 1$. Clip the scaled functions so that their values stay in the unit interval.

Step 3

Let $Q = \log(\tilde{m}_W(X)/(1 - \tilde{m}_W(X)))$. Maximize the following pseudo log-likelihood function

$$\max_{h_0, h_1} -\frac{1}{N} \sum_i \left[\tilde{Y}_i \log \left(1 + \exp(-Q_i - h_0 \frac{1-W}{1-\hat{e}(X_i)} - h_1 \frac{W}{\hat{e}(X_i)} \right) \right.$$
$$\left. + (1 - \tilde{Y}_i) \log \left(1 + \exp(Q_i + h_0 \frac{1-W}{1-\hat{e}(X_i)} + h_1 \frac{W}{\hat{e}(X_i)} \right) \right]$$

Step 4

Let

$$\tilde{Q}_0 = \frac{1}{1 + \exp\left(-Q - h_0 \frac{1}{1-\hat{e}(X)}\right)},$$
$$\tilde{Q}_1 = \frac{1}{1 + \exp\left(-Q - h_1 \frac{1}{\hat{e}(X)}\right)}.$$

The ATE estimate is the sample average of the differences of $\tilde{Q}_1$ and $\tilde{Q}_0$ after rescale to the original range.

CHAPTER

THREE

INSTALLATION

Installation with conda is recommended. conda environment files for Python 3.6, 3.7, 3.8 and 3.9 are available in the repository. To use models under the `inference.tf` module (e.g. `DragonNet`), additional dependency of `tensorflow` is required. For detailed instructions, see below.

3.1 Install using conda

This will create a new conda virtual environment named `causalml-[tf-]py3x`, where `x` is in `[6, 7, 8, 9]`. e.g. `causalml-py37` or `causalml-tf-py38`. If you want to change the name of the environment, update the relevant YAML file in `envs/`.

```
$ git clone https://github.com/uber/causalml.git
$ cd causalml/envs/
$ conda env create -f environment-py38.yml  # for the virtual environment with Python 3.
↪8 and CausalML
$ conda activate causalml-py38
(causalml-py38)
```

To install `causalml` with `tensorflow` using conda, use a relevant `causalml-[tf-]py3x` environment file as follows:

```
$ git clone https://github.com/uber/causalml.git
$ cd causalml/envs/
$ conda env create -f environment-tf-py38.yml       # for the virtual environment with␣
↪Python 3.8 and CausalML
$ conda activate causalml-tf-py38
(causalml-tf-py38) pip install -U numpy                      # this step is necessary to␣
↪fix [#338](https://github.com/uber/causalml/issues/338)
```

3.2 Install using pip

```
$ git clone https://github.com/uber/causalml.git
$ cd causalml
$ pip install -r requirements.txt
$ pip install causalml
```

To install `causalml` with `tensorflow` using `pip`, use `causalml[tf]` as follows:

```
$ git clone https://github.com/uber/causalml.git
$ cd causalml
$ pip install -r requirements-tf.txt
$ pip install causalml[tf]
$ pip install -U numpy                                                    # this step␣
↪is necessary to fix [#338](https://github.com/uber/causalml/issues/338)
```

3.3 Install from source

```
$ git clone https://github.com/uber/causalml.git
$ cd causalml
$ pip install -r requirements.txt
$ python setup.py build_ext --inplace
$ python setup.py install
```

CHAPTER

FOUR

EXAMPLES

Working example notebooks are available in the example folder.

4.1 Propensity Score

4.1.1 Propensity Score Estimation

```
from causalml.propensity import ElasticNetPropensityModel

pm = ElasticNetPropensityModel(n_fold=5, random_state=42)
ps = pm.fit_predict(X, y)
```

4.1.2 Propensity Score Matching

```
from causalml.match import NearestNeighborMatch, create_table_one

psm = NearestNeighborMatch(replace=False,
                           ratio=1,
                           random_state=42)
matched = psm.match_by_group(data=df,
                             treatment_col=treatment_col,
                             score_col=score_col,
                             groupby_col=groupby_col)

create_table_one(data=matched,
                 treatment_col=treatment_col,
                 features=covariates)
```

4.2 Average Treatment Effect (ATE) Estimation

4.2.1 Meta-learners and Uplift Trees

In addition to the Methodology section, you can find examples in the links below for *Meta-Learner Algorithms* and *Tree-Based Algorithms*

- Meta-learners (S/T/X/R): meta_learners_with_synthetic_data.ipynb
- Meta-learners (S/T/X/R) with multiple treatment: meta_learners_with_synthetic_data_multiple_treatment.ipynb
- Comparing meta-learners across simulation setups: benchmark_simulation_studies.ipynb
- Doubly Robust (DR) learner: dr_learner_with_synthetic_data.ipynb
- TMLE learner: validation_with_tmle.ipynb
- Uplift Trees: uplift_trees_with_synthetic_data.ipynb

```
from causalml.inference.meta import LRSRegressor
from causalml.inference.meta import XGBTRegressor, MLPTRegressor
from causalml.inference.meta import BaseXRegressor
from causalml.inference.meta import BaseRRegressor
from xgboost import XGBRegressor
from causalml.dataset import synthetic_data

y, X, treatment, _, _, e = synthetic_data(mode=1, n=1000, p=5, sigma=1.0)

lr = LRSRegressor()
te, lb, ub = lr.estimate_ate(X, treatment, y)
print('Average Treatment Effect (Linear Regression): {:.2f} ({:.2f}, {:.2f})'.
↪format(te[0], lb[0], ub[0]))

xg = XGBTRegressor(random_state=42)
te, lb, ub = xg.estimate_ate(X, treatment, y)
print('Average Treatment Effect (XGBoost): {:.2f} ({:.2f}, {:.2f})'.format(te[0], lb[0],␣
↪ub[0]))

nn = MLPTRegressor(hidden_layer_sizes=(10, 10),
                   learning_rate_init=.1,
                   early_stopping=True,
                   random_state=42)
te, lb, ub = nn.estimate_ate(X, treatment, y)
print('Average Treatment Effect (Neural Network (MLP)): {:.2f} ({:.2f}, {:.2f})'.
↪format(te[0], lb[0], ub[0]))

xl = BaseXRegressor(learner=XGBRegressor(random_state=42))
te, lb, ub = xl.estimate_ate(X, treatment, y, e)
print('Average Treatment Effect (BaseXRegressor using XGBoost): {:.2f} ({:.2f}, {:.2f})'.
↪format(te[0], lb[0], ub[0]))

rl = BaseRRegressor(learner=XGBRegressor(random_state=42))
te, lb, ub =  rl.estimate_ate(X=X, p=e, treatment=treatment, y=y)
print('Average Treatment Effect (BaseRRegressor using XGBoost): {:.2f} ({:.2f}, {:.2f})'.
↪format(te[0], lb[0], ub[0]))
```

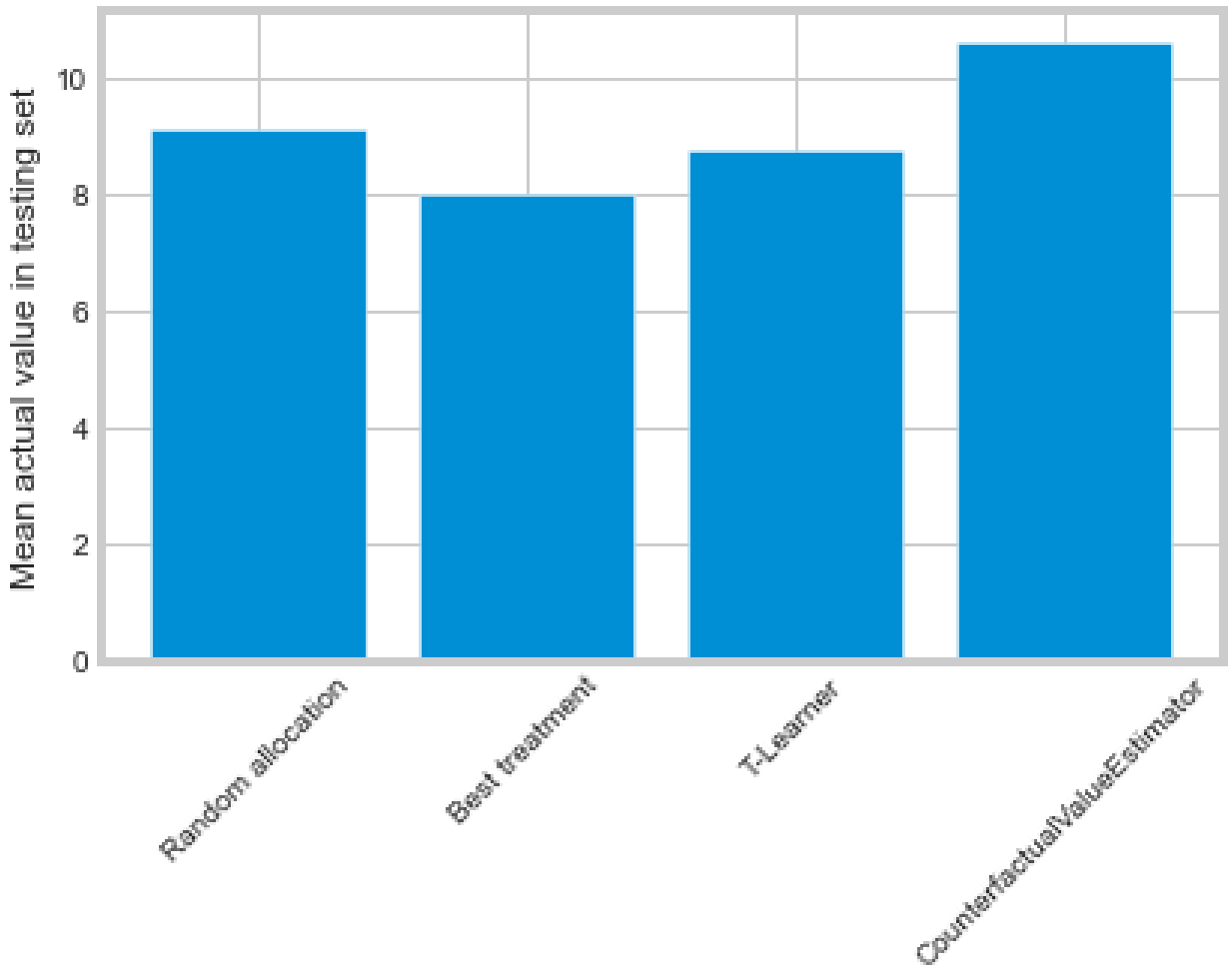
4.3 More algorithms

4.3.1 Treatment optimization algorithms

We have developed *Counterfactual Unit Selection* and *Counterfactual Value Estimator* methods, please find the code snippet below and details in the following notebooks:

- counterfactual_unit_selection.ipynb
- counterfactual_value_optimization.ipynb

```
from causalml.optimize import CounterfactualValueEstimator
from causalml.optimize import get_treatment_costs, get_actual_value

# load data set and train test split
df_train, df_test = train_test_split(df)
train_idx = df_train.index
test_idx = df_test.index
# some more code here to initiate and train the Model, and produce tm_pred
# please refer to the counterfactual_value_optimization notebook for complete example

# run the counterfactual calculation with TwoModel prediction
cve = CounterfactualValueEstimator(treatment=df_test['treatment_group_key'],
                                   control_name='control',
                                   treatment_names=conditions[1:],
                                   y_proba=y_proba,
                                   cate=tm_pred,
                                   value=conversion_value_array[test_idx],
                                   conversion_cost=cc_array[test_idx],
                                   impression_cost=ic_array[test_idx])

cve_best_idx = cve.predict_best()
cve_best = [conditions[idx] for idx in cve_best_idx]
actual_is_cve_best = df.loc[test_idx, 'treatment_group_key'] == cve_best
cve_value = actual_value.loc[test_idx][actual_is_cve_best].mean()

labels = [
    'Random allocation',
    'Best treatment',
    'T-Learner',
    'CounterfactualValueEstimator'
]
values  = [
    random_allocation_value,
    best_ate_value,
    tm_value,
    cve_value
]
# plot the result
plt.bar(labels, values)
```



4.3.2 Instrumental variables algorithms

- 2-Stage Least Squares (2SLS): iv_nlsym_synthetic_data.ipynb

4.3.3 Neural network based algorithms

- CEVAE: cevae_example.ipynb
- DragonNet: dragonnet_example.ipynb

4.4 Interpretation

Please see *Interpretable Causal ML* section

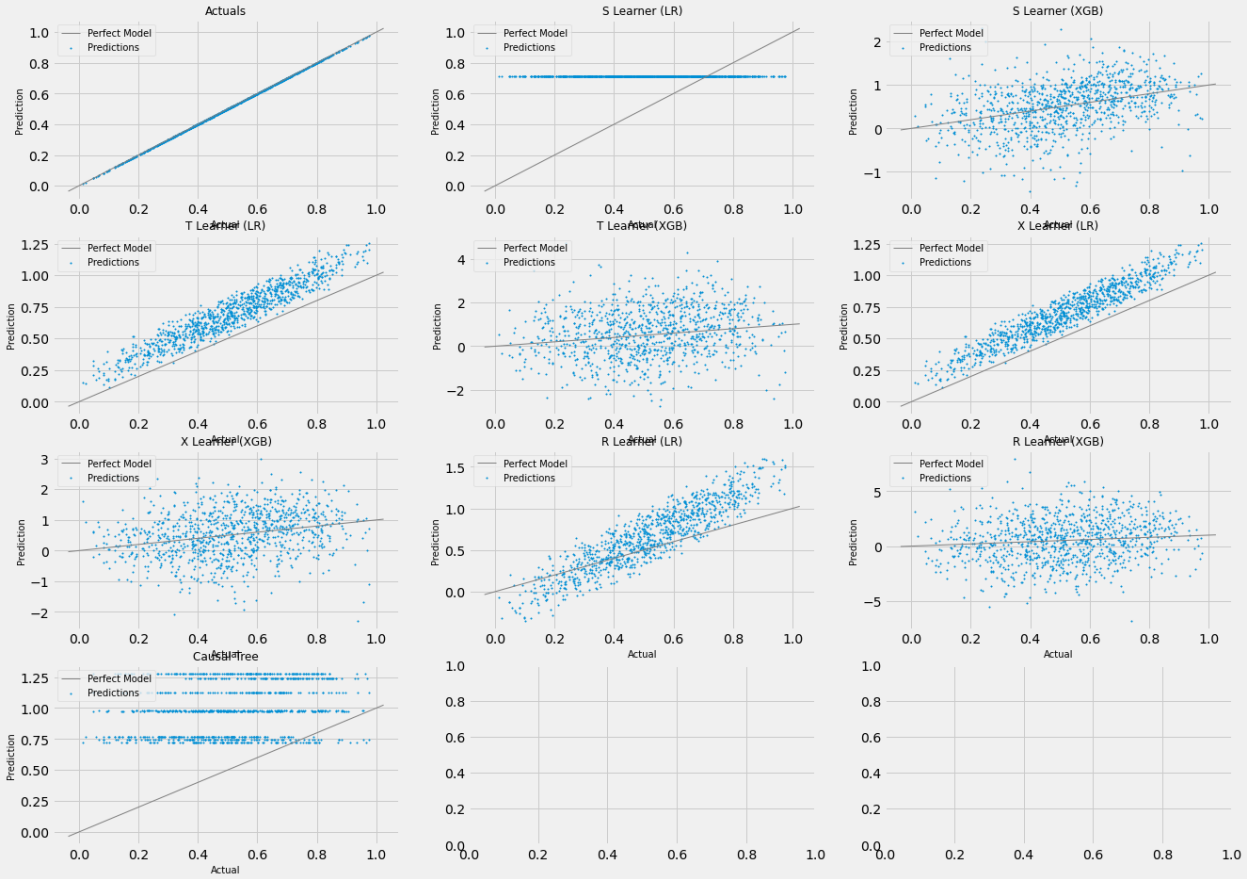
4.5 Validation

Please see *Validation* section

4.6 Synthetic Data Generation Process

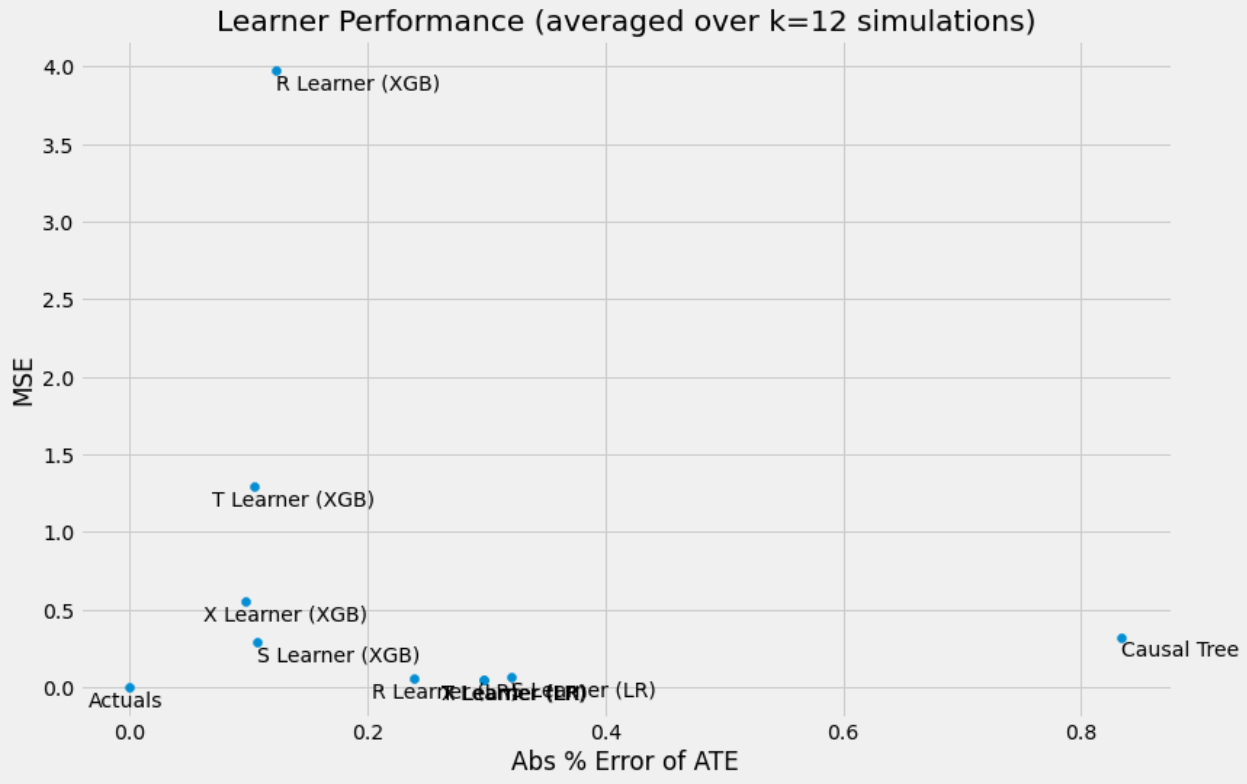
4.6.1 Single Simulation

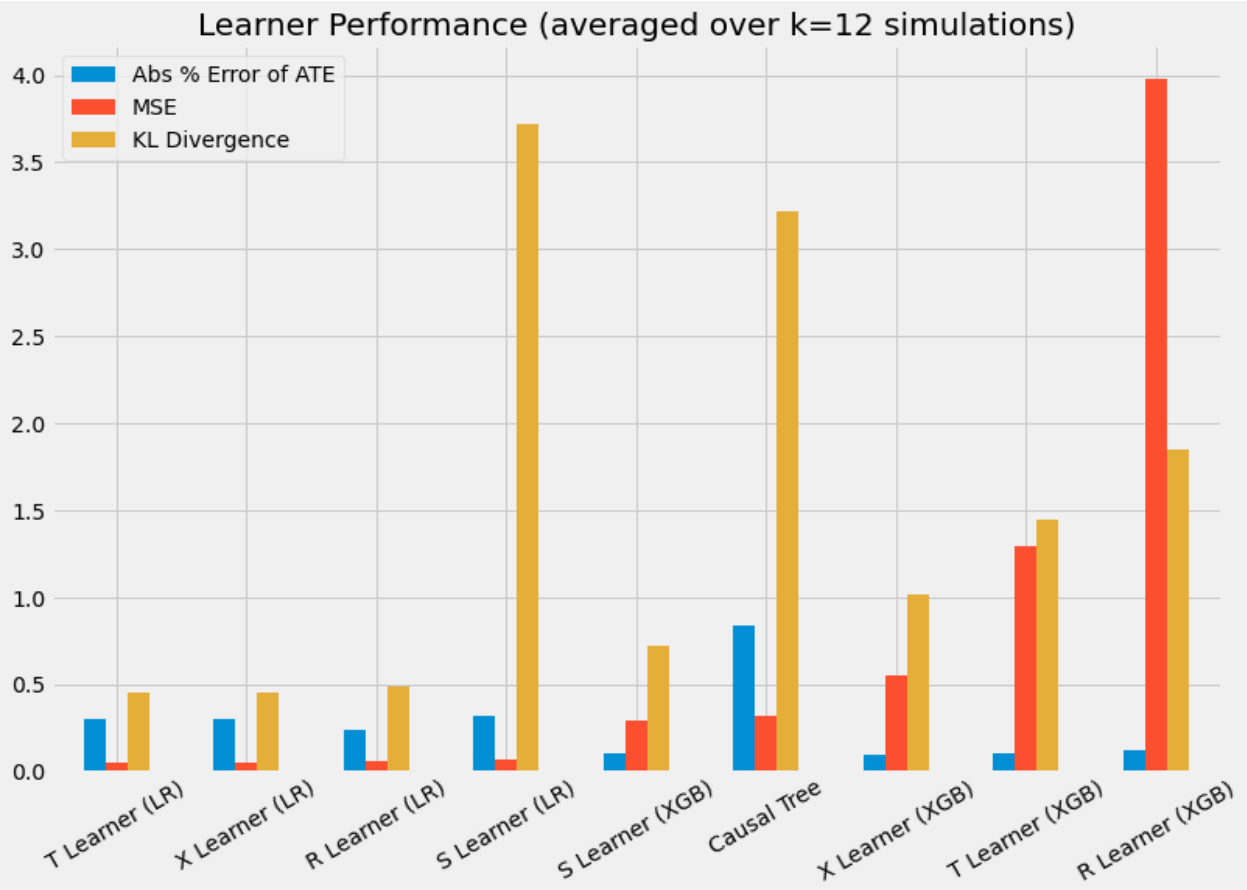
```
from causalml.dataset import *

# Generate synthetic data for single simulation
y, X, treatment, tau, b, e = synthetic_data(mode=1)
y, X, treatment, tau, b, e = simulate_nuisance_and_easy_treatment()

# Generate predictions for single simulation
single_sim_preds = get_synthetic_preds(simulate_nuisance_and_easy_treatment, n=1000)

# Generate multiple scatter plots to compare learner performance for a single simulation
scatter_plot_single_sim(single_sim_preds)

# Visualize distribution of learner predictions for a single simulation
distr_plot_single_sim(single_sim_preds, kind='kde')
```



4.6.2 Multiple Simulations

```
from causalml.dataset import *

# Generalize performance summary over k simulations
num_simulations = 12
preds_summary = get_synthetic_summary(simulate_nuisance_and_easy_treatment, n=1000,␣
↪k=num_simulations)

# Generate scatter plot of performance summary
scatter_plot_summary(preds_summary, k=num_simulations)

# Generate bar plot of performance summary
bar_plot_summary(preds_summary, k=num_simulations)
```

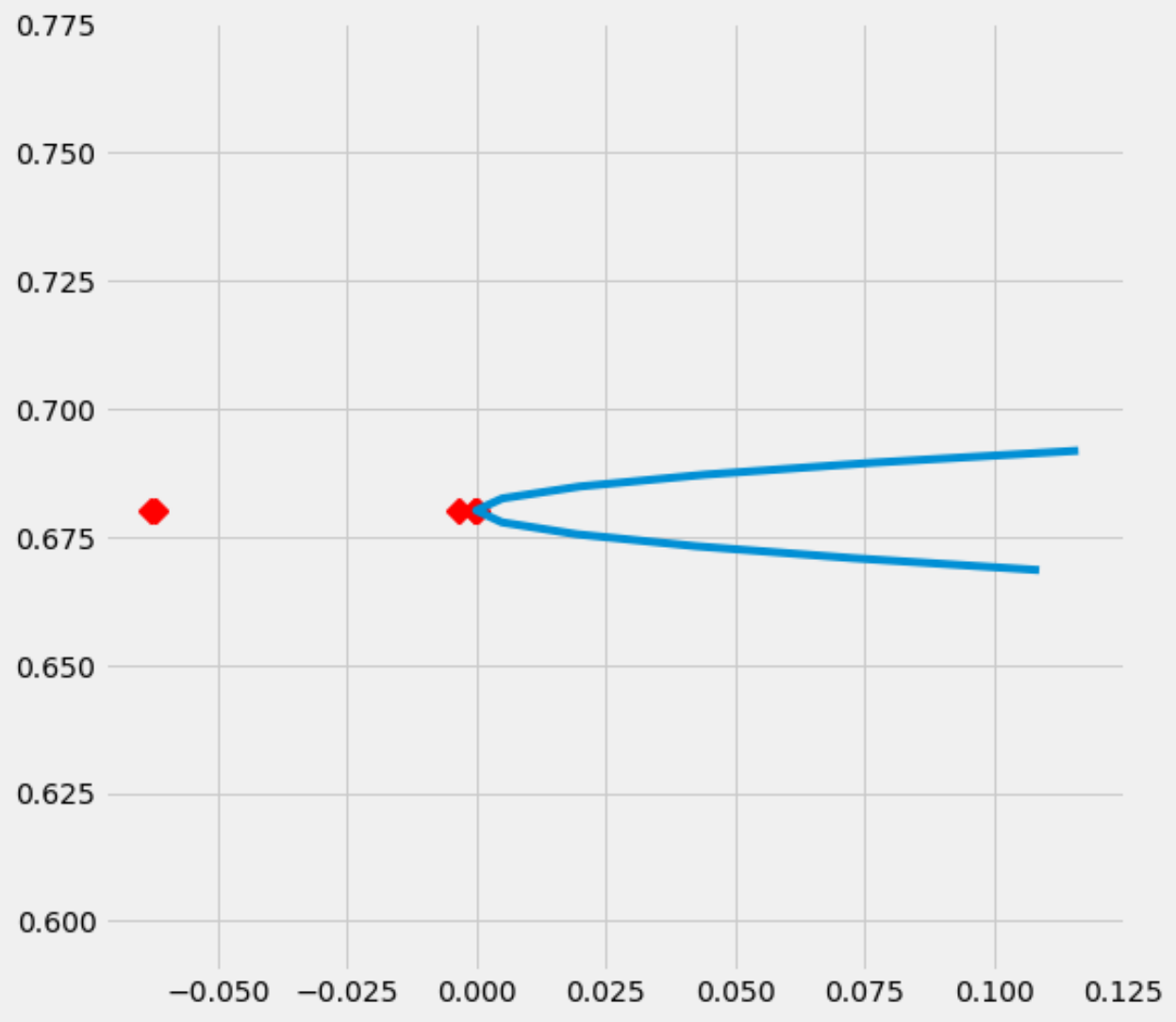

Learner Performance (averaged over k=12 simulations)
MSE
Abs % Error of ATE
R Learner (XGB)
T Learner (XGB)
X Learner (XGB)
S Learner (XGB)
Causal Tree
Actuals
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8


Learner Performance (averaged over k=12 simulations)
Abs % Error of ATE
MSE
KL Divergence
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
T Learner (LR)
X Learner (LR)
R Learner (LR)
S Learner (LR)
S Learner (XGB)
Causal Tree
X Learner (XGB)
T Learner (XGB)
R Learner (XGB)

4.7 Sensitivity Analysis

For more details, please refer to the sensitivity_example_with_synthetic_data.ipynb notebook.

```
from causalml.metrics.sensitivity import Sensitivity
from causalml.metrics.sensitivity import SensitivitySelectionBias
from causalml.inference.meta import BaseXLearner
from sklearn.linear_model import LinearRegression

# Calling the Base XLearner class and return the sensitivity analysis summary report
learner_x = BaseXLearner(LinearRegression())
sens_x = Sensitivity(df=df, inference_features=INFERENCE_FEATURES, p_col='pihat',
                     treatment_col=TREATMENT_COL, outcome_col=OUTCOME_COL,␣
↪learner=learner_x)
# Here for Selection Bias method will use default one-sided confounding function and␣
↪alpha (quantile range of outcome values) input
sens_sumary_x = sens_x.sensitivity_analysis(methods=['Placebo Treatment',
                                                     'Random Cause',
                                                     'Subset Data',
                                                     'Random Replace',
                                                     'Selection Bias'], sample_size=0.5)

# Selection Bias: Alignment confounding Function
sens_x_bias_alignment = SensitivitySelectionBias(df, INFERENCE_FEATURES, p_col='pihat',␣
↪treatment_col=TREATMENT_COL,
                                                 outcome_col=OUTCOME_COL, learner=learner_x,␣
↪confound='alignment',
                                                 alpha_range=None)
# Plot the results by rsquare with partial r-square results by each individual features
sens_x_bias_alignment.plot(lls_x_bias_alignment, partial_rsqs_x_bias_alignment, type='r.
↪squared', partial_rsqs=True)
```



4.8 Feature Selection

For more details, please refer to the feature_selection.ipynb notebook and the associated paper reference by Zhao, Zhenyu, et al.

```
from causalml.feature_selection.filters import FilterSelect
from causalml.dataset import make_uplift_classification

# define parameters for simulation
y_name = 'conversion'
treatment_group_keys = ['control', 'treatment1']
n = 100000
n_classification_features = 50
n_classification_informative = 10
n_classification_repeated = 0
n_uplift_increase_dict = {'treatment1': 8}
n_uplift_decrease_dict = {'treatment1': 4}
```

(continues on next page)

(continued from previous page)

```
delta_uplift_increase_dict = {'treatment1': 0.1}
delta_uplift_decrease_dict = {'treatment1': -0.1}

# make a synthetic uplift data set
random_seed = 20200808
df, X_names = make_uplift_classification(
    treatment_name=treatment_group_keys,
    y_name=y_name,
    n_samples=n,
    n_classification_features=n_classification_features,
    n_classification_informative=n_classification_informative,
    n_classification_repeated=n_classification_repeated,
    n_uplift_increase_dict=n_uplift_increase_dict,
    n_uplift_decrease_dict=n_uplift_decrease_dict,
    delta_uplift_increase_dict = delta_uplift_increase_dict,
    delta_uplift_decrease_dict = delta_uplift_decrease_dict,
    random_seed=random_seed
)

# Feature selection with Filter method
filter_f = FilterSelect()
method = 'F'
f_imp = filter_f.get_importance(df, X_names, y_name, method,
                        treatment_group = 'treatment1')
print(f_imp)

# Use likelihood ratio test method
method = 'LR'
lr_imp = filter_f.get_importance(df, X_names, y_name, method,
                        treatment_group = 'treatment1')
print(lr_imp)

# Use KL divergence method
method = 'KL'
kl_imp = filter_f.get_importance(df, X_names, y_name, method,
                        treatment_group = 'treatment1',
                        n_bins=10)
print(kl_imp)
```

CHAPTER

FIVE

INTERPRETABLE CAUSAL ML

Causal ML provides methods to interpret the treatment effect models trained, where we provide more sample code in feature_interpretations_example.ipynb notebook.

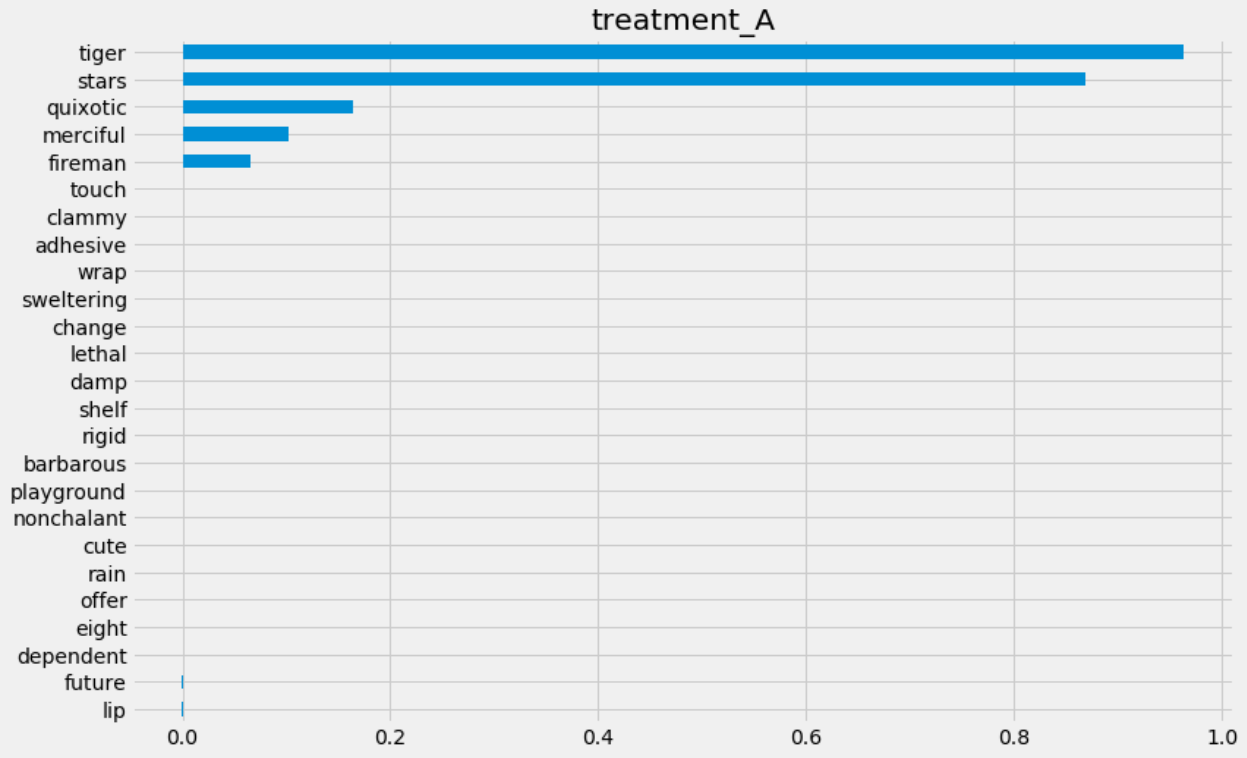
5.1 Meta-Learner Feature Importances

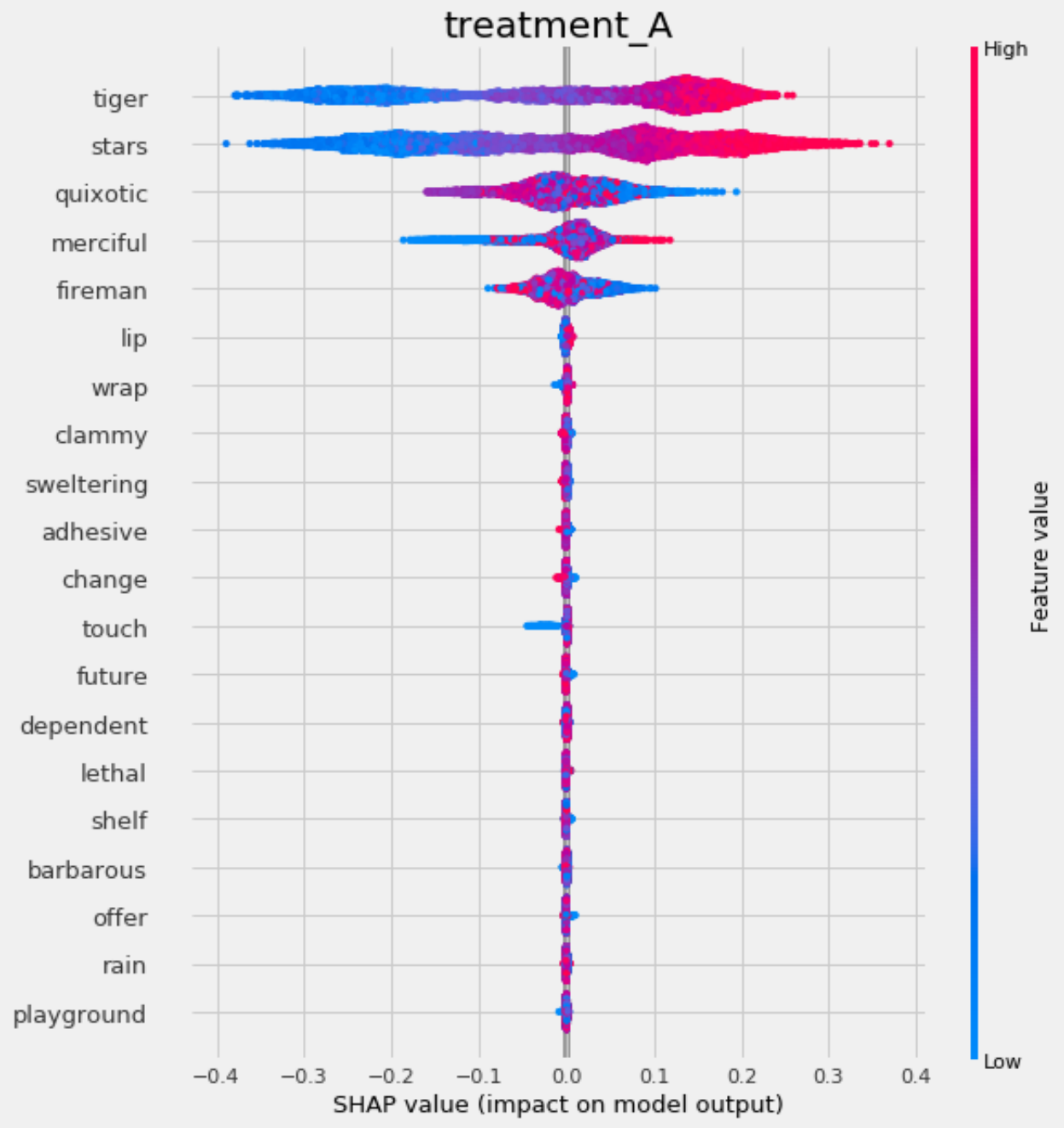
```
from causalml.inference.meta import BaseSRegressor, BaseTRegressor, BaseXRegressor,␣
↪BaseRRegressor

slearner = BaseSRegressor(LGBMRegressor(), control_name='control')
slearner.estimate_ate(X, w_multi, y)
slearner_tau = slearner.fit_predict(X, w_multi, y)

model_tau_feature = RandomForestRegressor()  # specify model for model_tau_feature

slearner.get_importance(X=X, tau=slearner_tau, model_tau_feature=model_tau_feature,
                        normalize=True, method='auto', features=feature_names)

# Using the feature_importances_ method in the base learner (LGBMRegressor() in this␣
↪example)
slearner.plot_importance(X=X, tau=slearner_tau, normalize=True, method='auto')

# Using eli5's PermutationImportance
slearner.plot_importance(X=X, tau=slearner_tau, normalize=True, method='permutation')

# Using SHAP
shap_slearner = slearner.get_shap_values(X=X, tau=slearner_tau)

# Plot shap values without specifying shap_dict
slearner.plot_shap_values(X=X, tau=slearner_tau)

# Plot shap values WITH specifying shap_dict
slearner.plot_shap_values(X=X, shap_dict=shap_slearner)

# interaction_idx set to 'auto' (searches for feature with greatest approximate␣
↪interaction)
slearner.plot_shap_dependence(treatment_group='treatment_A',
                              feature_idx=1,
                              X=X,
```

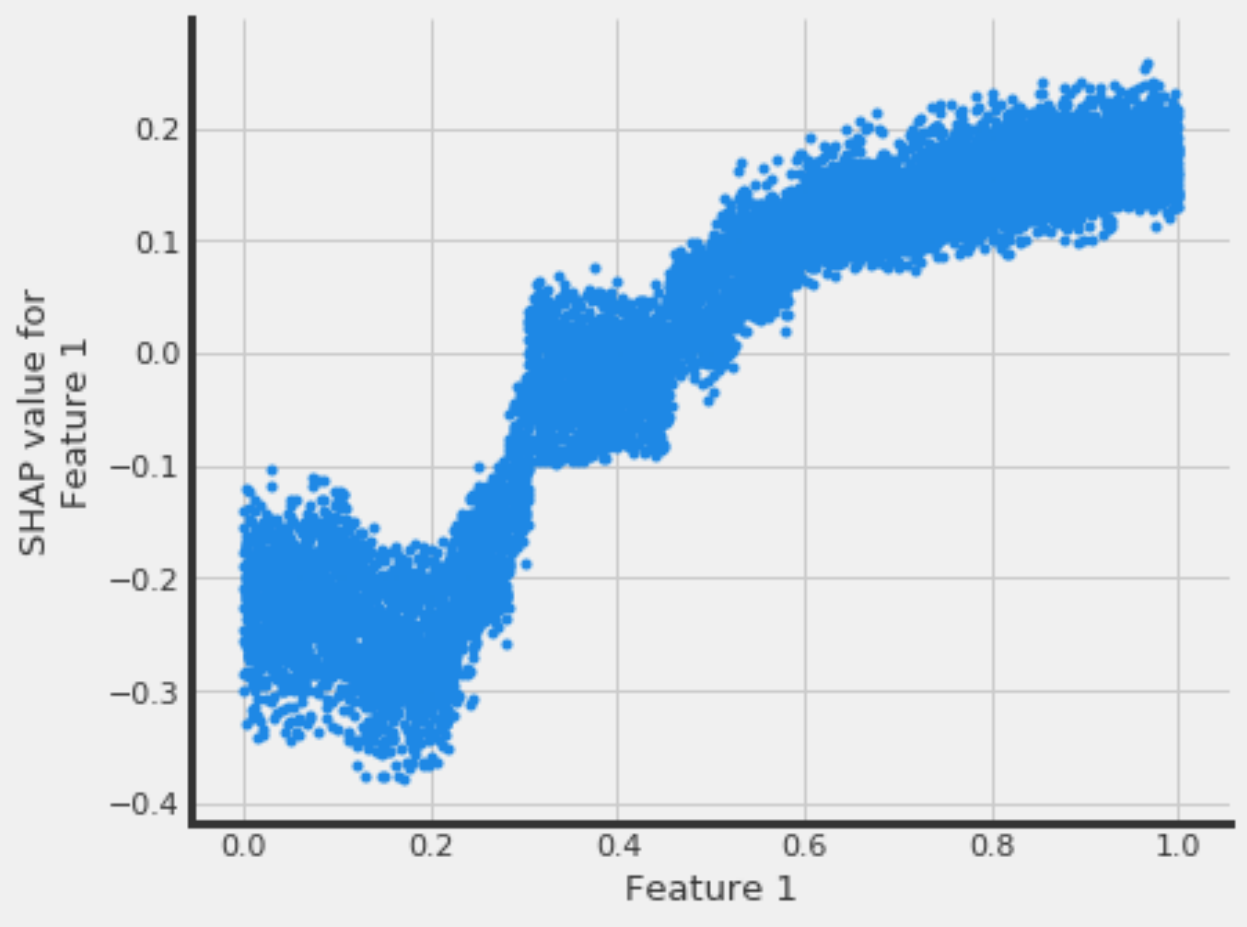
(continues on next page)

(continued from previous page)

```
                                tau=slearner_tau,
                                interaction_idx='auto')
```

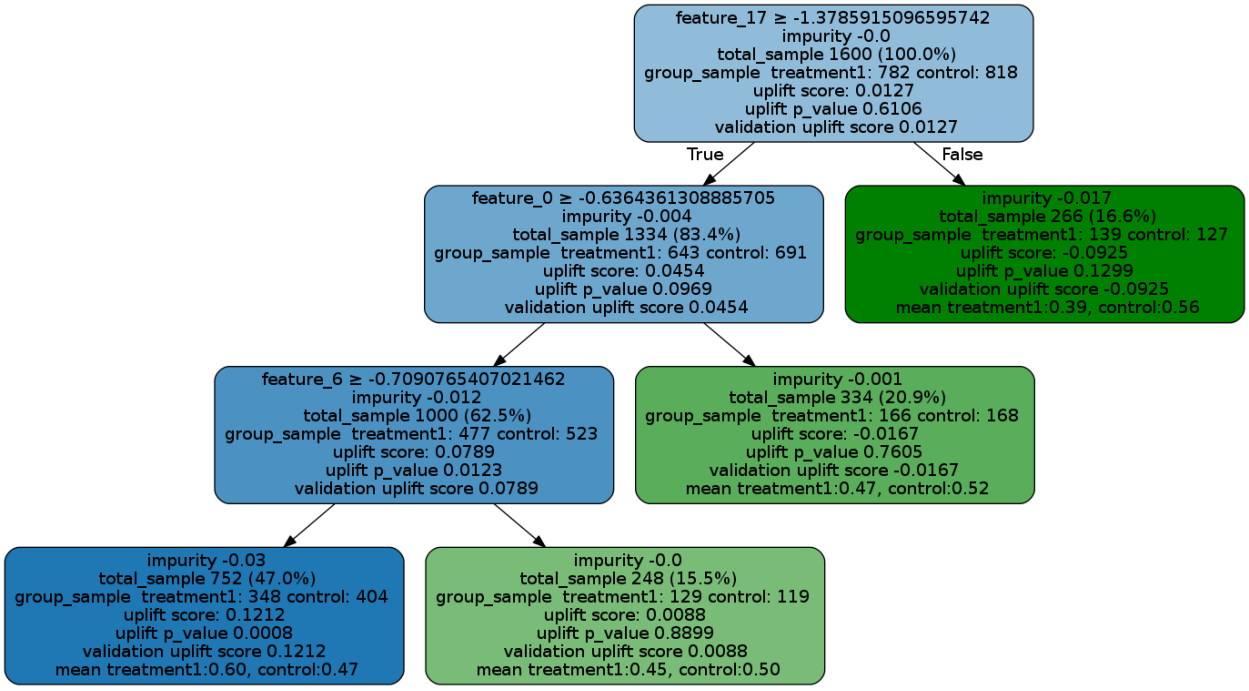



treatment_A
tiger
stars
quixotic
merciful
fireman
lip
wrap
clammy
sweltering
adhesive
change
touch
future
dependent
lethal
shelf
barbarous
offer
rain
playground
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
SHAP value (impact on model output)
High
Low
Feature value



5.2 Uplift Tree Visualization

```
from IPython.display import Image
from causalml.inference.tree import UpliftTreeClassifier, UpliftRandomForestClassifier
from causalml.inference.tree import uplift_tree_string, uplift_tree_plot
from causalml.dataset import make_uplift_classification

df, x_names = make_uplift_classification()
uplift_model = UpliftTreeClassifier(max_depth=5, min_samples_leaf=200, min_samples_
↪treatment=50,
                                    n_reg=100, evaluationFunction='KL', control_name=
↪'control')

uplift_model.fit(df[x_names].values,
                 treatment=df['treatment_group_key'].values,
                 y=df['conversion'].values)

graph = uplift_tree_plot(uplift_model.fitted_uplift_tree, x_names)
Image(graph.create_png())
```

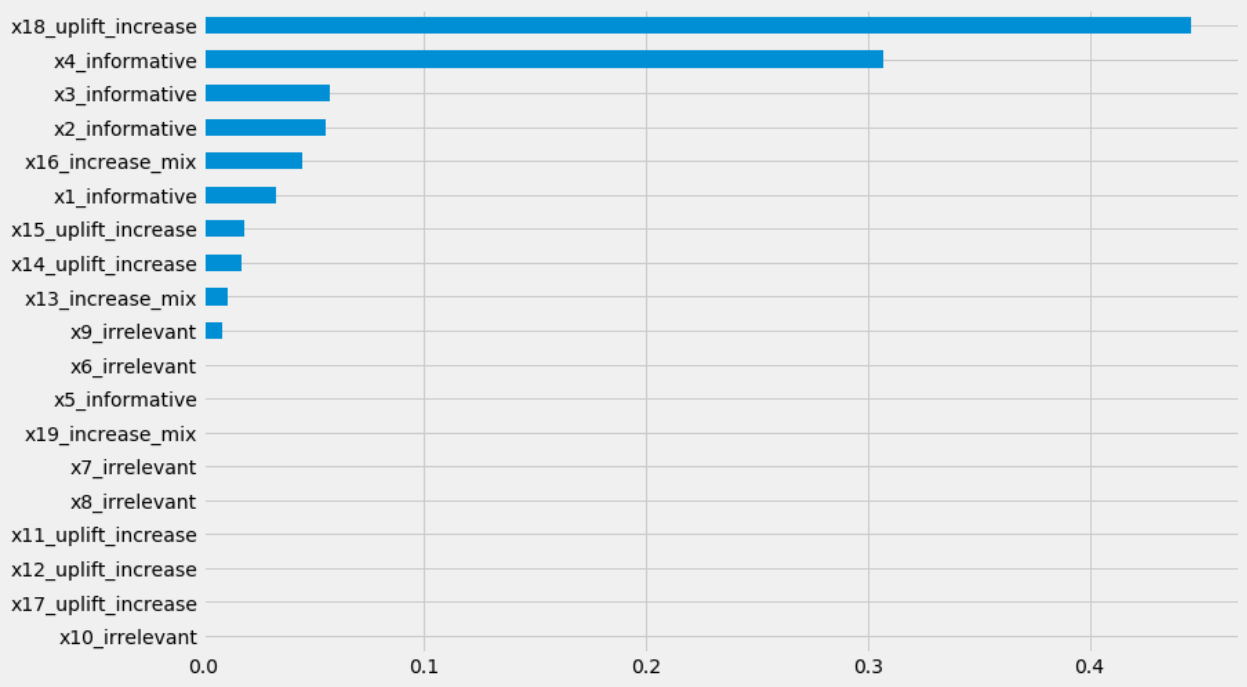


Please see below for how to read the plot, and uplift_tree_visualization.ipynb example notebook is provided in the repo.

- feature_name > threshold: For non-leaf node, the first line is an inequality indicating the splitting rule of this node to its children nodes.
- impurity: the impurity is defined as the value of the split criterion function (such as KL, Chi, or ED) evaluated at this current node
- total_sample: sample size in this node.
- group_sample: sample sizes by treatment groups
- uplift score: treatment effect in this node, if there are multiple treatment, it indicates the maximum (signed) of the treatment effects across all treatment vs control pairs.
- uplift p_value: p value of the treatment effect in this node
- validation uplift score: all the information above is static once the tree is trained (based on the trained trees), while the validation uplift score represents the treatment effect of the testing data when the method fill() is used. This score can be used as a comparison to the training uplift score, to evaluate if the tree has an overfitting issue.

5.3 Uplift Tree Feature Importances

```
pd.Series(uplift_model.feature_importances_, index=x_names).sort_values().plot(kind='barh
↪', figsize=(12,8))
```


x18_uplift_increase
x4_informative
x3_informative
x2_informative
x16_increase_mix
x1_informative
x15_uplift_increase
x14_uplift_increase
x13_increase_mix
x9_irrelevant
x6_irrelevant
x5_informative
x19_increase_mix
x7_irrelevant
x8_irrelevant
x11_uplift_increase
x12_uplift_increase
x17_uplift_increase
x10_irrelevant
0.0
0.1
0.2
0.3
0.4

CHAPTER

SIX

VALIDATION

Estimation of the treatment effect cannot be validated the same way as regular ML predictions because the true value is not available except for the experimental data. Here we focus on the internal validation methods under the assumption of unconfoundedness of potential outcomes and the treatment status conditioned on the feature set available to us.

6.1 Validation with Multiple Estimates

We can validate the methodology by comparing the estimates with other approaches, checking the consistency of estimates across different levels and cohorts.

6.1.1 Model Robustness for Meta Algorithms

In meta-algorithms we can assess the quality of user-level treatment effect estimation by comparing estimates from different underlying ML algorithms. We will report MSE, coverage (overlapping 95% confidence interval), uplift curve. In addition, we can split the sample within a cohort and compare the result from out-of-sample scoring and within-sample scoring.

6.1.2 User Level/Segment Level/Cohort Level Consistency

We can also evaluate user-level/segment level/cohort level estimation consistency by conducting T-test.

6.1.3 Stability between Cohorts

Treatment effect may vary from cohort to cohort but should not be too volatile. For a given cohort, we will compare the scores generated by model fit to another score with the ones generated by its own model.

6.2 Validation with Synthetic Data Sets

We can test the methodology with simulations, where we generate data with known causal and non-causal links between the outcome, treatment and some of confounding variables.

We implemented the following sets of synthetic data generation mechanisms based on [18]:

6.2.1 Mechanism 1

This generates a complex outcome regression model with easy treatment effect with input variables $X_i \sim Unif(0,1)^d$.
The treatment flag is a binomial variable, whose d.g.p. is:

:math:`P(W_i = 1 | X_i) = trim_{0.1}(sin(pi X_{i1} X_{i2})

With :

$$trim_\eta(x) = \max(\eta, \min(x, 1-\eta))$$

The outcome variable is:

$$y_i = sin(\pi X_{i1} X_{i2}) + 2(X_{i3} - 0.5)^2 + X_{i4} + 0.5 X_{i5} + (W_i - 0.5)(X_{i1} + X_{i2})/2 + \epsilon_i$$

6.2.2 Mechanism 2

This simulates a randomized trial. The input variables are generated by $X_i \sim N(0, I_{d \times d})$

The treatment flag is generated by a fair coin flip:

$$P(W_i = 1 | X_i) = 0.5$$

The outcome variable is

$$y_i = max(X_{i1} + X_{i2}, X_{i3}, 0) + max(X_{i4} + X_{i5}, 0) + (W_i - 0.5)(X_{i1} + \log(1 + e^{X_{i2}}))$$

6.2.3 Mechanism 3

This one has an easy propensity score but a difficult control outcome. The input variables follow $X_i \sim N(0, I_{d \times d})$

The treatment flag is a binomial variable, whose d.g.p is:

$$P(W_i = 1 | X_i) = \frac{1}{1 + \exp X_{i2} + X_{i3}}$$

The outcome variable is:

$$y_i = 2 \log(1 + e^{X_{i1} + X_{i2} + X_{i3}}) + (W_i - 0.5)$$

6.2.4 Mechanism 4

This contains an unrelated treatment arm and control arm, with input data generated by $X_i \sim N(0, I_{d\times d})$.

The treatment flag is a binomial variable whose d.g.p. is:

$$P(W_i = 1|X_i) = \frac{1}{1+\exp -X_{i1}+\exp -X_{i2}}$$

The outcome variable is:

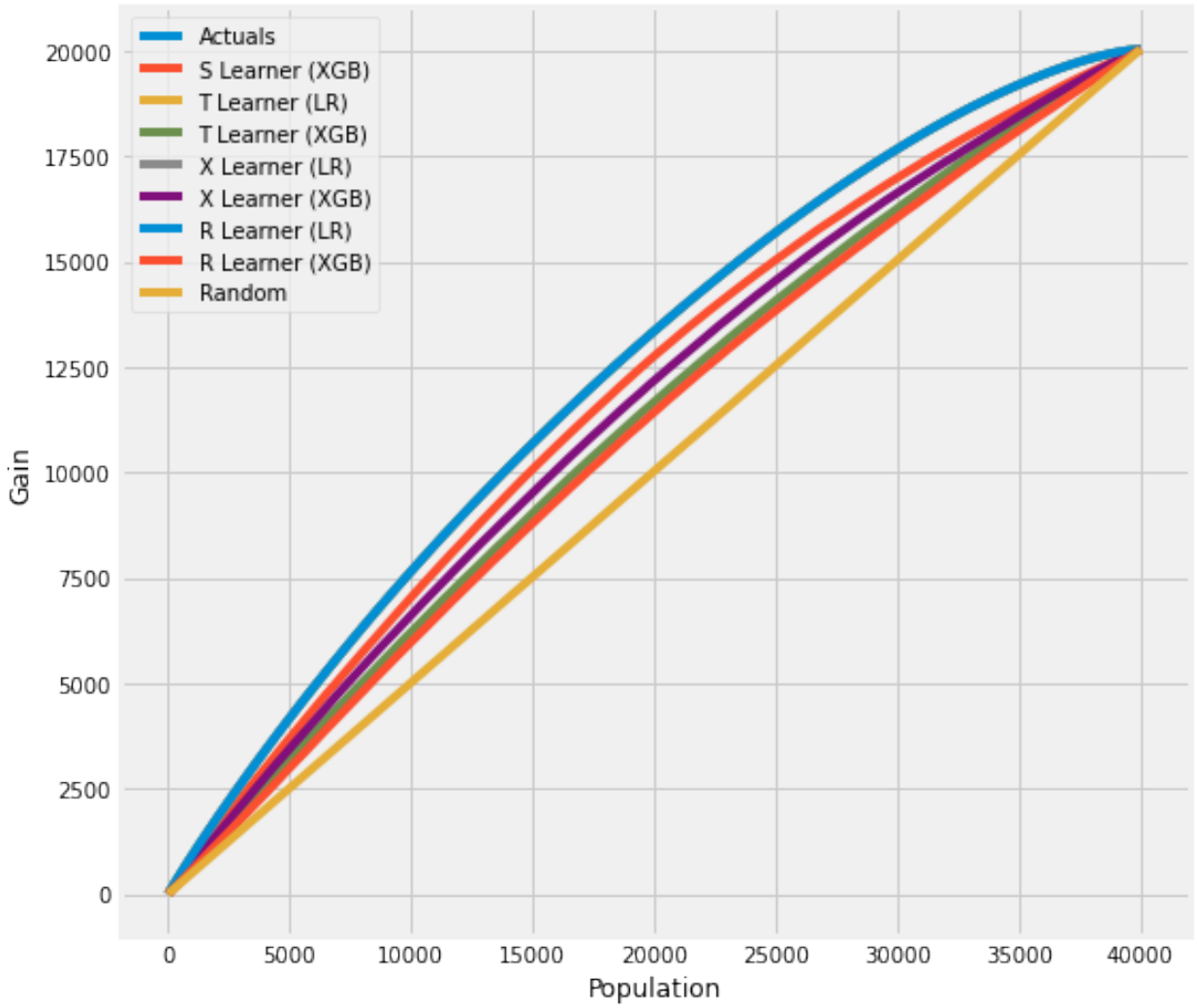
$$y_i = \frac{1}{2}\big(max(X_{i1}+X_{i2}+X_{i3},0)+max(X_{i4}+X_{i5},0)\big)+(W_i-0.5)(max(X_{i1}+X_{i2}+X_{i3},0)-max(X_{i4},X_{i5},0))$$

6.3 Validation with Uplift Curve (AUUC)

We can validate the estimation by evaluating and comparing the uplift gains with AUUC (Area Under Uplift Curve), it calculates cumulative gains. Please find more details in meta_learners_with_synthetic_data.ipynb example notebook.

```
from causalml.dataset import *
from causalml.metrics import *
# Single simulation
train_preds, valid_preds = get_synthetic_preds_holdout(simulate_nuisance_and_easy_
↪treatment,
                                                       n=50000,
                                                       valid_size=0.2)
# Cumulative Gain AUUC values for a Single Simulation of Validaiton Data
get_synthetic_auuc(valid_preds)
```

	Learner	cum_gain_auuc
0	Actuals	4.942619e+06
6	R Learner (LR)	4.941699e+06
2	T Learner (LR)	4.941643e+06
4	X Learner (LR)	4.941643e+06
1	S Learner (XGB)	4.723843e+06
5	X Learner (XGB)	4.580028e+06
3	T Learner (XGB)	4.446320e+06
7	R Learner (XGB)	4.364945e+06
8	Random	4.010939e+06



For data with skewed treatment, it is sometimes advantageous to use *Targeted maximum likelihood estimation (TMLE) for ATE* to generate the AUUC curve for validation, as TMLE provides a more accurate estimation of ATE. Please find validation_with_tmle.ipynb example notebook for details.

6.4 Validation with Sensitivity Analysis

Sensitivity analysis aim to check the robustness of the unconfoundeness assumption. If there is hidden bias (unobserved confounders), it detemineds how severe whould have to be to change conclusion by examine the average treatment effect estimation.

We implemented the following methods to conduct sensitivity analysis:

6.4.1 Placebo Treatment

Replace treatment with a random variable.

6.4.2 Irrelevant Additional Confounder

Add a random common cause variable.

6.4.3 Subset validation

Remove a random subset of the data.

6.4.4 Random Replace

Random replace a covariate with an irrelevant variable.

6.4.5 Selection Bias

Blackwell(2013) <https://www.mattblackwell.org/files/papers/sens.pdf> introduced an approach to sensitivity analysis for causal effects that directly models confounding or selection bias.

One Sided Confounding Function: here as the name implies, this function can detect sensitivity to one-sided selection bias, but it would fail to detect other deviations from ignobility. That is, it can only determine the bias resulting from the treatment group being on average better off or the control group being on average better off.

Alignment Confounding Function: this type of bias is likely to occur when units select into treatment and control based on their predicted treatment effects

The sensitivity analysis is rigid in this way because the confounding function is not identified from the data, so that the causal model in the last section is only identified conditional on a specific choice of that function. The goal of the sensitivity analysis is not to choose the "correct" confounding function, since we have no way of evaluating this correctness. By its very nature, unmeasured confounding is unmeasured. Rather, the goal is to identify plausible deviations from ignobility and test sensitivity to those deviations. The main harm that results from the incorrect specification of the confounding function is that hidden biases remain hidden.

CHAPTER

SEVEN

CAUSALML PACKAGE

7.1 Submodules

7.2 causalml.inference.tree module

class causalml.inference.tree.**CausalMSE**

Bases: sklearn.tree._criterion.RegressionCriterion

Causal Tree mean squared error impurity criterion.

CausalTreeMSE = right_effect + left_effect

where,

effect = alpha * tau^2 - (1 - alpha) * (1 + train_to_est_ratio) * (VAR_tr / p + VAR_cont / (1 - p))

class causalml.inference.tree.**CausalTreeRegressor**(*ate_alpha=0.05*, *control_name=0*, *max_depth=None*, *min_samples_leaf=100*, *random_state=None*)

Bases: object

A Causal Tree regressor class.

The Causal Tree is a decision tree regressor with a split criteria for treatment effects instead of outputs.

Details are available at Athey and Imbens (2015) (https://arxiv.org/abs/1504.01132)

bootstrap(*X*, *treatment*, *y*, *size=10000*)

Runs a single bootstrap.

Fits on bootstrapped sample, then predicts on whole population.

Parameters

- **X** (*np.matrix*) – a feature matrix
- **treatment** (*np.array*) – a treatment vector
- **y** (*np.array*) – an outcome vector
- **size** (*int, optional*) – bootstrap sample size

Returns bootstrap predictions

Return type (np.array)

estimate_ate(*X*, *treatment*, *y*)

Estimate the Average Treatment Effect (ATE).

Parameters

- **X** (*np.matrix*) – a feature matrix
- **treatment** (*np.array*) – a treatment vector
- **y** (*np.array*) – an outcome vector

Returns The mean and confidence interval (LB, UB) of the ATE estimate.

fit(*X*, *treatment*, *y*)
Fit the Causal Tree model

Parameters

- **X** (*np.matrix*) – a feature matrix
- **treatment** (*np.array*) – a treatment vector
- **y** (*np.array*) – an outcome vector

Returns self (CausalTree object)

fit_predict(*X*, *treatment*, *y*, *return_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*, *verbose=False*)
Fit the Causal Tree model and predict treatment effects.

Parameters

- **X** (*np.matrix*) – a feature matrix
- **treatment** (*np.array*) – a treatment vector
- **y** (*np.array*) – an outcome vector
- **return_ci** (*bool*) – whether to return confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap
- **verbose** (*str*) – whether to output progress logs

Returns

- te (numpy.ndarray): Predictions of treatment effects.
- te_lower (numpy.ndarray, optional): lower bounds of treatment effects
- te_upper (numpy.ndarray, optional): upper bounds of treatment effects

Return type (tuple)

predict(*X*)
Predict treatment effects.

Parameters **X** (*np.matrix*) – a feature matrix

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.tree.**DecisionTree**(*classes_*, *col=- 1*, *value=None*, *trueBranch=None*, *falseBranch=None*, *results=None*, *summary=None*, *maxDiffTreatment=None*, *maxDiffSign=1.0*, *nodeSummary=None*, *backupResults=None*, *bestTreatment=None*, *upliftScore=None*, *matchScore=None*)

Bases: object

Tree Node Class

Tree node class to contain all the statistics of the tree node.

Parameters

- **classes** (*list of str*) – A list of the control and treatment group names.
- **col** (*int, optional (default = -1)*) – The column index for splitting the tree node to children nodes.
- **value** (*float, optional (default = None)*) – The value of the feature column to split the tree node to children nodes.
- **trueBranch** (*object of DecisionTree*) – The true branch tree node (feature > value).
- **falseBranch** (*object of DecisionTree*) – The false branch tree node (feature > value).
- **results** (*list of float*) – The classification probability P(Y=1|T) for each of the control and treatment groups in the tree node.
- **summary** (*list of list*) – Summary statistics of the tree nodes, including impurity, sample size, uplift score, etc.
- **maxDiffTreatment** (*int*) – The treatment index generating the maximum difference between the treatment and control groups.
- **maxDiffSign** (*float*) – The sign of the maximum difference (1. or -1.).
- **nodeSummary** (*list of list*) – Summary statistics of the tree nodes [P(Y=1|T), N(T)], where y_mean stands for the target metric mean and n is the sample size.
- **backupResults** (*list of float*) – The positive probabilities in each of the control and treatment groups in the parent node. The parent node information is served as a backup for the children node, in case no valid statistics can be calculated from the children node, the parent node information will be used in certain cases.
- **bestTreatment** (*int*) – The treatment index providing the best uplift (treatment effect).
- **upliftScore** (*list*) – The uplift score of this node: [max_Diff, p_value], where max_Diff stands for the maximum treatment effect, and p_value stands for the p_value of the treatment effect.
- **matchScore** (*float*) – The uplift score by filling a trained tree with validation dataset or testing dataset.

class causalml.inference.tree.**UpliftRandomForestClassifier**(*control_name*, *n_estimators=10*, *max_features=10*, *random_state=None*, *max_depth=5*, *min_samples_leaf=100*, *min_samples_treatment=10*, *n_reg=10*, *evaluationFunction='KL'*, *normalization=True*, *n_jobs=- 1*)

Bases: object

Uplift Random Forest for Classification Task.

Parameters

- **n_estimators** (*integer, optional (default=10)*) – The number of trees in the uplift random forest.
- **evaluationFunction** (*string*) – Choose from one of the models: 'KL', 'ED', 'Chi', 'CTS', 'DDP'.
- **max_features** (*int, optional (default=10)*) – The number of features to consider when looking for the best split.

- **random_state** (*int, RandomState instance or None (default=None)*) – A random seed or *np.random.RandomState* to control randomness in building the trees and forest.
- **max_depth** (*int, optional (default=5)*) – The maximum depth of the tree.
- **min_samples_leaf** (*int, optional (default=100)*) – The minimum number of samples required to be split at a leaf node.
- **min_samples_treatment** (*int, optional (default=10)*) – The minimum number of samples required of the experiment group to be split at a leaf node.
- **n_reg** (*int, optional (default=10)*) – The regularization parameter defined in Rzepakowski et al. 2012, the weight (in terms of sample size) of the parent node influence on the child node, only effective for 'KL', 'ED', 'Chi', 'CTS' methods.
- **control_name** (*string*) – The name of the control group (other experiment groups will be regarded as treatment groups)
- **normalization** (*boolean, optional (default=True)*) – The normalization factor defined in Rzepakowski et al. 2012, correcting for tests with large number of splits and imbalanced treatment and control splits
- **n_jobs** (*int, optional (default=-1)*) – The parallelization parameter to define how many parallel jobs need to be created. This is passed on to joblib library for parallelizing uplift-tree creation.
- **Outputs** –
- **----------** –
- **df_res** (*pandas dataframe*) – A user-level results dataframe containing the estimated individual treatment effect.

static bootstrap(*X*, *treatment*, *y*, *tree*)

fit(*X*, *treatment*, *y*)

Fit the UpliftRandomForestClassifier.

Parameters

- **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.
- **treatment** (*array-like, shape = [num_samples]*) – An array containing the treatment group for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.

predict(*X*, *full_output=False*)

Returns the recommended treatment group and predicted optimal probability conditional on using the recommended treatment group.

Parameters

- **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.
- **full_output** (*bool, optional (default=False)*) – Whether the UpliftTree algorithm returns upliftScores, pred_nodes alongside the recommended treatment group and p_hat in the treatment group.

Returns

- **y_pred_list** (*ndarray, shape = (num_samples, num_treatments]*)) – An ndarray containing the predicted treatment effect of each treatment group for each sample
- **df_res** (*DataFrame, shape = [num_samples, (num_treatments * 2 + 3)]*) – If *full_output* is *True*, a DataFrame containing the predicted outcome of each treatment and control group, the treatment effect of each treatment group, the treatment group with the highest treatment effect, and the maximum treatment effect for each sample.

class causalml.inference.tree.**UpliftTreeClassifier**(*control_name*, *max_features=None*, *max_depth=3*, *min_samples_leaf=100*, *min_samples_treatment=10*, *n_reg=100*, *evaluationFunction='KL'*, *normalization=True*, *random_state=None*)

Bases: `object`

Uplift Tree Classifier for Classification Task.

A uplift tree classifier estimates the individual treatment effect by modifying the loss function in the classification trees.

The uplift tree classifier is used in uplift random forest to construct the trees in the forest.

Parameters

- **evaluationFunction** (*string*) – Choose from one of the models: 'KL', 'ED', 'Chi', 'CTS', 'DDP'.
- **max_features** (*int, optional (default=None)*) – The number of features to consider when looking for the best split.
- **max_depth** (*int, optional (default=3)*) – The maximum depth of the tree.
- **min_samples_leaf** (*int, optional (default=100)*) – The minimum number of samples required to be split at a leaf node.
- **min_samples_treatment** (*int, optional (default=10)*) – The minimum number of samples required of the experiment group to be split at a leaf node.
- **n_reg** (*int, optional (default=100)*) – The regularization parameter defined in Rzepakowski et al. 2012, the weight (in terms of sample size) of the parent node influence on the child node, only effective for 'KL', 'ED', 'Chi', 'CTS' methods.
- **control_name** (*string*) – The name of the control group (other experiment groups will be regarded as treatment groups).
- **normalization** (*boolean, optional (default=True)*) – The normalization factor defined in Rzepakowski et al. 2012, correcting for tests with large number of splits and imbalanced treatment and control splits.
- **random_state** (*int, RandomState instance or None (default=None)*) – A random seed or *np.random.RandomState* to control randomness in building a tree.

static **classify**(*observations*, *tree*, *dataMissing=False*)

Classifies (prediction) the observations according to the tree.

Parameters

- **observations** (*list of list*) – The internal data format for the training data (combining X, Y, treatment).
- **dataMissing** (*boolean, optional (default = False)*) – An indicator for if data are missing or not.

Returns The results in the leaf node.

Return type tree.results, tree.upliftScore

static divideSet(*X*, *treatment_idx*, *y*, *column*, *value*)
Tree node split.

Parameters

- **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.
- **treatment_idx** (*array-like, shape = [num_samples]*) – An array containing the treatment group index for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.
- **column** (*int*) – The column used to split the data.
- **value** (*float or int*) – The value in the column for splitting the data.

Returns **(X_l, X_r, treatment_l, treatment_r, y_l, y_r)** – The covariates, treatments and outcomes of left node and the right node.

Return type list of ndarray

static evaluate_CTS(*nodeSummary*)
Calculate CTS (conditional treatment selection) as split evaluation criterion for a given node.

Parameters **nodeSummary** (*list of list*) – The tree node summary statistics, [P(Y=1|T), N(T)], produced by tree_node_summary() method.

Returns **d_res**

Return type Chi-Square

static evaluate_Chi(*nodeSummary*)
Calculate Chi-Square statistic as split evaluation criterion for a given node.

Parameters **nodeSummary** (*dictionary*) – The tree node summary statistics, produced by tree_node_summary() method.

Returns **d_res**

Return type Chi-Square

static evaluate_DDP(*nodeSummary*)
Calculate Delta P as split evaluation criterion for a given node.

Parameters **nodeSummary** (*list of list*) – The tree node summary statistics, [P(Y=1|T), N(T)], produced by tree_node_summary() method.

Returns **d_res**

Return type Delta P

static evaluate_ED(*nodeSummary*)
Calculate Euclidean Distance as split evaluation criterion for a given node.

Parameters **nodeSummary** (*dictionary*) – The tree node summary statistics, produced by tree_node_summary() method.

Returns **d_res**

Return type Euclidean Distance

static evaluate_KL(*nodeSummary*)
Calculate KL Divergence as split evaluation criterion for a given node.

Parameters **nodeSummary** (*list of list*) – The tree node summary statistics, [P(Y=1|T), N(T)], produced by tree_node_summary() method.

Returns **d_res**

Return type KL Divergence

fill(*X*, *treatment*, *y*)

Fill the data into an existing tree. This is a higher-level function to transform the original data inputs into lower level data inputs (list of list and tree).

Parameters

- **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.
- **treatment** (*array-like, shape = [num_samples]*) – An array containing the treatment group for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.

Returns **self**

Return type object

fillTree(*X*, *treatment_idx*, *y*, *tree*)

Fill the data into an existing tree. This is a lower-level function to execute on the tree filling task.

Parameters

- **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.
- **treatment_idx** (*array-like, shape = [num_samples]*) – An array containing the treatment group index for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.
- **tree** (*object*) – object of DecisionTree class

Returns **self**

Return type object

fit(*X*, *treatment*, *y*)

Fit the uplift model.

Parameters

- **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.
- **treatment** (*array-like, shape = [num_samples]*) – An array containing the treatment group for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.

Returns **self**

Return type object

group_uniqueCounts(*treatment_idx*, *y*)

Count sample size by experiment group.

Parameters

- **treatment_idx** (*array-like, shape = [num_samples]*) – An array containing the treatment group index for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.

Returns **results** – The negative and positive outcome sample sizes for each of the control and treatment groups.

Return type list of list

growDecisionTreeFrom(*X*, *treatment_idx*, *y*, *max_depth=10*, *min_samples_leaf=100*, *depth=1*, *min_samples_treatment=10*, *n_reg=100*, *parentNodeSummary=None*)

Train the uplift decision tree.

Parameters

- **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.
- **treatment_idx** (*array-like, shape = [num_samples]*) – An array containing the treatment group idx for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.
- **max_depth** (*int, optional (default=10)*) – The maximum depth of the tree.
- **min_samples_leaf** (*int, optional (default=100)*) – The minimum number of samples required to be split at a leaf node.
- **depth** (*int, optional (default = 1)*) – The current depth.
- **min_samples_treatment** (*int, optional (default=10)*) – The minimum number of samples required of the experiment group to be split at a leaf node.
- **n_reg** (*int, optional (default=10)*) – The regularization parameter defined in Rzepakowski et al. 2012, the weight (in terms of sample size) of the parent node influence on the child node, only effective for 'KL', 'ED', 'Chi', 'CTS' methods.
- **parentNodeSummary** (*dictionary, optional (default = None)*) – Node summary statistics of the parent tree node.

Returns

Return type object of DecisionTree class

normI(*n_c: int*, *n_c_left: int*, *n_t: list*, *n_t_left: list*, *alpha: float = 0.9*) → float

Normalization factor.

Parameters

- **currentNodeSummary** (*list of list*) – The summary statistics of the current tree node, [P(Y=1|T), N(T)].
- **leftNodeSummary** (*list of list*) – The summary statistics of the left tree node, [P(Y=1|T), N(T)].
- **alpha** (*float*) – The weight used to balance different normalization parts.

Returns **norm_res** – Normalization factor.

Return type float

predict(*X*)

Returns the recommended treatment group and predicted optimal probability conditional on using the recommended treatment group.

Parameters **X** (*ndarray, shape = [num_samples, num_features]*) – An ndarray of the covariates used to train the uplift model.

Returns **pred** – An ndarray of predicted treatment effects across treatments.

Return type ndarray, shape = [num_samples, num_treatments]

prune(*X*, *treatment*, *y*, *minGain=0.0001*, *rule='maxAbsDiff'*)

Prune the uplift model. :param X: An ndarray of the covariates used to train the uplift model. :type X: ndarray, shape = [num_samples, num_features] :param treatment: An array containing the treatment group for each unit. :type treatment: array-like, shape = [num_samples] :param y: An array containing the outcome of interest for each unit. :type y: array-like, shape = [num_samples] :param minGain: The minimum gain required to make a tree node split. The children

tree branches are trimmed if the actual split gain is less than the minimum gain.

Parameters **rule** (*string, optional (default = 'maxAbsDiff')*) – The prune rules. Supported values are 'maxAbsDiff' for optimizing the maximum absolute difference, and 'bestUplift' for optimizing the node-size weighted treatment effect.

Returns **self**

Return type object

pruneTree(*X*, *treatment_idx*, *y*, *tree*, *rule='maxAbsDiff'*, *minGain=0.0*, *n_reg=0*, *parentNodeSummary=None*)

Prune one single tree node in the uplift model. :param X: An ndarray of the covariates used to train the uplift model. :type X: ndarray, shape = [num_samples, num_features] :param treatment_idx: An array containing the treatment group index for each unit. :type treatment_idx: array-like, shape = [num_samples] :param y: An array containing the outcome of interest for each unit. :type y: array-like, shape = [num_samples] :param rule: The prune rules. Supported values are 'maxAbsDiff' for optimizing the maximum absolute difference, and

'bestUplift' for optimizing the node-size weighted treatment effect.

Parameters

- **minGain** (*float, optional (default = 0.)*) – The minimum gain required to make a tree node split. The children tree branches are trimmed if the actual split gain is less than the minimum gain.
- **n_reg** (*int, optional (default=0)*) – The regularization parameter defined in Rzepakowski et al. 2012, the weight (in terms of sample size) of the parent node influence on the child node, only effective for 'KL', 'ED', 'Chi', 'CTS' methods.
- **parentNodeSummary** (*list of list, optional (default = None)*) – Node summary statistics, [P(Y=1|T), N(T)] of the parent tree node.

Returns **self**

Return type object

tree_node_summary(*treatment_idx*, *y*, *min_samples_treatment=10*, *n_reg=100*, *parentNodeSummary=None*)

Tree node summary statistics.

Parameters

- **treatment_idx** (*array-like, shape = [num_samples]*) – An array containing the treatment group index for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.
- **min_samples_treatment** (*int, optional (default=10)*) – The minimum number of samples required of the experiment group t be split at a leaf node.
- **n_reg** (*int, optional (default=10)*) – The regularization parameter defined in Rzepakowski et al. 2012, the weight (in terms of sample size) of the parent node influence on the child node, only effective for 'KL', 'ED', 'Chi', 'CTS' methods.
- **parentNodeSummary** (*list of list*) – The positive probabilities and sample sizes of each of the control and treatment groups in the parent node.

Returns **nodeSummary** – The positive probabilities and sample sizes of each of the control and treatment groups in the current node.

Return type list of list

uplift_classification_results(*treatment_idx*, *y*)

Classification probability for each treatment in the tree node.

Parameters

- **treatment_idx** (*array-like, shape = [num_samples]*) – An array containing the treatment group index for each unit.
- **y** (*array-like, shape = [num_samples]*) – An array containing the outcome of interest for each unit.

Returns **res** – The positive probabilities P(Y = 1) of each of the control and treatment groups

Return type list of list

causalml.inference.tree.**cat_continuous**(*x*, *granularity='Medium'*)

Categorize (bin) continuous variable based on percentile.

Parameters

- **x** (*list*) – Feature values.
- **granularity** (*string, optional, (default = 'Medium')*) – Control the granularity of the bins, optional values are: 'High', 'Medium', 'Low'.

Returns **res** – List of percentile bins for the feature value.

Return type list

causalml.inference.tree.**cat_group**(*dfx*, *kpix*, *n_group=10*)

Category Reduction for Categorical Variables

Parameters

- **dfx** (*dataframe*) – The inputs data dataframe.
- **kpix** (*string*) – The column of the feature.
- **n_group** (*int, optional (default = 10)*) – The number of top category values to be remained, other category values will be put into "Other".

Returns

Return type The transformed categorical feature value list.

causalml.inference.tree.**cat_transform**(*dfx*, *kpix*, *kpi1*)
Encoding string features.

Parameters

- **dfx** (*dataframe*) – The inputs data dataframe.
- **kpix** (*string*) – The column of the feature.
- **kpi1** (*list*) – The list of feature names.

Returns

- **dfx** (*DataFrame*) – The updated dataframe containing the encoded data.
- **kpi1** (*list*) – The updated feature names containing the new dummy feature names.

causalml.inference.tree.**cv_fold_index**(*n*, *i*, *k*, *random_seed=2018*)
Encoding string features.

Parameters

- **dfx** (*dataframe*) – The inputs data dataframe.
- **kpix** (*string*) – The column of the feature.
- **kpi1** (*list*) – The list of feature names.

Returns

- **dfx** (*DataFrame*) – The updated dataframe containing the encoded data.
- **kpi1** (*list*) – The updated feature names containing the new dummy feature names.

causalml.inference.tree.**kpi_transform**(*dfx*, *kpi_combo*, *kpi_combo_new*)
Feature transformation from continuous feature to binned features for a list of features

Parameters

- **dfx** (*DataFrame*) – DataFrame containing the features.
- **kpi_combo** (*list of string*) – List of feature names to be transformed
- **kpi_combo_new** (*list of string*) – List of new feature names to be assigned to the transformed features.

Returns **dfx** – Updated DataFrame containing the new features.

Return type DataFrame

causalml.inference.tree.**uplift_tree_plot**(*decisionTree*, *x_names*)
Convert the tree to dot graph for plots.

Parameters

- **decisionTree** (*object*) – object of DecisionTree class
- **x_names** (*list*) – List of feature names

Returns

Return type Dot class representing the tree graph.

causalml.inference.tree.**uplift_tree_string**(*decisionTree*, *x_names*)
Convert the tree to string for print.

Parameters

- **decisionTree** (*object*) – object of DecisionTree class

- **x_names** (*list*) – List of feature names

Returns

Return type A string representation of the tree.

7.3 causalml.inference.meta module

class causalml.inference.meta.**BaseDRLearner**(*learner=None*, *control_outcome_learner=None*, *treatment_outcome_learner=None*, *treatment_effect_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: causalml.inference.meta.base.BaseLearner

A parent class for DR-learner regressor classes.

A DR-learner estimates treatment effects with machine learning models.

Details of DR-learner are available at Kennedy (2020) (https://arxiv.org/abs/2004.14497).

estimate_ate(*X*, *treatment*, *y*, *p=None*, *bootstrap_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*, *seed=None*)

Estimate the Average Treatment Effect (ATE).

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **bootstrap_ci** (*bool*) – whether run bootstrap for confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap
- **seed** (*int*) – random seed for cross-fitting

Returns The mean and confidence interval (LB, UB) of the ATE estimate.

fit(*X*, *treatment*, *y*, *p=None*, *seed=None*)

Fit the inference model.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **seed** (*int*) – random seed for cross-fitting

fit_predict(*X*, *treatment*, *y*, *p=None*, *return_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*, *return_components=False*, *verbose=True*, *seed=None*)

Fit the treatment effect and outcome models of the R learner and predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **return_ci** (*bool*) – whether to return confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap
- **return_components** (*bool, optional*) – whether to return outcome for treatment and control seperately
- **verbose** (*str*) – whether to output progress logs
- **seed** (*int*) – random seed for cross-fitting

Returns

Predictions of treatment effects. Output dim: [n_samples, n_treatment] If return_ci, returns CATE [n_samples, n_treatment], LB [n_samples, n_treatment], UB [n_samples, n_treatment]

Return type (numpy.ndarray)

predict(*X*, *treatment=None*, *y=None*, *p=None*, *return_components=False*, *verbose=True*)

Predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series, optional*) – a treatment vector
- **y** (*np.array or pd.Series, optional*) – an outcome vector
- **verbose** (*bool, optional*) – whether to output progress logs

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseDRRegressor**(*learner=None*, *control_outcome_learner=None*, *treatment_outcome_learner=None*, *treatment_effect_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.drlearner.BaseDRLearner*

A parent class for DR-learner regressor classes.

class causalml.inference.meta.**BaseRClassifier**(*outcome_learner=None*, *effect_learner=None*, *propensity_learner=LogisticRegressionCV(Cs=array([1.00230524, 2.15608891, 4.63802765, 9.97700064]), cv=StratifiedKFold(n_splits=4, random_state=42, shuffle=True), l1_ratios=array([0.001, 0.33366667, 0.66633333, 0.999]), penalty='elasticnet', random_state=42, solver='saga')*, *ate_alpha=0.05*, *control_name=0*, *n_fold=5*, *random_state=None*)

Bases: causalml.inference.meta.rlearner.BaseRLearner

A parent class for R-learner classifier classes.

fit(*X*, *treatment*, *y*, *p=None*, *sample_weight=None*, *verbose=True*)

Fit the treatment effect and outcome models of the R learner.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **sample_weight** (*np.array or pd.Series, optional*) – an array of sample weights indicating the weight of each observation for *effect_learner*. If None, it assumes equal weight.
- **verbose** (*bool, optional*) – whether to output progress logs

predict(*X*, *p=None*)

Predict treatment effects.

Parameters **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseRLearner**(*learner=None*, *outcome_learner=None*, *effect_learner=None*, *propensity_learner=LogisticRegressionCV(Cs=array([1.00230524, 2.15608891, 4.63802765, 9.97700064]), cv=StratifiedKFold(n_splits=4, random_state=42, shuffle=True), l1_ratios=array([0.001, 0.33366667, 0.66633333, 0.999]), penalty='elasticnet', random_state=42, solver='saga')*, *ate_alpha=0.05*, *control_name=0*, *n_fold=5*, *random_state=None*)

Bases: causalml.inference.meta.base.BaseLearner

A parent class for R-learner classes.

An R-learner estimates treatment effects with two machine learning models and the propensity score.

Details of R-learner are available at Nie and Wager (2019) (https://arxiv.org/abs/1712.04912).

estimate_ate(*X*, *treatment*, *y*, *p=None*, *sample_weight=None*, *bootstrap_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*)

Estimate the Average Treatment Effect (ATE).

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **sample_weight** (*np.array or pd.Series, optional*) – an array of sample weights indicating the weight of each observation for *effect_learner*. If None, it assumes equal weight.
- **bootstrap_ci** (*bool*) – whether run bootstrap for confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap

Returns The mean and confidence interval (LB, UB) of the ATE estimate.

fit(*X*, *treatment*, *y*, *p=None*, *sample_weight=None*, *verbose=True*)

Fit the treatment effect and outcome models of the R learner.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **sample_weight** (*np.array or pd.Series, optional*) – an array of sample weights indicating the weight of each observation for *effect_learner*. If None, it assumes equal weight.
- **verbose** (*bool, optional*) – whether to output progress logs

fit_predict(*X*, *treatment*, *y*, *p=None*, *sample_weight=None*, *return_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*, *verbose=True*)

Fit the treatment effect and outcome models of the R learner and predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.

- **sample_weight** (*np.array or pd.Series, optional*) – an array of sample weights indicating the weight of each observation for *effect_learner*. If None, it assumes equal weight.
- **return_ci** (*bool*) – whether to return confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap
- **verbose** (*bool*) – whether to output progress logs

Returns

Predictions of treatment effects. Output dim: [n_samples, n_treatment]. If return_ci, returns CATE [n_samples, n_treatment], LB [n_samples, n_treatment], UB [n_samples, n_treatment]

Return type (numpy.ndarray)

predict(*X*, *p=None*)

Predict treatment effects.

Parameters **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseRRegressor**(*learner=None*, *outcome_learner=None*, *effect_learner=None*, *propensity_learner=LogisticRegressionCV(Cs=array([1.00230524, 2.15608891, 4.63802765, 9.97700064]), cv=StratifiedKFold(n_splits=4, random_state=42, shuffle=True), l1_ratios=array([0.001, 0.33366667, 0.66633333, 0.999]), penalty='elasticnet', random_state=42, solver='saga')*, *ate_alpha=0.05*, *control_name=0*, *n_fold=5*, *random_state=None*)

Bases: *causalml.inference.meta.rlearner.BaseRLearner*

A parent class for R-learner regressor classes.

class causalml.inference.meta.**BaseSClassifier**(*learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.slearner.BaseSLearner*

A parent class for S-learner classifier classes.

predict(*X*, *treatment=None*, *y=None*, *p=None*, *return_components=False*, *verbose=True*)

Predict treatment effects. :param X: a feature matrix :type X: np.matrix or np.array or pd.Dataframe :param treatment: a treatment vector :type treatment: np.array or pd.Series, optional :param y: an outcome vector :type y: np.array or pd.Series, optional :param return_components: whether to return outcome for treatment and control seperately :type return_components: bool, optional :param verbose: whether to output progress logs :type verbose: bool, optional

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseSLearner**(*learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: causalml.inference.meta.base.BaseLearner

A parent class for S-learner classes. An S-learner estimates treatment effects with one machine learning model. Details of S-learner are available at Kunzel et al. (2018) (https://arxiv.org/abs/1706.03461).

estimate_ate(*X*, *treatment*, *y*, *p=None*, *return_ci=False*, *bootstrap_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*)

Estimate the Average Treatment Effect (ATE).

Parameters

- **X** (*np.matrix, np.array, or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **return_ci** (*bool, optional*) – whether to return confidence intervals
- **bootstrap_ci** (*bool*) – whether to return confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap

Returns The mean and confidence interval (LB, UB) of the ATE estimate.

fit(*X*, *treatment*, *y*, *p=None*)

Fit the inference model :param X: a feature matrix :type X: np.matrix, np.array, or pd.Dataframe :param treatment: a treatment vector :type treatment: np.array or pd.Series :param y: an outcome vector :type y: np.array or pd.Series

fit_predict(*X*, *treatment*, *y*, *p=None*, *return_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*, *return_components=False*, *verbose=True*)

Fit the inference model of the S learner and predict treatment effects. :param X: a feature matrix :type X: np.matrix, np.array, or pd.Dataframe :param treatment: a treatment vector :type treatment: np.array or pd.Series :param y: an outcome vector :type y: np.array or pd.Series :param return_ci: whether to return confidence intervals :type return_ci: bool, optional :param n_bootstraps: number of bootstrap iterations :type n_bootstraps: int, optional :param bootstrap_size: number of samples per bootstrap :type bootstrap_size: int, optional :param return_components: whether to return outcome for treatment and control seperately :type return_components: bool, optional :param verbose: whether to output progress logs :type verbose: bool, optional

Returns

Predictions of treatment effects. Output dim: [n_samples, n_treatment]. If return_ci, returns CATE [n_samples, n_treatment], LB [n_samples, n_treatment], UB [n_samples, n_treatment]

Return type (numpy.ndarray)

predict(*X*, *treatment=None*, *y=None*, *p=None*, *return_components=False*, *verbose=True*)

Predict treatment effects. :param X: a feature matrix :type X: np.matrix or np.array or pd.Dataframe :param treatment: a treatment vector :type treatment: np.array or pd.Series, optional :param y: an outcome vector :type y: np.array or pd.Series, optional :param return_components: whether to return outcome for treatment and control seperately :type return_components: bool, optional :param verbose: whether to output progress logs :type verbose: bool, optional

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseSRegressor**(*learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.slearner.BaseSLearner*

A parent class for S-learner regressor classes.

class causalml.inference.meta.**BaseTClassifier**(*learner=None*, *control_learner=None*, *treatment_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.tlearner.BaseTLearner*

A parent class for T-learner classifier classes.

predict(*X*, *treatment=None*, *y=None*, *p=None*, *return_components=False*, *verbose=True*)

Predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series, optional*) – a treatment vector
- **y** (*np.array or pd.Series, optional*) – an outcome vector
- **verbose** (*bool, optional*) – whether to output progress logs

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseTLearner**(*learner=None*, *control_learner=None*, *treatment_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: causalml.inference.meta.base.BaseLearner

A parent class for T-learner regressor classes.

A T-learner estimates treatment effects with two machine learning models.

Details of T-learner are available at Kunzel et al. (2018) (https://arxiv.org/abs/1706.03461).

estimate_ate(*X*, *treatment*, *y*, *p=None*, *bootstrap_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*)

Estimate the Average Treatment Effect (ATE).

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **bootstrap_ci** (*bool*) – whether to return confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap

Returns The mean and confidence interval (LB, UB) of the ATE estimate.

fit(*X*, *treatment*, *y*, *p=None*)

Fit the inference model

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector

fit_predict(*X*, *treatment*, *y*, *p=None*, *return_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*, *return_components=False*, *verbose=True*)

Fit the inference model of the T learner and predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **return_ci** (*bool*) – whether to return confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap
- **return_components** (*bool, optional*) – whether to return outcome for treatment and control seperately
- **verbose** (*str*) – whether to output progress logs

Returns

Predictions of treatment effects. Output dim: [n_samples, n_treatment]. If return_ci, returns CATE [n_samples, n_treatment], LB [n_samples, n_treatment], UB [n_samples, n_treatment]

Return type (numpy.ndarray)

predict(*X*, *treatment=None*, *y=None*, *p=None*, *return_components=False*, *verbose=True*)
Predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series, optional*) – a treatment vector
- **y** (*np.array or pd.Series, optional*) – an outcome vector
- **return_components** (*bool, optional*) – whether to return outcome for treatment and control seperately
- **verbose** (*bool, optional*) – whether to output progress logs

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseTRegressor**(*learner=None*, *control_learner=None*, *treatment_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.tlearner.BaseTLearner*

A parent class for T-learner regressor classes.

class causalml.inference.meta.**BaseXClassifier**(*outcome_learner=None*, *effect_learner=None*, *control_outcome_learner=None*, *treatment_outcome_learner=None*, *control_effect_learner=None*, *treatment_effect_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.xlearner.BaseXLearner*

A parent class for X-learner classifier classes.

fit(*X*, *treatment*, *y*, *p=None*)
Fit the inference model.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.

predict(*X*, *treatment=None*, *y=None*, *p=None*, *return_components=False*, *verbose=True*)
Predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series, optional*) – a treatment vector
- **y** (*np.array or pd.Series, optional*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **return_components** (*bool, optional*) – whether to return outcome for treatment and control seperately
- **return_p_score** (*bool, optional*) – whether to return propensity score
- **verbose** (*bool, optional*) – whether to output progress logs

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseXLearner**(*learner=None*, *control_outcome_learner=None*, *treatment_outcome_learner=None*, *control_effect_learner=None*, *treatment_effect_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: causalml.inference.meta.base.BaseLearner

A parent class for X-learner regressor classes.

An X-learner estimates treatment effects with four machine learning models.

Details of X-learner are available at Kunzel et al. (2018) (https://arxiv.org/abs/1706.03461).

estimate_ate(*X*, *treatment*, *y*, *p=None*, *bootstrap_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*)
Estimate the Average Treatment Effect (ATE).

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to

propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.

- **bootstrap_ci** (*bool*) – whether run bootstrap for confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap

Returns The mean and confidence interval (LB, UB) of the ATE estimate.

fit(*X*, *treatment*, *y*, *p=None*)
Fit the inference model.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.

fit_predict(*X*, *treatment*, *y*, *p=None*, *return_ci=False*, *n_bootstraps=1000*, *bootstrap_size=10000*, *return_components=False*, *verbose=True*)
Fit the treatment effect and outcome models of the R learner and predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **return_ci** (*bool*) – whether to return confidence intervals
- **n_bootstraps** (*int*) – number of bootstrap iterations
- **bootstrap_size** (*int*) – number of samples per bootstrap
- **return_components** (*bool, optional*) – whether to return outcome for treatment and control seperately
- **verbose** (*str*) – whether to output progress logs

Returns

Predictions of treatment effects. Output dim: [n_samples, n_treatment] If return_ci, returns CATE [n_samples, n_treatment], LB [n_samples, n_treatment], UB [n_samples, n_treatment]

Return type (numpy.ndarray)

predict(*X*, *treatment=None*, *y=None*, *p=None*, *return_components=False*, *verbose=True*)
Predict treatment effects.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series, optional*) – a treatment vector
- **y** (*np.array or pd.Series, optional*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **return_components** (*bool, optional*) – whether to return outcome for treatment and control seperately
- **verbose** (*bool, optional*) – whether to output progress logs

Returns Predictions of treatment effects.

Return type (numpy.ndarray)

class causalml.inference.meta.**BaseXRegressor**(*learner=None*, *control_outcome_learner=None*, *treatment_outcome_learner=None*, *control_effect_learner=None*, *treatment_effect_learner=None*, *ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.xlearner.BaseXLearner*

A parent class for X-learner regressor classes.

class causalml.inference.meta.**LRSRegressor**(*ate_alpha=0.05*, *control_name=0*)

Bases: *causalml.inference.meta.slearner.BaseSRegressor*

estimate_ate(*X*, *treatment*, *y*, *p=None*)

Estimate the Average Treatment Effect (ATE). :param X: a feature matrix :type X: np.matrix, np.array, or pd.Dataframe :param treatment: a treatment vector :type treatment: np.array or pd.Series :param y: an outcome vector :type y: np.array or pd.Series

Returns The mean and confidence interval (LB, UB) of the ATE estimate.

class causalml.inference.meta.**MLPTRegressor**(*ate_alpha=0.05*, *control_name=0*, **args*, ***kwargs*)

Bases: *causalml.inference.meta.tlearner.BaseTRegressor*

class causalml.inference.meta.**TMLELearner**(*learner*, *ate_alpha=0.05*, *control_name=0*, *cv=None*, *calibrate_propensity=True*)

Bases: `object`

Targeted maximum likelihood estimation.

Ref: Gruber, S., & Van Der Laan, M. J. (2009). Targeted maximum likelihood estimation: A gentle introduction.

estimate_ate(*X*, *treatment*, *y*, *p*, *segment=None*, *return_ci=False*)

Estimate the Average Treatment Effect (ATE).

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **treatment** (*np.array or pd.Series*) – a treatment vector
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1)

- **segment** (*np.array, optional*) – An optional segment vector of int. If given, the ATE and its CI will be estimated for each segment.
- **return_ci** (*bool, optional*) – Whether to return confidence intervals

Returns The ATE and its confidence interval (LB, UB) for each treatment, t and segment, s

Return type (tuple)

class causalml.inference.meta.**XGBDRRegressor**(*ate_alpha=0.05*, *control_name=0*, **args*, ***kwargs*)

Bases: *causalml.inference.meta.drlearner.BaseDRRegressor*

class causalml.inference.meta.**XGBRRegressor**(*early_stopping=True*, *test_size=0.3*, *early_stopping_rounds=30*, *effect_learner_objective='rank:pairwise'*, *effect_learner_n_estimators=500*, *random_state=42*, **args*, ***kwargs*)

Bases: *causalml.inference.meta.rlearner.BaseRRegressor*

fit(*X*, *treatment*, *y*, *p=None*, *sample_weight=None*, *verbose=True*)

Fit the treatment effect and outcome models of the R learner.

Parameters

- **X** (*np.matrix or np.array or pd.Dataframe*) – a feature matrix
- **y** (*np.array or pd.Series*) – an outcome vector
- **p** (*np.ndarray or pd.Series or dict, optional*) – an array of propensity scores of float (0,1) in the single-treatment case; or, a dictionary of treatment groups that map to propensity vectors of float (0,1); if None will run ElasticNetPropensityModel() to generate the propensity scores.
- **sample_weight** (*np.array or pd.Series, optional*) – an array of sample weights indicating the weight of each observation for *effect_learner*. If None, it assumes equal weight.
- **verbose** (*bool, optional*) – whether to output progress logs

class causalml.inference.meta.**XGBTRegressor**(*ate_alpha=0.05*, *control_name=0*, **args*, ***kwargs*)

Bases: *causalml.inference.meta.tlearner.BaseTRegressor*

7.4 causalml.optimize module

class causalml.optimize.**CounterfactualUnitSelector**(*learner*, *nevertaker_payoff*, *alwaystaker_payoff*, *complier_payoff*, *defier_payoff*, *organic_conversion=None*)

Bases: `object`

A highly experimental implementation of the counterfactual unit selection model proposed by Li and Pearl (2019).

Parameters

- **learner** (*object*) – The base learner used to estimate the segment probabilities.
- **nevertaker_payoff** (*float*) – The payoff from targeting a never-taker
- **alwaystaker_payoff** (*float*) – The payoff from targeting an always-taker
- **complier_payoff** (*float*) – The payoff from targeting a complier

- **defier_payoff** (*float*) – The payoff from targeting a defier
- **organic_conversion** (*float, optional (default=None)*) – The organic conversion rate in the population without an intervention. If None, the organic conversion rate is obtained from tne control group.

  NB: The organic conversion in the control group is not always the same as the organic conversion rate without treatment.
- **data** (*DataFrame*) – A pandas DataFrame containing the features, treatment assignment indicator and the outcome of interest.
- **treatment** (*string*) – A string corresponding to the name of the treatment column. The assumed coding in the column is 1 for treatment and 0 for control.
- **outcome** (*string*) – A string corresponding to the name of the outcome column. The assumed coding in the column is 1 for conversion and 0 for no conversion.

References

Li, Ang, and Judea Pearl. 2019. "Unit Selection Based on Counterfactual Logic." https://ftp.cs.ucla.edu/pub/stat_ser/r488.pdf.

fit(*data*, *treatment*, *outcome*)
Fits the class.

predict(*data*, *treatment*, *outcome*)
Predicts an individual-level payoff. If gain equality is satisfied, uses the exact function; if not, uses the midpoint between bounds.

class causalml.optimize.**CounterfactualValueEstimator**(*treatment*, *control_name*, *treatment_names*, *y_proba*, *cate*, *value*, *conversion_cost*, *impression_cost*, **args*, ***kwargs*)

Bases: object

Parameters

- **treatment** (*array, shape = (num_samples,)*) – An array of treatment group indicator values.
- **control_name** (*string*) – The name of the control condition as a string. Must be contained in the treatment array.
- **treatment_names** (*list, length = cate.shape[1]*) – A list of treatment group names. NB: The order of the items in the list must correspond to the order in which the conditional average treatment effect estimates are in cate_array.
- **y_proba** (*array, shape = (num_samples,)*) – The predicted probability of conversion using the Y ~ X model across the total sample.
- **cate** (*array, shape = (num_samples, len(set(treatment)))*) – Conditional average treatment effect estimations from any model.
- **value** (*array, shape = (num_samples,)*) – Value of converting each unit.
- **conversion_cost** (*shape = (num_samples, len(set(treatment)))*) – The cost of a treatment that is triggered if a unit converts after having been in the treatment, such as a promotion code.
- **impression_cost** (*shape = (num_samples, len(set(treatment)))*) – The cost of a treatment that is the same for each unit whether or not they convert, such as a cost associated with a promotion channel.

Notes

Because we get the conditional average treatment effects from cate-learners relative to the control condition, we subtract the cate for the unit in their actual treatment group from y_proba for that unit, in order to recover the control outcome. We then add the cates to the control outcome to obtain y_proba under each condition. These outcomes are counterfactual because just one of them is actually observed.

predict_best()
Predict the best treatment group based on the highest counterfactual value for a treatment.

predict_counterfactuals()
Predict the counterfactual values for each treatment group.

class causalml.optimize.**PolicyLearner**(*outcome_learner=GradientBoostingRegressor()*, *treatment_learner=GradientBoostingClassifier()*, *policy_learner=DecisionTreeClassifier()*, *clip_bounds=(0.001, 0.999)*, *n_fold=5*, *random_state=None*, *calibration=False*)

Bases: object

A Learner that learns a treatment assignment policy with observational data using doubly robust estimator of causal effect for binary treatment.

Details of the policy learner are available at Athey and Wager (2018) (https://arxiv.org/abs/1702.02896).

fit(*X*, *treatment*, *y*, *p=None*, *dhat=None*)
Fit the treatment assignment policy learner.

Parameters

- **X** (*np.matrix*) – a feature matrix
- **treatment** (*np.array*) – a treatment vector (1 if treated, otherwise 0)
- **y** (*np.array*) – an outcome vector
- **p** (*optional, np.array*) – user provided propensity score vector between 0 and 1
- **dhat** (*optinal, np.array*) – user provided predicted treatment effect vector

Returns returns an instance of self.

Return type self

predict(*X*)
Predict treatment assignment that optimizes the outcome.

Parameters **X** (*np.matrix*) – a feature matrix

Returns predictions of treatment assignment.

Return type (numpy.ndarray)

predict_proba(*X*)
Predict treatment assignment score that optimizes the outcome.

Parameters **X** (*np.matrix*) – a feature matrix

Returns predictions of treatment assignment score.

Return type (numpy.ndarray)

causalml.optimize.**get_actual_value**(*treatment*, *observed_outcome*, *conversion_value*, *conditions*, *conversion_cost*, *impression_cost*)
Set the conversion and impression costs based on a dict of parameters.

Calculate the actual value of targeting a user with the actual treatment group using the above parameters.

treatment [array, shape = (num_samples,)] Treatment array.

observed_outcome [array, shape = (num_samples,)] Observed outcome array, aka y.

conversion_value [array, shape = (num_samples,)] The value of converting a given user.

conditions [list, len = len(set(treatment))] List of treatment conditions.

conversion_cost [array, shape = (num_samples, num_treatment)] Array of conversion costs for each unit in each treatment.

impression_cost [array, shape = (num_samples, num_treatment)] Array of impression costs for each unit in each treatment.

Returns

- **actual_value** (*array, shape = (num_samples,)*) – Array of actual values of havng a user in their actual treatment group.
- **conversion_value** (*array, shape = (num_samples,)*) – Array of payoffs from converting a user.

causalml.optimize.**get_treatment_costs**(*treatment*, *control_name*, *cc_dict*, *ic_dict*)

Set the conversion and impression costs based on a dict of parameters.

Calculate the actual cost of targeting a user with the actual treatment group using the above parameters.

treatment [array, shape = (num_samples,)] Treatment array.

control_name, str Control group name as string.

cc_dict [dict] Dict containing the conversion cost for each treatment.

ic_dict Dict containing the impression cost for each treatment.

Returns

- **conversion_cost** (*ndarray, shape = (num_samples, num_treatments)*) – An array of conversion costs for each treatment.
- **impression_cost** (*ndarray, shape = (num_samples, num_treatments)*) – An array of impression costs for each treatment.
- **conditions** (*list, len = len(set(treatment))*) – A list of experimental conditions.

causalml.optimize.**get_uplift_best**(*cate*, *conditions*)

Takes the CATE prediction from a learner, adds the control outcome array and finds the name of the argmax conditon.

cate [array, shape = (num_samples,)] The conditional average treatment effect prediction.

conditions : list, len = len(set(treatment))

Returns **uplift_recomm_name** – The experimental group recommended by the learner.

Return type array, shape = (num_samples,)

7.5 causalml.dataset module

causalml.dataset.**bar_plot_summary**(*synthetic_summary*, *k*, *drop_learners=[]*, *drop_cols=[]*, *sort_cols=['MSE', 'Abs % Error of ATE']*)

Generates a bar plot comparing learner performance.

Parameters

- **synthetic_summary** (*pd.DataFrame*) – summary generated by get_synthetic_summary()
- **k** (*int*) – number of simulations (used only for plot title text)
- **drop_learners** (*list, optional*) – list of learners (str) to omit when plotting
- **drop_cols** (*list, optional*) – list of metrics (str) to omit when plotting
- **sort_cols** (*list, optional*) – list of metrics (str) to sort on when plotting

causalml.dataset.**bar_plot_summary_holdout**(*train_summary*, *validation_summary*, *k*, *drop_learners=[]*, *drop_cols=[]*)

Generates a bar plot comparing learner performance by training and validation

Parameters

- **train_summary** (*pd.DataFrame*) – summary for training synthetic data generated by get_synthetic_summary_holdout()
- **validation_summary** (*pd.DataFrame*) – summary for validation synthetic data generated by get_synthetic_summary_holdout()
- **k** (*int*) – number of simulations (used only for plot title text)
- **drop_learners** (*list, optional*) – list of learners (str) to omit when plotting
- **drop_cols** (*list, optional*) – list of metrics (str) to omit when plotting

causalml.dataset.**distr_plot_single_sim**(*synthetic_preds*, *kind='kde'*, *drop_learners=[]*, *bins=50*, *histtype='step'*, *alpha=1*, *linewidth=1*, *bw_method=1*)

Plots the distribution of each learner's predictions (for a single simulation). Kernel Density Estimation (kde) and actual histogram plots supported.

Parameters

- **synthetic_preds** (*dict*) – dictionary of predictions generated by get_synthetic_preds()
- **kind** (*str, optional*) – 'kde' or 'hist'
- **drop_learners** (*list, optional*) – list of learners (str) to omit when plotting
- **bins** (*int, optional*) – number of bins to plot if kind set to 'hist'
- **histtype** (*str, optional*) – histogram type if kind set to 'hist'
- **alpha** (*float, optional*) – alpha (transparency) for plotting
- **linewidth** (*int, optional*) – line width for plotting
- **bw_method** (*float, optional*) – parameter for kde

causalml.dataset.**get_synthetic_auuc**(*synthetic_preds*, *drop_learners=[]*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *plot=True*)

Get auuc values for cumulative gains of model estimates in quantiles.

For details, reference get_cumgain() and plot_gain() :param synthetic_preds: dictionary of predictions generated by get_synthetic_preds() :type synthetic_preds: dict :param or get_synthetic_preds_holdout(): :param outcome_col: the column name for the actual outcome :type outcome_col: str, optional :param treatment_col: the

column name for the treatment indicator (0 or 1) :type treatment_col: str, optional :param treatment_effect_col: the column name for the true treatment effect :type treatment_effect_col: str, optional :param plot: plot the cumulative gain chart or not :type plot: boolean,optional

Returns auuc values by learner for cumulative gains of model estimates

Return type (pandas.DataFrame)

causalml.dataset.**get_synthetic_preds**(*synthetic_data_func*, *n=1000*, *estimators={}*)

Generate predictions for synthetic data using specified function (single simulation)

Parameters

- **synthetic_data_func** (*function*) – synthetic data generation function
- **n** (*int, optional*) – number of samples
- **estimators** (*dict of object*) – dict of names and objects of treatment effect estimators

Returns dict of the actual and estimates of treatment effects

Return type (dict)

causalml.dataset.**get_synthetic_preds_holdout**(*synthetic_data_func*, *n=1000*, *valid_size=0.2*, *estimators={}*)

Generate predictions for synthetic data using specified function (single simulation) for train and holdout

Parameters

- **synthetic_data_func** (*function*) – synthetic data generation function
- **n** (*int, optional*) – number of samples
- **valid_size** (*float,optional*) – validaiton/hold out data size
- **estimators** (*dict of object*) – dict of names and objects of treatment effect estimators

Returns

synthetic training and validation data dictionaries:

- preds_dict_train (dict): synthetic training data dictionary
- preds_dict_valid (dict): synthetic validation data dictionary

Return type (tuple)

causalml.dataset.**get_synthetic_summary**(*synthetic_data_func*, *n=1000*, *k=1*, *estimators={}*)

Generate a summary for predictions on synthetic data using specified function

Parameters

- **synthetic_data_func** (*function*) – synthetic data generation function
- **n** (*int, optional*) – number of samples per simulation
- **k** (*int, optional*) – number of simulations

causalml.dataset.**get_synthetic_summary_holdout**(*synthetic_data_func*, *n=1000*, *valid_size=0.2*, *k=1*)

Generate a summary for predictions on synthetic data for train and holdout using specified function

Parameters

- **synthetic_data_func** (*function*) – synthetic data generation function
- **n** (*int, optional*) – number of samples per simulation
- **valid_size** (*float,optional*) – validation/hold out data size

- **k** (*int, optional*) – number of simulations

Returns

summary evaluation metrics of predictions for train and validation:

- summary_train (pandas.DataFrame): training data evaluation summary
- summary_train (pandas.DataFrame): validation data evaluation summary

Return type (tuple)

causalml.dataset.**make_uplift_classification**(*n_samples=1000*, *treatment_name=['control', 'treatment1', 'treatment2', 'treatment3']*, *y_name='conversion'*, *n_classification_features=10*, *n_classification_informative=5*, *n_classification_redundant=0*, *n_classification_repeated=0*, *n_uplift_increase_dict={'treatment1': 2, 'treatment2': 2, 'treatment3': 2}*, *n_uplift_decrease_dict={'treatment1': 0, 'treatment2': 0, 'treatment3': 0}*, *delta_uplift_increase_dict={'treatment1': 0.02, 'treatment2': 0.05, 'treatment3': 0.1}*, *delta_uplift_decrease_dict={'treatment1': 0.0, 'treatment2': 0.0, 'treatment3': 0.0}*, *n_uplift_increase_mix_informative_dict={'treatment1': 1, 'treatment2': 1, 'treatment3': 1}*, *n_uplift_decrease_mix_informative_dict={'treatment1': 0, 'treatment2': 0, 'treatment3': 0}*, *positive_class_proportion=0.5*, *random_seed=20190101*)

Generate a synthetic dataset for classification uplift modeling problem.

Parameters

- **n_samples** (*int, optional (default=1000)*) – The number of samples to be generated for each treatment group.
- **treatment_name** (*list, optional (default = ['control','treatment1', 'treatment2','treatment3'])*) – The list of treatment names.
- **y_name** (*string, optional (default = 'conversion')*) – The name of the outcome variable to be used as a column in the output dataframe.
- **n_classification_features** (*int, optional (default = 10)*) – Total number of features for base classification
- **n_classification_informative** (*int, optional (default = 5)*) – Total number of informative features for base classification
- **n_classification_redundant** (*int, optional (default = 0)*) – Total number of redundant features for base classification
- **n_classification_repeated** (*int, optional (default = 0)*) – Total number of repeated features for base classification
- **n_uplift_increase_dict** (*dictionary, optional (default: {'treatment1': 2, 'treatment2': 2, 'treatment3': 2})*) – Number of features for generating positive treatment effects for corresponding treatment group. Dictionary of {treatment_key: number_of_features_for_increase_uplift}.
- **n_uplift_decrease_dict** (*dictionary, optional (default: {'treatment1': 0, 'treatment2': 0, 'treatment3': 0})*) – Number of features for generating

negative treatment effects for corresponding treatment group. Dictionary of {treatment_key: number_of_features_for_increase_uplift}.

- **delta_uplift_increase_dict** (*dictionary, optional (default: {'treatment1': .02, 'treatment2': .05, 'treatment3': .1})*) – Positive treatment effect created by the positive uplift features on the base classification label. Dictionary of {treatment_key: increase_delta}.
- **delta_uplift_decrease_dict** (*dictionary, optional (default: {'treatment1': 0., 'treatment2': 0., 'treatment3': 0.})*) – Negative treatment effect created by the negative uplift features on the base classification label. Dictionary of {treatment_key: increase_delta}.
- **n_uplift_increase_mix_informative_dict** (*dictionary, optional (default: {'treatment1': 1, 'treatment2': 1, 'treatment3': 1})*) – Number of positive mix features for each treatment. The positive mix feature is defined as a linear combination of a randomly selected informative classification feature and a randomly selected positive uplift feature. The linear combination is made by two coefficients sampled from a uniform distribution between -1 and 1.
- **n_uplift_decrease_mix_informative_dict** (*dictionary, optional (default: {'treatment1': 0, 'treatment2': 0, 'treatment3': 0})*) – Number of negative mix features for each treatment. The negative mix feature is defined as a linear combination of a randomly selected informative classification feature and a randomly selected negative uplift feature. The linear combination is made by two coefficients sampled from a uniform distribution between -1 and 1.
- **positive_class_proportion** (*float, optional (default = 0.5)*) – The proportion of positive label (1) in the control group.
- **random_seed** (*int, optional (default = 20190101)*) – The random seed to be used in the data generation process.

Returns

- **df_res** (*DataFrame*) – A data frame containing the treatment label, features, and outcome variable.
- **x_name** (*list*) – The list of feature names generated.

Notes

The algorithm for generating the base classification dataset is adapted from the make_classification method in the sklearn package, that uses the algorithm in Guyon [1] designed to generate the "Madelon" dataset.

References

causalml.dataset.**scatter_plot_single_sim**(*synthetic_preds*)

Creates a grid of scatter plots comparing each learner's predictions with the truth (for a single simulation).

Parameters **synthetic_preds** (*dict*) – dictionary of predictions generated by get_synthetic_preds() or get_synthetic_preds_holdout()

causalml.dataset.**scatter_plot_summary**(*synthetic_summary*, *k*, *drop_learners=[]*, *drop_cols=[]*)

Generates a scatter plot comparing learner performance. Each learner's performance is plotted as a point in the (Abs % Error of ATE, MSE) space.

Parameters

- **synthetic_summary** (*pd.DataFrame*) – summary generated by get_synthetic_summary()
- **k** (*int*) – number of simulations (used only for plot title text)
- **drop_learners** (*list, optional*) – list of learners (str) to omit when plotting
- **drop_cols** (*list, optional*) – list of metrics (str) to omit when plotting

causalml.dataset.**scatter_plot_summary_holdout**(*train_summary*, *validation_summary*, *k*, *label=['Train', 'Validation']*, *drop_learners=[]*, *drop_cols=[]*)

Generates a scatter plot comparing learner performance by training and validation.

Parameters

- **train_summary** (*pd.DataFrame*) – summary for training synthetic data generated by get_synthetic_summary_holdout()
- **validation_summary** (*pd.DataFrame*) – summary for validation synthetic data generated by get_synthetic_summary_holdout()
- **label** (*string, optional*) – legend label for plot
- **k** (*int*) – number of simulations (used only for plot title text)
- **drop_learners** (*list, optional*) – list of learners (str) to omit when plotting
- **drop_cols** (*list, optional*) – list of metrics (str) to omit when plotting

causalml.dataset.**simulate_easy_propensity_difficult_baseline**(*n=1000*, *p=5*, *sigma=1.0*, *adj=0.0*)

Synthetic data with easy propensity and a difficult baseline From Setup C in Nie X. and Wager S. (2018) 'Quasi-Oracle Estimation of Heterogeneous Treatment Effects'

Parameters

- **n** (*int, optional*) – number of observations
- **p** (*int optional*) – number of covariates (>=3)
- **sigma** (*float*) – standard deviation of the error term
- **adj** (*float*) – no effect. added for consistency

Returns

Synthetically generated samples with the following outputs:

- y ((n,)-array): outcome variable.
- X ((n,p)-ndarray): independent variables.
- w ((n,)-array): treatment flag with value 0 or 1.
- tau ((n,)-array): individual treatment effect.
- b ((n,)-array): expected outcome.
- e ((n,)-array): propensity of receiving treatment.

Return type (tuple)

causalml.dataset.**simulate_hidden_confounder**(*n=10000*, *p=5*, *sigma=1.0*, *adj=0.0*)

Synthetic dataset with a hidden confounder biasing treatment. From Louizos et al. (2018) "Causal Effect Inference with Deep Latent-Variable Models"

Parameters

- **n** (*int, optional*) – number of observations
- **p** (*int optional*) – number of covariates (>=3)
- **sigma** (*float*) – standard deviation of the error term
- **adj** (*float*) – no effect. added for consistency

Returns

Synthetically generated samples with the following outputs:

- y ((n,)-array): outcome variable.
- X ((n,p)-ndarray): independent variables.
- w ((n,)-array): treatment flag with value 0 or 1.
- tau ((n,)-array): individual treatment effect.
- b ((n,)-array): expected outcome.
- e ((n,)-array): propensity of receiving treatment.

Return type (tuple)

causalml.dataset.**simulate_nuisance_and_easy_treatment**(*n=1000*, *p=5*, *sigma=1.0*, *adj=0.0*)

Synthetic data with a difficult nuisance components and an easy treatment effect From Setup A in Nie X. and Wager S. (2018) 'Quasi-Oracle Estimation of Heterogeneous Treatment Effects'

Parameters

- **n** (*int, optional*) – number of observations
- **p** (*int optional*) – number of covariates (>=5)
- **sigma** (*float*) – standard deviation of the error term
- **adj** (*float*) – adjustment term for the distribution of propensity, e. Higher values shift the distribution to 0.

Returns

Synthetically generated samples with the following outputs:

- y ((n,)-array): outcome variable.
- X ((n,p)-ndarray): independent variables.
- w ((n,)-array): treatment flag with value 0 or 1.
- tau ((n,)-array): individual treatment effect.
- b ((n,)-array): expected outcome.
- e ((n,)-array): propensity of receiving treatment.

Return type (tuple)

causalml.dataset.**simulate_randomized_trial**(*n=1000*, *p=5*, *sigma=1.0*, *adj=0.0*)

Synthetic data of a randomized trial From Setup B in Nie X. and Wager S. (2018) 'Quasi-Oracle Estimation of Heterogeneous Treatment Effects'

Parameters

- **n** (*int, optional*) – number of observations
- **p** (*int optional*) – number of covariates (>=5)
- **sigma** (*float*) – standard deviation of the error term
- **adj** (*float*) – no effect. added for consistency

Returns

Synthetically generated samples with the following outputs:

- y ((n,)-array): outcome variable.
- X ((n,p)-ndarray): independent variables.
- w ((n,)-array): treatment flag with value 0 or 1.
- tau ((n,)-array): individual treatment effect.
- b ((n,)-array): expected outcome.
- e ((n,)-array): propensity of receiving treatment.

Return type (tuple)

causalml.dataset.**simulate_unrelated_treatment_control**(*n=1000*, *p=5*, *sigma=1.0*, *adj=0.0*)

Synthetic data with unrelated treatment and control groups. From Setup D in Nie X. and Wager S. (2018) 'Quasi-Oracle Estimation of Heterogeneous Treatment Effects'

Parameters

- **n** (*int, optional*) – number of observations
- **p** (*int optional*) – number of covariates (>=3)
- **sigma** (*float*) – standard deviation of the error term
- **adj** (*float*) – adjustment term for the distribution of propensity, e. Higher values shift the distribution to 0.

Returns

Synthetically generated samples with the following outputs:

- y ((n,)-array): outcome variable.
- X ((n,p)-ndarray): independent variables.
- w ((n,)-array): treatment flag with value 0 or 1.
- tau ((n,)-array): individual treatment effect.
- b ((n,)-array): expected outcome.
- e ((n,)-array): propensity of receiving treatment.

Return type (tuple)

causalml.dataset.**synthetic_data**(*mode=1*, *n=1000*, *p=5*, *sigma=1.0*, *adj=0.0*)

Synthetic data in Nie X. and Wager S. (2018) 'Quasi-Oracle Estimation of Heterogeneous Treatment Effects'

Parameters

- **mode** (*int, optional*) – mode of the simulation: 1 for difficult nuisance components and an easy treatment effect. 2 for a randomized trial. 3 for an easy propensity and a difficult baseline. 4 for unrelated treatment and control groups. 5 for a hidden confounder biasing treatment.
- **n** (*int, optional*) – number of observations
- **p** (*int optional*) – number of covariates (>=5)
- **sigma** (*float*) – standard deviation of the error term
- **adj** (*float*) – adjustment term for the distribution of propensity, e. Higher values shift the distribution to 0. It does not apply to mode == 2 or 3.

Returns

Synthetically generated samples with the following outputs:

- y ((n,)-array): outcome variable.
- X ((n,p)-ndarray): independent variables.
- w ((n,)-array): treatment flag with value 0 or 1.
- tau ((n,)-array): individual treatment effect.
- b ((n,)-array): expected outcome.
- e ((n,)-array): propensity of receiving treatment.

Return type (tuple)

7.6 causalml.match module

class causalml.match.**MatchOptimizer**(*treatment_col='is_treatment', ps_col='pihat', user_col=None, matching_covariates=['pihat'], max_smd=0.1, max_deviation=0.1, caliper_range=(0.01, 0.5), max_pihat_range=(0.95, 0.999), max_iter_per_param=5, min_users_per_group=1000, smd_cols=['pihat'], dev_cols_transformations={'pihat': <function mean>}, dev_factor=1.0, verbose=True*)

Bases: object

check_table_one(*tableone*, *matched*, *score_cols*, *pihat_threshold*, *caliper*)

match_and_check(*score_cols*, *pihat_threshold*, *caliper*)

search_best_match(*df*)

single_match(*score_cols*, *pihat_threshold*, *caliper*)

class causalml.match.**NearestNeighborMatch**(*caliper=0.2*, *replace=False*, *ratio=1*, *shuffle=True*, *random_state=None*, *n_jobs=- 1*)

Bases: object

Propensity score matching based on the nearest neighbor algorithm.

caliper
threshold to be considered as a match.

Type float

replace
whether to match with replacement or not

Type bool

ratio
ratio of control / treatment to be matched. used only if replace=True.

Type int

shuffle
whether to shuffle the treatment group data before matching

Type bool

random_state
RandomState or an int seed

Type numpy.random.RandomState or int

n_jobs
The number of parallel jobs to run for neighbors search. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors

Type int

match(*data*, *treatment_col*, *score_cols*)
Find matches from the control group by matching on specified columns (propensity preferred).

Parameters

- **data** (*pandas.DataFrame*) – total input data
- **treatment_col** (*str*) – the column name for the treatment
- **score_cols** (*list*) – list of column names for matching (propensity column should be included)

Returns

The subset of data consisting of matched treatment and control group data.

Return type (pandas.DataFrame)

match_by_group(*data*, *treatment_col*, *score_cols*, *groupby_col*)
Find matches from the control group stratified by groupby_col, by matching on specified columns (propensity preferred).

Parameters

- **data** (*pandas.DataFrame*) – total sample data
- **treatment_col** (*str*) – the column name for the treatment
- **score_cols** (*list*) – list of column names for matching (propensity column should be included)
- **groupby_col** (*str*) – the column name to be used for stratification

Returns

The subset of data consisting of matched treatment and control group data.

Return type (pandas.DataFrame)

causalml.match.**create_table_one**(*data*, *treatment_col*, *features*)
Report balance in input features between the treatment and control groups.

References

R's tableone at CRAN: https://github.com/kaz-yos/tableone Python's tableone at PyPi: https://github.com/tompollard/tableone

Parameters

- **data** (*pandas.DataFrame*) – total or matched sample data
- **treatment_col** (*str*) – the column name for the treatment
- **features** (*list of str*) – the column names of features

Returns

A table with the means and standard deviations in the treatment and control groups, and the SMD between two groups for the features.

Return type (pandas.DataFrame)

causalml.match.**smd**(*feature*, *treatment*)

Calculate the standard mean difference (SMD) of a feature between the treatment and control groups.

The definition is available at https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3144483/#s11title

Parameters

- **feature** (*pandas.Series*) – a column of a feature to calculate SMD for
- **treatment** (*pandas.Series*) – a column that indicate whether a row is in the treatment group or not

Returns The SMD of the feature

Return type (float)

7.7 causalml.propensity module

class causalml.propensity.**ElasticNetPropensityModel**(*clip_bounds=(0.001, 0.999)*, ***model_kwargs*)

Bases: *causalml.propensity.LogisticRegressionPropensityModel*

class causalml.propensity.**GradientBoostedPropensityModel**(*early_stop=False*, *clip_bounds=(0.001, 0.999)*, ***model_kwargs*)

Bases: *causalml.propensity.PropensityModel*

Gradient boosted propensity score model with optional early stopping.

Notes

Please see the xgboost documentation for more information on gradient boosting tuning parameters: https://xgboost.readthedocs.io/en/latest/python/python_api.html

fit(*X*, *y*, *early_stopping_rounds=10*, *stop_val_size=0.2*)

Fit a propensity model.

Parameters

- **X** (*numpy.ndarray*) – a feature matrix
- **y** (*numpy.ndarray*) – a binary target vector

predict(*X*)
Predict propensity scores.

Parameters **X** (*numpy.ndarray*) – a feature matrix

Returns Propensity scores between 0 and 1.

Return type (numpy.ndarray)

class causalml.propensity.**LogisticRegressionPropensityModel**(*clip_bounds=(0.001, 0.999)*, ***model_kwargs*)
Bases: *causalml.propensity.PropensityModel*

Propensity regression model based on the LogisticRegression algorithm.

class causalml.propensity.**PropensityModel**(*clip_bounds=(0.001, 0.999)*, ***model_kwargs*)
Bases: object

fit(*X*, *y*)
Fit a propensity model.

Parameters

- **X** (*numpy.ndarray*) – a feature matrix
- **y** (*numpy.ndarray*) – a binary target vector

fit_predict(*X*, *y*)
Fit a propensity model and predict propensity scores.

Parameters

- **X** (*numpy.ndarray*) – a feature matrix
- **y** (*numpy.ndarray*) – a binary target vector

Returns Propensity scores between 0 and 1.

Return type (numpy.ndarray)

predict(*X*)
Predict propensity scores.

Parameters **X** (*numpy.ndarray*) – a feature matrix

Returns Propensity scores between 0 and 1.

Return type (numpy.ndarray)

causalml.propensity.**calibrate**(*ps*, *treatment*)
Calibrate propensity scores with logistic GAM.

Ref: https://pygam.readthedocs.io/en/latest/api/logisticgam.html

Parameters

- **ps** (*numpy.array*) – a propensity score vector
- **treatment** (*numpy.array*) – a binary treatment vector (0: control, 1: treated)

Returns a calibrated propensity score vector

Return type (numpy.array)

causalml.propensity.**compute_propensity_score**(*X*, *treatment*, *p_model=None*, *X_pred=None*, *treatment_pred=None*, *calibrate_p=True*)
Generate propensity score if user didn't provide

Parameters

- **X** (*np.matrix*) – features for training
- **treatment** (*np.array or pd.Series*) – a treatment vector for training
- **p_model** (*propensity model object, optional*) – ElasticNetPropensityModel (default) / GradientBoostedPropensityModel
- **X_pred** (*np.matrix, optional*) – features for prediction
- **treatment_pred** (*np.array or pd.Series, optional*) – a treatment vector for prediciton
- **calibrate_p** (*bool, optional*) – whether calibrate the propensity score

Returns

(tuple)

- p (numpy.ndarray): propensity score
- p_model (PropensityModel): a trained PropensityModel object

7.8 causalml.metrics module

class causalml.metrics.**Sensitivity**(*df*, *inference_features*, *p_col*, *treatment_col*, *outcome_col*, *learner*, **args*, ***kwargs*)

Bases: object

A Sensitivity Check class to support Placebo Treatment, Irrelevant Additional Confounder and Subset validation refutation methods to verify causal inference.

Reference: https://github.com/microsoft/dowhy/blob/master/dowhy/causal_refuters/

get_ate_ci(*X*, *p*, *treatment*, *y*)

Return the confidence intervals for treatment effects prediction.

Parameters

- **X** (*np.matrix*) – a feature matrix
- **p** (*np.array*) – a propensity score vector between 0 and 1
- **treatment** (*np.array*) – a treatment vector (1 if treated, otherwise 0)
- **y** (*np.array*) – an outcome vector

Returns Mean and confidence interval (LB, UB) of the ATE estimate.

Return type (numpy.ndarray)

static get_class_object(*method_name*, **args*, ***kwargs*)

Return class object based on input method :param method_name: a list of sensitivity analysis method :type method_name: list of str

Returns Sensitivy Class

Return type (class)

get_prediction(*X*, *p*, *treatment*, *y*)

Return the treatment effects prediction.

Parameters

- **X** (*np.matrix*) – a feature matrix
- **p** (*np.array*) – a propensity score vector between 0 and 1
- **treatment** (*np.array*) – a treatment vector (1 if treated, otherwise 0)
- **y** (*np.array*) – an outcome vector

Returns Predictions of treatment effects

Return type (numpy.ndarray)

sensitivity_analysis(*methods*, *sample_size=None*, *confound='one_sided'*, *alpha_range=None*)
Return the sensitivity data by different method

Parameters

- **method** (*list of str*) – a list of sensitivity analysis method
- **sample_size** (*float, optional*) – ratio for subset the original data
- **confound** (*string, optional*) – the name of confouding function
- **alpha_range** (*np.array, optional*) – a parameter to pass the confounding function

Returns a feature matrix p (np.array): a propensity score vector between 0 and 1 treatment (np.array): a treatment vector (1 if treated, otherwise 0) y (np.array): an outcome vector

Return type X (np.matrix)

sensitivity_estimate()

summary(*method*)
Summary report :param method_name: sensitivity analysis method :type method_name: str

Returns a summary dataframe

Return type (pd.DataFrame)

class causalml.metrics.**SensitivityPlaceboTreatment**(**args*, ***kwargs*)
Bases: *causalml.metrics.sensitivity.Sensitivity*

Replaces the treatment variable with a new variable randomly generated.

sensitivity_estimate()
Summary report :param return_ci: sensitivity analysis method :type return_ci: str

Returns a summary dataframe

Return type (pd.DataFrame)

class causalml.metrics.**SensitivityRandomCause**(**args*, ***kwargs*)
Bases: *causalml.metrics.sensitivity.Sensitivity*

Adds an irrelevant random covariate to the dataframe.

sensitivity_estimate()

class causalml.metrics.**SensitivityRandomReplace**(**args*, ***kwargs*)
Bases: *causalml.metrics.sensitivity.Sensitivity*

Replaces a random covariate with an irrelevant variable.

sensitivity_estimate()
Replaces a random covariate with an irrelevant variable.

class causalml.metrics.**SensitivitySelectionBias**(**args*, *confound='one_sided'*, *alpha_range=None*, *sensitivity_features=None*, ***kwargs*)

Bases: *causalml.metrics.sensitivity.Sensitivity*

Reference:

[1] Blackwell, Matthew. "A selection bias approach to sensitivity analysis for causal effects." Political Analysis 22.2 (2014): 169-182. https://www.mattblackwell.org/files/papers/causalsens.pdf

[2] Confouding parameter alpha_range using the same range as in: https://github.com/mattblackwell/causalsens/blob/master/R/causalsens.R

causalsens()

static partial_rsqs_confounding(*sens_df*, *feature_name*, *partial_rsqs_value*, *range=0.01*)

Check partial rsqs values of feature corresponding confounding amonunt of ATE :param sens_df: a data frame output from causalsens :type sens_df: pandas.DataFrame :param feature_name: feature name to check :type feature_name: str :param partial_rsqs_value: partial rsquare value of feature :type partial_rsqs_value: float :param range: range to search from sens_df :type range: float

Return: min and max value of confounding amount

static plot(*sens_df*, *partial_rsqs_df=None*, *type='raw'*, *ci=False*, *partial_rsqs=False*)

Plot the results of a sensitivity analysis against unmeasured :param sens_df: a data frame output from causalsens :type sens_df: pandas.DataFrame :param partial_rsqs_d: a data frame output from causalsens including partial rsqure :type partial_rsqs_d: pandas.DataFrame :param type: the type of plot to draw, 'raw' or 'r.squared' are supported :type type: str, optional :param ci: whether plot confidence intervals :type ci: bool, optional :param partial_rsqs: whether plot partial rsquare results :type partial_rsqs: bool, optional

summary(*method='Selection Bias'*)

Summary report for Selection Bias Method :param method_name: sensitivity analysis method :type method_name: str

Returns a summary dataframe

Return type (pd.DataFrame)

class causalml.metrics.**SensitivitySubsetData**(**args*, ***kwargs*)

Bases: *causalml.metrics.sensitivity.Sensitivity*

Takes a random subset of size sample_size of the data.

sensitivity_estimate()

causalml.metrics.**ape**(*y*, *p*)

Absolute Percentage Error (APE). :param y: target :type y: float :param p: prediction :type p: float

Returns APE

Return type e (float)

causalml.metrics.**auuc_score**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *normalize=True*, *tmle=False*, **args*, ***kwarg*)

Calculate the AUUC (Area Under the Uplift Curve) score.

Args: df (pandas.DataFrame): a data frame with model estimates and actual data as columns outcome_col (str, optional): the column name for the actual outcome treatment_col (str, optional): the column name for the treatment indicator (0 or 1) treatment_effect_col (str, optional): the column name for the true treatment effect normalize (bool, optional): whether to normalize the y-axis to 1 or not

Returns the AUUC score

Return type (float)

causalml.metrics.**classification_metrics**(*y*, *p*, *w=None*, *metrics={'AUC': <function roc_auc_score>, 'Log Loss': <function logloss>}*)

Log metrics for classifiers.

Parameters

- **y** (*numpy.array*) – target
- **p** (*numpy.array*) – prediction
- **w** (*numpy.array, optional*) – a treatment vector (1 or True: treatment, 0 or False: control). If given, log metrics for the treatment and control group separately
- **metrics** (*dict, optional*) – a dictionary of the metric names and functions

causalml.metrics.**get_cumgain**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *normalize=False*, *random_seed=42*)

Get cumulative gains of model estimates in population.

If the true treatment effect is provided (e.g. in synthetic data), it's calculated as the cumulative gain of the true treatment effect in each population. Otherwise, it's calculated as the cumulative difference between the mean outcomes of the treatment and control groups in each population.

For details, see Section 4.1 of Gutierrez and G{ 'e}rardy (2016), *Causal Inference and Uplift Modeling: A review of the literature*.

For the former, *treatment_effect_col* should be provided. For the latter, both *outcome_col* and *treatment_col* should be provided.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **treatment_effect_col** (*str, optional*) – the column name for the true treatment effect
- **normalize** (*bool, optional*) – whether to normalize the y-axis to 1 or not
- **random_seed** (*int, optional*) – random seed for numpy.random.rand()

Returns cumulative gains of model estimates in population

Return type (pandas.DataFrame)

causalml.metrics.**get_cumlift**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *random_seed=42*)

Get average uplifts of model estimates in cumulative population.

If the true treatment effect is provided (e.g. in synthetic data), it's calculated as the mean of the true treatment effect in each of cumulative population. Otherwise, it's calculated as the difference between the mean outcomes of the treatment and control groups in each of cumulative population.

For details, see Section 4.1 of Gutierrez and G{ 'e}rardy (2016), *Causal Inference and Uplift Modeling: A review of the literature*.

For the former, *treatment_effect_col* should be provided. For the latter, both *outcome_col* and *treatment_col* should be provided.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **treatment_effect_col** (*str, optional*) – the column name for the true treatment effect
- **random_seed** (*int, optional*) – random seed for numpy.random.rand()

Returns average uplifts of model estimates in cumulative population

Return type (pandas.DataFrame)

causalml.metrics.**get_qini**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *normalize=False*, *random_seed=42*)

Get Qini of model estimates in population.

If the true treatment effect is provided (e.g. in synthetic data), it's calculated as the cumulative gain of the true treatment effect in each population. Otherwise, it's calculated as the cumulative difference between the mean outcomes of the treatment and control groups in each population.

For details, see Radcliffe (2007), *Using Control Group to Target on Predicted Lift: Building and Assessing Uplift Models*

For the former, *treatment_effect_col* should be provided. For the latter, both *outcome_col* and *treatment_col* should be provided.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **treatment_effect_col** (*str, optional*) – the column name for the true treatment effect
- **normalize** (*bool, optional*) – whether to normalize the y-axis to 1 or not
- **random_seed** (*int, optional*) – random seed for numpy.random.rand()

Returns cumulative gains of model estimates in population

Return type (pandas.DataFrame)

causalml.metrics.**get_tmlegain**(*df*, *inference_col*, *learner=LGBMRegressor(learning_rate=0.05, n_estimators=300, num_leaves=64)*, *outcome_col='y'*, *treatment_col='w'*, *p_col='p'*, *n_segment=5*, *cv=None*, *calibrate_propensity=True*, *ci=False*)

Get TMLE based average uplifts of model estimates of segments.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **inferenece_col** (*list of str*) – a list of columns that used in learner for inference
- **learner** (*optional*) – a model used by TMLE to estimate the outcome
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **p_col** (*str, optional*) – the column name for propensity score
- **n_segment** (*int, optional*) – number of segment that TMLE will estimated for each

- **cv** (*sklearn.model_selection._BaseKFold, optional*) – sklearn CV object
- **calibrate_propensity** (*bool, optional*) – whether calibrate propensity score or not
- **ci** (*bool, optional*) – whether return confidence intervals for ATE or not

Returns cumulative gains of model estimates based of TMLE

Return type (pandas.DataFrame)

causalml.metrics.**get_tmleqini**(*df*, *inference_col*, *learner=LGBMRegressor(learning_rate=0.05, n_estimators=300, num_leaves=64)*, *outcome_col='y'*, *treatment_col='w'*, *p_col='p'*, *n_segment=5*, *cv=None*, *calibrate_propensity=True*, *ci=False*, *normalize=False*)

Get TMLE based Qini of model estimates by segments.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **inferenece_col** (*list of str*) – a list of columns that used in learner for inference
- **learner** (*optional*) – a model used by TMLE to estimate the outcome
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **p_col** (*str, optional*) – the column name for propensity score
- **n_segment** (*int, optional*) – number of segment that TMLE will estimated for each
- **cv** (*sklearn.model_selection._BaseKFold, optional*) – sklearn CV object
- **calibrate_propensity** (*bool, optional*) – whether calibrate propensity score or not
- **ci** (*bool, optional*) – whether return confidence intervals for ATE or not

Returns cumulative gains of model estimates based of TMLE

Return type (pandas.DataFrame)

causalml.metrics.**gini**(*y*, *p*)

Normalized Gini Coefficient.

Parameters

- **y** (*numpy.array*) – target
- **p** (*numpy.array*) – prediction

Returns normalized Gini coefficient

Return type e (numpy.float64)

causalml.metrics.**logloss**(*y*, *p*)

Bounded log loss error. :param y: target :type y: numpy.array :param p: prediction :type p: numpy.array

Returns bounded log loss error

causalml.metrics.**mae**(*y_true*, *y_pred*, *, *sample_weight=None*, *multioutput='uniform_average'*)

Mean absolute error regression loss.

Read more in the User Guide.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Ground truth (correct) target values.

- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Estimated target values.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **multioutput** (*{'raw_values', 'uniform_average'} or array-like of shape (n_outputs,), default='uniform_average'*) – Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

  'raw_values' : Returns a full set of errors in case of multioutput input.

  'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss – If multioutput is 'raw_values', then mean absolute error is returned for each output separately. If multioutput is 'uniform_average' or an ndarray of weights, then the weighted average of all output errors is returned.

MAE output is non-negative floating point. The best value is 0.0.

Return type float or ndarray of floats

Examples

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.85...
```

causalml.metrics.**mape**(*y*, *p*)

Mean Absolute Percentage Error (MAPE). :param y: target :type y: numpy.array :param p: prediction :type p: numpy.array

Returns MAPE

Return type e (numpy.float64)

causalml.metrics.**plot**(*df*, *kind='gain'*, *tmle=False*, *n=100*, *figsize=(8, 8)*, **args*, ***kwarg*)

Plot one of the lift/gain/Qini charts of model estimates.

A factory method for *plot_lift()*, *plot_gain()*, *plot_qini()*, *plot_tmlegain()* and *plot_tmleqini()*. For details, pleas see docstrings of each function.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns.
- **kind** (*str, optional*) – the kind of plot to draw. 'lift', 'gain', and 'qini' are supported.
- **n** (*int, optional*) – the number of samples to be used for plotting.

causalml.metrics.**plot_gain**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *normalize=False*, *random_seed=42*, *n=100*, *figsize=(8, 8)*)

Plot the cumulative gain chart (or uplift curve) of model estimates.

If the true treatment effect is provided (e.g. in synthetic data), it's calculated as the cumulative gain of the true treatment effect in each population. Otherwise, it's calculated as the cumulative difference between the mean outcomes of the treatment and control groups in each population.

For details, see Section 4.1 of Gutierrez and G{'e}rardy (2016), *Causal Inference and Uplift Modeling: A review of the literature*.

For the former, *treatment_effect_col* should be provided. For the latter, both *outcome_col* and *treatment_col* should be provided.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **treatment_effect_col** (*str, optional*) – the column name for the true treatment effect
- **normalize** (*bool, optional*) – whether to normalize the y-axis to 1 or not
- **random_seed** (*int, optional*) – random seed for numpy.random.rand()
- **n** (*int, optional*) – the number of samples to be used for plotting

causalml.metrics.**plot_lift**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *random_seed=42*, *n=100*, *figsize=(8, 8)*)

Plot the lift chart of model estimates in cumulative population.

If the true treatment effect is provided (e.g. in synthetic data), it's calculated as the mean of the true treatment effect in each of cumulative population. Otherwise, it's calculated as the difference between the mean outcomes of the treatment and control groups in each of cumulative population.

For details, see Section 4.1 of Gutierrez and G{'e}rardy (2016), *Causal Inference and Uplift Modeling: A review of the literature*.

For the former, *treatment_effect_col* should be provided. For the latter, both *outcome_col* and *treatment_col* should be provided.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **treatment_effect_col** (*str, optional*) – the column name for the true treatment effect
- **random_seed** (*int, optional*) – random seed for numpy.random.rand()
- **n** (*int, optional*) – the number of samples to be used for plotting

causalml.metrics.**plot_qini**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *normalize=False*, *random_seed=42*, *n=100*, *figsize=(8, 8)*)

Plot the Qini chart (or uplift curve) of model estimates.

If the true treatment effect is provided (e.g. in synthetic data), it's calculated as the cumulative gain of the true treatment effect in each population. Otherwise, it's calculated as the cumulative difference between the mean outcomes of the treatment and control groups in each population.

For details, see Radcliffe (2007), *Using Control Group to Target on Predicted Lift: Building and Assessing Uplift Models*

For the former, *treatment_effect_col* should be provided. For the latter, both *outcome_col* and *treatment_col* should be provided.

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **treatment_effect_col** (*str, optional*) – the column name for the true treatment effect
- **normalize** (*bool, optional*) – whether to normalize the y-axis to 1 or not
- **random_seed** (*int, optional*) – random seed for numpy.random.rand()
- **n** (*int, optional*) – the number of samples to be used for plotting
- **ci** (*bool, optional*) – whether return confidence intervals for ATE or not

causalml.metrics.**plot_tmlegain**(*df*, *inference_col*, *learner=LGBMRegressor(learning_rate=0.05, n_estimators=300, num_leaves=64)*, *outcome_col='y'*, *treatment_col='w'*, *p_col='tau'*, *n_segment=5*, *cv=None*, *calibrate_propensity=True*, *ci=False*, *figsize=(8, 8)*)

Plot the lift chart based of TMLE estimation

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns
- **inferenece_col** (*list of str*) – a list of columns that used in learner for inference
- **learner** (*optional*) – a model used by TMLE to estimate the outcome
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **p_col** (*str, optional*) – the column name for propensity score
- **n_segment** (*int, optional*) – number of segment that TMLE will estimated for each
- **cv** (*sklearn.model_selection._BaseKFold, optional*) – sklearn CV object
- **calibrate_propensity** (*bool, optional*) – whether calibrate propensity score or not
- **ci** (*bool, optional*) – whether return confidence intervals for ATE or not

causalml.metrics.**plot_tmleqini**(*df*, *inference_col*, *learner=LGBMRegressor(learning_rate=0.05, n_estimators=300, num_leaves=64)*, *outcome_col='y'*, *treatment_col='w'*, *p_col='tau'*, *n_segment=5*, *cv=None*, *calibrate_propensity=True*, *ci=False*, *figsize=(8, 8)*)

Plot the qini chart based of TMLE estimation

Parameters

- **df** (*pandas.DataFrame*) – a data frame with model estimates and actual data as columns

- **inferenece_col** (*list of str*) – a list of columns that used in learner for inference
- **learner** (*optional*) – a model used by TMLE to estimate the outcome
- **outcome_col** (*str, optional*) – the column name for the actual outcome
- **treatment_col** (*str, optional*) – the column name for the treatment indicator (0 or 1)
- **p_col** (*str, optional*) – the column name for propensity score
- **n_segment** (*int, optional*) – number of segment that TMLE will estimated for each
- **cv** (*sklearn.model_selection._BaseKFold, optional*) – sklearn CV object
- **calibrate_propensity** (*bool, optional*) – whether calibrate propensity score or not
- **ci** (*bool, optional*) – whether return confidence intervals for ATE or not

causalml.metrics.**qini_score**(*df*, *outcome_col='y'*, *treatment_col='w'*, *treatment_effect_col='tau'*, *normalize=True*, *tmle=False*, **args*, ***kwarg*)

Calculate the Qini score: the area between the Qini curves of a model and random.

For details, see Radcliffe (2007), *Using Control Group to Target on Predicted Lift: Building and Assessing Uplift Models*

Args: df (pandas.DataFrame): a data frame with model estimates and actual data as columns outcome_col (str, optional): the column name for the actual outcome treatment_col (str, optional): the column name for the treatment indicator (0 or 1) treatment_effect_col (str, optional): the column name for the true treatment effect normalize (bool, optional): whether to normalize the y-axis to 1 or not

Returns the Qini score

Return type (float)

causalml.metrics.**r2_score**(*y_true*, *y_pred*, ***, *sample_weight=None*, *multioutput='uniform_average'*)

R^2 (coefficient of determination) regression score function.

Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Read more in the User Guide.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Ground truth (correct) target values.
- **y_pred** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – Estimated target values.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **multioutput** (*{'raw_values', 'uniform_average', 'variance_weighted'}, array-like of shape (n_outputs,) or None, default='uniform_average'*) – Defines aggregating of multiple output scores. Array-like value defines weights used to average scores. Default is "uniform_average".

  'raw_values' : Returns a full set of scores in case of multioutput input.

  'uniform_average' : Scores of all outputs are averaged with uniform weight.

  'variance_weighted' : Scores of all outputs are averaged, weighted by the variances of each individual output.

Changed in version 0.19: Default value of multioutput is 'uniform_average'.

Returns **z** – The R^2 score or ndarray of scores if 'multioutput' is 'raw_values'.

Return type float or ndarray of floats

Notes

This is not a symmetric function.

Unlike most other scores, R^2 score may be negative (it need not actually be the square of a quantity R).

This metric is not well-defined for single samples and will return a NaN value if n_samples is less than two.

References

Examples

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred,
...          multioutput='variance_weighted')
0.938...
>>> y_true = [1, 2, 3]
>>> y_pred = [1, 2, 3]
>>> r2_score(y_true, y_pred)
1.0
>>> y_true = [1, 2, 3]
>>> y_pred = [2, 2, 2]
>>> r2_score(y_true, y_pred)
0.0
>>> y_true = [1, 2, 3]
>>> y_pred = [3, 2, 1]
>>> r2_score(y_true, y_pred)
-3.0
```

causalml.metrics.**regression_metrics**(*y*, *p*, *w=None*, *metrics={'Gini': <function gini>, 'RMSE': <function rmse>, 'sMAPE': <function smape>}*)

Log metrics for regressors.

Parameters

- **y** (*numpy.array*) – target
- **p** (*numpy.array*) – prediction
- **w** (*numpy.array, optional*) – a treatment vector (1 or True: treatment, 0 or False: control). If given, log metrics for the treatment and control group separately
- **metrics** (*dict, optional*) – a dictionary of the metric names and functions

causalml.metrics.**rmse**(*y*, *p*)

Root Mean Squared Error (RMSE). :param y: target :type y: numpy.array :param p: prediction :type p: numpy.array

Returns RMSE

Return type e (numpy.float64)

causalml.metrics.**roc_auc_score**(*y_true*, *y_score*, ***, *average='macro'*, *sample_weight=None*, *max_fpr=None*, *multi_class='raise'*, *labels=None*)

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

Note: this implementation can be used with binary, multiclass and multilabel classification, but some restrictions apply (see Parameters).

Read more in the User Guide.

Parameters

- **y_true** (*array-like of shape (n_samples,) or (n_samples, n_classes)*) – True labels or binary label indicators. The binary and multiclass cases expect labels with shape (n_samples,) while the multilabel case expects binary label indicators with shape (n_samples, n_classes).
- **y_score** (*array-like of shape (n_samples,) or (n_samples, n_classes)*) – Target scores.
  - In the binary case, it corresponds to an array of shape *(n_samples,)*. Both probability estimates and non-thresholded decision values can be provided. The probability estimates correspond to the **probability of the class with the greater label**, i.e. *estimator.classes_[1]* and thus *estimator.predict_proba(X, y)[:, 1]*. The decision values corresponds to the output of *estimator.decision_function(X, y)*. See more information in the User guide;
  - In the multiclass case, it corresponds to an array of shape *(n_samples, n_classes)* of probability estimates provided by the *predict_proba* method. The probability estimates **must** sum to 1 across the possible classes. In addition, the order of the class scores must correspond to the order of `labels`, if provided, or else to the numerical or lexicographical order of the labels in `y_true`. See more information in the User guide;
  - In the multilabel case, it corresponds to an array of shape *(n_samples, n_classes)*. Probability estimates are provided by the *predict_proba* method and the non-thresholded decision values by the *decision_function* method. The probability estimates correspond to the **probability of the class with the greater label for each output** of the classifier. See more information in the User guide.
- **average** (*{'micro', 'macro', 'samples', 'weighted'} or None, default='macro'*) – If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data: Note: multiclass ROC AUC currently only handles the 'macro' and 'weighted' averages.

  'micro': Calculate metrics globally by considering each element of the label indicator matrix as a label.

  'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

  'samples': Calculate metrics for each instance, and find their average.

  Will be ignored when `y_true` is binary.

- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.
- **max_fpr** (*float > 0 and <= 1, default=None*) – If not None, the standardized partial AUC[2] over the range [0, max_fpr] is returned. For the multiclass case, max_fpr, should be either equal to None or 1.0 as AUC ROC partial computation currently is not supported for multiclass.
- **multi_class** (*{'raise', 'ovr', 'ovo'}, default='raise'*) – Only used for multiclass targets. Determines the type of configuration to use. The default value raises an error, so either 'ovr' or 'ovo' must be passed explicitly.

  'ovr': Stands for One-vs-rest. Computes the AUC of each class against the rest[3][4]. This treats the multiclass case in the same way as the multilabel case. Sensitive to class imbalance even when average == 'macro', because class imbalance affects the composition of each of the 'rest' groupings.

  'ovo': Stands for One-vs-one. Computes the average AUC of all possible pairwise combinations of classes[5]. Insensitive to class imbalance when average == 'macro'.
- **labels** (*array-like of shape (n_classes,), default=None*) – Only used for multiclass targets. List of labels that index the classes in y_score. If None, the numerical or lexicographical order of the labels in y_true is used.

Returns **auc**

Return type float

References

See also:

average_precision_score Area under the precision-recall curve.

roc_curve Compute Receiver operating characteristic (ROC) curve.

RocCurveDisplay.from_estimator Plot Receiver Operating Characteristic (ROC) curve given an estimator and some data.

RocCurveDisplay.from_predictions Plot Receiver Operating Characteristic (ROC) curve given the true and predicted values.

Examples

Binary case:

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.metrics import roc_auc_score
>>> X, y = load_breast_cancer(return_X_y=True)
>>> clf = LogisticRegression(solver="liblinear", random_state=0).fit(X, y)
```

(continues on next page)

[2] Analyzing a portion of the ROC curve. McClish, 1989

[3] Provost, F., Domingos, P. (2000). Well-trained PETs: Improving probability estimation trees (Section 6.2), CeDER Working Paper #IS-00-04, Stern School of Business, New York University.

[4] Fawcett, T. (2006). An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861-874.

[5] Hand, D.J., Till, R.J. (2001). A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. Machine Learning, 45(2), 171-186.

(continued from previous page)

```
>>> roc_auc_score(y, clf.predict_proba(X)[:, 1])
0.99...
>>> roc_auc_score(y, clf.decision_function(X))
0.99...
```

Multiclass case:

```
>>> from sklearn.datasets import load_iris
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(solver="liblinear").fit(X, y)
>>> roc_auc_score(y, clf.predict_proba(X), multi_class='ovr')
0.99...
```

Multilabel case:

```
>>> import numpy as np
>>> from sklearn.datasets import make_multilabel_classification
>>> from sklearn.multioutput import MultiOutputClassifier
>>> X, y = make_multilabel_classification(random_state=0)
>>> clf = MultiOutputClassifier(clf).fit(X, y)
>>> # get a list of n_output containing probability arrays of shape
>>> # (n_samples, n_classes)
>>> y_pred = clf.predict_proba(X)
>>> # extract the positive columns for each output
>>> y_pred = np.transpose([pred[:, 1] for pred in y_pred])
>>> roc_auc_score(y, y_pred, average=None)
array([0.82..., 0.86..., 0.94..., 0.85... , 0.94...])
>>> from sklearn.linear_model import RidgeClassifierCV
>>> clf = RidgeClassifierCV().fit(X, y)
>>> roc_auc_score(y, clf.decision_function(X), average=None)
array([0.81..., 0.84... , 0.93..., 0.87..., 0.94...])
```

causalml.metrics.**smape**(*y*, *p*)

Symmetric Mean Absolute Percentage Error (sMAPE). :param y: target :type y: numpy.array :param p: prediction :type p: numpy.array

Returns sMAPE

Return type e (numpy.float64)

7.9 Module contents

CHAPTER

EIGHT

REFERENCES

8.1 Open Source Software Projects

8.1.1 Python Packages

- DoWhy: a package for causal inference based on causal graphs.
- CausalLift: a package for uplift modeling based on T-learner [15].
- PyLift: a package for uplift modeling based on the transformed outcome method in [4].
- EconML: a package for treatment effect estimation with orthogonal random forest [19], DeepIV [11] and other ML methods.

8.1.2 R Packages

- uplift: a package for treatment effect estimation with ML.
- grf: a package for forest-based honest estimation from [5].

8.2 Papers

CHAPTER

NINE

CHANGELOG

9.1 0.11.0 (2021-07-28)

- CausalML surpassed 2K stars!
- We have 3 new community contributors, Jannik (@jroessler), Mohamed (@ibraaaa), and Leo (@lleiou). Thanks for the contribution!

9.1.1 Major Updates

- Make tensorflow dependency optional and add python 3.9 support by @jeongyoonlee (#343)
- Add delta-delta-p (ddp) tree inference approach by @jroessler (#327)
- Add conda env files for Python 3.6, 3.7, and 3.8 by @jeongyoonlee (#324)

9.1.2 Minor Updates

- Fix inconsistent feature importance calculation in uplift tree by @paullo0106 (#372)
- Fix filter method failure with NaNs in the data issue by @manojbalaji1 (#367)
- Add automatic package publish by @jeongyoonlee (#354)
- Fix typo in unit_selection optimization by @jeongyoonlee (#347)
- Fix docs build failure by @jeongyoonlee (#335)
- Convert pandas inputs to numpy in S/T/R Learners by @jeongyoonlee (#333)
- Require scikit-learn as a dependency of setup.py by @ibraaaa (#325)
- Fix AttributeError when passing in Outcome and Effect learner to R-Learner by @paullo0106 (#320)
- Fix error when there is no positive class for KL Divergence filter by @lleiou (#311)
- Add versions to cython and numpy in setup.py for requirements.txt accordingly by @maccam912 (#306)

9.2 0.10.0 (2021-02-18)

- CausalML surpassed 235,000 downloads!
- We have 5 new community contributors, Suraj (@surajiyer), Harsh (@HarshCasper), Manoj (@manojbalaji1), Matthew (@maccam912) and Václav (@vaclavbelak). Thanks for the contribution!

9.2.1 Major Updates

- Add Policy learner, DR learner, DRIV learner by @huigangchen (#292)
- Add wrapper for CEVAE, a deep latent-variable and variational autoencoder based model by @ppstacy(#276)

9.2.2 Minor Updates

- Add propensity_learner to R-learner by @jeongyoonlee (#297)
- Add BaseLearner class for other meta-learners to inherit from without duplicated code by @jeongyoonlee (#295)
- Fix installation issue for Shap>=0.38.1 by @paullo0106 (#287)
- Fix import error for sklearn>= 0.24 by @jeongyoonlee (#283)
- Fix KeyError issue in Filter method for certain dataset by @surajiyer (#281)
- Fix inconsistent cumlift score calculation of multiple models by @vaclavbelak (#273)
- Fix duplicate values handling in feature selection method by @manojbalaji1 (#271)
- Fix the color spectrum of SHAP summary plot for feature interpretations of meta-learners by @paullo0106 (#269)
- Add IIA and value optimization related documentation by @t-tte (#264)
- Fix StratifiedKFold arguments for propensity score estimation by @paullo0106 (#262)
- Refactor the code with string format argument and is to compare object types, and change methods not using bound instance to static methods by @harshcasper (#256, #260)

9.3 0.9.0 (2020-10-23)

- CausalML won the 1st prize at the poster session in UberML'20
- DoWhy integrated CausalML starting v0.4 (release note)
- CausalML team welcomes new project leadership, Mert Bay
- We have 4 new community contributors, Mario Wijaya (@mwijaya3), Harry Zhao (@deeplaunch), Christophe (@ccrndn) and Georg Walther (@waltherg). Thanks for the contribution!

9.3.1 Major Updates

- Add feature importance and its visualization to UpliftDecisionTrees and UpliftRF by @yungmsh (#220)
- Add feature selection example with Filter methods by @paullo0106 (#223)

9.3.2 Minor Updates

- Implement propensity model abstraction for common interface by @waltherg (#223)
- Fix bug in BaseSClassifier and BaseXClassifier by @yungmsh and @ppstacy (#217), (#218)
- Fix parentNodeSummary for UpliftDecisionTrees by @paullo0106 (#238)
- Add pd.Series for propensity score condition check by @paullo0106 (#242)
- Fix the uplift random forest prediction output by @ppstacy (#236)
- Add functions and methods to init for optimization module by @mwijaya3 (#228)
- Install GitHub Stale App to close inactive issues automatically @jeongyoonlee (#237)
- Update documentation by @deeplaunch, @ccrndn, @ppstacy(#214, #231, #232)

9.4 0.8.0 (2020-07-17)

CausalML surpassed 100,000 downloads! Thanks for the support.

9.4.1 Major Updates

- Add value optimization to *optimize* by @t-tte (#183)
- Add counterfactual unit selection to *optimize* by @t-tte (#184)
- Add sensitivity analysis to *metrics* by @ppstacy (#199, #212)
- Add the *iv* estimator submodule and add 2SLS model to it by @huigangchen (#201)

9.4.2 Minor Updates

- Add *GradientBoostedPropensityModel* by @yungmsh (#193)
- Add covariate balance visualization by @yluogit (#200)
- Fix bug in the X learner propensity model by @ppstacy (#209)
- Update package dependencies by @jeongyoonlee (#195, #197)
- Update documentation by @jeongyoonlee, @ppstacy and @yluogit (#181, #202, #205)

9.5 0.7.1 (2020-05-07)

Special thanks to our new community contributor, Katherine (@khof312)!

9.5.1 Major Updates

- Adjust matching distances by a factor of the number of matching columns in propensity score matching by @yungmsh (#157)
- Add TMLE-based AUUC/Qini/lift calculation and plotting by @ppstacy (#165)

9.5.2 Minor Updates

- Fix typos and update documents by @paullo0106, @khof312, @jeongyoonlee (#150, #151, #155, #163)
- Fix error in *UpliftTreeClassifier.kl_divergence()* for *pk == 1 or 0* by @jeongyoonlee (#169)
- Fix error in *BaseRRegressor.fit()* without propensity score input by @jeongyoonlee (#170)

9.6 0.7.0 (2020-02-28)

Special thanks to our new community contributor, Steve (@steveyang90)!

9.6.1 Major Updates

- Add a new *nn* inference submodule with *DragonNet* implementation by @yungmsh
- Add a new *feature selection* submodule with filter feature selection methods by @zhenyuz0500

9.6.2 Minor Updates

- Make propensity scores optional in all meta-learners by @ppstacy
- Replace *eli5* permutation importance with *sklearn*'s by @yluogit
- Replace *ElasticNetCV* with *LogisticRegressionCV* in *propensity.py* by @yungmsh
- Fix the normalized uplift curve plot with negative ATE by @jeongyoonlee
- Fix the TravisCI FOSSA error for PRs from forked repo by @steveyang90
- Add documentation about tree visualization by @zhenyuz0500

9.7 0.6.0 (2019-12-31)

Special thanks to our new community contributors, Fritz (@fritzo), Peter (@peterfoley) and Tomasz (@TomaszZamacinski)!

- Improve *UpliftTreeClassifier*'s speed by 4 times by @jeongyoonlee
- Fix impurity computation in *CausalTreeRegressor* by @TomaszZamacinski
- Fix XGBoost related warnings by @peterfoley
- Fix typos and improve documentation by @peterfoley and @fritzo

9.8 0.5.0 (2019-11-26)

Special thanks to our new community contributors, Paul (@paullo0106) and Florian (@FlorianWilhelm)!

- Add *TMLELearner*, targeted maximum likelihood estimator to *inference.meta* by @huigangchen
- Add an option to DGPs for regression to simulate imbalanced propensity distribution by @huigangchen
- Fix incorrect edge connections, and add more information in the uplift tree plot by @paullo0106
- Fix an installation error related to *Cython* and *numpy* by @FlorianWilhelm
- Drop Python 2 support from *setup.py* by @jeongyoonlee
- Update *causaltree.pyx* Cython code to be compatible with *scikit-learn>=0.21.0* by @jeongyoonlee

9.9 0.4.0 (2019-10-21)

- Add *uplift_tree_plot()* to *inference.tree* to visualize *UpliftTreeClassifier* by @zhenyuz0500
- Add the *Explainer* class to *inference.meta* to provide feature importances using *SHAP* and *eli5*'s *PermutationImportance* by @yungmsh
- Add bootstrap confidence intervals for the average treatment effect estimates of meta learners by @ppstacy

9.10 0.3.0 (2019-09-17)

- Extend meta-learners to support classification by @t-tte
- Extend meta-learners to support multiple treatments by @yungmsh
- Fix a bug in uplift curves and add Qini curves/scores to *metrics* by @jeongyoonlee
- Add *inference.meta.XGBRRegressor* with early stopping and ranking optimization by @yluogit

9.11 0.2.0 (2019-08-12)

- Add *optimize.PolicyLearner* based on Athey and Wager 2017 [6]
- Add the *CausalTreeRegressor* estimator based on Athey and Imbens 2016 [4] (experimental)
- Add missing imports in *features.py* to enable label encoding with grouping of rare values in *LabelEncoder()*
- Fix a bug that caused the mismatch between training and prediction features in *inference.meta.tlearner.predict()*

9.12 0.1.0 (unreleased)

- Initial release with the Uplift Random Forest, and S/T/X/R-learners.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Ahmed Alaa and Mihaela Schaar. Limits of estimating heterogeneous treatment effects: guidelines for practical algorithm design. In *International Conference on Machine Learning*, 129–138. 2018.

[2] Joshua D Angrist and Jörn-Steffen Pischke. *Mostly harmless econometrics: An empiricist's companion*. Princeton university press, 2008.

[3] Joshua D. Angrist and Alan B. Krueger. Instrumental variables and the search for identification: from supply and demand to natural experiments. *Journal of Economic Perspectives*, 15(4):69–85, December 2001. URL: https://www.aeaweb.org/articles?id=10.1257/jep.15.4.69, doi:10.1257/jep.15.4.69.

[4] Susan Athey and Guido Imbens. Recursive partitioning for heterogeneous causal effects. *Proceedings of the National Academy of Sciences*, 113(27):7353–7360, 2016.

[5] Susan Athey, Julie Tibshirani, Stefan Wager, and others. Generalized random forests. *The Annals of Statistics*, 47(2):1148–1178, 2019.

[6] Susan Athey and Stefan Wager. Efficient policy learning. *arXiv preprint arXiv:1702.02896*, 2017.

[7] Peter C. Austin and Elizabeth A. Stuart. Moving towards best practice when using inverse probability of treatment weighting (iptw) using the propensity score to estimate causal treatment effects in observational studies. *Statistics in Medicine*, 34(28):3661–3679, 2015. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.6607, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sim.6607, doi:https://doi.org/10.1002/sim.6607.

[8] Hansotia Behram and Rukstales Brad. Incremental value modeling. *Journal of Interactive Marketing*, 16:35–46, 2002.

[9] Victor Chernozhukov, Denis Chetverikov, Mert Demirer, Esther Duflo, Christian Hansen, Whitney Newey, and James Robins. Double/debiased machine learning for treatment and structural parameters. *The Econometrics Journal*, 21(1):C1–C68, 01 2018. URL: https://doi.org/10.1111/ectj.12097, arXiv:https://academic.oup.com/ectj/article-pdf/21/1/C1/27684918/ectj00c1.pdf, doi:10.1111/ectj.12097.

[10] Pierre Gutierrez and Jean-Yves Gerardy. Causal inference and uplift modeling a review of the literature. *JMLR: Workshop and Conference Proceedings 67*, 2016.

[11] Jason Hartford, Greg Lewis, Kevin Leyton-Brown, and Matt Taddy. Deep iv: a flexible approach for counterfactual prediction. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1414–1423. JMLR. org, 2017.

[12] Keisuke Hirano, Guido W. Imbens, and Geert Ridder. Efficient estimation of average treatment effects using the estimated propensity score. *Econometrica*, 71(4):1161–1189, 2003. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-0262.00442, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/1468-0262.00442, doi:https://doi.org/10.1111/1468-0262.00442.

[13] Guido W Imbens and Jeffrey M Wooldridge. Recent developments in the econometrics of program evaluation. *Journal of economic literature*, 47(1):5–86, 2009.

[14] Edward H. Kennedy. Optimal doubly robust estimation of heterogeneous causal effects. 2020. arXiv:2004.14497.

[15] Sören R Künzel, Jasjeet S Sekhon, Peter J Bickel, and Bin Yu. Metalearners for estimating heterogeneous treatment effects using machine learning. *Proceedings of the National Academy of Sciences*, 116(10):4156–4165, 2019.

[16] Mark Laan and Sherri Rose. *Targeted Learning: Causal Inference for Observational and Experimental Data*. Springer-Verlag New York, 01 2011. ISBN 978-1-4419-9781-4. doi:10.1007/978-1-4419-9782-1.

[17] Ang Li and Judea Pearl. Unit selection based on counterfactual logic. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 1793–1799. International Joint Conferences on Artificial Intelligence Organization, 7 2019. URL: https://doi.org/10.24963/ijcai.2019/248, doi:10.24963/ijcai.2019/248.

[18] Xinkun Nie and Stefan Wager. Quasi-oracle estimation of heterogeneous treatment effects. *arXiv preprint arXiv:1712.04912*, 2017.

[19] Miruna Oprescu, Vasilis Syrgkanis, and Zhiwei Steven Wu. Orthogonal random forest for heterogeneous treatment effect estimation. *CoRR*, 2018. URL: http://arxiv.org/abs/1806.03467, arXiv:1806.03467.

[20] Judea Pearl. *Causality*. Cambridge university press, 2009.

[21] Piotr Rzepakowski and Szymon Jaroszewicz. Decision trees for uplift modeling with single and multiple treatments. *Knowl. Inf. Syst.*, 32(2):303–327, August 2012.

[22] Elizabeth A Stuart. Matching methods for causal inference: a review and a look forward. *Statistical science: a review journal of the Institute of Mathematical Statistics*, 25(1):1, 2010.

[23] Yan Zhao, Xiao Fang, and David Simchi-Levi. Uplift modeling with multiple treatments and general response types. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, 588–596. SIAM, 2017.

[24] Zhenyu Zhao and Totte Harinen. Uplift modeling for multiple treatments with cost optimization. In *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 422–431. IEEE, 2019.

[25] P. Richard Hahn, Jared S. Murray, and Carlos Carvalho. Bayesian regression tree models for causal inference: regularization, confounding, and heterogeneous effects. *arXiv e-prints*, pages arXiv:1706.09523, Jun 2017. arXiv:1706.09523.

PYTHON MODULE INDEX

C

INDEX

F

G

S

T

U

X