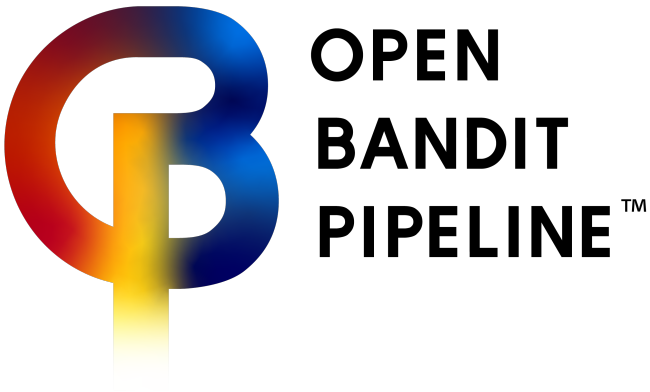
obp Documentation

Release latest

usaito

Dec 11, 2020

INTRODUCTION:


OPEN
BANDIT
PIPELINE™

CHAPTER

ONE

OVERVIEW

Open Bandit Pipeline (OBP) is an open source python library for bandit algorithms and off-policy evaluation (OPE). The toolkit comes with the *Open Bandit Dataset* , a large-scale logged bandit feedback data collected on a fashion e-commerce platform, ZOZOTOWN. The purpose of the open data and library is to enable easy, realistic, and reproducible evaluation of bandit algorithms and OPE. OBP has a series of implementations of dataset preprocessing, bandit policy interfaces, and a variety of OPE estimators.

Our open data and pipeline facilitate evaluation and comparison related to the following research topics.

- **Bandit Algorithms**: Our data include the probabilities of each action being selected by behavior policies (the true propensity scores).

Therefore, it enables the evaluation of new online bandit algorithms, including contextual and combinatorial algorithms, in a large real-world setting.

- **Off-Policy Evaluation**: We present implementations of behavior policies used when collecting datasets as a part of our pipeline.

Our open data also contains logged bandit feedback data generated by multiple behavior policies. Therefore, it enables the evaluation of off-policy evaluation with ground-truths for the performances of evaluation policies.

This website contains pages with example analyses to help demonstrate the usage of this library. Additionally, it presents examples of evaluating counterfactual bandit algorithms and OPE itself. The reference page contains the full reference documentation for the current functions of this toolkit.

CHAPTER

TWO

ALGORITHMS AND OPE ESTIMATORS SUPPORTED

2.1 Bandit Algorithms

- Online
  - Context-free
    - Random
    - Epsilon Greedy
    - Bernoulli Thompson Sampling
  - Contextual (Linear)
    - Linear Epsilon Greedy
    - Linear Thompson Sampling [10]
    - Linear Upper Confidence Bound [11]
  - Contextual (Logistic)
    - Logistic Epsilon Greedy
    - Logistic Thompson Sampling [12]
    - Logistic Upper Confidence Bound [13]
- Offline (Off-Policy Learning) [4]
  - Inverse Probability Weighting

2.2 OPE Estimators

- Replay Method (RM) [14]
- Direct Method (DM) [15]
- Inverse Probability Weighting (IPW) [2] [3]
- Self-Normalized Inverse Probability Weighting (SNIPW) [16]
- Doubly Robust (DR) [4]
- Switch Estimators [8]
- Doubly Robust with Optimistic Shrinkage (DRos) [9]
- More Robust Doubly Robust (MRDR) [1]

- Double Machine Learning (DML) [17]

CHAPTER

THREE

CITATION

If you use our dataset and pipeline in your work, please cite our paper below.

@article{saito2020large, title={Large-scale Open Dataset, Pipeline, and Benchmark for Bandit Algorithms}, author={Saito, Yuta, Shunsuke Aihara, Megumi Matsutani, Yusuke Narita}, journal={arXiv preprint arXiv:2008.07146}, year={2020}

}

CHAPTER

FOUR

GOOGLE GROUP

If you are interested in the Open Bandit Project, we can follow the updates at its google group: https://groups.google.com/g/open-bandit-project

CHAPTER

FIVE

CONTACT

For any question about the paper, data, and pipeline, feel free to contact: saito@hanjuku-kaso.com

CHAPTER

SIX

TABLE OF CONTENTS

6.1 About

Motivated by the paucity of real-world data and implementation enabling the evaluation and comparison of OPE, we release the following open-source dataset and pipeline software for research uses.

6.1.1 Open Bandit Dataset (OBD)

Open Bandit Dataset is a public real-world logged bandit feedback data. The dataset is provided by ZOZO, Inc., the largest Japanese fashion e-commerce company with over 5 billion USD market capitalization (as of May 2020). The company uses multi-armed bandit algorithms to recommend fashion items to users in a large-scale fashion e-commerce platform called ZOZOTOWN. The following figure presents examples of displayed fashion items as actions.

We collected the data in a 7-days experiment in late November 2019 on three campaigns, corresponding to "all", "men's", and "women's" items, respectively. Each campaign randomly uses either the Random policy or the Bernoulli Thompson Sampling (Bernoulli TS) policy for each user impression. Note that we pre-trained Bernoulli TS for over a month before the data collection process and the policy well converges to a fixed one. Thus, we suppose our data is generated by a fixed policy and apply the standard OPE formulation that assumes static behavior and evaluation

policies. These policies select three of the possible fashion items to each user. Let $\mathcal{I} := \{0, \ldots, n\}$ be a set of $n+1$ items and $\mathcal{K} := \{0, \ldots, k\}$ be a set of $k+1$ positions. The above figure shows that $k+1 = 3$ for our data. We assume that the reward (click indicator) depends only on the item and its position, which is a general assumption on the click generative model in the web industry:cite:*Li2018*. Under the assumption, the action space is simply the product of the item set and the position set, i.e., $= \mathcal{I} \times \mathcal{K}$. Then, we can apply the standard OPE setup and estimators to our setting. We describe some statistics of the dataset in the following.

Table 1: Statistics of the Open Bandit Dataset

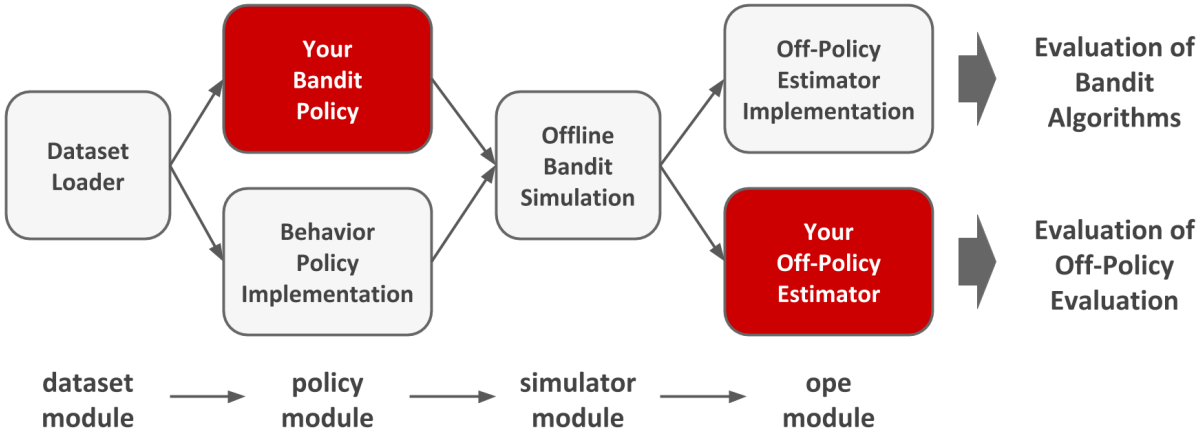
Campaigns	Behavior Policies	#Data	#Items	Average Age	CTR (V^{π}) ±95% CI	Relative-CTR
ALL	RANDOM	1,374,327	80	37.93	0.35% ±0.010	1.00
	BERNOULLI TS	12,168,084			0.50% ±0.004	1.43
MEN'S	RANDOM	452,949	34	37.68	0.51% ±0.021	1.48
	BERNOULLI TS	4,077,727			0.67% ±0.008	1.94
WOMEN'S	RANDOM	864,585	46	37.99	0.48% ±0.014	1.39
	BERNOULLI TS	7,765,497			0.64% ±0.056	1.84

Notes: Bernoulli TS stands for Bernoulli Thompson Sampling. **#Data** is the total number of recommended items observed during the 7-day experiment. **#Items** is the total number of items having a positive probability of being recommended by each behavior policy. **Average Age** is the average age of users in each campaign. **CTR** is the percentage of a click being observed in log data, and this is the ground-truth performance of behavior policies in each campaign. 95% confidence interval (CI) of CTR is calculated based on a normal approximation of Bernoulli sampling. **Relative-CTR** is CTR relative to that of the Random policy for the "All" campaign.

The data is large and contains many millions of recommendation instances. It also includes the true action choice probabilities by behavior policies computed by Monte Carlo simulations based on the policy parameters (e.g., parameters of the beta distribution used by Bernoulli TS) used during the data collection process. The number of actions is also sizable, so this setting is challenging for bandit algorithms and their OPE. We share the full version of our data at https://research.zozo.com/data.html

6.1.2 Open Bandit Pipeline (OBP)

Open Bandit Pipeline is a series of implementations of dataset preprocessing, policy learning, and evaluation of OPE estimators. This pipeline allows researchers to focus on building their bandit algorithm or OPE estimator and easily compare them with others' methods in realistic and reproducible ways. Thus, it facilitates reproducible research on bandit algorithms and off-policy evaluation.



Open Bandit Pipeline consists of the following main modules.

- **dataset module**: This module provides a data loader for Open Bandit Dataset and a flexible interface for handling logged bandit feedback. It also provides tools to generate synthetic bandit datasets.
- **policy module**: This module provides interfaces for online and offline bandit algorithms. It also implements several standard algorithms.
- **simulator module**: This module provides functions for conducting offline bandit simulation.
- **ope module**: This module provides interfaces for OPE estimators. It also implements several standard OPE estimators.

In addition to the above algorithms and estimators, the pipeline also provides flexible interfaces. Therefore, researchers can easily implement their own algorithms or estimators and evaluate them with our data and pipeline. Moreover, the pipeline provides an interface for handling logged bandit feedback datasets. Thus, practitioners can combine their own datasets with the pipeline and easily evaluate bandit algorithms' performances in their settings.

Please see package reference for detailed information about Open Bandit Pipeline.

To our knowledge, our real-world dataset and pipeline are the first to include logged bandit datasets collected by running *multiple* different policies, policy implementations used in production, and their ground-truth policy values. These features enable the **evaluation of OPE** for the first time.

6.2 Related Resources

We summarize existing related resources for bandit algorithms and off-policy evaluation.

6.2.1 Related Datasets

Our dataset is most closely related to those of [Lefortier2016] and [11]. [Lefortier2016] introduces a large-scale logged bandit feedback data (Criteo data) from a leading company in the display advertising, Criteo. The data contains context vectors of user impressions, advertisements (ads) as actions, and click indicators as reward. It also provides the ex ante probability of each ad being selected by the behavior policy. Therefore, this data can be used to compare different *off-policy learning* methods, which aim to learn a new bandit policy using only log data generated by a behavior policy. In contrast, [11] introduces a dataset (Yahoo! data) collected on a news recommendation interface of the the Yahoo! Today Module. The data contains context vectors of user impressions, presented news as actions, and click indicators as reward. It was collected by running uniform random policy on the new recommendation platform, allowing researchers to evaluate their own bandit algorithms.

However, the Criteo and Yahoo! data have limitations, which we overcome as follows:

- The previous datasets do not provide the code (production implementation) of their behavior policy. Moreover, the data was collected by running only a single behavior policy. As a result, these data cannot be used for the evaluation and comparison of different OPE estimators.

→ In contrast, we provide the code of our behavior policies (i.e., Bernoulli TS and Random) in our pipeline, which allows researchers to re-run the same behavior policies on the log data. Our open data also contains logged bandit feedback data generated by *multiple* behavior policies. It enables the evaluation and comparison of different OPE estimators. This is the first large-scale bandit dataset that enables such evaluation of OPE with the ground-truth policy value of behavior policies.

- The previous datasets do not provide a pipeline implementation to handle their data. Researchers have to re-implement the experimental environment by themselves before implementing their own methods. This may lead to inconsistent experimental conditions across different studies, potentially causing reproducibility issues.

→ We implement the Open Bandit Pipeline to simplify and standardize the experimental processing of bandit algorithms and OPE with our open data. This tool thus contributes to the reproducible and transparent use of our data.

The following table summarizes key differences between our data and existing ones.

Table 2. Comparison of Currently Available Large-scale Bandit Datasets

	Criteo Data (Lefortier et al., 2016)	**Yahoo! Data** (Li et al., 2010)	**Open Bandit Dataset** (ours)
Domain	Display Advertising	News Recommendation	Fashion E-Commerce
#Data	$\geq$ 103M	$\geq$ 40M	$\geq$ 26M (will increase)
#Behavior Policies	1	1	2 (will increase)
Random A/B Test Data	✗	✔	✔
Behavior Policy Code	✗	✗	✔
Evaluation of Bandit Algorithms	✔	✔	✔
Evaluation of OPE	✗	✗	✔
Pipeline Implementation	✗	✗	✔

Notes: **#Data** is the total number of samples included in the data. **#Behavior Policies** is the number of behavior policies that were used to collect the data. **Random A/B Test Data** is whether the data contains a subset of data generated by the uniform random policy. **Behavior Policy Code** is whether the code (production implementation) of behavior policies is publicized along with the data. **Evaluation of Bandit Algorithms** is whether it is possible to use the data to evaluate a new bandit algorithm. **Evaluation of OPE** is whether it is possible to use the data to evaluate a new OPE estimator. **Pipeline Implementation** is whether a pipeline tool to handle the data is available.

6.2.2 Related Packages

There are several existing Python packages related to our Open Bandit Pipeline. For example, *contextualbandits* package (https://github.com/david-cortes/contextualbandits) contains implementations of several contextual bandit algorithms [Cortes2018]. It aims to provide an easy procedure to compare bandit algorithms to reproduce research papers that do not provide easily-available implementations. In addition, *RecoGym* (https://github.com/criteo-research/reco-gym) focuses on providing simulation bandit environments imitating the e-commerce recommendation setting [Rohde2018]. This package also implements an online bandit algorithm based on epsilon greedy and off-policy learning method based on IPW.

However, the following features differentiate our pipeline from the previous ones:

- The previous packages focus on implementing and comparing online bandit algorithms or off-policy learning method. Instead, they **cannot** be used to implement and compare the off-policy evaluation methods.

→ Our package implements a wide variety of OPE estimators including advanced ones such as Switch Estimators [8], More Robust Doubly Robust [1], and Doubly Robust with Shrinkage [9]. Moreover, it is possible to compare the estimation accuracies of these estimators with our package in a fair manner. Our package also provides flexible interfaces for implementing new OPE estimators. Thus, researchers can easily compare their own estimators with other methods using our packages.

- The previous packages cannot handle real-world bandit datasets.

→ Our package comes with the Open Bandit Dataset and includes the **dataset module**. This enables the evaluation of bandit algorithms and off-policy estimators using our real-world data. This function contributes to realistic experiments on these topics.

The following table summarizes key differences between our pipeline and existing ones.

Table 3. Comparison of Currently Available Packages of Bandit Algorithms

	contextualbandits (Cortes, 2018)	**RecoGym** (Rohde et al., 2018)	**Open Bandit Pipeline** (ours)
Synthetic Data Generator	✗	✔	✔
Support for Real-World Data	✗	✗	✔
Implementation of Bandit Algorithms	✔	✔	✔
Implementation of Basic Off-Policy Estimators	✔	✗	✔
Implementation of Advanced Off-Policy Estimators	✗	✗	✔
Evaluation of OPE	✗	✗	✔

Notes: **Synthetic Data Generator** is whether it is possible to create synthetic bandit datasets with the package. **Support for Real-World Data** is whether it is possible to handle real-world bandit datasets with the package. **Implementation of Bandit Algorithms** is whether the package includes implementation of online and offline bandit algorithms. **Implementation of Basic Off-Policy Estimators** is whether the package includes implementation of *basic* off-policy estimators such as DM, IPW, and DR described in Section 2.3. **Implementation of Advanced Off-Policy Estimators** is whether the package includes implementation of *advanced* off-policy estimators such as Switch Estimators and More Robust Doubly Robust. **Evaluation of OPE** is whether it is possible to evaluate the accuracy of off-policy estimators with the package.

6.3 Overview

6.3.1 Setup

We consider a general contextual bandit setting. Let $r \in [0, R_{\max}]$ denote a reward or outcome variable (e.g., whether a fashion item as an action results in a click). We let $x \in$ be a context vector (e.g., the user's demographic profile) that the decision maker observes when picking an action. Rewards and contexts are sampled from the unknown probability distributions $p(r \mid x, a)$ and $p(x)$, respectively. Let $:= \{0, \ldots, m\}$ be a finite set of $m + 1$ actions. We call a function $\pi :\rightarrow \Delta()$ a *policy*. It maps each context $x \in$ into a distribution over actions, where $\pi(a \mid x)$ is the probability of taking action a given x.

Let $:= \{(x_t, a_t, r_t)\}_{t=1}^T$ rounds of observations. a_t is a discrete variable indicating which action in is chosen in round t. r_t and x_t denote the reward and the context observed in round t, respectively. We assume that a logged bandit feedback is generated by a behavior policy π_b as follows:

$$\{(x_t, a_t, r_t)\}_{i=1}^T \sim \prod_{i=1}^T p(x_t)\pi_b(a_t \mid x_t)p(r_t \mid x_t, a_t),$$

where each context-action-reward triplets are sampled independently from the product distribution. Note that we assume a_t is independent of r_t conditional on x_t.

We let $\pi(x, a, r) := p(x)\pi(a \mid x)p(r \mid x, a)$ be the product distribution by a policy π. For a function $f(x, a, r)$, we use $[f] := ||^{-1} \sum_{(x_t, a_t, r_t) \in} f(x_t, a_t, r_t)$ to denote its empirical expectation over T observations in . Then, for a function $g(x, a)$, we let $g(x, \pi) :=_{a \sim \pi(a|x)} [g(x, a) \mid x]$. We also use $q(x, a) :=_{r \sim p(r|x,a)} [r \mid x, a]$ to denote the mean reward function.

6.3.2 Estimation Target

We are interested in using the historical logged bandit data to estimate the following *policy value* of any given *evaluation policy* π_e which might be different from π_b:

$$V(\pi_e) :=_{(x,a,r) \sim \pi_e(x,a,r)} [r].$$

where the last equality uses the independence of A and $Y(\cdot)$ conditional on X and the definition of $\pi_b(\cdot|X)$. We allow the evaluation policy π_e to be degenerate, i.e., it may choose a particular action with probability 1. Estimating $V(\pi_e)$ before implementing π_e in an online environment is valuable because π_e may perform poorly and damage user satisfaction. Additionally, it is possible to select an evaluation policy that maximizes the policy value by comparing their estimated performances without having additional implementation cost.

6.4 Estimators

6.4.1 Direct Method (DM)

A widely-used method, DM, first learns a supervised machine learning model, such as random forest, ridge regression, and gradient boosting, to estimate the mean reward function. DM then uses it to estimate the policy value as

$$\hat{V}_{\mathrm{DM}}(\pi_e; , \hat{q}) := [\hat{q}(x_t, \pi_e)],$$

where $\hat{q}(a \mid x)$ is the estimated reward function. If $\hat{q}(a \mid x)$ is a good approximation to the mean reward function, this estimator accurately estimates the policy value of the evaluation policy V^{π}. If $\hat{q}(a \mid x)$ fails to approximate the mean reward function well, however, the final estimator is no longer consistent. The model misspecification issue is problematic because the extent of misspecification cannot be easily quantified from data [1].

6.4.2 Inverse Probability Weighting (IPW)

To alleviate the issue with DM, researchers often use another estimator called IPW [2] [3]. IPW re-weights the rewards by the ratio of the evaluation policy and behavior policy as

$$\hat{V}_{\mathrm{IPW}}(\pi_e;) := [w(x_t, a_t) r_t],$$

where $w(x, a) := \pi_e(a \mid x)/\pi_b(a \mid x)$ is the importance weight given x and a. When the behavior policy is known, the IPW estimator is unbiased and consistent for the policy value. However, it can have a large variance, especially when the evaluation policy significantly deviates from the behavior policy.

6.4.3 Doubly Robust (DR)

The final approach is DR [4], which combines the above two estimators as

$$\hat{V}_{\mathrm{DR}} := [\hat{q}(x_t, \pi_e) + w(x_t, a_t)(r_t - \hat{q}(x_t, a_t))].$$

DR mimics IPW to use a weighted version of rewards, but DR also uses the estimated mean reward function as a control variate to decrease the variance. It preserves the consistency of IPW if either the importance weight or the mean reward estimator is accurate (a property called *double robustness*). Moreover, DR is *semiparametric efficient* [5] when the mean reward estimator is correctly specified. On the other hand, when it is wrong, this estimator can have larger asymptotic mean-squared-error than IPW [6] and perform poorly in practice [7].

6.4.4 Self-Normalized Estimators

Self-Normalized Inverse Probability Weighting (SNIPW) is an approach to address the variance issue with the original IPW. It estimates the policy value by dividing the sum of weighted rewards by the sum of importance weights as:

$$\hat{V}_{\mathrm{SNIPW}}(\pi_e;) := \frac{[w(x_t, a_t) r_t]}{[w(x_t, a_t)]}.$$

SNIPW is more stable than IPW, because estimated policy value by SNIPW is bounded in the support of rewards and its conditional variance given action and context is bounded by the conditional variance of the rewards:cite:*kallus2019*. IPW does not have these properties. We can define Self-Normalized Doubly Robust (SNDR) in a similar manner as follows.

$$\hat{V}_{\mathrm{SNDR}}(\pi_e;) := \left[\hat{q}(x_t, \pi_e) + \frac{w(x_t, a_t)(r_t - \hat{q}(x_t, a_t))}{[w(x_t, a_t)]}\right].$$

6.4.5 Switch Estimators

The DR estimator can still be subject to the variance issue, particularly when the importance weights are large due to low overlap. Switch-DR aims to reduce the effect of the variance issue by using DM where importance weights are large as:

$$\hat{V}_{\mathrm{SwitchDR}}(\pi_e; , \hat{q}, \tau) := [\hat{q}(x_t, \pi_e) + w(x_t, a_t)(r_t - \hat{q}(x_t, a_t))\mathbb{I}\{w(x_t, a_t) \leq \tau\}] ,$$

where $\mathbb{I}\{\cdot\}$ is the indicator function and $\tau \geq 0$ is a hyperparameter. Switch-DR interpolates between DM and DR. When $\tau = 0$, it coincides with DM, while $\tau \to \infty$ yields DR. This estimator is minimax optimal when τ is appropriately chosen [8].

6.4.6 More Robust Doubly Robust (MRDR)

MRDR uses a specialized reward estimator ($\hat{q}_{\mathrm{MRDR}}$) that minimizes the variance of the resulting policy value estimator:cite:*Farajtabar2018*. This estimator estimates the policy value as:

$$\hat{V}_{\mathrm{MRDR}}(\pi_e; , \hat{q}_{\mathrm{MRDR}}) := \hat{V}_{\mathrm{DR}}(\pi_e; , \hat{q}_{\mathrm{MRDR}}),$$

where $\mathcal{Q}$ is a function class for the reward estimator. When $\mathcal{Q}$ is well-specified, then $\hat{q}_{\mathrm{MRDR}} = q$. Here, even if $\mathcal{Q}$ is misspecified, the derived reward estimator is expected to behave well since the target function is the resulting variance.

6.4.7 Doubly Robust with Optimistic Shrinkage (DRos)

[9] proposes DRs based on a new weight function $w_o : \times \to \mathbb{R}_+$ that directly minimizes sharp bounds on the MSE of the resulting estimator. DRs is defined as

$$\hat{V}_{\mathrm{DRs}}(\pi_e; , \hat{q}, \lambda) := [\hat{q}(x_t, \pi_e) + w_o(x_t, a_t; \lambda)(r_t - \hat{q}(x_t, a_t))],$$

where $\lambda \geq 0$ is a hyperparameter and the new weight is

$$w_o(x, a; \lambda) := \frac{\lambda}{w^2(x, a) + \lambda} w(x, a).$$

When $\lambda = 0$, $w_o(x, a; \lambda) = 0$ leading to the standard DM. On the other hand, as $\lambda \to \infty$, $w_o(x, a; \lambda) = w(x, a)$ leading to the original DR.

6.5 Evaluation of OPE

Here we describe an experimental protocol to evaluate OPE estimators and use it to compare a wide variety of existing estimators.

We can empirically evaluate OPE estimators' performances by using two sources of logged bandit feedback collected by two different policies $\pi^{(he)}$ (hypothetical evaluation policy) and $\pi^{(hb)}$ (hypothetical behavior policy). We denote log data generated by $\pi^{(he)}$ and $\pi^{(hb)}$ as ${}^{(he)} := \{(x_t^{(he)}, a_t^{(he)}, r_t^{(he)})\}_{t=1}^T$ and ${}^{(hb)} := \{(x_t^{(hb)}, a_t^{(hb)}, r_t^{(hb)})\}_{t=1}^T$, respectively. By applying the following protocol to several different OPE estimators, we can compare their estimation performances:

1. Define the evaluation and test sets as:
   - in-sample case: ${}_{\mathrm{ev}} :=_{1:T}^{(hb)}$, ${}_{\mathrm{te}} :=_{1:T}^{(he)}$
   - out-sample case: ${}_{\mathrm{ev}} :=_{1:\tilde{t}}^{(hb)}$, ${}_{\mathrm{te}} :=_{\tilde{t}+1:T}^{(he)}$

   where ${}_{a:b} := \{(x_t, a_t, r_t)\}_{t=a}^b$.

2. Estimate the policy value of $\pi^{(he)}$ using $_{\mathrm{ev}}$ by an estimator $\hat{V}$. We can represent an estimated policy value by $\hat{V}$ as $\hat{V}(\pi^{(he)};_{\mathrm{ev}})$.
3. Estimate $V(\pi^{(he)})$ by the *on-policy estimation* and regard it as the ground-truth as

$$V_{\mathrm{on}}(\pi^{(he)};_{\mathrm{te}}) :=_{\mathrm{te}} [r_t^{(he)}].$$

4. Compare the off-policy estimate $\hat{V}(\pi^{(he)};_{\mathrm{ev}})$ with its ground-truth $V_{\mathrm{on}}(\pi^{(he)};_{\mathrm{te}})$. We can evaluate the estimation accuracy of $\hat{V}$ by the following *relative estimation error* (relative-EE):

$$\textit{relative-EE}(\hat{V};_{\mathrm{ev}}) := \left| \frac{\hat{V}(\pi^{(he)};_{\mathrm{ev}}) - V_{\mathrm{on}}(\pi^{(he)};_{\mathrm{te}})}{V_{\mathrm{on}}(\pi^{(he)};_{\mathrm{te}})} \right|.$$

5. To estimate standard deviation of relative-EE, repeat the above process several times with different bootstrap samples of the logged bandit data created by sampling data *with replacement* from $_{\mathrm{ev}}$.

We call the problem setting **without** the sample splitting by time series as in-sample case. In contrast, we call that **with** the sample splitting as out-sample case where OPE estimators aim to estimate the policy value of an evaluation policy in the test data.

The following algorithm describes the detailed experimental protocol to evaluate OPE estimators.

Algorithm 1 Experimental Protocol for Evaluating Off-Policy Estimators

Require: a policy $\pi^{(he)}$; two different logged bandit feedback datasets $\mathcal{D}^{(he)} = \{(x_t^{(he)}, a_t^{(he)}, r_t^{(he)})\}_{t=1}^T$ and $\mathcal{D}^{(hb)} = \{(x_t^{(hb)}, a_t^{(hb)}, r_t^{(hb)})\}_{t=1}^T$ where $\mathcal{D}^{(he)}$ is collected by $\pi^{(he)}$ and $\mathcal{D}^{(hb)}$ is collected by a different one $\pi^{(hb)}$; an off-policy estimator to be evaluated $\hat{V}$; *split-point* $\tilde{t}$

Ensure: the mean and standard deviations of *relative-EE*$(\hat{V})$

$\mathcal{S} \leftarrow \emptyset$
Define the evaluation set: $\mathcal{D}_{\mathrm{ev}} := \mathcal{D}_{1:T}^{(hb)}$ (*in*-sample case), $\mathcal{D}_{\mathrm{ev}} := \mathcal{D}_{1:\tilde{t}}^{(hb)}$ (*out*-sample case)
Define the test set: $\mathcal{D}_{\mathrm{te}} := \mathcal{D}_{1:T}^{(he)}$ (*in*-sample case), $\mathcal{D}_{\mathrm{te}} := \mathcal{D}_{\tilde{t}+1:T}^{(he)}$ (*out*-sample case)
Approximate $V(\pi^{(he)})$ by its on-policy estimation using $\mathcal{D}_{\mathrm{te}}$, i.e, $V_{\mathrm{on}}(\pi^{(he)};\mathcal{D}_{\mathrm{te}}) = \mathbb{E}_{\mathcal{D}_{\mathrm{te}}}[r_t^{(he)}]$
for $b = 1, \ldots, K$ **do**
    Sample data from $\mathcal{D}_{\mathrm{ev}}$ with *replacement* and construct b-th bootstrapped samples $\mathcal{D}_{\mathrm{ev}}^{(b,*)}$
    Estimate the policy value of $\pi^{(he)}$ by $\hat{V}(\pi^{(he)};\mathcal{D}_{\mathrm{ev}}^{(b,*)})$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{\textit{relative-EE}\ (\hat{V};\mathcal{D}_{\mathrm{ev}}^{(b,*)})\}$
end for
Estimate the mean and standard deviations of *relative-EE*$(\hat{V})$ using $\mathcal{S}$

Using the above protocol, our real-world data, and pipeline, we have performed extensive benchmark experiments on a variety of existing off-policy estimators. The experimental results and the relevant discussion can be found in our paper. The code for running the benchmark experiments can be found at zr-obp/benchmark/ope.

6.6 Installation

obp is available on PyPI, and can be installed from pip or source as follows:

From pip:

```
pip install obp
```

From source:

```
git clone https://github.com/st-tech/zr-obp
cd zr-obp
python setup.py install
```

6.7 Quickstart

We show an example of conducting offline evaluation of the performance of Bernoulli Thompson Sampling (BernoulliTS) as an evaluation policy using *Inverse Probability Weighting (IPW)* and logged bandit feedback generated by the Random policy (behavior policy). We see that only ten lines of code are sufficient to complete OPE from scratch. In this example, it is assumed that the *obd/random/all* directory exists under the present working directory. Please clone the repository in advance.

```
# a case for implementing OPE of the BernoulliTS policy using log data generated by␣
↪the Random policy
>>> from obp.dataset import OpenBanditDataset
>>> from obp.policy import BernoulliTS
>>> from obp.ope import OffPolicyEvaluation, InverseProbabilityWeighting as IPW

# (1) Data loading and preprocessing
>>> dataset = OpenBanditDataset(behavior_policy='random', campaign='all')
>>> bandit_feedback = dataset.obtain_batch_bandit_feedback()

# (2) Off-Policy Learning
>>> evaluation_policy = BernoulliTS(
    n_actions=dataset.n_actions,
    len_list=dataset.len_list,
    is_zozotown_prior=True,
    campaign="all",
    random_state=12345
)
>>> action_dist = evaluation_policy.compute_batch_action_dist(
    n_sim=100000, n_rounds=bandit_feedback["n_rounds"]
)

# (3) Off-Policy Evaluation
>>> ope = OffPolicyEvaluation(bandit_feedback=bandit_feedback, ope_estimators=[IPW()])
>>> estimated_policy_value = ope.estimate_policy_values(action_dist=action_dist)

# estimated performance of BernoulliTS relative to the ground-truth performance of␣
↪Random
>>> relative_policy_value_of_bernoulli_ts = estimated_policy_value['ipw'] / bandit_
↪feedback['reward'].mean()
>>> print(relative_policy_value_of_bernoulli_ts)
1.198126...
```

A detailed introduction with the same example can be found at quickstart. Below, we explain some important features in the example flow.

6.7.1 Data loading and preprocessing

We prepare an easy-to-use data loader for Open Bandit Dataset.

```
# load and preprocess raw data in "ALL" campaign collected by the Random policy
>>> dataset = OpenBanditDataset(behavior_policy='random', campaign='all')
# obtain logged bandit feedback generated by the behavior policy
>>> bandit_feedback = dataset.obtain_batch_bandit_feedback()

>>> print(bandit_feedback.keys())
dict_keys(['n_rounds', 'n_actions', 'action', 'position', 'reward', 'pscore', 'context
↪', 'action_context'])
```

Users can implement their own feature engineering in the `pre_process` method of `obp.dataset.OpenBanditDataset` class. We show an example of implementing some new feature engineering processes in custom_dataset.py.

Moreover, by following the interface of `obp.dataset.BaseBanditDataset` class, one can handle their own or future open datasets for bandit algorithms other than our OBD.

6.7.2 Off-Policy Learning

After preparing a dataset, we now compute the action choice probability of BernoulliTS in the ZOZOTOWN production. Then, we can use it as the evaluation policy.

```
# define evaluation policy (the Bernoulli TS policy here)
# by activating the `is_zozotown_prior` argument of BernoulliTS, we can replicate␣
↪BernoulliTS used in ZOZOTOWN production.
>>> evaluation_policy = BernoulliTS(
    n_actions=dataset.n_actions,
    len_list=dataset.len_list,
    is_zozotown_prior=True, # replicate the policy in the ZOZOTOWN production
    campaign="all",
    random_state=12345
)
# compute the distribution over actions by the evaluation policy using Monte Carlo␣
↪simulation
# action_dist is an array of shape (n_rounds, n_actions, len_list)
# representing the distribution over actions made by the evaluation policy
>>> action_dist = evaluation_policy.compute_batch_action_dist(
    n_sim=100000, n_rounds=bandit_feedback["n_rounds"]
)
```

The `compute_batch_action_dist` method of `BernoulliTS` computes the action choice probabilities based on given hyperparameters of the beta distribution. `action_dist` is an array representing the distribution over actions made by the evaluation policy.

6.7.3 Off-Policy Evaluation

Our final step is **off-policy evaluation** (OPE), which attempts to estimate the performance of decision making policy using log data generated by offline bandit simulation. Our pipeline also provides an easy procedure for doing OPE as follows.

```
# estimate the policy value of BernoulliTS based on the distribution over actions by␣
↪that policy
# it is possible to set multiple OPE estimators to the `ope_estimators` argument
>>> ope = OffPolicyEvaluation(bandit_feedback=bandit_feedback, ope_
↪estimators=[ReplayMethod()])
>>> estimated_policy_value = ope.estimate_policy_values(action_dist=action_dist)
>>> print(estimated_policy_value)
{'ipw': 0.004553...} # dictionary containing estimated policy values by each OPE␣
↪estimator.

# compare the estimated performance of BernoulliTS (evaluation policy)
# with the ground-truth performance of Random (behavior policy)
>>> relative_policy_value_of_bernoulli_ts = estimated_policy_value['ipw'] / bandit_
↪feedback['reward'].mean()
# our OPE procedure suggests that BernoulliTS improves Random by 19.81%
>>> print(relative_policy_value_of_bernoulli_ts)
1.198126...
```

Users can implement their own OPE estimator by following the interface of `obp.ope.BaseOffPolicyEstimator` class. `obp.ope.OffPolicyEvaluation` class summarizes and compares the estimated policy values by several off-policy estimators. A detailed usage of this class can be found at quickstart. `bandit_feedback['reward'].mean()` is the empirical mean of factual rewards (on-policy estimate of the policy value) in the log and thus is the ground-truth performance of the behavior policy (the Random policy in this example.).

6.8 OBP Package Reference

6.8.1 ope module

obp.ope.estimators	Off-Policy Estimators.
obp.ope.meta	Off-Policy Evaluation Class to Streamline OPE.
obp.ope.regression_model	Regression Model Class for Estimating Mean Reward Functions.

obp.ope.estimators

Off-Policy Estimators.

Classes

BaseOffPolicyEstimator()	Base class for OPE estimators.
DirectMethod(estimator_name)	Estimate the policy value by Direct Method (DM).
DoublyRobust(estimator_name)	Estimate the policy value by Doubly Robust (DR).
DoublyRobustWithShrinkage(estimator_name, ...)	Estimate the policy value by Doubly Robust with optimistic shrinkage (DRos).
InverseProbabilityWeighting(estimator_name)	Estimate the policy value by Inverse Probability Weighting (IPW).
ReplayMethod(estimator_name)	Estimate the policy value by Relpay Method (RM).
SelfNormalizedDoublyRobust(estimator_name)	Estimate the policy value by Self-Normalized Doubly Robust (SNDR).
SelfNormalizedInverseProbabilityWeighting(...)	Estimate the policy value by Self-Normalized Inverse Probability Weighting (SNIPW).
SwitchDoublyRobust(estimator_name, tau)	Estimate the policy value by Switch Doubly Robust (Switch-DR).
SwitchInverseProbabilityWeighting(...)	Estimate the policy value by Switch Inverse Probability Weighting (Switch-IPW).

class obp.ope.estimators.**BaseOffPolicyEstimator**

Bases: object

Base class for OPE estimators.

abstract estimate_interval() → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

abstract estimate_policy_value() → float

Estimate policy value of an evaluation policy.

class obp.ope.estimators.**DirectMethod**(*estimator_name: str = 'dm'*)

Bases: *obp.ope.estimators.BaseOffPolicyEstimator*

Estimate the policy value by Direct Method (DM).

Note: DM first learns a supervised machine learning model, such as ridge regression and gradient boosting, to estimate the mean reward function ($q(x, a) = \mathbb{E}[r|x, a]$). It then uses it to estimate the policy value as follows.

$$\hat{V}_{\mathrm{DM}}(\pi_e; \mathcal{D}, \hat{q}) := \mathbb{E}_{\mathcal{D}} \left[\sum_{a \in \mathcal{A}} \hat{q}(x_t, a) \pi_e(a|x_t) \right], \\ = \mathbb{E}_{\mathcal{D}}[\hat{q}(x_t, \pi_e)],$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$. $\hat{q}(x, a)$ is an estimated expected reward given x and a. $\hat{q}(x_t, \pi) := \mathbb{E}_{a \sim \pi(a|x)}[\hat{q}(x, a)]$ is the expectation of the estimated reward function over π. To estimate the mean reward function, please use *obp.ope.regression_model.RegressionModel*, which supports several fitting methods specific to OPE.

If the regression model ($\hat{q}$) is a good approximation to the true mean reward function, this estimator accurately estimates the policy value of the evaluation policy. If the regression function fails to approximate the mean reward function well, however, the final estimator is no longer consistent.

Parameters **estimator_name** (*str, default='dm'.*) – Name of off-policy estimator.

References

Alina Beygelzimer and John Langford. "The offset tree for learning with partial labels.", 2009.

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

estimate_interval(*position: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*position: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*, ***kwargs*) → float

Estimate policy value of an evaluation policy.

Parameters

- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.

Returns **V_hat** – Estimated policy value (performance) of a given evaluation policy.

Return type float

class obp.ope.estimators.**DoublyRobust**(*estimator_name: str = 'dr'*)

Bases: *obp.ope.estimators.InverseProbabilityWeighting*

Estimate the policy value by Doubly Robust (DR).

Note: Similar to DM, DR first learns a supervised machine learning model, such as ridge regression and gradient boosting, to estimate the mean reward function ($q(x, a) = \mathbb{E}[r|x, a]$). It then uses it to estimate the

policy value as follows.

$$\hat{V}_{\mathrm{DR}}(\pi_e; \mathcal{D}, \hat{q}) := \mathbb{E}_{\mathcal{D}}[\hat{q}(x_t, \pi_e) + w(x_t, a_t)(r_t - \hat{q}(x_t, a_t))],$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $w(x, a) := \pi_e(a|x)/\pi_b(a|x)$ is the importance weight given x and a. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$. $\hat{q}(x, a)$ is an estimated expected reward given x and a. $\hat{q}(x_t, \pi) := \mathbb{E}_{a \sim \pi(a|x)}[\hat{q}(x, a)]$ is the expectation of the estimated reward function over π.

To estimate the mean reward function, please use *obp.ope.regression_model.RegressionModel*, which supports several fitting methods specific to OPE such as *more robust doubly robust*.

DR mimics IPW to use a weighted version of rewards, but DR also uses the estimated mean reward function (the regression model) as a control variate to decrease the variance. It preserves the consistency of IPW if either the importance weight or the mean reward estimator is accurate (a property called double robustness). Moreover, DR is semiparametric efficient when the mean reward estimator is correctly specified.

Parameters **estimator_name** (*str, default='dr'.*) – Name of off-policy estimator.

References

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

Mehrdad Farajtabar, Yinlam Chow, and Mohammad Ghavamzadeh. "More Robust Doubly Robust Off-policy Evaluation.", 2018.

estimate_interval(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.

- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*) → float

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.

Returns **V_hat** – Estimated policy value by the DR estimator.

Return type float

class obp.ope.estimators.**DoublyRobustWithShrinkage**(*estimator_name: str = 'dr-os'*, *lambda_: float = 0.0*)

Bases: *obp.ope.estimators.DoublyRobust*

Estimate the policy value by Doubly Robust with optimistic shrinkage (DRos).

Note: DR with (optimistic) shrinkage replaces the importance weight in the original DR estimator with a new weight mapping found by directly optimizing sharp bounds on the resulting MSE.

$$\hat{V}_{\mathrm{DRos}}(\pi_e; \mathcal{D}, \hat{q}, \lambda) := \mathbb{E}_{\mathcal{D}}[\hat{q}(x_t, \pi_e) + w_o(x_t, a_t; \lambda)(r_t - \hat{q}(x_t, a_t))],$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $w(x, a) := \pi_e(a|x)/\pi_b(a|x)$ is the importance weight given x and a. $\hat{q}(x_t, \pi) := \mathbb{E}_{a \sim \pi(a|x)}[\hat{q}(x, a)]$ is the expectation of the estimated reward function over π. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$. $\hat{q}(x, a)$ is an estimated expected reward given x and a. To estimate the mean reward function, please use *obp.ope.regression_model.RegressionModel*.

$w_o(x_t, a_t; \lambda)$ is a new weight by the shrinkage technique which is defined as

$$w_o(x_t, a_t; \lambda) := \frac{\lambda}{w^2(x_t, a_t) + \lambda} w(x_t, a_t).$$

When $\lambda = 0$, we have $w_o(x, a; \lambda) = 0$ corresponding to the DM estimator. In contrast, as $\lambda \to \infty$, $w_o(x, a; \lambda)$ increases and in the limit becomes equal to the original importance weight, corresponding to the standard DR estimator.

Parameters

- **lambda_** (*float*) – Shrinkage hyperparameter. This hyperparameter should be larger than or equal to 0., otherwise it is meaningless.
- **estimator_name** (*str, default='dr-os'.*) – Name of off-policy estimator.

References

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

Yi Su, Maria Dimakopoulou, Akshay Krishnamurthy, and Miroslav Dudik. "Doubly Robust Off-Policy Evaluation with Shrinkage.", 2020.

estimate_interval(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*) → float

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.

Returns **V_hat** – Estimated policy value by the DR estimator.

Return type float

class obp.ope.estimators.**InverseProbabilityWeighting**(*estimator_name: str = 'ipw'*)

Bases: *obp.ope.estimators.BaseOffPolicyEstimator*

Estimate the policy value by Inverse Probability Weighting (IPW).

Note: Inverse Probability Weighting (IPW) estimates the policy value of a given evaluation policy π_e by

$$\hat{V}_{\mathrm{IPW}}(\pi_e; \mathcal{D}) := \mathbb{E}_{\mathcal{D}}[w(x_t, a_t) r_t],$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $w(x, a) := \pi_e(a|x)/\pi_b(a|x)$ is the importance weight given x and a. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$.

IPW re-weights the rewards by the ratio of the evaluation policy and behavior policy (importance weight). When the behavior policy is known, IPW is unbiased and consistent for the true policy value. However, it can have a large variance, especially when the evaluation policy significantly deviates from the behavior policy.

Parameters **estimator_name** (*str, default='ipw'.*) – Name of off-policy estimator.

References

Alex Strehl, John Langford, Lihong Li, and Sham M Kakade. "Learning from Logged Implicit Exploration Data"., 2010.

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

estimate_interval(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, ***kwargs*) → numpy.ndarray

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.

Returns **V_hat** – Estimated policy value (performance) of a given evaluation policy.

Return type float

class obp.ope.estimators.**ReplayMethod**(*estimator_name: str = 'rm'*)

Bases: *obp.ope.estimators.BaseOffPolicyEstimator*

Estimate the policy value by Relpay Method (RM).

Note: Replay Method (RM) estimates the policy value of a given evaluation policy π_e by

$$\hat{V}_{\mathrm{RM}}(\pi_e; \mathcal{D}) := \frac{\mathbb{E}_{\mathcal{D}}[\mathbb{I}\{\pi_e(x_t) = a_t\} r_t]}{\mathbb{E}_{\mathcal{D}}[\mathbb{I}\{\pi_e(x_t) = a_t\}]},$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $\pi_e : \mathcal{X} \to \mathcal{A}$ is the function representing action choices by the evaluation policy realized during offline bandit simulation. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$.

Parameters **estimator_name** (*str, default='rm'.*) – Name of off-policy estimator.

References

Lihong Li, Wei Chu, John Langford, and Xuanhui Wang. "Unbiased Offline Evaluation of Contextual-bandit-based News Article Recommendation Algorithms.", 2011.

estimate_interval (*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *action_dist: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 100*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value (*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *action_dist: numpy.ndarray*, ***kwargs*) → float

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.

Returns **V_hat** – Estimated policy value (performance) of a given evaluation policy.

Return type float

class obp.ope.estimators.**SelfNormalizedDoublyRobust** (*estimator_name: str = 'sndr'*)

Bases: *obp.ope.estimators.DoublyRobust*

Estimate the policy value by Self-Normalized Doubly Robust (SNDR).

Note: Self-Normalized Doubly Robust estimates the policy value of a given evaluation policy π_e by

$$\hat{V}_{\mathrm{SNDR}}(\pi_e; \mathcal{D}, \hat{q}) := \mathbb{E}_{\mathcal{D}} \left[\hat{q}(x_t, \pi_e) + \frac{w(x_t, a_t)(r_t - \hat{q}(x_t, a_t))}{\mathbb{E}_{\mathcal{D}}[w(x_t, a_t)]} \right],$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $w(x, a) := \pi_e(a|x)/\pi_b(a|x)$ is the importance weight given x and a. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$. $\hat{q}(x, a)$ is an estimated expected reward given x and a. $\hat{q}(x_t, \pi) := \mathbb{E}_{a \sim \pi(a|x)}[\hat{q}(x, a)]$ is the expectation of the estimated reward function over π. To estimate the mean reward function, please use *obp.ope.regression_model.RegressionModel*.

Similar to Self-Normalized Inverse Probability Weighting, SNDR estimator applies the self-normalized importance weighting technique to increase the stability of the original Doubly Robust estimator.

Parameters **estimator_name** (*str, default='sndr'.*) – Name of off-policy estimator.

References

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

Nathan Kallus and Masatoshi Uehara. "Intrinsically Efficient, Stable, and Bounded Off-Policy Evaluation for Reinforcement Learning.", 2019.

estimate_interval(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*) → float

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.

Returns **V_hat** – Estimated policy value by the DR estimator.

Return type float

class obp.ope.estimators.**SelfNormalizedInverseProbabilityWeighting**(*estimator_name: str = 'snipw'*)

Bases: *obp.ope.estimators.InverseProbabilityWeighting*

Estimate the policy value by Self-Normalized Inverse Probability Weighting (SNIPW).

Note: Self-Normalized Inverse Probability Weighting (SNIPW) estimates the policy value of a given evaluation policy π_e by

$$\hat{V}_{\mathrm{SNIPW}}(\pi_e; \mathcal{D}) := \frac{\mathbb{E}_{\mathcal{D}}[w(x_t, a_t) r_t]}{\mathbb{E}_{\mathcal{D}}[w(x_t, a_t)]},$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $w(x, a) := \pi_e(a|x)/\pi_b(a|x)$ is the importance weight given x and a. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$.

SNIPW re-weights the observed rewards by the self-normalized importance weihgt. This estimator is not unbiased even when the behavior policy is known. However, it is still consistent for the true policy value and increases the stability in some senses. See the references for the detailed discussions.

Parameters **estimator_name** (*str, default='snipw'.*) – Name of off-policy estimator.

References

Adith Swaminathan and Thorsten Joachims. "The Self-normalized Estimator for Counterfactual Learning.", 2015.

Nathan Kallus and Masatoshi Uehara. "Intrinsically Efficient, Stable, and Bounded Off-Policy Evaluation for Reinforcement Learning.", 2019.

estimate_interval(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, ***kwargs*) → numpy.ndarray

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.

Returns **V_hat** – Estimated policy value (performance) of a given evaluation policy.

Return type float

class obp.ope.estimators.**SwitchDoublyRobust**(*estimator_name: str = 'switch-dr'*, *tau: float = 1*)

Bases: *obp.ope.estimators.DoublyRobust*

Estimate the policy value by Switch Doubly Robust (Switch-DR).

Note: Switch-DR aims to reduce the variance of the DR estimator by using direct method when the importance weight is large. This estimator estimates the policy value of a given evaluation policy π_e by

$$\hat{V}_{\mathrm{SwitchDR}}(\pi_e; \mathcal{D}, \hat{q}, \tau) := \mathbb{E}_{\mathcal{D}}[\hat{q}(x_t, \pi_e) + w(x_t, a_t)(r_t - \hat{q}(x_t, a_t))\mathbb{I}\{w(x_t, a_t) \leq \tau\}],$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $w(x, a) := \pi_e(a|x)/\pi_b(a|x)$ is the importance weight given x and a. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$. $\tau (\geq 0)$ is a switching hyperparameter, which decides the threshold for the importance weight. $\hat{q}(x, a)$ is an estimated expected reward given x and a. $\hat{q}(x_t, \pi) := \mathbb{E}_{a \sim \pi(a|x)}[\hat{q}(x, a)]$ is the expectation of the estimated reward function over π. To estimate the mean reward function, please use *obp.ope.regression_model.RegressionModel*.

Parameters

- **tau** (*float, default=1*) – Switching hyperparameter. When importance weight is larger than this parameter, the DM estimator is applied, otherwise the DR estimator is applied. This hyperparameter should be larger than or equal to 0., otherwise it is meaningless.
- **estimator_name** (*str, default='switch-dr'.*) – Name of off-policy estimator.

References

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

Yu-Xiang Wang, Alekh Agarwal, and Miroslav Dudík. "Optimal and Adaptive Off-policy Evaluation in Contextual Bandits", 2016.

estimate_interval(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.

- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*) → float

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.

Returns **V_hat** – Estimated policy value by the DR estimator.

Return type float

class obp.ope.estimators.**SwitchInverseProbabilityWeighting**(*estimator_name: str = 'switch-ipw'*, *tau: float = 1*)

Bases: *obp.ope.estimators.DoublyRobust*

Estimate the policy value by Switch Inverse Probability Weighting (Switch-IPW).

Note: Switch-IPW aims to reduce the variance of the IPW estimator by using direct method when the importance weight is large. This estimator estimates the policy value of a given evaluation policy π_e by

$$\hat{V}_{\mathrm{SwitchIPW}}(\pi_e; \mathcal{D}, \tau) := \mathbb{E}_{\mathcal{D}} \left[\sum_{a \in \mathcal{A}} \hat{q}(x_t, a) \pi_e(a|x_t) \mathbb{I}\{w(x_t, a) > \tau\} + w(x_t, a_t) r_t \mathbb{I}\{w(x_t, a_t) \leq \tau\} \right],$$

where $\mathcal{D} = \{(x_t, a_t, r_t)\}_{t=1}^{T}$ is logged bandit feedback data with T rounds collected by a behavior policy π_b. $w(x,a) := \pi_e(a|x)/\pi_b(a|x)$ is the importance weight given x and a. $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$. $\tau (\geq 0)$ is a switching hyperparameter, which decides the threshold for the importance weight. To estimate the mean reward function, please use *obp.ope.regression_model.RegressionModel*.

Parameters

- **tau** (*float, default=1*) – Switching hyperparameter. When importance weight is larger than this parameter, the DM estimator is applied, otherwise the IPW estimator is applied. This hyperparameter should be larger than 1., otherwise it is meaningless.
- **estimator_name** (*str, default='switch-ipw'.*) – Name of off-policy estimator.

References

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

Yu-Xiang Wang, Alekh Agarwal, and Miroslav Dudík. "Optimal and Adaptive Off-policy Evaluation in Contextual Bandits", 2016.

estimate_interval(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*, ***kwargs*) → Dict[str, float]

Estimate confidence interval of policy value by nonparametric bootstrap procedure.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

estimate_policy_value(*reward: numpy.ndarray*, *action: numpy.ndarray*, *position: numpy.ndarray*, *pscore: numpy.ndarray*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: numpy.ndarray*) → float

Estimate policy value of an evaluation policy.

Parameters

- **reward** (*array-like, shape (n_rounds,)*) – Reward observed in each round of the logged bandit feedback, i.e., r_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **position** (*array-like, shape (n_rounds,)*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,)*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list)*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$.

Returns **V_hat** – Estimated policy value by the DR estimator.

Return type float

obp.ope.meta

Off-Policy Evaluation Class to Streamline OPE.

Classes

OffPolicyEvaluation(bandit_feedback, ...)	Class to conduct off-policy evaluation by multiple off-policy estimators simultaneously.

class obp.ope.meta.**OffPolicyEvaluation**(*bandit_feedback: Dict[str, Union[int, numpy.ndarray]]*, *ope_estimators: List[*obp.ope.estimators.BaseOffPolicyEstimator*]*)

Bases: object

Class to conduct off-policy evaluation by multiple off-policy estimators simultaneously.

Parameters

- **bandit_feedback** (*BanditFeedback*) – Logged bandit feedback data used for off-policy evaluation.
- **ope_estimators** (*List[BaseOffPolicyEstimator]*) – List of OPE estimators used to evaluate the policy value of evaluation policy. Estimators must follow the interface of *obp.ope.BaseOffPolicyEstimator*.

Examples

```
# a case for implementing OPE of the BernoulliTS policy
# using log data generated by the Random policy
>>> from obp.dataset import OpenBanditDataset
>>> from obp.policy import BernoulliTS
>>> from obp.ope import OffPolicyEvaluation, InverseProbabilityWeighting as IPW

# (1) Data loading and preprocessing
>>> dataset = OpenBanditDataset(behavior_policy='random', campaign='all')
>>> bandit_feedback = dataset.obtain_batch_bandit_feedback()
>>> bandit_feedback.keys()
dict_keys(['n_rounds', 'n_actions', 'action', 'position', 'reward', 'pscore',
↪'context', 'action_context'])

# (2) Off-Policy Learning
>>> evaluation_policy = BernoulliTS(
    n_actions=dataset.n_actions,
    len_list=dataset.len_list,
    is_zozotown_prior=True, # replicate the policy in the ZOZOTOWN production
    campaign="all",
    random_state=12345
)
>>> action_dist = evaluation_policy.compute_batch_action_dist(
    n_sim=100000, n_rounds=bandit_feedback["n_rounds"]
)

# (3) Off-Policy Evaluation
>>> ope = OffPolicyEvaluation(bandit_feedback=bandit_feedback, ope_
↪estimators=[IPW()])
>>> estimated_policy_value = ope.estimate_policy_values(action_dist=action_dist)
>>> estimated_policy_value
{'ipw': 0.004553...}

# policy value improvement of BernoulliTS over the Random policy estimated by IPW
>>> estimated_policy_value_improvement = estimated_policy_value['ipw'] / bandit_
↪feedback['reward'].mean()
# our OPE procedure suggests that BernoulliTS improves Random by 19.81%
>>> print(estimated_policy_value_improvement)
1.198126...
```

estimate_intervals(*action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: Optional[numpy.ndarray] = None*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 100*, *random_state: Optional[int] = None*) → Dict[str, Dict[str, float]]

Estimate confidence intervals of estimated policy values using a nonparametric bootstrap procedure.

Parameters

- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$. When it is not given, model-dependent estimators such as DM and DR cannot be used.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=100*) – Number of resampling performed in the boot-

strap procedure.

- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **policy_value_interval_dict** – Dictionary containing confidence intervals of estimated policy value estimated using a nonparametric bootstrap procedure.

Return type Dict[str, Dict[str, float]]

estimate_policy_values(*action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: Optional[numpy.ndarray] = None*) → Dict[str, float]

Estimate policy value of an evaluation policy.

Parameters

- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$. When None is given, model-dependent estimators such as DM and DR cannot be used.

Returns **policy_value_dict** – Dictionary containing estimated policy values by OPE estimators.

Return type Dict[str, float]

evaluate_performance_of_estimators(*ground_truth_policy_value: float*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: Optional[numpy.ndarray] = None*, *metric: str = 'relative-ee'*) → Dict[str, float]

Evaluate estimation performances of OPE estimators.

Note: Evaluate the estimation performances of OPE estimators by relative estimation error (relative-EE) or squared error (SE):

$$\text{Relative-EE}(\hat{V}; \mathcal{D}) = \left| \frac{\hat{V}(\pi; \mathcal{D}) - V(\pi)}{V(\pi)} \right|,$$

$$\text{SE}(\hat{V}; \mathcal{D}) = \left(\hat{V}(\pi; \mathcal{D}) - V(\pi) \right)^2,$$

where $V(\pi)$ is the ground-truth policy value of the evalation policy π_e (often estimated using on-policy estimation). $\hat{V}(\pi; \mathcal{D})$ is an estimated policy value by an OPE estimator $\hat{V}$ and logged bandit feedback $\mathcal{D}$.

Parameters

- **ground_truth policy value** (*float*) – Ground_truth policy value of an evaluation policy, i.e., $V(\pi)$. With Open Bandit Dataset, in general, we use an on-policy estimate of the policy value as its ground-truth.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$. When it is not given, model-dependent estimators such as DM and DR cannot be used.
- **metric** (*str, default="relative-ee"*) – Evaluation metric to evaluate and compare the estimation performance of OPE estimators. Must be "relative-ee" or "se".

Returns **eval_metric_ope_dict** – Dictionary containing evaluation metric for evaluating the estimation performance of OPE estimators.

Return type Dict[str, float]

summarize_estimators_comparison(*ground_truth_policy_value: float*, *action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: Optional[numpy.ndarray] = None*, *metric: str = 'relative-ee'*) → pandas.core.frame.DataFrame

Summarize performance comparisons of OPE estimators.

Parameters

- **ground_truth policy value** (*float*) – Ground_truth policy value of an evaluation policy, i.e., $V(\pi)$. With Open Bandit Dataset, in general, we use an on-policy estimate of the policy value as ground-truth.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$. When it is not given, model-dependent estimators such as DM and DR cannot be used.
- **metric** (*str, default="relative-ee"*) – Evaluation metric to evaluate and compare the estimation performance of OPE estimators. Must be either "relative-ee" or "se".

Returns **eval_metric_ope_df** – Evaluation metric for evaluating the estimation performance of OPE estimators.

Return type DataFrame

summarize_off_policy_estimates(*action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: Optional[numpy.ndarray] = None*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 100*, *random_state: Optional[int] = None*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame]

Summarize policy values estimated by OPE estimators and their confidence intervals.

Parameters

- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$. When it is not given, model-dependent estimators such as DM and DR cannot be used.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=100*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **(policy_value_df, policy_value_interval_df)** – Estimated policy values and their confidence intervals by OPE estimators.

Return type Tuple[DataFrame, DataFrame]

visualize_off_policy_estimates(*action_dist: numpy.ndarray*, *estimated_rewards_by_reg_model: Optional[numpy.ndarray] = None*, *alpha: float = 0.05*, *is_relative: bool = False*, *n_bootstrap_samples: int = 100*, *random_state: Optional[int] = None*, *fig_dir: Optional[pathlib.Path] = None*, *fig_name: str = 'estimated_policy_value.png'*) → None

Visualize policy values estimated by OPE estimators.

Parameters

- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list)*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$.
- **estimated_rewards_by_reg_model** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Expected rewards for each round, action, and position estimated by a regression model, i.e., $\hat{q}(x_t, a_t)$. When it is not given, model-dependent estimators such as DM and DR cannot be used.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=100*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.
- **is_relative** (*bool, default=False,*) – If True, the method visualizes the estimated policy values of evaluation policy relative to the ground-truth policy value of behavior policy.
- **fig_dir** (*Path, default=None*) – Path to store the bar figure. If 'None' is given, the figure will not be saved.
- **fig_name** (*str, default="estimated_policy_value.png"*) – Name of the bar figure.

obp.ope.regression_model

Regression Model Class for Estimating Mean Reward Functions.

Classes

RegressionModel(base_model, n_actions, ...)	Machine learning model to estimate the mean reward function ($q(x, a) := \mathbb{E}[r|x, a]$).

class obp.ope.regression_model.**RegressionModel**(*base_model: sklearn.base.BaseEstimator*, *n_actions: int*, *len_list: int = 1*, *action_context: Optional[numpy.ndarray] = None*, *fitting_method: str = 'normal'*)

Bases: sklearn.base.BaseEstimator

Machine learning model to estimate the mean reward function ($q(x, a) := \mathbb{E}[r|x, a]$).

Note: Reward (or outcome) r must be either binary or continuous.

Parameters

- **base_model** (*BaseEstimator*) – A machine learning model used to estimate the mean reward function.
- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **action_context** (*array-like, shape (n_actions, dim_action_context), default=None*) – Context vector characterizing each action, vector representation of each action. If not given, then one-hot encoding of the action variable is automatically used.
- **fitting_method** (*str, default='normal'*) – Method to fit the regression model. Must be one of ['normal', 'iw', 'mrdr'] where 'iw' stands for importance weighting and 'mrdr' stands for more robust doubly robust.

References

Mehrdad Farajtabar, Yinlam Chow, and Mohammad Ghavamzadeh. "More Robust Doubly Robust Off-policy Evaluation.", 2018.

Yi Su, Maria Dimakopoulou, Akshay Krishnamurthy, and Miroslav Dudik. "Doubly Robust Off-Policy Evaluation with Shrinkage.", 2020.

Yusuke Narita, Shota Yasui, and Kohei Yata. "Off-policy Bandit and Reinforcement Learning.", 2020.

fit(*context: numpy.ndarray*, *action: numpy.ndarray*, *reward: numpy.ndarray*, *pscore: Optional[numpy.ndarray] = None*, *position: Optional[numpy.ndarray] = None*, *action_dist: Optional[numpy.ndarray] = None*) → None

Fit the regression model on given logged bandit feedback data.

Parameters

- **context** (*array-like, shape (n_rounds, dim_context)*) – Context vectors in each round, i.e., x_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **reward** (*array-like, shape (n_rounds,)*) – Observed rewards (or outcome) in each round, i.e., r_t.
- **pscore** (*array-like, shape (n_rounds,), default=None*) – Action choice probabilities (propensity score) of a behavior policy in the training logged bandit feedback. When None is given, the the behavior policy is assumed to be a uniform one.
- **position** (*array-like, shape (n_rounds,), default=None*) – Positions of each round in the given logged bandit feedback. If None is set, a regression model assumes that there is only one position. When *len_list* > 1, this position argument has to be set.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$. When either of 'iw' or 'mrdr' is used as the 'fitting_method' argument, then *action_dist* must be given.

fit_predict(*context: numpy.ndarray*, *action: numpy.ndarray*, *reward: numpy.ndarray*, *pscore: Optional[numpy.ndarray] = None*, *position: Optional[numpy.ndarray] = None*, *action_dist: Optional[numpy.ndarray] = None*, *n_folds: int = 1*, *random_state: Optional[int] = None*) → None

Fit the regression model on given logged bandit feedback data and predict the reward function of the same data.

Note: When *n_folds* is larger than 1, then the cross-fitting procedure is applied. See the reference for the details about the cross-fitting technique.

Parameters

- **context** (*array-like, shape (n_rounds, dim_context)*) – Context vectors in each round, i.e., x_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **reward** (*array-like, shape (n_rounds,)*) – Observed rewards (or outcome) in each round, i.e., r_t.
- **pscore** (*array-like, shape (n_rounds,), default=None*) – Action choice probabilities (propensity score) of a behavior policy in the training logged bandit feedback. When None is given, the the behavior policy is assumed to be a uniform one.
- **position** (*array-like, shape (n_rounds,), default=None*) – Positions of each round in the given logged bandit feedback. If None is set, a regression model assumes that there is only one position. When *len_list* > 1, this position argument has to be set.
- **action_dist** (*array-like, shape (n_rounds, n_actions, len_list), default=None*) – Action choice probabilities by the evaluation policy (can be deterministic), i.e., $\pi_e(a_t|x_t)$. When either of 'iw' or 'mrdr' is used as the 'fitting_method' argument, then *action_dist* must be given.
- **n_folds** (*int, default=1*) – Number of folds in the cross-fitting procedure. When 1 is given, the regression model is trained on the whole logged bandit feedback data.
- **random_state** (*int, default=None*) – *random_state* affects the ordering of the indices, which controls the randomness of each fold. See https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html for the details.

Returns **estimated_rewards_by_reg_model** – Estimated expected rewards for new data by the regression model.

Return type array-like, shape (n_rounds, n_actions, len_list)

get_params(*deep=True*)
Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type mapping of string to any

predict(*context: numpy.ndarray*) → numpy.ndarray
Predict the mean reward function.

Parameters **context** (*array-like, shape (n_rounds_of_new_data, dim_context)*) – Context vectors for new data.

Returns **estimated_rewards_by_reg_model** – Estimated expected rewards for new data by the regression model.

Return type array-like, shape (n_rounds_of_new_data, n_actions, len_list)

set_params(**params)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters **params (*dict*) – Estimator parameters.

Returns self – Estimator instance.

Return type object

6.8.2 policy module

obp.policy.base	Base Interfaces for Bandit Algorithms.
obp.policy.contextfree	Context-Free Bandit Algorithms.
obp.policy.linear	Contextual Linear Bandit Algorithms.
obp.policy.logistic	Contextual Logistic Bandit Algorithms.
obp.policy.offline	Offline Bandit Algorithms.

obp.policy.base

Base Interfaces for Bandit Algorithms.

Classes

BaseContextFreePolicy(n_actions, len_list, ...)	Base class for context-free bandit policies.
BaseContextualPolicy(dim, n_actions, ...)	Base class for contextual bandit policies.
BaseOfflinePolicyLearner(n_actions, len_list)	Base class for off-policy learners.

class obp.policy.base.**BaseContextFreePolicy**(*n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *random_state: Optional[int] = None*)

Bases: object

Base class for context-free bandit policies.

Parameters

- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.

initialize() → None
Initialize Parameters.

abstract select_action() → numpy.ndarray
Select a list of actions.

abstract update_params(*action: int*, *reward: float*) → None
Update policy parameters.

property policy_type
Type of the bandit policy.

class obp.policy.base.**BaseContextualPolicy**(*dim: int*, *n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *alpha_: float = 1.0*, *lambda_: float = 1.0*, *random_state: Optional[int] = None*)
Bases: object

Base class for contextual bandit policies.

Parameters

- **dim** (*int*) – Number of dimensions of context vectors.
- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **alpha_** (*float, default=1.*) – Prior parameter for the online logistic regression.
- **lambda_** (*float, default=1.*) – Regularization hyperparameter for the online logistic regression.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.

initialize() → None
Initialize policy parameters.

abstract select_action(*context: numpy.ndarray*) → numpy.ndarray
Select a list of actions.

abstract update_params(*action: float*, *reward: float*, *context: numpy.ndarray*) → None
Update policy parameters.

property policy_type
Type of the bandit policy.

class obp.policy.base.**BaseOfflinePolicyLearner**(*n_actions: int*, *len_list: int = 1*)
Bases: object

Base class for off-policy learners.

Parameters

- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.

abstract fit() → None
Fits an offline bandit policy using the given logged bandit feedback data.

abstract predict(*context: numpy.ndarray*) → numpy.ndarray
Predict best action for new data.

Parameters **context** (*array-like, shape (n_rounds_of_new_data, dim_context)*) – Context vectors for new data.

Returns **action** – Action choices by a policy trained by calling the *fit* method.

Return type array-like, shape (n_rounds_of_new_data, n_actions, len_list)

property policy_type
Type of the bandit policy.

obp.policy.contextfree

Context-Free Bandit Algorithms.

Classes

BernoulliTS(n_actions, len_list, batch_size, ...)	Bernoulli Thompson Sampling Policy
EpsilonGreedy(n_actions, len_list, ...)	Epsilon Greedy policy.
Random(n_actions, len_list, batch_size, ...)	Random policy

class obp.policy.contextfree.**BernoulliTS**(*n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *random_state: Optional[int] = None*, *alpha: Optional[numpy.ndarray] = None*, *beta: Optional[numpy.ndarray] = None*, *is_zozotown_prior: bool = False*, *campaign: Optional[str] = None*, *policy_name: str = 'bts'*)

Bases: *obp.policy.base.BaseContextFreePolicy*

Bernoulli Thompson Sampling Policy

Parameters

- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.
- **alpha** (*array-like, shape (n_actions,), default=None*) – Prior parameter vector for Beta distributions.
- **beta** (*array-like, shape (n_actions,), default=None*) – Prior parameter vector for Beta distributions.
- **is_zozotown_prior** (*bool, default=False*) – Whether to use hyperparameters for the beta distribution used at the start of the data collection period in ZOZOTOWN.
- **campaign** (*str, default=None*) – One of the three possible campaigns considered in ZOZOTOWN, "all", "men", and "women".
- **policy_name** (*str, default='bts'*) – Name of bandit policy.

compute_batch_action_dist(*n_rounds: int = 1*, *n_sim: int = 100000*) → numpy.ndarray
Compute the distribution over actions by Monte Carlo simulation.

Parameters

- **n_rounds** (*int, default=1*) – Number of rounds in the distribution over actions. (the size of the first axis of *action_dist*)
- **n_sim** (*int, default=100000*) – Number of simulations in the Monte Carlo simulation to compute the distribution over actions.

Returns **action_dist** – Probability estimates of each arm being the best one for each sample, action, and position.

Return type array-like, shape (n_rounds, n_actions, len_list)

initialize() → None
Initialize Parameters.

select_action() → numpy.ndarray
Select a list of actions.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.

property policy_type
Type of the bandit policy.

class obp.policy.contextfree.**EpsilonGreedy**(*n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *random_state: Optional[int] = None*, *epsilon: float = 1.0*)
Bases: *obp.policy.base.BaseContextFreePolicy*

Epsilon Greedy policy.

Parameters

- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.
- **epsilon** (*float, default=1.*) – Exploration hyperparameter that must take value in the range of [0., 1.].
- **policy_name** (*str, default=f'egreedy_{epsilon}'.*) – Name of bandit policy.

initialize() → None
Initialize Parameters.

select_action() → numpy.ndarray
Select a list of actions.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.

property policy_type
Type of the bandit policy.

class obp.policy.contextfree.**Random**(*n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *random_state: Optional[int] = None*, *epsilon: float = 1.0*, *policy_name: str = 'random'*)
Bases: *obp.policy.contextfree.EpsilonGreedy*

Random policy

Parameters

- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.
- **epsilon** (*float, default=1.*) – Exploration hyperparameter that must take value in the range of [0., 1.].
- **policy_name** (*str, default='random'.*) – Name of bandit policy.

compute_batch_action_dist(*n_rounds: int = 1*, *n_sim: int = 100000*) → numpy.ndarray
Compute the distribution over actions by Monte Carlo simulation.

Parameters **n_rounds** (*int, default=1*) – Number of rounds in the distribution over actions. (the size of the first axis of *action_dist*)

Returns **action_dist** – Probability estimates of each arm being the best one for each sample, action, and position.

Return type array-like, shape (n_rounds, n_actions, len_list)

initialize() → None
Initialize Parameters.

select_action() → numpy.ndarray
Select a list of actions.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.

property policy_type
Type of the bandit policy.

obp.policy.linear

Contextual Linear Bandit Algorithms.

Classes

LinEpsilonGreedy(dim, n_actions, len_list, ...)	Linear Epsilon Greedy.
LinTS(dim, n_actions, len_list, batch_size, ...)	Linear Thompson Sampling.
LinUCB(dim, n_actions, len_list, batch_size, ...)	Linear Upper Confidence Bound.

class obp.policy.linear.**LinEpsilonGreedy**(*dim: int*, *n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *alpha_: float = 1.0*, *lambda_: float = 1.0*, *random_state: Optional[int] = None*, *epsilon: float = 0.0*)

Bases: *obp.policy.base.BaseContextualPolicy*

Linear Epsilon Greedy.

Parameters

- **dim** (*int*) – Number of dimensions of context vectors.
- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **n_trial** (*int, default=0*) – Current number of trials in a bandit simulation.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.
- **epsilon** (*float, default=0.*) – Exploration hyperparameter that must take value in the range of [0., 1.].

References

L. Li, W. Chu, J. Langford, and E. Schapire. A contextual-bandit approach to personalized news article recommendation. In Proceedings of the 19th International Conference on World Wide Web, pp. 661–670. ACM, 2010.

initialize() → None
Initialize policy parameters.

select_action(*context: numpy.ndarray*) → numpy.ndarray
Select action for new data.

Parameters **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*, *context: numpy.ndarray*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.

- **reward** (*float*) – Observed reward for the chosen action and position.
- **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

property policy_type
Type of the bandit policy.

class obp.policy.linear.**LinTS**(*dim: int*, *n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *alpha_: float = 1.0*, *lambda_: float = 1.0*, *random_state: Optional[int] = None*)
Bases: *obp.policy.base.BaseContextualPolicy*

Linear Thompson Sampling.

Parameters

- **dim** (*int*) – Number of dimensions of context vectors.
- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **alpha_** (*float, default=1.*) – Prior parameter for the online logistic regression.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.

initialize() → None
Initialize policy parameters.

select_action(*context: numpy.ndarray*) → numpy.ndarray
Select action for new data.

Parameters **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*, *context: numpy.ndarray*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.
- **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

property policy_type
Type of the bandit policy.

class obp.policy.linear.**LinUCB**(*dim: int*, *n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *alpha_: float = 1.0*, *lambda_: float = 1.0*, *random_state: Optional[int] = None*, *epsilon: float = 0.0*)
Bases: *obp.policy.base.BaseContextualPolicy*

Linear Upper Confidence Bound.

Parameters

- **dim** (*int*) – Number of dimensions of context vectors.
- **n_actions** (*int*) – Number of actions.

- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.
- **epsilon** (*float, default=0.*) – Exploration hyperparameter that must take value in the range of [0., 1.].

References

L. Li, W. Chu, J. Langford, and E. Schapire. A contextual-bandit approach to personalized news article recommendation. In Proceedings of the 19th International Conference on World Wide Web, pp. 661–670. ACM, 2010.

initialize() → None
Initialize policy parameters.

select_action(*context: numpy.ndarray*) → numpy.ndarray
Select action for new data.

Parameters **context** (*array*) – Observed context vector.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*, *context: numpy.ndarray*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.
- **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

property policy_type
Type of the bandit policy.

obp.policy.logistic

Contextual Logistic Bandit Algorithms.

Classes

LogisticEpsilonGreedy(dim, n_actions, ...)	Logistic Epsilon Greedy.
LogisticTS(dim, n_actions, len_list, ...)	Logistic Thompson Sampling.
LogisticUCB(dim, n_actions, len_list, ...)	Logistic Upper Confidence Bound.
MiniBatchLogisticRegression(lambda_, alpha, ...)	MiniBatch Online Logistic Regression Model.

class obp.policy.logistic.**LogisticEpsilonGreedy**(*dim: int*, *n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *alpha_: float = 1.0*, *lambda_: float = 1.0*, *random_state: Optional[int] = None*, *epsilon: float = 0.0*)

Bases: *obp.policy.base.BaseContextualPolicy*

Logistic Epsilon Greedy.

Parameters

- **dim** (*int*) – Number of dimensions of context vectors.
- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **alpha_** (*float, default=1.*) – Prior parameter for the online logistic regression.
- **lambda_** (*float, default=1.*) – Regularization hyperparameter for the online logistic regression.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.
- **epsilon** (*float, default=0.*) – Exploration hyperparameter that must take value in the range of [0., 1.].

initialize() → None
Initialize policy parameters.

select_action(*context: numpy.ndarray*) → numpy.ndarray
Select action for new data.

Parameters **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*, *context: numpy.ndarray*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.
- **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

property policy_type
Type of the bandit policy.

class obp.policy.logistic.**LogisticTS**(*dim: int*, *n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *alpha_: float = 1.0*, *lambda_: float = 1.0*, *random_state: Optional[int] = None*, *policy_name: str = 'logistic_ts'*)

Bases: *obp.policy.base.BaseContextualPolicy*

Logistic Thompson Sampling.

Parameters

- **dim** (*int*) – Number of dimensions of context vectors.

- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **alpha_** (*float, default=1.*) – Prior parameter for the online logistic regression.
- **lambda_** (*float, default=1.*) – Regularization hyperparameter for the online logistic regression.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.

References

Olivier Chapelle and Lihong Li. "An empirical evaluation of thompson sampling," 2011.

initialize() → None
Initialize policy parameters.

select_action(*context: numpy.ndarray*) → numpy.ndarray
Select action for new data.

Parameters **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*, *context: numpy.ndarray*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.
- **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

property policy_type
Type of the bandit policy.

class obp.policy.logistic.**LogisticUCB**(*dim: int*, *n_actions: int*, *len_list: int = 1*, *batch_size: int = 1*, *alpha_: float = 1.0*, *lambda_: float = 1.0*, *random_state: Optional[int] = None*, *epsilon: float = 0.0*)
Bases: *obp.policy.base.BaseContextualPolicy*

Logistic Upper Confidence Bound.

Parameters

- **dim** (*int*) – Number of dimensions of context vectors.
- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **batch_size** (*int, default=1*) – Number of samples used in a batch parameter update.
- **alpha_** (*float, default=1.*) – Prior parameter for the online logistic regression.

- **lambda_** (*float, default=1.*) – Regularization hyperparameter for the online logistic regression.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.
- **epsilon** (*float, default=0.*) – Exploration hyperparameter that must take value in the range of [0., 1.].

References

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. "A Contextual-bandit Approach to Personalized News Article Recommendation," 2010.

initialize() → None
Initialize policy parameters.

select_action(*context: numpy.ndarray*) → numpy.ndarray
Select action for new data.

Parameters **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

Returns **selected_actions** – List of selected actions.

Return type array-like, shape (len_list,)

update_params(*action: int*, *reward: float*, *context: numpy.ndarray*) → None
Update policy parameters.

Parameters

- **action** (*int*) – Selected action by the policy.
- **reward** (*float*) – Observed reward for the chosen action and position.
- **context** (*array-like, shape (1, dim_context)*) – Observed context vector.

property policy_type
Type of the bandit policy.

class obp.policy.logistic.**MiniBatchLogisticRegression**(*lambda_: float*, *alpha: float*, *dim: int*, *random_state: Optional[int] = None*)

Bases: object

MiniBatch Online Logistic Regression Model.

fit(*X: numpy.ndarray*, *y: numpy.ndarray*)
Update coefficient vector by the mini-batch data.

grad(*w: numpy.ndarray*, **args*) → numpy.ndarray
Calculate gradient.

loss(*w: numpy.ndarray*, **args*) → float
Calculate loss function.

predict_proba(*X: numpy.ndarray*) → numpy.ndarray
Predict extected probability by the expected coefficient.

predict_proba_with_sampling(*X: numpy.ndarray*) → numpy.ndarray
Predict extected probability by the sampled coefficient.

sample() → numpy.ndarray
Sample coefficient vector from the prior distribution.

sd() → numpy.ndarray
Standard deviation for the coefficient vector.

obp.policy.offline

Offline Bandit Algorithms.

Classes

IPWLearner(n_actions, len_list, base_classifier)	Off-policy learner with Inverse Probability Weighting.

class obp.policy.offline.**IPWLearner**(*n_actions: int*, *len_list: int = 1*, *base_classifier: Optional[sklearn.base.ClassifierMixin] = None*)
Bases: *obp.policy.base.BaseOfflinePolicyLearner*

Off-policy learner with Inverse Probability Weighting.

Parameters

- **n_actions** (*int*) – Number of actions.
- **len_list** (*int, default=1*) – Length of a list of actions recommended in each impression. When Open Bandit Dataset is used, 3 should be set.
- **base_classifier** (*ClassifierMixin*) – Machine learning classifier used to train an offline decision making policy.

References

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

Damien Lefortier, Adith Swaminathan, Xiaotao Gu, Thorsten Joachims, and Maarten de Rijke. "Large-scale Validation of Counterfactual Learning Methods: A Test-Bed.", 2016.

fit(*context: numpy.ndarray*, *action: numpy.ndarray*, *reward: numpy.ndarray*, *pscore: Optional[numpy.ndarray] = None*, *position: Optional[numpy.ndarray] = None*) → None
Fits an offline bandit policy using the given logged bandit feedback data.

Note: This *fit* method trains a deterministic policy $\pi : \mathcal{X} \to \mathcal{A}$ via a cost-sensitive classification reduction as follows:

$$\begin{aligned}\hat{\pi} &\in \arg\max_{\pi \in \Pi} \hat{V}_{\mathrm{IPW}}(\pi; \mathcal{D}) \\ &= \arg\max_{\pi \in \Pi} \mathbb{E}_{\mathcal{D}} \left[\frac{\mathbb{I}\{\pi(x_i) = a_i\}}{\pi_b(a_i|x_i)} r_i \right] \\ &= \arg\min_{\pi \in \Pi} \mathbb{E}_{\mathcal{D}} \left[\frac{r_i}{\pi_b(a_i|x_i)} \mathbb{I}\{\pi(x_i) \neq a_i\} \right],\end{aligned}$$

where $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over observations in $\mathcal{D}$. See the reference for the details.

Parameters

- **context** (*array-like, shape (n_rounds, dim_context)*) – Context vectors in each round, i.e., x_t.

- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **reward** (*array-like, shape (n_rounds,)*) – Observed rewards (or outcome) in each round, i.e., r_t.
- **pscore** (*array-like, shape (n_rounds,), default=None*) – Action choice probabilities by a behavior policy (propensity scores), i.e., $\pi_b(a_t|x_t)$.
- **position** (*array-like, shape (n_rounds,), default=None*) – Positions of each round in the given logged bandit feedback. If None is given, a learner assumes that there is only one position. When *len_list* > 1, position has to be set.

predict (*context: numpy.ndarray*) → numpy.ndarray
Predict best actions for new data.

Note: Action set predicted by this *predict* method can contain duplicate items. If you want a non-repetitive action set, then please use the *sample_action* method.

Parameters **context** (*array-like, shape (n_rounds_of_new_data, dim_context)*) – Context vectors for new data.

Returns **action_dist** – Action choices by a classifier, which can contain duplicate items. If you want a non-repetitive action set, please use the *sample_action* method.

Return type array-like, shape (n_rounds_of_new_data, n_actions, len_list)

predict_proba (*context: numpy.ndarray*, *tau: Union[int, float] = 1.0*) → numpy.ndarray
Obtains action choice probabilities for new data based on scores predicted by a classifier.

Note: This *predict_proba* method obtains action choice probabilities for new data $x \in \mathcal{X}$ by first computing non-negative scores for all possible candidate actions $a \in \mathcal{A}$ (where $\mathcal{A}$ is an action set), and using a Plackett-Luce ranking model as follows:

$$P(A = a|x) = \frac{\exp(f(x,a)/\tau)}{\sum_{a' \in \mathcal{A}} \exp(f(x,a')/\tau)},$$

where A is a random variable representing an action, and τ is a temperature hyperparameter. $f : \mathcal{X} \times \mathcal{A} \to \mathbb{R}_+$ is a scoring function which is now implemented in the *predict_score* method.

Note that this method can be used only when `len_list=1`, please use the `sample_action` method otherwise.

Parameters

- **context** (*array-like, shape (n_rounds_of_new_data, dim_context)*) – Context vectors for new data.
- **tau** (*int or float, default=1.0*) – A temperature parameter, controlling the randomness of the action choice. As $\tau \to \infty$, the algorithm will select arms uniformly at random.

Returns **choice_prob** – Action choice probabilities obtained by a trained classifier.

Return type array-like, shape (n_rounds_of_new_data, n_actions, len_list)

predict_score (*context: numpy.ndarray*) → numpy.ndarray
Predict non-negative scores for all possible products of action and position.

Parameters **context** (*array-like, shape (n_rounds_of_new_data, dim_context)*) – Context vectors for new data.

Returns **score_predicted** – Scores for all possible pairs of action and position predicted by a classifier.

Return type array-like, shape (n_rounds_of_new_data, n_actions, len_list)

sample_action(*context: numpy.ndarray*, *tau: Union[int, float] = 1.0*, *random_state: Optional[int] = None*) → numpy.ndarray

Sample (non-repetitive) actions based on scores predicted by a classifier.

Note: This *sample_action* method samples a **non-repetitive** set of actions for new data $x \in \mathcal{X}$ by first computing non-negative scores for all possible candidate products of action and position $(a, k) \in \mathcal{A} \times \mathcal{K}$ (where $\mathcal{A}$ is an action set and $\mathcal{K}$ is a position set), and using softmax function as follows:

$$P(A_1 = a_1|x) = \frac{\exp(f(x, a_1, 1)/\tau)}{\sum_{a' \in \mathcal{A}} \exp(f(x, a', 1)/\tau)},$$
$$P(A_2 = a_2|A_1 = a_1, x) = \frac{\exp(f(x, a_2, 2)/\tau)}{\sum_{a' \in \mathcal{A} \setminus \{a_1\}} \exp(f(x, a', 2)/\tau)}, \ldots$$

where A_k is a random variable representing an action at a position k. τ is a temperature hyperparameter. $f : \mathcal{X} \times \mathcal{A} \times \mathcal{K} \to \mathbb{R}_+$ is a scoring function which is now implemented in the *predict_score* method.

Parameters

- **context** (*array-like, shape (n_rounds_of_new_data, dim_context)*) – Context vectors for new data.
- **tau** (*int or float, default=1.0*) – A temperature parameter, controlling the randomness of the action choice. As $\tau \to \infty$, the algorithm will select arms uniformly at random.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.

Returns **action** – Action sampled by a trained classifier.

Return type array-like, shape (n_rounds_of_new_data, n_actions, len_list)

property policy_type

Type of the bandit policy.

6.8.3 dataset module

obp.dataset.base	Abstract Base Class for Logged Bandit Feedback.
obp.dataset.real	Dataset Class for Real-World Logged Bandit Feedback.
obp.dataset.synthetic	Class for Generating Synthetic Logged Bandit Feedback.
obp.dataset.multiclass	Class for Multi-Class Classification to Bandit Reduction.

obp.dataset.base

Abstract Base Class for Logged Bandit Feedback.

Classes

BaseRealBanditDataset()	Base Class for Real-World Bandit Dataset.
BaseSyntheticBanditDataset()	Base Class for Synthetic Bandit Dataset.

class obp.dataset.base.**BaseRealBanditDataset**

Bases: object

Base Class for Real-World Bandit Dataset.

abstract load_raw_data() → None
Load raw dataset.

abstract obtain_batch_bandit_feedback() → None
Obtain batch logged bandit feedback.

abstract pre_process() → None
Preprocess raw dataset.

class obp.dataset.base.**BaseSyntheticBanditDataset**

Bases: object

Base Class for Synthetic Bandit Dataset.

abstract obtain_batch_bandit_feedback() → None
Obtain batch logged bandit feedback.

obp.dataset.real

Dataset Class for Real-World Logged Bandit Feedback.

Classes

OpenBanditDataset(behavior_policy, campaign, ...)	Class for loading and preprocessing Open Bandit Dataset.

class obp.dataset.real.**OpenBanditDataset**(*behavior_policy: str*, *campaign: str*, *data_path: pathlib.Path = PosixPath('obd')*, *dataset_name: str = 'obd'*)

Bases: *obp.dataset.base.BaseRealBanditDataset*

Class for loading and preprocessing Open Bandit Dataset.

Note: Users are free to implement their own feature engineering by overriding the *pre_process* method.

Parameters

- **behavior_policy** (*str*) – Name of the behavior policy that generated the logged bandit feedback data. Must be either 'random' or 'bts'.

- **campaign** (*str*) – One of the three possible campaigns considered in ZOZOTOWN, “all”, “men”, and “women”.
- **data_path** (*Path, default=Path('./obd')*) – Path that stores Open Bandit Dataset.
- **dataset_name** (*str, default='obd'*) – Name of the dataset.

References

Yuta Saito, Shunsuke Aihara, Megumi Matsutani, Yusuke Narita. “Large-scale Open Dataset, Pipeline, and Benchmark for Bandit Algorithms.”, 2020.

classmethod calc_on_policy_policy_value_estimate(*behavior_policy: str*, *campaign: str*, *data_path: pathlib.Path = PosixPath('obd')*, *test_size: float = 0.3*, *is_timeseries_split: bool = False*) → float

Calculate on-policy policy value estimate (used as a ground-truth policy value).

Parameters

- **behavior_policy** (*str*) – Name of the behavior policy that generated the log data. Must be either ‘random’ or ‘bts’.
- **campaign** (*str*) – One of the three possible campaigns considered in ZOZOTOWN (i.e., “all”, “men”, and “women”).
- **data_path** (*Path, default=Path('./obd')*) – Path that stores Open Bandit Dataset.
- **test_size** (*float, default=0.3*) – If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split.
- **is_timeseries_split** (*bool, default=False*) – If true, split the original logged bandit feedback data by time series.

Returns **on_policy_policy_value_estimate** – Policy value of the behavior policy estimated by on-policy estimation, i.e., $\mathbb{E}_{\mathcal{D}}[r_t]$. where $\mathbb{E}_{\mathcal{D}}[\cdot]$ is the empirical average over T observations in $\mathcal{D}$. This parameter is used as a ground-truth policy value in the evaluation of OPE estimators.

Return type float

load_raw_data() → None

Load raw open bandit dataset.

obtain_batch_bandit_feedback(*test_size: float = 0.3*, *is_timeseries_split: bool = False*) → Dict[str, Union[int, numpy.ndarray]]

Obtain batch logged bandit feedback.

Parameters

- **test_size** (*float, default=0.3*) – If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the evaluation split.
- **is_timeseries_split** (*bool, default=False*) – If true, split the original logged bandit feedback data by time series.

Returns **bandit_feedback** – Batch logged bandit feedback collected by a behavior policy.

Return type BanditFeedback

pre_process() → None
Preprocess raw open bandit dataset.

> **Note:** This is the default feature engineering and please override this method to implement your own preprocessing. see https://github.com/st-tech/zr-obp/blob/master/examples/examples_with_obd/custom_dataset.py for example.

sample_bootstrap_bandit_feedback(*test_size: float = 0.3*, *is_timeseries_split: bool = False*, *random_state: Optional[int] = None*) → Dict[str, Union[int, numpy.ndarray]]
Obtain bootstrap logged bandit feedback.

Parameters

- **test_size** (*float, default=0.3*) – If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the evaluation split.
- **is_timeseries_split** (*bool, default=False*) – If true, split the original logged bandit feedback data by time series.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **bandit_feedback** – Logged bandit feedback sampled independently from the original data with replacement.

Return type BanditFeedback

property dim_context
Dimensions of context vectors.

property len_list
Length of recommendation lists.

property n_actions
Number of actions.

property n_rounds
Total number of rounds contained in the logged bandit dataset.

obp.dataset.synthetic

Class for Generating Synthetic Logged Bandit Feedback.

Functions

linear_behavior_policy(context, action_context)	Linear contextual behavior policy for synthetic bandit datasets.
linear_reward_function(context, action_context)	Linear mean reward function for synthetic bandit datasets.
logistic_reward_function(context, action_context)	Logistic mean reward function for synthetic bandit datasets.

Classes

SyntheticBanditDataset(n_actions, ...)	Class for generating synthetic bandit dataset.

class obp.dataset.synthetic.**SyntheticBanditDataset**(*n_actions: int*, *dim_context: int = 1*, *reward_type: str = 'binary'*, *reward_function: Optional[Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]] = None*, *behavior_policy_function: Optional[Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]] = None*, *random_state: Optional[int] = None*, *dataset_name: str = 'synthetic_bandit_dataset'*)

Bases: *obp.dataset.base.BaseSyntheticBanditDataset*

Class for generating synthetic bandit dataset.

Note: By calling the *obtain_batch_bandit_feedback* method several times, we have different bandit samples with the same setting. This can be used to estimate confidence intervals of the performances of OPE estimators.

If None is set as *behavior_policy_function*, the synthetic data will be context-free bandit feedback.

Parameters

- **n_actions** (*int*) – Number of actions.
- **dim_context** (*int, default=1*) – Number of dimensions of context vectors.
- **reward_type** (*str, default='binary'*) – Type of reward variable, must be either 'binary' or 'continuous'. When 'binary' is given, rewards are sampled from the Bernoulli distribution. When 'continuous' is given, rewards are sampled from the truncated Normal distribution with *scale=1*.
- **reward_function** (*Callable[[np.ndarray, np.ndarray], np.ndarray]], default=None*) – Function generating expected reward with context and action context vectors, i.e., $\mu : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$. If None is set, context **independent** expected reward for each action will be sampled from the uniform distribution automatically.
- **behavior_policy_function** (*Callable[[np.ndarray, np.ndarray], np.ndarray], default=None*) – Function generating probability distribution over action space, i.e., $\pi : \mathcal{X} \to \Delta(\mathcal{A})$. If None is set, context **independent** uniform distribution will be used (uniform random behavior policy).
- **random_state** (*int, default=None*) – Controls the random seed in sampling synthetic bandit dataset.
- **dataset_name** (*str, default='synthetic_bandit_dataset'*) – Name of the dataset.

Examples

```
>>> import numpy as np
>>> from obp.dataset import (
    SyntheticBanditDataset,
    linear_reward_function,
    linear_behavior_policy
)

# generate synthetic contextual bandit feedback with 10 actions.
>>> dataset = SyntheticBanditDataset(
        n_actions=10,
        dim_context=5,
        reward_function=logistic_reward_function,
        behavior_policy=linear_behavior_policy,
        random_state=12345
    )
>>> bandit_feedback = dataset.obtain_batch_bandit_feedback(n_rounds=100000)
>>> bandit_feedback
{
    'n_rounds': 100000,
    'n_actions': 10,
    'context': array([[-0.20470766,  0.47894334, -0.51943872, -0.5557303 ,  1.
↪96578057],
            [ 1.39340583,  0.09290788,  0.28174615,  0.76902257,  1.24643474],
            [ 1.00718936, -1.29622111,  0.27499163,  0.22891288,  1.35291684],
            ...,
            [ 1.36946256,  0.58727761, -0.69296769, -0.27519988, -2.10289159],
            [-0.27428715,  0.52635353,  1.02572168, -0.18486381,  0.72464834],
            [-1.25579833, -1.42455203, -0.26361242,  0.27928604,  1.21015571]]),
    'action_context': array([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]),
    'action': array([7, 4, 0, ..., 7, 9, 6]),
    'position': array([0, 0, 0, ..., 0, 0, 0]),
    'reward': array([0, 1, 1, ..., 0, 1, 0]),
    'expected_reward': array([[0.80210203, 0.73828559, 0.83199558, ..., 0.
↪81190503, 0.70617705,
            0.68985306],
            [0.94119582, 0.93473317, 0.91345213, ..., 0.94140688, 0.93152449,
            0.90132868],
            [0.87248862, 0.67974991, 0.66965669, ..., 0.79229752, 0.82712978,
            0.74923536],
            ...,
            [0.64856003, 0.38145901, 0.84476094, ..., 0.40962057, 0.77114661,
            0.65752798],
            [0.73208527, 0.82012699, 0.78161352, ..., 0.72361416, 0.8652249 ,
            0.82571751],
            [0.40348366, 0.24485417, 0.24037926, ..., 0.49613133, 0.30714854,
            0.5527749 ]]),
```

(continues on next page)

(continued from previous page)

```
    'pscore': array([0.05423855, 0.10339675, 0.09756788, ..., 0.05423855, 0.
↪07250876,
            0.14065505])
}
```

obtain_batch_bandit_feedback(*n_rounds: int*) → Dict[str, Union[int, numpy.ndarray]]
Obtain batch logged bandit feedback.

Parameters **n_rounds** (*int*) – Number of rounds for synthetic bandit feedback data.

Returns **bandit_feedback** – Generated synthetic bandit feedback dataset.

Return type BanditFeedback

sample_contextfree_expected_reward() → numpy.ndarray
Sample expected reward for each action from the uniform distribution.

property len_list
Length of recommendation lists.

obp.dataset.synthetic.**linear_behavior_policy**(*context: numpy.ndarray*, *action_context: numpy.ndarray*, *random_state: Optional[int] = None*) → numpy.ndarray
Linear contextual behavior policy for synthetic bandit datasets.

Parameters

- **context** (*array-like, shape (n_rounds, dim_context)*) – Context vectors characterizing each round (such as user information).
- **action_context** (*array-like, shape (n_actions, dim_action_context)*) – Vector representation for each action.
- **random_state** (*int, default=None*) – Controls the random seed in sampling dataset.

Returns **behavior_policy** – Action choice probabilities given context (x), i.e., $\pi : \mathcal{X} \to \Delta(\mathcal{A})$.

Return type array-like, shape (n_rounds, n_actions)

obp.dataset.synthetic.**linear_reward_function**(*context: numpy.ndarray*, *action_context: numpy.ndarray*, *random_state: Optional[int] = None*) → numpy.ndarray
Linear mean reward function for synthetic bandit datasets.

Parameters

- **context** (*array-like, shape (n_rounds, dim_context)*) – Context vectors characterizing each round (such as user information).
- **action_context** (*array-like, shape (n_actions, dim_action_context)*) – Vector representation for each action.
- **random_state** (*int, default=None*) – Controls the random seed in sampling dataset.

Returns **expected_reward** – Expected reward given context (x) and action (a), i.e., $q(x, a) := \mathbb{E}[r|x, a]$.

Return type array-like, shape (n_rounds, n_actions)

obp.dataset.synthetic.**logistic_reward_function**(*context: numpy.ndarray*, *action_context: numpy.ndarray*, *random_state: Optional[int] = None*) → numpy.ndarray
Logistic mean reward function for synthetic bandit datasets.

Parameters

- **context** (*array-like, shape (n_rounds, dim_context)*) – Context vectors characterizing each round (such as user information).
- **action_context** (*array-like, shape (n_actions, dim_action_context)*) – Vector representation for each action.
- **random_state** (*int, default=None*) – Controls the random seed in sampling dataset.

Returns **expected_reward** – Expected reward given context (x) and action (a), i.e., $q(x, a) := \mathbb{E}[r|x, a]$.

Return type array-like, shape (n_rounds, n_actions)

obp.dataset.multiclass

Class for Multi-Class Classification to Bandit Reduction.

Classes

MultiClassToBanditReduction(X, y, ...)	Class for handling multi-class classification data as logged bandit feedback data.

class obp.dataset.multiclass.**MultiClassToBanditReduction**(*X: numpy.ndarray*, *y: numpy.ndarray*, *base_classifier_b: sklearn.base.ClassifierMixin*, *alpha_b: float = 0.8*, *dataset_name: Optional[str] = None*)

Bases: *obp.dataset.base.BaseSyntheticBanditDataset*

Class for handling multi-class classification data as logged bandit feedback data.

Note: A machine learning classifier such as logistic regression is used to construct behavior and evaluation policies as follows.

1. Split the original data into training ($\mathcal{D}_{\mathrm{tr}}$) and evaluation ($\mathcal{D}_{\mathrm{ev}}$) sets.
2. Train classifiers on $\mathcal{D}_{\mathrm{tr}}$ and obtain base deterministic policies $\pi_{\mathrm{det},b}$ and $\pi_{\mathrm{det},e}$.
3. Construct behavior (π_b) and evaluation (π_e) policies based on $\pi_{\mathrm{det},b}$ and $\pi_{\mathrm{det},e}$ as

$$\pi_b(a|x) := \alpha_b \cdot \pi_{\mathrm{det},b}(a|x) + (1.0 - \alpha_b) \cdot \pi_u(a|x)$$

$$\pi_e(a|x) := \alpha_e \cdot \pi_{\mathrm{det},e}(a|x) + (1.0 - \alpha_e) \cdot \pi_u(a|x)$$

where π_u is a uniform random policy and α_b and α_e are set by the user.

4. Measure the accuracy of the evaluation policy on $\mathcal{D}_{\mathrm{ev}}$ with its fully observed rewards and use it as the evaluation policy's ground truth policy value.
5. Using $\mathcal{D}_{\mathrm{ev}}$, an estimator $\hat{V}$ estimates the policy value of the evaluation policy, i.e.,

$$V(\pi_e) \approx \hat{V}(\pi_e; \mathcal{D}_{\mathrm{ev}})$$

6. Evaluate the estimation performance of $\hat{V}$ by comparing its estimate with the ground-truth policy value.

Parameters

- **X** (*array-like, shape (n_rounds,n_features)*) – Training vector of the original multi-class classification data, where n_rounds is the number of samples and n_features is the number of features.
- **y** (*array-like, shape (n_rounds,)*) – Target vector (relative to X) of the original multi-class classification data.
- **base_classifier_b** (*ClassifierMixin*) – Machine learning classifier used to construct a behavior policy.
- **alpha_b** (*float, default=0.9*) – Ration of a uniform random policy when constructing a **behavior** policy. Must be in the [0, 1) interval to make the behavior policy a stochastic one.
- **dataset_name** (*str, default=None*) – Name of the dataset.

Examples

```
# evaluate the estimation performance of IPW using the `digits` data in sklearn
>>> import numpy as np
>>> from sklearn.datasets import load_digits
>>> from sklearn.linear_model import LogisticRegression
# import open bandit pipeline (obp)
>>> from obp.dataset import MultiClassToBanditReduction
>>> from obp.ope import OffPolicyEvaluation, InverseProbabilityWeighting as IPW

# load raw digits data
>>> X, y = load_digits(return_X_y=True)
# convert the raw classification data into the logged bandit dataset
>>> dataset = MultiClassToBanditReduction(
    X=X,
    y=y,
    base_classifier_b=LogisticRegression(random_state=12345),
    alpha_b=0.8,
    dataset_name="digits",
)
# split the original data into the training and evaluation sets
>>> dataset.split_train_eval(eval_size=0.7, random_state=12345)
# obtain logged bandit feedback generated by behavior policy
>>> bandit_feedback = dataset.obtain_batch_bandit_feedback(random_state=12345)
>>> bandit_feedback
{
    'n_actions': 10,
    'n_rounds': 1258,
    'context': array([[ 0.,  0.,  0., ..., 16.,  1.,  0.],
            [ 0.,  0.,  7., ..., 16.,  3.,  0.],
            [ 0.,  0., 12., ...,  8.,  0.,  0.],
            ...,
```

(continues on next page)

(continued from previous page)

```
            [ 0.,  1., 13., ...,  8., 11.,  1.],
            [ 0.,  0., 15., ...,  0.,  0.,  0.],
            [ 0.,  0.,  4., ..., 15.,  3.,  0.]]),
    'action': array([6, 8, 5, ..., 2, 5, 9]),
    'reward': array([1., 1., 1., ..., 1., 1., 1.]),
    'position': array([0, 0, 0, ..., 0, 0, 0]),
    'pscore': array([0.82, 0.82, 0.82, ..., 0.82, 0.82, 0.82])
}

# obtain action choice probabilities by an evaluation policy and its ground-truth␣
↪policy value
>>> action_dist = dataset.obtain_action_dist_by_eval_policy(
    base_classifier_e=LogisticRegression(C=100, random_state=12345),
    alpha_e=0.9,
)
>>> ground_truth = dataset.calc_ground_truth_policy_value(action_dist=action_dist)
>>> ground_truth
0.865643879173291

# off-policy evaluation using IPW
>>> ope = OffPolicyEvaluation(bandit_feedback=bandit_feedback, ope_
↪estimators=[IPW()])
>>> estimated_policy_value = ope.estimate_policy_values(action_dist=action_dist)
>>> estimated_policy_value
{'ipw': 0.8662705029276045}

# evaluate the estimation performance (accuracy) of IPW by relative estimation␣
↪error (relative-ee)
>>> relative_estimation_errors = ope.evaluate_performance_of_estimators(
        ground_truth_policy_value=ground_truth,
        action_dist=action_dist,
    )
>>> relative_estimation_errors
{'ipw': 0.000723881690137968}
```

References

Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. "Doubly Robust Policy Evaluation and Optimization.", 2014.

calc_ground_truth_policy_value (*action_dist: numpy.ndarray*) → numpy.ndarray

Calculate the ground-truth policy value of a given action distribution.

Parameters **action_dist** (*array-like, shape (n_rounds_ev, n_actions, 1)*) – Action distribution or action choice probabilities of a policy whose ground-truth is to be caliculated here. where n_rounds_ev is the number of samples in the evaluation set given the current train-eval split. n_actions is the number of actions. axis 2 of action_dist represents the length of list; it is always 1 in the current implementation.

Returns **ground_truth_policy_value** – policy value of a given action distribution (mostly evaluation policy).

Return type float

obtain_action_dist_by_eval_policy (*base_classifier_e: Optional[sklearn.base.ClassifierMixin] = None, alpha_e: float = 1.0*) → numpy.ndarray

Obtain action choice probabilities by an evaluation policy.

Parameters

- **base_classifier_e** (*ClassifierMixin, default=None*) – Machine learning classifier used to construct a behavior policy.
- **alpha_e** (*float, default=1.0*) – Ration of a uniform random policy when constructing an **evaluation** policy. Must be in the [0, 1] interval (evaluation policy can be deterministic).

Returns **action_dist_by_eval_policy** – action_dist_by_eval_policy is an action choice probabilities by an evaluation policy. where n_rounds_ev is the number of samples in the evaluation set given the current train-eval split. n_actions is the number of actions. axis 2 represents the length of list; it is always 1 in the current implementation.

Return type array-like, shape (n_rounds_ev, n_actions, 1)

obtain_batch_bandit_feedback(*random_state: Optional[int] = None*) → Dict[str, Union[int, numpy.ndarray]]

Obtain batch logged bandit feedback, an evaluation policy, and its ground-truth policy value.

Note: Please call *self.split_train_eval()* before calling this method.

Parameters

- **eval_size** (*float or int, default=0.25*) – If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples.
- **random_state** (*int, default=None*) – Controls the random seed in sampling actions.

Returns **bandit_feedback** – bandit_feedback is logged bandit feedback data generated from a multi-class classification dataset.

Return type BanditFeedback

split_train_eval(*eval_size: Union[int, float] = 0.25*, *random_state: Optional[int] = None*) → None

Split the original data into the training (used for policy learning) and evaluation (used for OPE) sets.

Parameters

- **eval_size** (*float or int, default=0.25*) – If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the evaluation split. If int, represents the absolute number of test samples.
- **random_state** (*int, default=None*) – Controls the random seed in train-evaluation split.

property len_list

Length of recommendation lists.

property n_actions

Number of actions (number of classes).

property n_rounds

Number of samples in the original multi-class classification data.

6.8.4 simulator module

obp.simulator.simulator	Bandit Simulator.

obp.simulator.simulator

Bandit Simulator.

Functions

run_bandit_simulation(bandit_feedback, policy)	Run an online bandit algorithm on the given logged bandit feedback data.

obp.simulator.simulator.**run_bandit_simulation**(*bandit_feedback: Dict[str, Union[int, numpy.ndarray]]*, *policy: Union[*obp.policy.base.BaseContextFreePolicy, obp.policy.base.BaseContextualPolicy*]*) → numpy.ndarray

Run an online bandit algorithm on the given logged bandit feedback data.

Parameters

- **bandit_feedback** (*BanditFeedback*) – Logged bandit feedback data used in offline bandit simulation.
- **policy** (*BanditPolicy*) – Online bandit policy evaluated in offline bandit simulation (i.e., evaluation policy).

Returns **action_dist** – Action choice probabilities (can be deterministic).

Return type array-like, shape (n_rounds, n_actions, len_list)

6.8.5 others

obp.utils	Useful Tools.

obp.utils

Useful Tools.

Functions

check_bandit_feedback_inputs(context, ... [, ...])	Check inputs for bandit learning or simulation.
check_is_fitted(estimator[, attributes, ...])	Perform is_fitted validation for estimator.
convert_to_action_dist(n_actions, ...)	Convert selected actions (output of *run_bandit_simulation*) to distribution over actions.
estimate_confidence_interval_by_bootstrap(samples, ...)	Estimate confidence interval by nonparametric bootstrap-like procedure.
sigmoid(x)	Calculate sigmoid function.
softmax(x)	Calculate softmax function.

obp.utils.**check_bandit_feedback_inputs**(*context: numpy.ndarray*, *action: numpy.ndarray*, *reward: numpy.ndarray*, *position: Optional[numpy.ndarray] = None*, *pscore: Optional[numpy.ndarray] = None*, *action_context: Optional[numpy.ndarray] = None*) → Optional[AssertionError]

Check inputs for bandit learning or simulation.

Parameters

- **context** (*array-like, shape (n_rounds, dim_context)*) – Context vectors in each round, i.e., x_t.
- **action** (*array-like, shape (n_rounds,)*) – Action sampled by a behavior policy in each round of the logged bandit feedback, i.e., a_t.
- **reward** (*array-like, shape (n_rounds,)*) – Observed rewards (or outcome) in each round, i.e., r_t.
- **position** (*array-like, shape (n_rounds,), default=None*) – Positions of each round in the given logged bandit feedback.
- **pscore** (*array-like, shape (n_rounds,), default=None*) – Propensity scores, the probability of selecting each action by behavior policy, in the given logged bandit feedback.
- **action_context** (*array-like, shape (n_actions, dim_action_context)*) – Context vectors characterizing each action.

obp.utils.**check_is_fitted**(*estimator: sklearn.base.BaseEstimator*, *attributes=None*, ***, *msg: str = None*, *all_or_any=<built-in function all>*) → bool

Perform is_fitted validation for estimator.

Note: Checks if the estimator is fitted by verifying the presence of fitted attributes (ending with a trailing underscore) and otherwise raises a NotFittedError with the given message. This utility is meant to be used internally by estimators themselves, typically in their own predict / transform methods.

Parameters

- **estimator** (*estimator instance.*) – estimator instance for which the check is performed.
- **attributes** (*str, list or tuple of str, default=None*) – Attribute name(s) given as string or a list/tuple of strings Eg.: `["coef_", "estimator_", ...], "coef_"` If *None*, *estimator* is considered fitted if there exist an attribute that ends with a underscore and does not start with double underscore.

- **msg** (*string*) – The default error message is, "This %(name)s instance is not fitted yet. Call 'fit' with appropriate arguments before using this estimator." For custom messages if "%(name)s" is present in the message string, it is substituted for the estimator name. Eg. : "Estimator, %(name)s, must be fitted before sparsifying".
- **all_or_any** (*callable, {all, any}, default all*) – Specify whether all or any of the given attributes must exist.

Returns **is_fitted** – Whether the given estimator is fitted or not.

Return type bool

References

https://scikit-learn.org/stable/modules/generated/sklearn.utils.validation.check_is_fitted.html

obp.utils.**convert_to_action_dist**(*n_actions: int*, *selected_actions: numpy.ndarray*) → numpy.ndarray

Convert selected actions (output of *run_bandit_simulation*) to distribution over actions.

Parameters

- **n_actions** (*int*) – Number of actions.
- **selected_actions** (*array-like, shape (n_rounds, len_list)*) – Sequence of actions selected by evaluation policy at each round in offline bandit simulation.

Returns **action_dist** – Action choice probabilities (can be deterministic).

Return type array-like, shape (n_rounds, n_actions, len_list)

obp.utils.**estimate_confidence_interval_by_bootstrap**(*samples: numpy.ndarray*, *alpha: float = 0.05*, *n_bootstrap_samples: int = 10000*, *random_state: Optional[int] = None*) → Dict[str, float]

Estimate confidence interval by nonparametric bootstrap-like procedure.

Parameters

- **samples** (*array-like*) – Empirical observed samples to be used to estimate cumulative distribution function.
- **alpha** (*float, default=0.05*) – P-value.
- **n_bootstrap_samples** (*int, default=10000*) – Number of resampling performed in the bootstrap procedure.
- **random_state** (*int, default=None*) – Controls the random seed in bootstrap sampling.

Returns **estimated_confidence_interval** – Dictionary storing the estimated mean and upper-lower confidence bounds.

Return type Dict[str, float]

obp.utils.**sigmoid**(*x: Union[float, numpy.ndarray]*) → Union[float, numpy.ndarray]

Calculate sigmoid function.

obp.utils.**softmax**(*x: Union[float, numpy.ndarray]*) → Union[float, numpy.ndarray]

Calculate softmax function.

6.9 References

6.9.1 Papers

6.9.2 Projects

This project is strongly inspired by **Open Graph Benchmark** –a collection of benchmark datasets, data loaders, and evaluators for graph machine learning: [github] [project page] [paper].

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Mehrdad Farajtabar, Yinlam Chow, and Mohammad Ghavamzadeh. More robust doubly robust off-policy evaluation. In *Proceedings of the 35th International Conference on Machine Learning*, 1447–1456. 2018.

[2] Doina Precup, Richard S. Sutton, and Satinder Singh. Eligibility Traces for Off-Policy Policy Evaluation. In *Proceedings of the 17th International Conference on Machine Learning*, 759–766. 2000.

[3] Alex Strehl, John Langford, Lihong Li, and Sham M Kakade. Learning from Logged Implicit Exploration Data. In *Advances in Neural Information Processing Systems*, 2217–2225. 2010.

[4] Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. Doubly Robust Policy Evaluation and Optimization. *Statistical Science*, 29:485–511, 2014.

[5] Yusuke Narita, Shota Yasui, and Kohei Yata. Efficient counterfactual learning from bandit feedback. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4634–4641. 2019.

[6] Nathan Kallus and Masatoshi Uehara. Intrinsically efficient, stable, and bounded off-policy evaluation for reinforcement learning. In *Advances in Neural Information Processing Systems*. 2019.

[7] Joseph DY Kang, Joseph L Schafer, and others. Demystifying double robustness: a comparison of alternative strategies for estimating a population mean from incomplete data. *Statistical science*, 22(4):523–539, 2007.

[8] Yu-Xiang Wang, Alekh Agarwal, and Miroslav Dudik. Optimal and Adaptive Off-policy Evaluation in Contextual Bandits. In *Proceedings of the 34th International Conference on Machine Learning*, 3589–3597. 2017.

[9] Yi Su, Maria Dimakopoulou, Akshay Krishnamurthy, and Miroslav Dudík. Doubly robust off-policy evaluation with shrinkage. *arXiv preprint arXiv:1907.09623*, 2019.

[10] Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *International Conference on Machine Learning*, 127–135. 2013.

[11] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A Contextual-bandit Approach to Personalized News Article Recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, 661–670. ACM, 2010.

[12] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, 2249–2257. 2011.

[13] Dhruv Kumar Mahajan, Rajeev Rastogi, Charu Tiwari, and Adway Mitra. Logucb: an explore-exploit algorithm for comments recommendation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, 6–15. 2012.

[14] Lihong Li, Wei Chu, John Langford, Taesup Moon, and Xuanhui Wang. An Unbiased Offline Evaluation of Contextual Bandit Algorithms with Generalized Linear Models. In *Journal of Machine Learning Research: Workshop and Conference Proceedings*, volume 26, 19–36. 2012.

[15] Alina Beygelzimer and John Langford. The offset tree for learning with partial labels. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 129–138. 2009.

[16] Adith Swaminathan and Thorsten Joachims. The Self-normalized Estimator for Counterfactual Learning. In *Advances in Neural Information Processing Systems*, 3231–3239. 2015.

[17] Yusuke Narita, Shota Yasui, and Kohei Yata. Off-policy bandit and reinforcement learning. *arXiv preprint arXiv:2002.08536*, 2020.

PYTHON MODULE INDEX

O

INDEX

F

G

I

L

M

N

O

P

R

S

U

V