DRO: A Python Library for Distributionally Robust Optimization in Machine Learning

Jiashuo Liu* LIUJIASHUO77@GMAIL.COM
Tianyu Wang* TW2837@COLUMBIA.EDU
Henry Lam† KHL2114@COLUMBIA.EDU
Hongseok Namkoong† NAMKOONG@GSB.COLUMBIA.EDU
Jose Blanchet† JOSE.BLANCHET@STANFORD.EDU

Abstract

We introduce `dro`, an open-source Python library for distributionally robust optimization (DRO) for regression and classification problems. The library implements 14 DRO formulations and 9 backbone models, enabling 79 distinct DRO methods. Furthermore, `dro` is compatible with both `scikit-learn` and `PyTorch`. Through vectorization and optimization approximation techniques, `dro` reduces runtime by 10x to over 1000x compared to baseline implementations on large-scale datasets. Comprehensive documentation is available at `https://python-dro.org`.

1. Introduction

Robustness is critical in high-stakes application domains of machine learning and decision making, such as finance [4, 16], healthcare [40] and supply chain [1, 36]. A central problem in these settings is the presence of distribution shifts between the training data and deployment environment [29]. To address this, Distributionally Robust Optimization (DRO) has emerged as a key framework for training predictive models and making decisions that perform well under worst-case scenarios over a set of plausible deployment distributions [21]. With various specifications of distributional uncertainty sets [6, 25], a growing number of DRO formulations have been developed to tackle various types of distribution shifts.

However, the practical adoption of DRO remains limited due to lack of a general practical computational software. A key challenge lies in the intrinsic difficulty in solving DRO problems, which typically involve min-max optimization. The tractability of these problems crucially depends on the structure of the predictive model and loss function [21]. Even in simple linear settings – where reformulations may allow the use of standard solvers – popular variants of Wasserstein DRO [13, 39] often suffer large computational overhead to compute near-exact solutions, rendering them impractical for large-scale modern machine learning tasks. For more complex models, including widely used neural networks and tree-based ensembles, only heuristic or approximation algorithms exist. Correspondingly, existing software tools often fall short in two key aspects. (i) Scalability and ML integration: Many rely on symbolic reformulations of distributionally robust objectives or constraints to more standard representations and solve the resulting problems using general-purpose solvers [7, 8, 17, 37]. While helpful, these tools lack encapsulation of the DRO workflow within a unified interface — users must still manually construct the worst-case scenario for the optimization

* Equal contribution, orders listed alphabetically † . Equal Advising

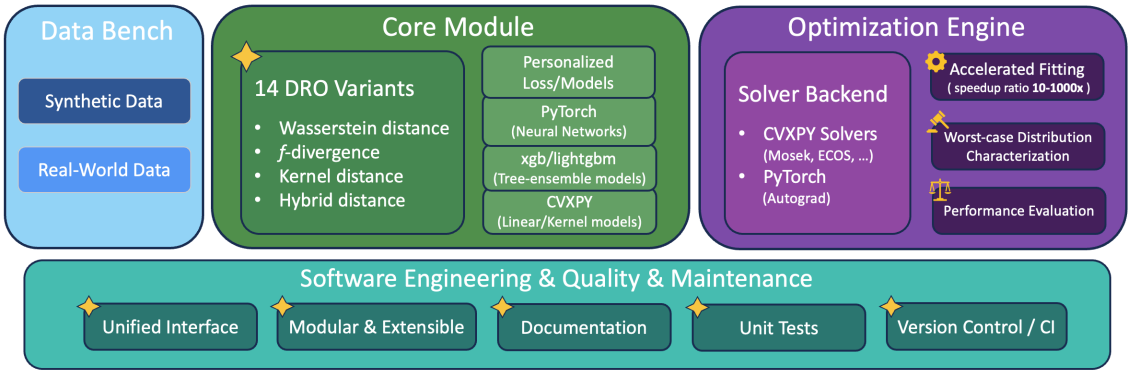

Figure 1: Overview of the `dro` library.

problem and call the corresponding optimization solver, which is computationally expensive and difficult to scale or integrate into modern ML workflows. (ii) Flexibility in modeling: some tools focus narrowly on one single DRO formulation (e.g., [38]), limiting their applicability in handling general distribution shifts. As a result, DRO's strengths are not yet fully realized in real-world applications, particularly in settings that demand both statistical robustness and computational efficiency.

This paper introduces `dro`: a library which provides the first comprehensive set of tools by offering modular implementations of different DRO formulations tailored for ML problems, compatible with both `scikit-learn` and `PyTorch`. This package supports 79 methodological combinations (14 formulations $\times$ 9 algorithmic backbones), encompassing a wide range of formulations including the Wasserstein distance and various f-divergences, and algorithmic backbones including linear, kernel, tree-based, and neural networks, with flexible user customization. To allow scalable experimentation while solving reformulated optimization near-exactly, we apply advanced vectorization and approximation techniques. The library achieves 10–1000$\times$ speedups over baseline implementations, enabling efficient deployment without sacrificing computational precision. This design translates state-of-the-art theoretical DRO research into reproducible, high-performance software. We provide comprehensive documentation, a modular architecture, and seamless integration with modern ML workflows in the library to both researchers and practitioners. By unifying engineering efficiency and optimization precision, `dro` enables principled, scalable deployment of robust learning systems and fosters new cross-disciplinary collaborations in trustworthy AI.

2. Package Overview

The `dro` library is designed to provide an easy-to-use framework for DRO, including data generation, model fitting, and model-based diagnosis, and is built on top of `CVXPY` [9] and `PyTorch` [28]. The releases of the `dro` package are available via PyPI at `https://pypi.org/project/dro/`. Our documentation is at `https://python-dro.org/`, which contains detailed installation instructions, tutorials, examples, and API reference. An overview of the `dro` library is shown in Figure 1, and the following summarizes our supported methods. We defer additional components (data generation and model-based diagnostics, including generating the worst-case distribution and performance

evaluation of some DRO methods) to Appendix A.1 and A.2, and full modeling details of the 79 methodological combinations to Appendix A.3.

Problem Formulation. The `dro` library addresses supervised learning problems with training samples $\{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathcal{X}$ denotes feature vectors and $y_i \in \mathcal{Y}$ the corresponding labels. Given a model class $\mathcal{F}$ of predictors $f\colon \mathcal{X} \to \mathcal{Y}$ and a loss function $\ell\colon \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$, the library solves DRO problems as:

$$\min_{f\in\mathcal{F}} \sup_{Q\in\mathcal{P}} \mathbb{E}_Q\big[\ell(f(X), Y)\big], \tag{1}$$

where $\mathcal{P}$ denotes the *ambiguity set* containing the empirical distribution and its neighborhood. This formulation guarantees robustness against distribution shifts between training and deployment environments. The ambiguity set $\mathcal{P}(d, \epsilon) = \{Q : d(Q, \widehat{P}_n) \le \epsilon\}$ characterizes distributions within ϵ-radius of the empirical distribution $\widehat{P}_n := \frac{1}{n}\sum_{i=1}^n \delta_{(x_i,y_i)}$, where $d(\cdot,\cdot)$ is a probability distance metric (e.g., Wasserstein distance) and $\epsilon > 0$ controls the robustness level. This construction provides coverage guarantees against distribution shifts while maintaining computational tractability. The library's DRO formulations adapt to three core components: the *distance metric* d, *model class* $f(\cdot)$, and *loss function* ℓ.

Supported Distance Metrics. We support 4 principal types of distance metrics and 14 DRO formulations in total. Complete method specifications are detailed in Table 1.

Table 1: Different DRO methods supported in `dro` package.

	Exact Optimization						Approximate Optimization		
	LAD	OLS	SVM	Logistic	Kernel	Personal	Tree-based	NN	Personal
WDRO	✓	✓	✓	✓	✓	✓		✓	✓
RS-WDRO	✓	✓	✓	✓	✓	✓			
χ^2-DRO	✓	✓	✓	✓	✓	✓	✓	✓	✓
KL-DRO	✓	✓	✓	✓	✓	✓	✓		
Bayesian-DRO	✓	✓	✓	✓					
CVaR-DRO	✓	✓	✓	✓	✓	✓	✓	✓	✓
TV-DRO	✓	✓	✓	✓	✓	✓			
Marginal(-CVaR)-DRO	✓	✓	✓	✓	✓	✓			
Conditional(-CVaR)-DRO	✓	✓	✓	✓	✓	✓			
MMD-DRO	✓	✓	✓	✓		✓			
HR-DRO	✓		✓					✓	
Sinkhorn-DRO	✓	✓	✓	✓				✓	
OutlierRobust(OR)-WDRO	✓		✓						
MOT-DRO	✓		✓						

(1) *Wasserstein distance* [3, 14, 33] via allowing changes in X and (or) Y;
(2) *f-divergence* including KL-divergence (KL) [18], χ^2-divergence [10, 22], Total Variation [20], Conditional Value-at-Risk (CVaR) with its variants in marginal X [12] and conditional $Y|X$ shifts [32];
(3) *Kernel distance* methods [41];
(4) *Hybrid distance* measures combining multiple metrics, such as Holistic Robust (HR)-DRO [2], Sinkhorn distance [39], Outlier-robust Wasserstein Distance [27], and Moment Optimal Transport Distance (MOT) [5].

Supported Model Classes & Loss Functions. The `dro` library supports linear models, kernel methods, tree methods, and neural networks, with (i) *Exact optimization* (convex optimization for linear cases: classification via hinge/logistic loss, regression via ℓ_1/ℓ_2 and their extension in kernel methods) for all distance-based methods. To ensure optimization fidelity, each model is fitted through disciplined convex programming using `CVXPY` backends. and (ii) *Approximate optimization* methods (tree-based ensembles and neural networks) for classical Wasserstein and f-divergence-based DRO methods. To ensure efficient deployment, the tree-based ensemble model is fitted on top of `lightgbm` or `xgboost` and the neural network model is fitted on top of `PyTorch`.

Personalization. Beyond the standard implementations, the library provides extensible support for specialized variants, including robust satisficing WDRO (RS-WDRO, [26]) and Bayesian DRO ([34]). The framework accommodates user-defined loss functions and model architectures — spanning linear, tree-based, and neural network paradigms — particularly for Wasserstein distance and standard f-divergence ambiguity sets through a unified and modular abstraction, where users typically personalize losses via only needing to update `self._loss()` (with details deferred to Appendix A.4). This flexibility enables tailored DRO solutions for specific objectives.

3. Modular Design Merits

In this section, we highlight three modular design merits.

Computational Performance Optimization. In the exact optimization module, we apply two key acceleration strategies: (1) systematic constraint vectorization and approximation for large-scale optimization problems, reducing solver processing time through batched operations, and (2) Nyström approximation for kernel methods, achieving orders-of-magnitude efficiency gains. These optimizations enable efficient handling of industrial-scale datasets while achieving near-exact solutions to DRO problems. Table 2 shows our method achieves order-of-magnitude speedups (10-1000×) compared with the current implementation of these corresponding methods, with greater gains observed at larger sample sizes. Acceleration details are in Appendix B.

Standard Interface. All methods provide a consistent API following `scikit-learn`'s estimator interface, including standard functions such as `update()`, `fit()`, `predict()`, and `score()`. This design ensures full compatibility with `scikit-learn`'s ecosystem while maintaining each method's specific parameters through inheritance. The interface strictly follows `scikit-learn`'s API guidelines to guarantee seamless integration with existing machine learning workflows. We provide a demo example in Appendix C.

Software Engineering Compliance. The package adheres to rigorous software engineering standards, featuring: (1) comprehensive unit testing with 91% line coverage across core algorithms; (2) static type checking via Python's MyPy at compile-time; (3) fully documented APIs including usage examples and mathematical specifications; and (4) continuous integration testing across Python 3.8+ environments (Linux/Windows/macOS). These practices ensure the implementation maintains both precision and production reliability.

Table 2: Effect of vectorization and approximation on computational performance. We report the average running time (and the standard error, in seconds) for DRO methods with and without vectorization and approximation, along with corresponding speedup ratios.

Sample	Speedup	KL-DRO	TV-DRO	MMD-DRO	OR-WDRO	Marginal-DRO	MOT-DRO	HR-DRO
1000	w/o	1.84±0.05	4.03±0.78	260.54±61.01	114.19±19.26	40.82±12.42	13.05±1.76	21.97±0.66
	w/	0.18±0.01	0.08±0.01	0.76±0.24	11.76±5.43	0.14±0.01	0.76±0.40	0.21±0.02
	Ratio	**10.2x**	**50.4x**	**342.8x**	**9.7x**	**291.6x**	**17.2x**	**104.6x**
10000	w/o	32.23±0.26	44.66±0.99	>7200	9977.41±342.74	>7200	351.43±53.88	900.60±26.88
	w/	2.25±0.10	0.94±0.02	31.22±2.68	141.16±73.94	2.68±0.10	11.38±5.15	2.63±0.31
	Ratio	**14.3x**	**47.5x**	**>230.6x**	**70.8x**	**>2686.6x**	**30.9x**	**342.4x**

4. Conclusion

Currently, the `dro v.0.3.3` library provides the most comprehensive implementation of data-driven DRO methods for supervised learning, supporting both diverse distance metrics and model architectures. In the future, we plan to include other methods, e.g., Group DRO [31] and efficient support for reinforcement learning settings.

References

[1] Ahmed Shihab Albahri, Ali M Duhaim, Mohammed A Fadhel, Alhamzah Alnoor, Noor S Baqer, Laith Alzubaidi, Osamah Shihab Albahri, Abdullah Hussein Alamoodi, Jinshuai Bai, Asma Salhi, et al. A systematic review of trustworthy and explainable artificial intelligence in healthcare: Assessment of quality, bias risk, and data fusion. *Information Fusion*, 96:156–191, 2023.

[2] Amine Bennouna and Bart Van Parys. Holistic robust data-driven decisions. *arXiv preprint arXiv:2207.09560*, 2022.

[3] Jose Blanchet, Yang Kang, Karthyek Murthy, and Fan Zhang. Data-driven optimal transport cost selection for distributionally robust optimization. In *2019 winter simulation conference (WSC)*, pages 3740–3751. IEEE, 2019.

[4] Jose Blanchet, Lin Chen, and Xun Yu Zhou. Distributionally robust mean-variance portfolio selection with wasserstein distances. *Management Science*, 68(9):6382–6410, 2022.

[5] Jose Blanchet, Daniel Kuhn, Jiajin Li, and Bahar Taskesen. Unifying distributionally robust optimization via optimal transport theory. *arXiv preprint arXiv:2308.05414*, 2023.

[6] Jose Blanchet, Jiajin Li, Sirui Lin, and Xuhui Zhang. Distributionally robust optimization and robust statistics. *arXiv preprint arXiv:2401.14655*, 2024.

[7] Zhi Chen and Peng Xiong. Rsome in python: An open-source package for robust stochastic optimization made easy. *INFORMS journal on computing*, 35(4):717–724, 2023.

[8] Zhi Chen, Melvyn Sim, and Peng Xiong. Robust stochastic optimization made easy with rsome. *Management Science*, 66(8):3329–3339, 2020.

[9] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[10] John Duchi and Hongseok Namkoong. Variance-based regularization with convex objectives. *Journal of Machine Learning Research*, 20(68):1–55, 2019.

[11] John Duchi, Tatsunori Hashimoto, and Hongseok Namkoong. Distributionally robust losses for latent covariate mixtures. *Operations Research*, 71(2):649–664, 2023.

[12] John C Duchi and Hongseok Namkoong. Learning models with uniform performance via distributionally robust optimization. *The Annals of Statistics*, 49(3):1378–1406, 2021.

[13] Peyman Mohajerin Esfahani and Daniel Kuhn. Data-driven distributionally robust optimization using the wasserstein metric: Performance guarantees and tractable reformulations. *Mathematical Programming*, 171(1):115–166, 2018.

[14] Rui Gao and Anton Kleywegt. Distributionally robust stochastic optimization with wasserstein distance. *Mathematics of Operations Research*, 48(2):603–655, 2023.

[15] Josh Gardner, Zoran Popovic, and Ludwig Schmidt. Benchmarking distribution shift in tabular data with tableshift. *Advances in Neural Information Processing Systems*, 2023.

[16] Paolo Giudici and Emanuela Raffinetti. Safe artificial intelligence in finance. *Finance Research Letters*, 56:104088, 2023.

[17] Joel Goh and Melvyn Sim. Robust optimization made easy with rome. *Operations Research*, 59(4):973–985, 2011.

[18] Zhaolin Hu and L Jeff Hong. Kullback-leibler divergence constrained distributionally robust optimization. *Available at Optimization Online*, 1(2):9, 2013.

[19] Garud Iyengar, Henry Lam, and Tianyu Wang. Optimizer's information criterion: Dissecting and correcting bias in data-driven optimization. *arXiv preprint arXiv:2306.10081*, 2023.

[20] Ruiwei Jiang and Yongpei Guan. Risk-averse two-stage stochastic program with distributional ambiguity. *Operations Research*, 66(5):1390–1405, 2018.

[21] Daniel Kuhn, Soroosh Shafiee, and Wolfram Wiesemann. Distributionally robust optimization. *arXiv preprint arXiv:2411.02549*, 2024.

[22] Henry Lam. Recovering best statistical guarantees via the empirical divergence-based distributionally robust optimization. *Operations Research*, 67(4):1090–1105, 2019.

[23] Jiashuo Liu, Jiayun Wu, Bo Li, and Peng Cui. Distributionally robust optimization with data geometry. *Advances in neural information processing systems*, 35:33689–33701, 2022.

[24] Jiashuo Liu, Tianyu Wang, Peng Cui, and Hongseok Namkoong. On the need for a language describing distribution shifts: Illustrations on tabular datasets. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[25] Jiashuo Liu, Tianyu Wang, Peng Cui, and Hongseok Namkoong. Rethinking distribution shifts: Empirical analysis and inductive modeling for tabular data. *arXiv preprint arXiv:2307.05284*, 2023.

[26] Daniel Zhuoyu Long, Melvyn Sim, and Minglong Zhou. Robust satisficing. *Operations Research*, 71(1):61–82, 2023.

[27] Sloan Nietert, Ziv Goldfeld, and Soroosh Shafiee. Outlier-robust wasserstein dro. *Advances in Neural Information Processing Systems*, 37, 2023.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8024–8035, 2019.

[29] Joaquin Quinonero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D Lawrence. *Dataset shift in machine learning*. MIT Press, 2008.

[30] R Tyrrell Rockafellar, Stanislav Uryasev, et al. Optimization of conditional value-at-risk. *Journal of risk*, 2:21–42, 2000.

[31] Shiori Sagawa, Pang Wei Koh, Tatsunori B. Hashimoto, and Percy Liang. Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization. *International Conference on Learning Representations*, 2019.

[32] Roshni Sahoo, Lihua Lei, and Stefan Wager. Learning from a biased sample. *arXiv preprint arXiv:2209.01754*, 2022.

[33] Soroosh Shafieezadeh-Abadeh, Daniel Kuhn, and Peyman Mohajerin Esfahani. Regularization via mass transportation. *Journal of Machine Learning Research*, 20(103):1–68, 2019.

[34] Alexander Shapiro, Enlu Zhou, and Yifan Lin. Bayesian distributionally robust optimization. *SIAM Journal on Optimization*, 33(2):1279–1304, 2023.

[35] Aman Sinha, Hongseok Namkoong, and John C. Duchi. Certifying some distributional robustness with principled adversarial training. In *International Conference on Learning Representations*. OpenReview.net, 2018.

[36] Christopher S Tang. Robust strategies for mitigating supply chain disruptions. *International Journal of Logistics: Research and Applications*, 9(1):33–45, 2006.

[37] Phebe Vayanos, Qing Jin, and George Elissaios. Roc++: Robust optimization in c++. *INFORMS Journal on Computing*, 34(6):2873–2888, 2022.

[38] Florian Vincent, Waïss Azizian, Franck Iutzeler, and Jérôme Malick. skwdro: a library for wasserstein distributionally robust machine learning. *arXiv preprint arXiv:2410.21231*, 2024.

[39] Jie Wang, Rui Gao, and Yao Xie. Sinkhorn distributionally robust optimization. *arXiv preprint arXiv:2109.11926*, 2021.

[40] Andrew Wong, Erkin Otles, John P Donnelly, Andrew Krumm, Jeffrey McCullough, Olivia DeTroyer-Cooley, Justin Pestrue, Marie Phillips, Judy Konye, Carleen Penoza, et al. External validation of a widely implemented proprietary sepsis prediction model in hospitalized patients. *JAMA Internal Medicine*, 181(8):1065–1070, 2021.

[41] Jia-Jie Zhu, Wittawat Jitkrittum, Moritz Diehl, and Bernhard Schölkopf. Kernel distributionally robust optimization: Generalized duality theorem and stochastic approximation. In *International Conference on Artificial Intelligence and Statistics*, pages 280–288. PMLR, 2021.

Appendix Contents

A Detailed Components of the Package **8**
A.1 Data Generating 8
A.2 Model-based Diagnostics. 9
A.3 Full Details of DRO Models 9

B Acceleration Details **11**
B.1 Vectorization 11
B.2 Kernel Approximation. 12
B.3 Constrain Reduction 13

C Documentation and User Example **14**

Appendix A. Detailed Components of the Package

In this section, we demonstrate in detail the components of our `dro` package.

A.1. Data Generation

In addition to the core module and optimization engine, our `dro` library also supports various synthetic data generating mechanisms that are widely used in DRO literatures, which we refer to as the "Data Bench" (see Figure 1).

Synthetic Classification Datasets. We implement four synthetic classification datasets to evaluate model robustness under different structural challenges, inspired by recent DRO literature. Each dataset construction is encapsulated in a standalone function, and their origins are summarized in Table 3.

Table 3: Summary of synthetic classification datasets and their sources.

Function	Source	Description
`classification_basic`	Custom	Multi-class Gaussian blobs on a sphere; baseline data generator.
`classification_DN21`	[12] (Sec 3.1.1)	Linear decision boundary with controlled label noise.
`classification_SNVD20`	[35] (Sec 5.1)	Ring-shaped classification with uncertain margin regions.
`classification_LWLC`	[23] (Sec 4.1)	High-dimensional mixed features with geometric scrambling.

All generators include a visualization option via a shared utility `draw_classification`, and support reproducibility through random seed control.

Synthetic Regression Datasets. We implement five synthetic regression datasets, reflecting a range of structural challenges, inspired by recent DRO literature. The data generation functions and their sources are summarized in Table 4.

Each function returns covariates and targets in `NumPy` format, which simulates challenging distribution shifts for robust model evaluation.

Real-World Datasets. Real-world datasets can be easily imported using existing Python libraries such as `whyshift` [24] and `tableshift` [15].

Table 4: Summary of synthetic regression datasets and their sources.

Function	Source	Description
`regression_basic`	Custom	Standard linear regression with Gaussian noise; serves as a baseline.
`regression_DN20_1`	[12] (Sec 3.1.2)	Linear model with heteroscedastic noise based on covariate threshold.
`regression_DN20_2`	[12] (Sec 3.1.3)	Mixture of linear models simulating group shifts.
`regression_DN20_3`	[12] (Sec 3.3)	Real-world UCI crime dataset with pre-processing.
`regression_LWLC`	[23] (Sec 4.1)	Complex high-dimensional regression with group imbalance and spurious correlation.

A.2. Model-based Diagnostics

To understand the robustness of different methods, we provide two functions given samples post training a model $\hat{f}$ in the linear modules: (i) Generating Worst-case Distribution: Computing the worst-case distribution (i.e., Q that maximizes $\mathbb{E}_Q[\ell(\hat{f}(X);Y)]$) in Wasserstein distance (based on the approximate worst-case distribution in [33]) or f-divergence (based on a direct inner maximization); (ii) Evaluating Out-of-sample Model Performance: Estimate the out-of-sample loss $\mathbb{E}_P[\ell(\hat{f}(X);Y)]$ based on training samples $\{(x_i,y_i)\}_{i\in[n]}$ by correcting in-sample bias in standard f-divergence-based DRO models (based on the bias results in [19]). Furthermore, the methods in our package also support the hyperparameter tuning in `scikit-learn`.

A.3. Full Details of DRO Models

Here, we specify details of each ambiguity set of the general DRO problem in (1) with the ambiguity set $\mathcal{P}$ (or $\mathcal{P}(d,\epsilon)$ more specifically). We refer readers to the complete modeling details in https://python-dro.org/api/tutorials.html.

(1) Wasserstein Distance. In the standard Wasserstein-DRO (**WDRO**), we apply $d(P,Q) = W(P,Q)$, where $W(\cdot,\cdot)$ is the Wasserstein distance (for $Z_1=(X_1,Y_1), Z_2=(X_2,Y_2)$):

$$W(P_1,P_2) = \inf_{\pi\sim(P_1,P_2)} \mathbb{E}_\pi[c(Z_1,Z_2)].$$

For "lad", "svm", "logistic", the inner distance is captured by the norm: $c((X_1,Y_1),(X_2,Y_2)) = \|(X_1-X_2,Y_1-Y_2)\|$. For "ols", the inner distance is captured by the norm square: $c((X_1,Y_1),(X_2,Y_2)) = \|(X_1-X_2,Y_1-Y_2)\|^2$. No matter in each case, the norm is defined on the product space $\mathcal{X}\times\mathbb{R}$ by:

$$\|(x,y)\| = \|x\|_{\Sigma,p} + \kappa|y|.$$

Here $\|x\|_{\Sigma,p} = \|\Sigma^{1/2}x\|_p$. Σ is the identity matrix in the default setup and κ denotes the robustness parameter for the perturbation Y ($\kappa=0$ means that we do not allow perturbation of Y); p is the norm parameter for controlling the perturbation moment of X.

For the Robust Satisficing Wasserstein-DRO (**RS-WDRO**) method [26], following the same configuration as before, we show the optimization problem of RS-WDRO can be approximately reformulated as:

$$\max\left\{\|\theta\|_{\Sigma^{-1/2},p}, \quad \text{s.t. } E_{(X,Y)\sim P}[\ell_{tr}(\theta;(X,Y))] \le \tau + \epsilon W_c(P,\widehat{P}), \forall P\right\}.$$

Besides the standard configuration, we set τ as another hyperparameter, as the multiplication (i.e., the so-called *target ratio*, >1) of the best empirical performance with $E_{(X,Y)\sim\widehat{P}}[\ell(\widehat{\theta};(X,Y))]$ with $\widehat{\theta}\in\operatorname{argmin}_{\theta\in\Theta}\mathbb{E}_{\widehat{P}}[\ell(\widehat{\theta};(X,Y))]$ obtained in the corresponding empirical risk minimization problem.

(2) (Generalized) f-divergence. When d is set as the generalized f-divergence (including CVaR), all distances there can be formulated as follows:

$$d(P, Q) = E_Q \left[f\left(\frac{dP}{dQ}\right)\right].$$

For the **KL-DRO** method [18], we apply $f(x) = x \log x - (x - 1)$;
For the χ^2**-DRO** method [10], we apply $f(x) = (x - 1)^2$;
For the **TV-DRO** method [20], we apply $f(x) = |x - 1|$;
In each of the three methods above, the hyperparameter is ϵ as the uncertainty set size.
For the (joint) **CVaR-DRO** problem [30], we apply $f(x) = 0$ if $x \in [\frac{1}{\alpha}, \alpha]$ and ∞ otherwise (an augmented definition of the standard f-DRO problem). Here the hyperparameter α denotes the worst-case ratio. And we denote such induced d as the so-called *CVaR distance*.

Building on our codebase, we implement **Bayesian DRO** [34], via incorporating a nested structure for the DRO model based on the distribution prior:

$$\min_{\theta\in\Theta} \mathbb{E}_{\zeta\sim\zeta_N}[\sup_{Q\in\mathcal{Q}_\zeta(d,\epsilon)} \mathbb{E}_{\xi\sim Q}[\ell(\theta;\xi)]],$$

where ζ_N denotes the posterior distribution of the parametric distribution of ξ given $\{\xi_i\}_{i\in[N]}$. And $\mathcal{Q}_\zeta(d, \epsilon) = \{Q : d(Q, P_\zeta) \le \epsilon\}$ with P_ζ denotes the distribution parametrized by ζ.

In practice, we approximate the outer expectation $\zeta \sim \zeta_N$ via finite samples generated from the posterior distribution of ζ_N. In this sense, the optimization problem can be reformulated as a optimization problem with finite variables.

Besides these standard (generalized) f-divergence DRO models, we also include DRO models modeling partial distribution shifts on (X, Y) based on CVaR metrics. Here, we directly use $\mathcal{P}(\alpha)$ as the ambiguity set where α is the CVaR parameter.

If we only consider the shifts in the marginal distribution X, we formulate the **Marginal-CVaR-DRO** model:

$$\mathcal{P}(\alpha) = \{Q_0 : P_X = \alpha Q_0 + (1 - \alpha)Q_1, \text{for some } \alpha \ge \alpha_0 \text{ and distribution } Q_1 \text{ and } \mathcal{X}\}.$$

Specifically, we follow the formulation of (27) in [11] to fit the model.

If we consider the shift in the conditional distribution $Y|X$, we formulate the **Conditional-CVaR-DRO** model:

$$\mathcal{P}(\alpha) = \{Q_0 : P_{Y|X} = \alpha Q_0 + (1 - \alpha)Q_1, \text{for some } \alpha \ge \alpha_0 \text{ and distribution } Q_1 \text{ and } \mathcal{Y}\}.$$

Specifically, we follow the formulation of Theorem 2 in [32] to fit the model where approximating $\alpha(x) = \theta^\top x$.

(3) Kernel distance. We include Maximum Mean Discrepancy DRO (**MMD-DRO**) model with the ambiguity set $\mathcal{P}(d, \epsilon)$, where $d(P, Q) = \|\mu_P - \mu_Q\|_{\mathcal{H}}$ is the kernel distance with the Gaussian kernel and defined as:

$$\|\mu_P - \mu_Q\|^2_{\mathcal{H}} = \mathbb{E}_{x,x'\sim P}[k(x, x')] + \mathbb{E}_{y,y'\sim Q}[k(y, y')] - 2\mathbb{E}_{x\sim P,y\sim Q}[k(x, y)],$$

with $k(x, y) = \exp(-\|x - y\|_2^2/(2\sigma^2))$. In the computation, we apply Equation (7) of Section 3.1.1 in [41].

(4) Hybrid distance. We include other ambiguity set design $\mathcal{P}$ induced by some hybrid distances involved in the ambiguity set $\mathcal{P}$:

- **Sinkhorn-DRO**: $\mathcal{P}(W_\epsilon; \rho, \epsilon) = \{P : W_\epsilon(P, \hat{P}) \leq \rho\}$. Here $W_\epsilon(\cdot, \cdot)$ denotes the Sinkhorn Distance, defined as:

$$W_\epsilon(P, Q) = \inf_{\gamma \in \Pi(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}[c(x, y)] + \epsilon \cdot H(\gamma | \mu \otimes \nu),$$

  where $c(x, y) = \|x - y\|_2$ and μ, ν are reference measures satisfying $P \ll \mu$ and $Q \ll \nu$.

- **Holistic-DRO**: $\mathcal{P}(LP_\mathcal{N}, D_{KL}; \alpha, r) = \{P : P, Q \in \mathcal{P}, LP_\mathcal{N}(\hat{P}, Q) \leq \alpha, D_{KL}(Q\|P) \leq r\}$, where $LP(\cdot, \cdot)$ is the Levy-Prokhorov metric $LP_\mathcal{N}(P, Q) = \inf_{\gamma\in\Pi(P,Q)} \mathbb{I}(\xi - \xi' \notin \mathcal{N}) d\gamma(\xi, \xi')$ with $\mathcal{N}$ denoting the perturbed (inaccuracy) set of each sample; $D_{KL}(\cdot\|\cdot)$ is the KL-divergence $D_{KL}(Q\|P) = \int_Q \log \frac{dQ}{dP} dQ$. In our setup, we set the perturbed (inaccuracy) set as a ball $\mathcal{N} = B_2(0, \epsilon) \times \{0\}$ with parameter ϵ.

- **MOT-DRO**: We implement with the ambiguity set $\mathcal{P}(M_c; \epsilon) = \{(Q, \delta) : M_c((Q, \delta), \tilde{P}) \leq \epsilon\}$ uses the OT-discrepancy with moment constraints, defined as:

$$M_c(P, Q) = \inf_\pi \mathbb{E}_\pi[c((Z, W), (\hat{Z}, \hat{W}))],$$

  where $\pi_{(Z,W)} = P, \pi_{(\hat{Z},\hat{W})} = Q$, and $\mathbb{E}_\pi[W] = 1$. Taking the cost function as

$$c((z, w), (\hat{z}, \hat{w})) = \theta_1 \cdot w \cdot \|\hat{z} - z\|^p + \theta_2 \cdot (\phi(w) - \phi(\hat{w}))_+,$$

  where $\tilde{P} = \hat{P} \otimes \delta_1$.

- **OutlierRobust-WDRO**: We implement with the ambiguity set $\mathcal{P}(W_p^\eta; \epsilon) = \{Q : W_p^\eta(Q, \hat{P}) \leq \epsilon\}$, where:

$$W_p^\eta(P, Q) = \inf_{Q' \in \mathcal{P}(R^d), \|Q - Q'\|_{TV} \leq \eta} W_p(P, Q'),$$

  where p is the p-Wasserstein distance and $\eta \in [0, 0.5)$ denotes the corruption ratio.

A.4. Personalization

When fitting the exact linear models, users only need to rewrite the corresponding `self._loss()` and `self._cvx_loss()` functions to change the corresponding losses in the training procedure when creating a new problem instance. This personalization is useful for a variety of f-divergence-based DRO (including KL-divergence, χ^2-divergence, CVaR, TV-DRO). For Wasserstein DRO, besides modifying these two loss functions, we also need to modify the `self._penalization()` function to adjust the regularization component, where the regularization component denotes the additional part besides the empirical objective in the Wasserstein DRO objective after the problem reformulation.

When fitting the tree-based ensemble models, users only need to rewrite the corresponding `self._loss()` and `self._cvx_loss()` functions to change the corresponding losses in the training procedure when creating a new problem instance. This personalization helps for both KL, CVaR, χ^2-divergence-based robust XGBoost or LightGBM.

When fitting the neural network models, users only need to rewrite the `self._compute_individual_loss()` and `self._criterion()` to change the corresponding losses in the training procedure when creating a new problem instance for f-divergence-based DRO. In Wasserstein DRO, users only need to rewrite the `self._loss()` to change corresponding losses. Furthermore, users can pass their own models via `self.update()` function,

Appendix B. Acceleration Details

This section provides a detailed overview of the acceleration strategies employed in our `dro` package, including vectorization, kernel approximation, and formulation-specific constraint reductions designed to improve computational efficiency across different DRO variants. As shown in Table 2, these strategies significantly improve the computation efficiency.

B.1. Vectorization

A significant performance bottleneck in `CVXPY` arises from how symbolic constraints are constructed, especially in large-scale DRO problems involving N training samples.

Here we take the constraints of KL-DRO as an example. A naive approach adds one constraint at a time using a for-loop:

```
# Inefficient loop-based constraint construction
for i in range(sample_size):
    constraints.append(
        cp.constraints.exponential.ExpCone(per_loss[i] - t, eta, epi_g[i
            ])
    )
```

This loop-based construction incurs substantial overhead, as `CVXPY` must individually parse and track each constraint's symbolic graph structure in Python. To address this, our implementation uses a vectorized formulation that creates all N constraints in a single batched expression:

```
# Efficient vectorized constraint construction
constraints.append(
    cp.constraints.exponential.ExpCone(
        per_loss - t,
        eta * np.ones(sample_size),
        epi_g
    )
)
```

This vectorized approach significantly reduces Python-level overhead and enables `CVXPY` to construct and optimize the constraint graph more efficiently. In practice, it leads to substantial speedups in both model formulation and solver pre-processing. We apply this vectorization strategy consistently across all DRO formulations implemented in our `dro` library.

B.2. Kernel Approximation

To accelerate kernel matrix computations, our implementation supports an optional Nyström approximation. Specifically, when `n_components` is provided, we apply the `Nystroem` transformer from `scikit-learn` to approximate the kernel mapping via a low-rank feature embedding. This significantly reduces computational cost compared to computing the full kernel matrix, especially when the number of support vectors is large. The kernel computation logic is summarized as follows:

```
# Kernel computation with optional Nystrom approximation
if self.n_components is None:
    K = pairwise_kernels(X, self.support_vectors_,
                         metric=self.kernel, gamma=self.kernel_gamma)
```

```
else:
    nystroem = Nystroem(kernel=self.kernel, gamma=self.kernel_gamma,
                        n_components=self.n_components)
    K = nystroem.fit(self.support_vectors_).transform(X)
```

When `n_components` is not specified, we compute the exact kernel matrix using `pairwise_kernels`. Otherwise, the Nyström method produces a lower-dimensional approximation, enabling faster kernel feature computation while preserving empirical performance.

Parallel Kernel Approximation for MMD-DRO To ensure scalability of MMD-DRO across large datasets and high-dimensional settings, we further parallelize the kernel approximation. Given that kernel transformations remain the bottleneck in high-dimensional DRO models, we batch the data and apply the Nyström approximation in parallel across multiple CPU cores using `joblib`:

```
batches = [zeta[i:i+5000] for i in range(0, len(zeta), 5000)]
K_approx_list = Parallel(n_jobs=4)(
    delayed(nystroem.fit_transform)(batch) for batch in batches)
```

This reduces wall-clock time significantly while keeping memory usage controlled.

B.3. Constrain Reduction

For MMD-DRO and Marginal-DRO that are quite time-consuming, we apply constrain reduction to improve the efficiency (with approximation).

Constraint Subsampling for MMD-DRO. For MMD-DRO, to further reduce the problem size passed to the convex solver, we randomly subsample a small set of constraints from the full certify set. All loss expressions and constraint terms are computed in a vectorized form:

```
losses = compute_loss(X_selected, y_selected, theta, b)
rhs = f0 + K_approx[selected_indices] @ a
constraints = [losses <= rhs]
```

This avoids Python loops and allows `CVXPY` to efficiently compile the computational graph.

Sparse Reformulation for Marginal-DRO. The original formulation of Marginal-DRO in [11] involves a full $n \times n$ coupling matrix $B \in \mathbb{R}_+^{n \times n}$ over all pairs of training samples, leading to $\mathcal{O}(n^2)$ memory and time complexity. Specifically, the loss adjustment term takes the form:

$$s_i \geq \ell_i - \frac{1}{n}\left(\sum_{j=1}^{n} B_{ij} - \sum_{j=1}^{n} B_{ji}\right) - \eta \quad \text{for } i = 1, \ldots, n.$$

This dense matrix variable is computationally prohibitive for moderate-scale datasets, with solver time dominated by the evaluation of the `CVXPY` graph and the optimization of tens of thousands of primal and dual variables.

We replace the full matrix B with its row and column marginals $B^{\text{row}} \in \mathbb{R}^n$ and $B^{\text{col}} \in \mathbb{R}^n$, respectively. The pairwise distance information is encoded sparsely via a k-nearest neighbor (k-NN) graph over the control features:

$$s_i \geq \ell_i - \frac{1}{n}\left(B_i^{\text{row}} - B_i^{\text{col}}\right) - \eta,$$

$$\text{Cost} \propto \sum_{(i,j)\in \text{kNN}} d_{ij} \cdot \frac{B_i^{\text{row}} + B_j^{\text{col}}}{2},$$

where d_{ij} is a sparse upper-triangular matrix of local distances. This reformulation reduces both memory and computation from quadratic to linear in n, and allows the solver to handle much larger datasets.

Appendix C. Documentation and User Example

Figure 2 provides a visual overview of the documentation website. Below, we present a code demo demonstrating how to use `dro` given a typical used dataset for robust training for data loading (using synthetic data from the package), model fitting and diagnostics (computing worst-case distributions and out-of-sample cost performance). More elaborate examples and a full API documentation can be found in https://python-dro.org.

```
# User example of chi-square DRO
from dro.src.linear_model import *
from dro.src.data.dataloader_classification import classification_basic
from dro.src.chi2_dro import *
X, y = classification_basic(d = 2, num_samples = 100, radius = 2)
model = Chi2DRO(input_dim = 2, model_type = 'logistic')
model.update({'eps':1}) # update parameter
model.fit(X, y) # fit model
model.worst_distribution(X, y) # return the worst-case distribution
model.evaluate(X, y) # compute loss over the whole distribution
```

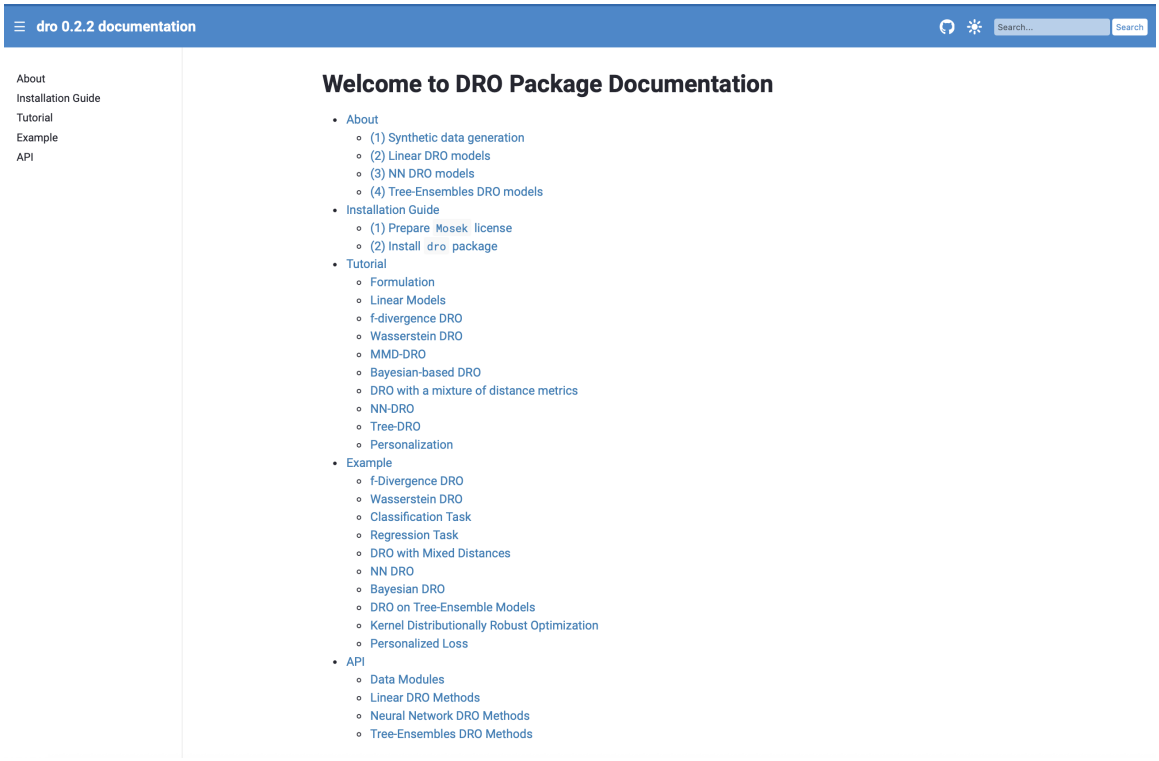

Figure 2: Overview of the documentation website.