Research Article

A Constraint-Compiled, Knowledge-Infused Framework for Robust and Explainable Discovery Across Heterogeneous Data Modalities

Yuusuke Harada[1]

1. Hiroshima University, Japan

Purely data-driven discovery can be brittle: it often captures spurious correlations, offers limited transparency, and degrades under distribution shifts or small sample regimes. This document proposes a data-agnostic framework specification that integrates domain knowledge as typed constraints and compiles them into train- and inference-time mechanisms. The framework, KID-CC (Knowledge-Infused Discovery via Constraint Compilation), standardizes (i) a typed knowledge interface, (ii) a constraint compiler that maps constraints to losses, projections, repairs, or constrained decoding, and (iii) an audit-first explainability contract that reports both predictive rationales and knowledge-compliance metrics. To evaluate without reliance on any particular dataset, we define a synthetic benchmark suite with controllable spurious correlations, environment shifts, missingness patterns, and knowledge contamination tests.

Correspondence: papers@team.qeios.com — Qeios will forward to the authors

1. Introduction

Big data and machine learning have accelerated scientific and socio-technical discovery, yet purely data-driven pipelines can fail in three recurring ways: (a) they learn shortcut signals that do not generalize (spurious correlation), (b) they produce predictions without actionable explanations, and (c) they become unstable when data are scarce, noisy, or shifted across environments [1][2][3][4][5].

This work is a methods-oriented paper: it does not propose a new predictor architecture, nor does it claim state-of-the-art performance on a specific dataset. Instead, it proposes a reusable framework

specification that treats domain knowledge as constraints, and a constraint compiler that translates knowledge into enforceable learning and inference procedures. Importantly, KID-CC is not a causal discovery tool: it does not guarantee causal identifiability, but it provides interfaces to inject and audit causal or physical knowledge when such knowledge is available [6].

1.1. Contributions

C1. Typed knowledge specification: represent domain knowledge as typed constraint objects (hard/soft, differentiable/non-differentiable, continuous/discrete, scope). C2. Constraint compilation and governance: a normative rule matrix that compiles constraints into loss terms, projections, repairs, constrained decoding, or rejection policies, with deterministic handling of constraint conflicts and optional budget-based weight tuning. C3. Audit-first explainability contract: standardize outputs to include predictive explanations and per-constraint compliance (violations, budgets/multipliers, and unsatisfiable/conflict events), enabling comparable audits across modalities. C4. Dataset-independent evaluation protocol: define synthetic benchmark generators and conformance tests to validate robustness, knowledge compliance, and explanation quality without committing to a specific dataset.

1.2. Positioning and related work (non-exhaustive)

KID-CC sits at the intersection of theory-guided learning, constrained optimization, robustness, and explainability. Its goal is not to replace existing approaches, but to provide a unified and implementable interface that makes knowledge integration explicit, auditable, and comparable across domains.

Theory-guided and physics-informed learning: prior work injects scientific laws and domain theory into models, often via specialized objectives or architectures (e.g., TGDS and PINNs). KID-CC generalizes this idea by treating knowledge as typed constraints whose enforcement strategy is compiled, not hard-coded to a single domain [7][8]. Constraint-based learning and posterior regularization: constraints have been used as regularizers or auxiliary objectives during training. KID-CC expands the design space by explicitly compiling constraints into training-time losses and inference-time mechanisms (projection/repair/constrained decoding), and by requiring per-constraint audits as outputs [9]. Robustness via invariance: invariance objectives (e.g., IRM) can be viewed as a specific class of cross-environment constraints. In KID-CC they are represented as scope=cross_env constraints, making the audit and budget interface uniform with other constraint types [10]. Acronym collision note: the acronym KIDD is used elsewhere; this document uses KID-CC to avoid ambiguity. [11]

1.3. Normative language and compliance levels

This document is written as a framework specification. We use normative keywords as follows: MUST indicates a requirement for compliance; SHOULD indicates a strong recommendation; MAY indicates optional behavior. Where practical, requirements are phrased so that compliance can be checked via conformance tests and audit outputs.

KID-CC compliant implementations MUST represent knowledge K as typed constraint objects (Definition 2) and implement a compiler interface Compile(k_j) -> (L_K,j, Enforce_j, Audit_j) (Definition 3). KID-CC compliant implementations MUST emit the unified explanation tuple z = (z_pred, z_K, z_audit) for each instance or batch (Definition 6), including per-constraint compliance metrics (at least VRate and MeanV). When hard constraints cannot be satisfied within bounded resources, implementations MUST record unsatisfiable/conflict events (UnsatFlag, ConflictIDs, Resolution) and follow a documented relaxation or fail-safe policy (Section 4.4). For the synthetic evaluation protocol (KID-Synth), reported results MUST include a completed reproducibility declaration (Checklist 6.1 or Checklist F1) or an equivalent statement covering all MUST items (Section 6.7 and Appendix F).

1.4. Companion implementation note and reproducibility commitments

To facilitate adoption while keeping this paper focused on a normative specification, we provide a separate companion implementation note. The companion consolidates (i) a Knowledge Constraint Language (KCL) sketch, (ii) a unified explanation/audit schema, (iii) modality-specific injection sketches (including constrained decoding and repair for text), and (iv) a benchmark documentation card template for KID-Synth.

To support transparency and reproducibility, this manuscript includes (i) a minimum reproducibility declaration (Section 6.7) and (ii) a standalone KID-Synth specification and reproducibility checklist (Appendix F). The accompanying documentation templates follow established dataset/model documentation frameworks (datasheets, model cards, data statements, data cards, and dataset nutrition labels) [12][13][14][15][16], and community reproducibility checklists [17].

2. Framework Overview

Definition 1 (Unified analysis mapping). Any analysis task is treated as a mapping from an observational dataset space and prior knowledge to a solution space:

$A : (D, K) \rightarrow S$

Here, D denotes observed data, K denotes prior/domain knowledge, and S denotes a solution space (e.g., predictors, rules, discovered structures, and explanations).

2.1. Data contract

The framework assumes a minimal dataset contract that is intentionally modality-agnostic:

D = {(x_i, y_i, e_i, m_i)}_{i=1..n}

where x is an observation (tabular, image, text, time-series, graph), y is an optional target, e is an optional environment/domain label, and m is optional metadata (missingness, sensor quality, annotator, acquisition conditions, etc.).

2.2. Three-layer modular pipeline

KID-CC is implemented as three composable layers with explicit input/output contracts:

h = Adapter(x) | $\tilde{h}$ = Filter(h; K) | ($\hat{y}$, z) = Solver($\tilde{h}$)

The key design choice is that knowledge K is not embedded implicitly in model weights, but carried as explicit constraint objects that can be compiled, audited, and reused.

2.3. Walkthrough example (from constraints to compiled enforcement and audit)

This section provides an end-to-end walkthrough on a minimal tabular predictor. The example is illustrative (non-normative): its purpose is to show what artifacts a KID-CC implementation produces, and how Table 1/2 guide compilation choices.

Task. Predict a scalar risk score $\hat{y}$ in [0,1] from inputs x = (age, biomarker). Optionally, each record carries an environment label e. Knowledge base K. Declare three constraints: (k_range_output) hard output range $\hat{y} \in [0,1]$; (k_monotone_age) soft monotonicity $\partial\hat{y}/\partial\text{age} \geq 0$; (k_unit_consistency) hard input validation (biomarker units must be consistent). Compilation. The compiler translates each constraint into a training term, an optional enforcement mechanism, and an audit function: range is enforced by construction (e.g., sigmoid head or projection), monotonicity is compiled to a differentiable penalty (or a monotone architecture), and unit consistency is compiled to preprocessing validation/repair or bounded rejection. (Table 1 and Table 2). Outputs and audit. For each prediction, the solver emits z = (z_pred, z_K,

z_audit), where z_pred explains the prediction, z_K reports per-constraint activity/violations, and z_audit aggregates compliance (and any conflicts) for traceability.

Example (illustrative) output tuple z
z_pred: attribution {age: 0.62, biomarker: 0.38}
z_K:
range_output (hard/output): enforced=true, v=0.00
monotone_age (soft/output): enforced=false, v=0.03
unit_consistency (hard/input): enforced=true, v=0.00
z_audit (batch): VRate(monotone_age)=0.12, MeanV(monotone_age)=0.01, UnsatFlag=false

In this example, the key benefit of the specification is that violations (even when tolerated) are measurable and comparable: the audit distinguishes performance degradation from knowledge non-compliance, and makes conflicts/fail-safe behavior explicit.

3. Typed Knowledge Specification

Definition 2 (Typed constraint object). A knowledge base is represented as a set of constraint objects, each carrying both content and implementability metadata:

```
K = {k_j}_{j=1..m},  k_j = (c_j, w_j, hardness τ_j, properties δ_j, scope σ_j)
```

Each constraint provides a violation score $v \geq 0$ (or a boolean that can be mapped to a violation score):

```
v_ij = viol_j(x_i, h̃_i, ŷ_i, z_i, e_i)  ≥  0
```

Hard constraints require $v_{ij} = 0$ by specification; soft constraints penalize v_{ij} but may allow violations.

Optional governance fields (recommended). The minimal typed constraint object can be extended with additional metadata to make behavior deterministic under conflicts and practical in deployment: Priority π_j: an ordinal used to resolve hard-hard conflicts; higher priority constraints are preserved when feasible. Violation budget ε_j: a target threshold (e.g., $\text{VRate}_j \leq \varepsilon_j$ or $\text{MeanV}_j \leq \varepsilon_j$) enabling optional automatic tuning of multipliers (Section 4.5). Relaxation strategy r_j: a policy invoked when constraints become unsatisfiable (e.g., demote-to-soft, drop, tolerance, bounded rejection), recorded in audit outputs.

These fields are optional in the specification, but recommended for any system intended for governance and safety-critical use.

3.1. Constraint catalog (non-exhaustive)

Range / bounds: $\hat{y} \in [a, b]$. Sign / non-negativity: $\hat{y} \geq 0$. Monotonicity: $\partial\hat{y}/\partial x\_k \geq 0$ (or ≤ 0). Conservation: $\Sigma\_t\ \hat{y}\_t$ = const (exact or approximate). Logical implication: $A \Rightarrow B$ (probabilistic or symbolic). Structural constraints: acyclic graphs, hierarchy constraints, sparsity in concept layers. Consistency with evidence (text): generated claims must be supported by retrieved evidence. Cross-environment invariance: relations that should remain stable across domains e.

4. Constraint Compilation and Enforcement

Definition 3 (Constraint compilation). A constraint compiler translates each typed constraint into three artifacts: (i) a training-time loss term (possibly a surrogate), (ii) an optional enforcement step at training/inference time, and (iii) a required audit function that reports compliance metrics. This abstraction generalizes prior "constraints as regularization" formulations such as posterior regularization[9] and logic/semantic loss approaches for symbolic constraints[18].

```
Compile(k_j) → (L_K,j , Enforce_j , Audit_j)
```

This design makes knowledge integration explicit and inspectable: when knowledge is imperfect, the audit output reveals where and how it breaks.

Definition 4 (Knowledge-infused objective). The framework optimizes a unified objective:

```
min_θ  L_task + λ·Σ_j L_K,j  + β·L_exp + γ·L_rob
```

where L_task is task loss (classification/regression/generation/etc.), L_K,j are compiled knowledge losses, L_exp regularizes explainability, and L_rob encodes robustness (e.g., cross-environment stability).

4.1. Compilation rule matrix

Table 1 provides normative recommendations for compiling constraints based on hardness and implementability (assuming constraints are satisfiable; conflicts are handled in Section 4.4). Constrained decoding for sequence generation is a canonical instance of inference-time enforcement[19].

Constraint attributes	Typical examples	Training term ($L_{K,j}$)	Enforcement ($Enforce_j$)	Notes
soft × differentiable × continuous	range; monotonicity; conservation (surrogate)	Penalty $\rho(v)$ (square / hinge / Huber)	Optional	Default and most reusable path
soft × differentiable × discrete	probabilistic logic; calibrated classification rules	Differentiable surrogate over probabilities	Optional	Probabilistic relaxation is key
soft × non-differentiable × continuous	rule checkers; dictionary validation; external validators	Smooth surrogate or score-based loss	Optional repair	Audit compensates approximation bias
soft × non-differentiable × discrete	constrained text generation; forbidden tokens	Score-based surrogate	Constrained decoding / repair / rejection	Inference-time constraints often dominate
hard × differentiable × continuous	nonnegativity; simplex; strict conservation	Optional barrier / augmented Lagrangian	Projection or reparameterization	Prefer “cannot violate by construction”
hard × differentiable × discrete	mandatory rules; forbidden combinations	Auxiliary surrogate (optional)	Constrained optimization / constrained decoding	Avoid relying only on penalties
hard × non-differentiable × continuous	exact verifier; simulator acceptance	Optional surrogate (optional)	Projection/repair or search with fallback	Define fail-safe behavior
hard × non-differentiable × discrete	formal grammar; strict logical consistency	Can be omitted	Constrained decoding / rejection / repair	Guarantee-first; always audit

Table 1. Constraint compilation rule matrix (normative recommendations).

4.2. Scope (injection point) matrix

Table 2 maps a constraint's scope σ to recommended injection points and implementation primitives.

Scope σ	Typical examples	Recommended injection point	Primitive
input	unit consistency; missingness rules	Pre-processing (before adapter)	Clipping; imputation; input repair
latent	concept separation; sparsity; hierarchy	Constraint Filter	Gating; regularization; projection
output	range; simplex; nonnegativity	Solver output head	Reparameterization; projection; barrier
explanation	brevity; stability; evidence requirement	Explainable Solver	Sparsity; faithfulness loss; stability loss
cross-environment	invariance; stable relations across e	Robustness term + auditing	Worst-env; invariance penalty; stratified audit

Table 2. Scope matrix: mapping constraint scope to injection points.

4.3. Compiler decision procedure (specification)

Algorithm 1 defines a deterministic decision procedure for compiling a constraint object into training loss, enforcement step, and audit output.

Algorithm 1. *Constraint compilation decision procedure (normative).*

Input: k = (c, w, hardness τ, properties δ, scope σ, optional: priority π, budget ε, relaxation r)

Output: (L_K, Enforce, Audit)

1. Define violation v := viol(x, $\tilde{h}$, $\hat{y}$, z, e) ≥ 0 (or boolean mapped to v)
2. Audit := { VRate = mean[v > 0], MeanV = mean[v], MaxV = max[v],
   Priority = π (optional), Budget = ε (optional),
   UnsatFlag = false, ConflictIDs = [], Resolution = "" }
3. If τ == soft:
4. If δ indicates differentiable:
5. L_K := w * mean(ρ(v)) # penalty (square / hinge / Huber)
6. Enforce := None (optional)
7. Else:
8. L_K := w * mean(surrogate(v)) # smooth surrogate or score-based loss
9. Enforce := optional repair / constrained decoding
10. Else (τ == hard):
11. L_K := optional auxiliary (barrier or augmented Lagrangian)
12. If projection / repair / constrained decoding exists:
13. Enforce := bounded projection / repair / constrained decoding
14. If Enforce fails to reach v = 0 within bounds:
15. Apply relaxation r (e.g., demote-to-soft, drop, tolerance, or UnsatisfiableError)
16. Set Audit.UnsatFlag = true; record ConflictIDs and Resolution
17. Else:
18. Enforce := reject-and-resample (max retries) or fail-safe output
19. If retries exceeded: set Audit.UnsatFlag = true; record Resolution
20. Return (L_K, Enforce, Audit)

Algorithm 1. Constraint compilation specification (with governance hooks)

4.4. Constraint conflicts and relaxation (normative)

In realistic deployments, multiple constraints may be mutually inconsistent. A framework specification must therefore define deterministic behavior when constraints conflict, rather than leaving the outcome

to implementation quirks or optimizer dynamics.

Definition 5 (Constraint conflict). Let H be the set of hard constraints. A conflict event occurs when no output can satisfy all hard constraints simultaneously within a specified tolerance after applying the configured enforcement procedures under bounded effort (e.g., bounded projection iterations, bounded rejection/repair attempts).

Equivalently, if the per-instance feasible set is empty:

```
F_i = { (ŷ_i, z_i) : v_ij = 0  for all hard constraints j ∈ H (within tolerance) }
;  conflict if F_i = ∅
```

To make conflict handling predictable, we recommend extending each constraint with (i) a priority π_j, and (ii) a relaxation strategy r_j. Priorities resolve hard-hard conflicts; relaxation strategies define what to do when constraints become unsatisfiable (e.g., demote-to-soft, drop, or raise an UnsatisfiableError event). All such events must be recorded in z_audit.

Conflict type	Detection signal	Default resolution	Audit output (required)
soft vs soft	High violation persists; optimization unstable	Trade-off via weights; optionally cap penalty gradients	VRate/MeanV per constraint; note active constraints
hard vs soft	Hard constraint violated after enforce	Enforce hard; keep soft as penalty (or pause soft if it destabilizes)	Hard violation flag; soft violation stats
hard vs hard (simple, statically checkable)	Feasible set intersection empty (e.g., disjoint ranges)	Resolve by priority π: keep higher π, relax lower π per r_j	UnsatFlag=true; conflicting IDs; chosen resolution
hard vs hard (complex, runtime)	Projection/repair fails within bounded steps	Apply relaxation policy: demote/drop lowest π; or fail-safe output	UnsatFlag=true; attempts; IDs; fail-safe action
nondiff hard (discrete constraints)	Constrained decoding / verifier rejects repeatedly	Bounded rejection; repair; or return conservative fallback	RejectCount; UnsatFlag if exceeded; fallback used

Table 3. Conflict handling strategies (normative recommendations).

Static feasibility checks are recommended when constraints admit closed-form checks (e.g., range intersections, simplex feasibility). For complex constraints (e.g., external validators, symbolic rules), conflict detection is operational: if Enforce cannot reach feasibility within bounded resources, the system must follow r_j and emit an unsatisfiable/conflict record in the audit output.

4.5. Weight governance and automatic tuning (optional)

Weights w_j and the global scaling λ should be treated as governance knobs rather than ordinary hyperparameters. In many settings, the most usable interface is to specify acceptable violation budgets ε_j and let the system tune multipliers automatically. Related non-convex constrained optimization formulations include proxy-Lagrangian / two-player game approaches[20].

Budgeted form (example using mean violation):

```
min_θ  L_task(θ)   subject to   MeanV_j(θ) ≤ ε_j  for all j
```

One practical option is a dual (Lagrangian) update, which adjusts per-constraint multipliers λ_j based on current violation summaries:

```
λ_j ← max(0, λ_j + η_λ · (g_j - ε_j))   where g_j is VRate_j or MeanV_j
```

Algorithm 2. *Budgeted constraint tuning via dual updates (optional).*

Input:

dataset D

constraints $K = \{k_j\}$ with budgets ε_j (target VRate or MeanV)

base weights w_j (confidence/importance)

initial multipliers $\lambda_j \geq 0$, multiplier step η_λ

Repeat for epochs:

1) Train model parameters Θ on $L_{total} = L_{task} + \Sigma_j \lambda_j \cdot w_j \cdot \rho(v_{ij}) + (\text{other regularizers})$

2) Compute audit summaries g_j on a held-out batch or moving window:

$g_j = VRate_j$ or $g_j = MeanV_j$

3) Dual update (projected):

$\lambda_j \leftarrow \max(0, \lambda_j + \eta_\lambda \cdot (g_j - \varepsilon_j))$

4) Emit audit: budgets ε_j, multipliers λ_j, and current violations

Output:

tuned multipliers λ_j and full audit logs

Algorithm 2. Dual update for constraint budgets (optional)

This option improves usability: instead of hand-tuning λ and w, a user may specify acceptable violation budgets ε_j, and the system automatically searches for multipliers that meet them as closely as possible while optimizing task loss. The tuned multipliers λ_j should be emitted in z_K and summarized in z_{audit} for traceability.

5. Explainability and Audit Specification

Explainability is treated as an output specification (contract), not a post-hoc add-on. The solver must emit prediction rationales and knowledge-compliance diagnostics in a unified format. This stance is aligned with prior calls for rigorous, context-dependent definitions of interpretability and caution against post-hoc explanations in high-stakes settings[4][5]. It also reflects concerns that visually

appealing explanations can be misleading without principled validation (e.g., randomization-based sanity checks for saliency)[21].

Definition 6 (Unified explanation tuple). For each prediction, the explanation object is defined as:

```
z = ( z_pred , z_K , z_audit )
```

z_pred explains the prediction (e.g., feature attributions, rules, rationales). z_K reports constraint activity (e.g., per-constraint violations, penalties, multipliers/budgets, enforcement status, and conflict resolutions). z_audit summarizes compliance metrics and governance events (including unsatisfiable/conflict flags) for the instance or batch.

5.1. Audit metrics

For each constraint j, the framework reports at least the violation rate and mean violation:

```
VRate_j = (1/n)·Σ_i 1[v_ij > 0]    |    MeanV_j = (1/n)·Σ_i v_ij
```

When environment labels e are available, audits must be stratified by environment to expose distributional failure modes.

When constraint conflicts occur (Section 4.4), the audit output must additionally report UnsatFlag, the set of conflicting constraints, and the applied resolution (e.g., demotion, drop, tolerance, or fail-safe fallback).

5.2. Audit-to-action interface (human-in-the-loop)

Audit outputs are only operationally useful if they connect to actions. Therefore, we recommend an explicit Audit-to-Action interface that consumes z_audit and produces governance updates (constraint patches, budget/priority changes, or data triage decisions).

Recommended feedback loop (conceptual): Run → Audit → Triage → Patch/Adjust → Recompile → Re-run

Trigger (from z_audit)	Typical action	Interface hook (examples)	Notes
UnsatFlag=true; ConflictIDs present	Resolve conflicts by priority/relaxation; escalate to expert	set_priority(id, π), set_relaxation(id, r), approve_override(...)	Must be logged; optionally block deployment until cleared
VRate_j or MeanV_j exceeds budget ε_j	Tighten enforcement or revise constraint scope	set_budget(id, ε), enable_dual_update(...), patch_constraint(...)	Prefer budget-driven tuning (Algorithm 2) before manual λ search
Violation spikes in a specific environment e	Investigate distribution shift; add environment-aware checks	stratify_audit(e), quarantine_samples(selector), add_env_constraint(...)	Treat as OOD signal; do not average away
Explanation instability increases	Increase stability regularization; revise explanation type	set_exp_regularizer(...), switch_explainer(type), patch_constraint(...)	Avoid brittle post-hoc explanations; enforce output contract
Persistent drift in audits over time	Trigger re-review of K and data pipeline	open_review_ticket(...), freeze_model(), recompile_and_rerun()	Audit drift is a governance event, not a mere metric change

Table 4. Audit-to-action hooks (recommended interface operations).

The interface above is intentionally implementation-agnostic: it may be realized as a command-line tool, a configuration file update (KCL/YAML), or an interactive dashboard. Crucially, every action must be traceable via change logs that link (i) audit evidence, (ii) the applied change, and (iii) the subsequent re-evaluation results.

6. Evaluation Protocol without Dataset Dependence

To avoid reliance on any particular real-world dataset, the evaluation is built on (i) synthetic benchmark generators with known ground truth and controllable shifts, and (ii) conformance tests that validate

framework-level guarantees (e.g., hard constraint satisfaction, audit availability).

6.1. Evaluation goals

Resistance to spurious correlations (shortcut learning). Robustness under distribution shifts and missingness changes. Knowledge compliance and auditability (constraint-level metrics). Explainability quality (faithfulness and stability against ground truth).

6.2. Synthetic benchmark generator (Latent–Environment–Renderer)

We define a unified generator that produces (x, y, e, m) with known ground truth. A full, standalone specification (KID-Synth) including a reproducibility checklist is provided in Appendix F:

```
u ~ N(0, I)  |  y = f(u) + ε  |  x = [x_c, x_s, x_n]
```

Causal features x_c depend on u, while spurious features x_s depend on y with environment-dependent strength α_e. Test environments flip or remove the spurious correlation to induce out-of-distribution failures[1][2][3].

6.3. Shift and degradation scenarios (spec)

S1: Spurious correlation sign reversal (train α=+a, test α=−a). S2: Spurious correlation removal (test α=0). S3: Covariate shift in latent distribution u ~ N(μ_e, Σ_e). S4: Missingness pattern shift (MCAR vs MNAR; change missingness rate). S5: Knowledge contamination (flip/perturb a subset of constraints; vary weights w).

6.4. Metrics

Performance: Accuracy/AUROC/AUPRC (classification) or RMSE/MAE (regression). Knowledge compliance: VRate_j, MeanV_j (overall and stratified by environment). Robustness: mean across environments, worst-environment performance, OOD gap. Explainability: Precision@k and Recall@k of top-k attribution vs causal feature set; faithfulness by deletion; stability under small perturbations.

6.5. Synthetic generator pseudocode

Algorithm 3. *Synthetic benchmark generator (normative).*

Input:

number of environments E

samples per environment n_e

dimensions (d, d_c, d_s, d_n)

shift scenario (S1–S5), functions f and g

modality renderer R_m (tabular / image-like / text-like)

For each environment e in {1..E}:

1) sample u_i ~ N(0, I_d) for i=1..n_e (or N(μ_e, Σ_e) under S3)

2) generate causal features: x_c = A·u + η_c

3) generate label: y = f(u) + ε (or Bernoulli for classification)

4) set spurious strength α_e:

train env: α_e = +a

test env : α_e = -a (S1) or 0 (S2)

5) generate spurious features: x_s = α_e·g(y) + η_s

6) generate noise features: x_n ~ N(0, I)

7) compose x = [x_c, x_s, x_n]

8) render observation x^(m) = R_m(x, e)

9) optionally apply missingness/noise (S4), store metadata m

Return:

dataset D = {(x^(m)_i, y_i, e_i, m_i)}

knowledge base K derived from generator (range/monotonicity/invariance/logic)

Algorithm 3. Synthetic benchmark generator (KID-Synth specification)

6.6. Conformance tests (framework-level)

Hard-constraint satisfaction: after Enforce_j, verify v_ij = 0 (or within tolerance) for all i. Soft-constraint effectiveness: increasing λ should reduce MeanV_j on controlled inputs. Audit availability: every run emits {VRate_j, MeanV_j, ...} for all constraints, and stratified audits by environment when e exists. Budget-based tuning (optional): when budgets ε_j are provided and dual updates are enabled (Algorithm 2), the run emits budgets and multipliers λ_j, and violations track budgets under controlled settings. Conflict

handling: when Enforce cannot satisfy hard constraints within bounded resources, the run emits UnsatFlag, ConflictIDs, and Resolution, and follows a documented fail-safe policy. Explanation contract: z = (z_pred, z_K, z_audit) is emitted for all instances/batches.

6.7. Recommended reporting format

As a framework paper, results should prioritize interpretability of effects rather than leaderboard performance. We recommend reporting: (i) in-environment vs OOD performance (mean and worst-env), (ii) per-constraint violation metrics (overall and stratified), (iii) explanation quality metrics, and (iv) ablations (no knowledge, no filter, no enforce, no audit, noisy knowledge). Best practices for OOD/worst-group reporting and the pitfalls of domain generalization evaluation motivate this focus[3][10][22].

Minimum reproducibility declaration (required for KID-Synth): KID-Synth results MUST include either (i) a completed copy of Checklist F1 (Appendix F) or (ii) an equivalent statement containing, at minimum, all MUST items listed below. Missing MUST items should be treated as non-compliant (results incomplete). [17]

Implementation note: If space is limited, authors may place the completed Checklist F1 in an appendix/supplement and cite it here. The template below is intended as a minimal, copy-paste reproducibility declaration for the main text or supplement.

Item (MUST)	Provided? (☐/☑)	Where (section/appendix/link)
Config object (Table F1) with all required fields	☐	
Exact definitions of f and g (with parameters)	☐	
A matrix construction (and seed if random)	☐	
Scenario set (S1–S5) and per-scenario parameter changes	☐	
Ground truth export: S_c, S_s and any renderer mappings	☐	
Knowledge base K (IDs, hardness, scope, weights, priorities, relaxation; budgets ε_j if used)	☐	
Evaluation metrics and attribution setting k (e.g., k=5/10)	☐	

Checklist 6.1 (minimum). KID-Synth reproducibility declaration template (MUST items; copy-paste and fill in).

Strongly recommended (SHOULD): To improve interpretability and comparability, also report the following SHOULD items (or justify deviations in the audit/report).

Renderer specification and mapping to causal/spurious sources (needed for explanation evaluation). Conformance tests results (hard satisfaction, audit availability, conflicts, budgets), summarized as pass/fail with notes. Compute budget (samples per environment, number of runs/seeds), to interpret stability and variance.

7. Limitations and Non-goals

This framework does not guarantee causal identifiability; it provides a disciplined interface for injecting and auditing assumptions. Constraint quality depends on correctness and scope of K. Therefore, knowledge contamination (S5) and audit outputs are treated as mandatory. Non-differentiable hard constraints (e.g., repair steps or constrained decoding) can increase inference latency and operational cost. KID-CC therefore allows implementations to treat inference-time budgets as first-class (optional) constraints (e.g., t_budget or cost_budget) that influence compilation choices: when strict enforcement

would exceed a budget, implementations MAY switch to bounded/approximate enforcement or conservative rejection, and MUST record the policy and any budget violations in z_audit. Over-constraining can reduce predictive performance; weights w and λ (or multipliers λ_j) should be treated as governance knobs. Budget-based tuning (Algorithm 2) reduces manual tuning but may require step-size control and monitoring to avoid oscillations.

Conflicting hard constraints are possible; deterministic resolution requires priorities and relaxation policies, and unsatisfiable events must be surfaced via audit outputs and handled by fail-safe mechanisms.

7.1. Broader Impact Statement

This work proposes a specification and evaluation protocol for knowledge-infused, auditable ML pipelines. Its primary intended benefit is to make data-centric assumptions (validity rules, invariances, evidence requirements) explicit and measurable, reducing silent failures due to spurious correlations or untracked constraint violations.

Potential positive impacts include: Improved transparency and accountability in safety- or policy-relevant ML deployments via per-constraint audit outputs (violations, conflicts, and fail-safe behavior). Better reproducibility of benchmarking and evaluation, since KID-Synth specifies controllable environments and a standardized reporting checklist. Lower barrier for domain experts to contribute knowledge about data quality and plausible ranges, supporting data cleaning and governance workflows.

Potential negative impacts and risks include: Incorrect or biased constraints can hard-code harmful assumptions (e.g., systematically disadvantaging subpopulations) and may appear 'legitimized' by compliance metrics. Audit outputs could be misused as a substitute for substantive validation ('audit theater') if stakeholders treat low violation rates as a blanket safety guarantee. Constraint enforcement (repair/constrained decoding) can increase latency and energy use, potentially shifting costs to smaller organizations or raising environmental impact. Synthetic benchmarks can be gamed if participants optimize to the generator rather than the underlying robustness goals, or if synthetic scenarios are overinterpreted as real-world coverage.

Mitigations: (i) require knowledge contamination tests (Scenario S5) and report uncertainty around knowledge bases; (ii) treat audits as observability, not guarantees, and explicitly report fail-safe policies and budgets; (iii) support human-in-the-loop review and revision of constraints based on audit evidence; (iv) publish artifacts with clear licenses and maintenance plans, and document limitations of synthetic

coverage. We intend this work to comply with relevant venue and community ethics guidelines (e.g., DMLR code of conduct and NeurIPS code of ethics).

7.2. Discussion and practical considerations

KID-CC's "audit-first" and "constraint compilation" design choices are intentionally data-centric: they turn tacit domain assumptions (validity rules, invariances, evidence constraints) into typed, executable artifacts that can be documented, versioned, and reviewed. In practice, compiled constraints and their audits serve as a bridge between dataset documentation and model reporting traditions, complementing frameworks such as datasheets, model cards, data statements, data cards, and dataset nutrition labels.[12][13][14][15][16]

Inference-time enforcement is not free. For discrete and/or hard constraints, constrained decoding and repair can dominate latency, and the chosen enforcement strategy should be treated as a deployment decision (not an implementation detail). We therefore recommend optionally exposing an explicit latency/compute budget as part of the audit output, and reporting it alongside violation metrics. This trade-off is already visible in the broader constrained decoding literature (e.g., grid beam search for lexically constrained generation).[19]

Because λ and w (or per-constraint multipliers λ_j) function as governance knobs, we recommend a "budget-first" interface whenever possible: users specify acceptable violation budgets ε_j, while the solver tunes multipliers via dual updates. For non-convex settings and non-differentiable constraints, proxy-Lagrangian and game-based constrained optimization methods offer a useful reference point for making such tuning stable and implementable.[20]

KID-Synth is designed to reduce dependence on any single real-world benchmark, but synthetic evaluation also has known pitfalls: participants can overfit to generator idiosyncrasies, and synthetic coverage can be mistaken for real-world coverage. To mitigate this, we recommend reporting worst-environment (or worst-group) metrics and clearly stating model selection procedures, since domain generalization research has shown that evaluation outcomes can hinge on selection protocols and experimental details[3][22]. A minimal reproducibility checklist should be treated as part of the benchmark contract rather than an afterthought.[17]

Finally, the framework boundary should be explicit: KID-CC is not a causal discovery tool, and knowledge infusion does not by itself guarantee causal identifiability[6]. Likewise, explanation methods can be

fragile; randomization-based sanity checks highlight that appealing explanations may not reflect the learned model[21]. For this reason, the audit outputs are best interpreted as observability signals that support governance and debugging, not as a blanket safety certificate.[4][5]

8. Conclusion

We proposed KID-CC, a data-agnostic framework specification for knowledge-infused discovery via constraint compilation. The key contributions are (i) typed constraints, (ii) normative compilation rules and decision procedures, and (iii) an audit-first explainability contract. A dataset-independent evaluation protocol based on synthetic generators and conformance tests is defined to validate robustness and compliance.

Appendix

The appendices provide (A) a concrete constraint catalog with compilation notes, (D) reporting templates and checklists for framework evaluations, (E) a naming note on acronym collisions, and (F) a standalone specification and reproducibility checklist for the Latent–Environment–Renderer (KID-Synth) benchmark. Implementation-oriented materials (KCL examples, unified schemas, injection sketches, and benchmark card templates) are provided in the companion implementation note.

Appendix A. Constraint Catalog Examples (with compilation hints)

This appendix lists example constraints in a modality-agnostic manner. Each entry includes a brief note on typical compilation targets.

A1. Range (soft, diff, continuous)

Constraint: $\hat{y} \in [a, b]$.
Violation: $v = \max(0, a-\hat{y}) + \max(0, \hat{y}-b)$.
Compile: $L_K = w \cdot \text{mean}(v^2)$; optional output projection for hard variant.

A2. Non-negativity (hard, diff, continuous)

Constraint: $\hat{y} \geq 0$.

Compile: reparameterize $\hat{y}$ = softplus(t); or project negative outputs to 0.

A3. Simplex (hard, diff, continuous)

Constraint: $\hat{y}$ is a probability vector, $\hat{y}_k \geq 0$ and $\Sigma_k \hat{y}_k = 1$.

Compile: softmax head; audit calibration and constraint satisfaction.

A4. Monotonicity (soft, diff, continuous)

Constraint: $\partial\hat{y}/\partial x_k \geq 0$.

Violation: $v = \max(0, -\partial\hat{y}/\partial x_k)$.

Compile: gradient penalty; optionally use monotone networks.

A5. Lipschitz / smoothness (soft, diff)

Constraint: model sensitivity bounded, $||\partial\hat{y}/\partial x|| \leq L$.

Compile: gradient norm penalty; used to stabilize explanations.

A6. Conservation (hard/soft, diff)

Constraint: $\Sigma_t \hat{y}_t = \text{const}$.

Compile: add penalty (soft) or projection to enforce sum constraint (hard).

A7. Symmetry / invariance (soft, diff)

Constraint: f(T(x)) = f(x) for known transformation T. Compile: consistency loss between outputs; augment audit with invariance gap.

A8. Ordering consistency (soft, nondiff or diff)

Constraint: if x_a preferred to x_b, then score(x_a) ≥ score(x_b). Compile: pairwise hinge loss; for nondiff labels, use ranking surrogates.

A9. Logical implication (soft, discrete)

Constraint: A ⇒ B. Compile: probabilistic relaxation; penalty on P(A)-P(A∧B).

A10. Mutual exclusion (hard, discrete)

Constraint: not(A and B). Compile: constrained decoding / structured prediction; audit violation counts.

A11. Coverage (soft, discrete)

Constraint: at least one of {A,B,C} must hold. Compile: surrogate on probabilities; enforce via repair at inference.

A12. Structural acyclicity (hard, nondiff)

Constraint: learned graph must be acyclic. Compile: projection or specialized parameterization; otherwise rejection+repair.

A13. Sparsity in concept layer (soft, diff)

Constraint: explanation uses at most k concepts. Compile: L1 penalty or top-k gating; audit explanation length.

A14. Group sparsity (soft, diff)

Constraint: select features in groups (e.g., sensor groups). Compile: group lasso-style penalty; improves stability.

A15. Evidence support (soft/hard, nondiff, text)

Constraint: each factual claim must be supported by retrieved evidence. Compile: score-based surrogate; enforce via constrained decoding or refusal policy.

A16. Citation requirement (hard, discrete, text)

Constraint: answer must include citations for factual claims. Compile: constrained decoding templates; audit missing-citation rate.

A17. Unit consistency (hard, input)

Constraint: inputs must respect units (e.g., mg/dL). Compile: preprocessing validation; rejection or repair.

A18. Missingness rule (soft/hard, input)

Constraint: certain fields must not be missing, or missingness indicates a specific state. Compile: imputation policy + audit stratified by missingness.

A19. Cross-environment invariance (soft, cross-env)

Constraint: stable relations across environments. Compile: invariance penalty; report worst-env performance and violation stratification.

A20. Knowledge weight governance (meta-constraint)

Constraint: high-confidence constraints must not be violated above threshold. Compile: set τ=hard or dynamic λ; audit triggers alerts when exceeded.

Appendix B. Reporting Templates and Checklists

The tables below can be copied directly into a results section. They emphasize interpretability of framework effects: robustness, compliance, and explanation quality.

B1. Performance across environments (template)

Method	Train (mean)	Test/OOD (mean)	Worst-env	OOD gap

Template B1. Report in-environment vs OOD performance (mean and worst-env).

B2. Constraint compliance (template)

Constraint ID	Hard/Soft	Scope	VRate	MeanV

Template B2. Report compliance per constraint (overall and stratified by environment in an additional table).

B3. Explanation quality (template)

Method	Precision@k	Recall@k	Deletion (faithfulness)	Insertion (faithfulness)	Stability (Δ explanation)

Template B3. Report explanation quality against ground truth (synthetic) or proxy metrics (real data).

B4. Ablation and compliance checklist

A0 Full framework (Adapter + Filter + Solver + Compile + Audit). A1 No knowledge: $\lambda=0$ (remove $L_{K,j}$). A2 No filter: bypass Constraint Filter. A3 No enforcement: disable Enforce_j (keep audits). A4 No audit: remove audit outputs (should degrade trustworthiness; included to demonstrate necessity). A5 Noisy knowledge: apply S5 knowledge contamination and vary weights w.

Appendix C. Naming note (acronym collision)

If the acronym KIDD is used for “Knowledge-Infused Data-Driven Discovery”, note that KIDD is also used in other communities (e.g., Kernel Ridge Regression-Based Graph Dataset Distillation at KDD ’23). To reduce ambiguity, this document uses KID-CC for the proposed framework.

Appendix D. KID-Synth: Latent-Environment-Renderer Specification and Reproducibility Checklist

This appendix makes the dataset-independent evaluation protocol operational by defining a standalone synthetic benchmark specification based on a Latent-Environment-Renderer (LER) generator. The goal is not to produce a new dataset, but to make framework evaluations comparable across implementations by standardizing (i) generator configuration, (ii) shift scenarios, (iii) modality renderers, (iv) knowledge derivation, and (v) reporting and reproducibility checklists.

Normative keywords used in this appendix: MUST: required for compliance with this specification. SHOULD: strongly recommended; deviations must be justified in the audit/report. MAY: optional; included to support broader use cases.

D1. Benchmark configuration object (required fields)

A KID-Synth run MUST be fully described by a single configuration object (Config). All fields below MUST be logged (or explicitly set to null/none) so that the generator and derived knowledge base are reproducible.

Field	Type / example	Req.	Notes
seed	int	MUST	Global random seed for all stochastic components (u sampling, noise, renderer randomness).
env.train_envs, env.test_envs	list[int]	MUST	Environment IDs used for training vs test/OOD evaluation.
n_per_env	int	MUST	Samples per environment (or a per-env mapping); report if imbalanced.
dims.d, dims.d_c, dims.d_s, dims.d_n	int	MUST	Latent, causal, spurious, and noise dimensions; defines x = [x_c, x_s, x_n].
alpha.train, alpha.test	float / dict	MUST	Spurious strength α_e for each env; supports S1 sign reversal and S2 removal.
noise.sigma_y, sigma_c, sigma_s, sigma_n	float	SHOULD	Noise scales for label and features; if omitted, defaults MUST be stated.
f.family, f.params	enum + dict	MUST	Label function family (linear / nonlinear / interaction) and parameters.
g.type	enum	MUST	Spurious link g(y) (e.g., identity for regression, 2y-1 for binary).
A.spec	string/dict	MUST	How A is constructed (fixed random matrix, orthonormal, sparse, etc.) and any seed.
renderer.modality, renderer.params	enum + dict	MUST	Renderer choice (tabular / image-like / text-like) and parameters.
scenarios.enabled	list[str]	MUST	Which shift/degradation scenarios are active (subset of S1–S5).
missingness.spec (S4)	dict or none	SHOULD	MCAR/MNAR mechanism, rate(s), and whether it changes between train/test.
knowledge_contamination.spec (S5)	dict or none	SHOULD	Contamination type (flip/widen/wrong-scope), fraction, and severity.

Field	Type / example	Req.	Notes
export.ground_truth	bool	MUST	Whether to export causal indices S_c and mappings needed for explanation evaluation.

Table D1. KID-Synth configuration schema (normative).

D2. Core generator model (LER)

The LER generator produces (x, y, e, m) with known ground truth by explicitly separating causal features x_c and spurious features x_s. This separation enables controlled shortcut learning in training and predictable failures under environment shifts.

Latent and label:

`u_i ~ N(0, I_d)  ;  y_i = f(u_i) + ε_i  (or Bernoulli for classification)`

`ε_i ~ N(0, σ_y^2). Under covariate shift (S3), u_i ~ N(μ_e, Σ_e) with environment-specific parameters.`

Feature construction:

`x_c = A u + η_c  ;  x_s = α_e · g(y) + η_s  ;  x_n ~ N(0, I)`

`x = [x_c, x_s, x_n]  ;  m encodes metadata (e.g., missingness mask, quality flags).`

Ground truth MUST include at least: (i) indices (or masks) of causal features S_c and spurious features S_s, (ii) environment parameters (α_e, μ_e, Σ_e when applicable), and (iii) any renderer-specific mappings needed to evaluate explanations (e.g., which pixels/tokens correspond to causal vs spurious sources).

D3. Shift and degradation scenarios (S1–S5)

A benchmark instance is defined by enabling one or more scenarios below. For each enabled scenario, the configuration MUST specify the exact parameter changes between training and test environments.

S1 Spurious sign reversal: Train: α_e = +α. Test: α_e = −α (same magnitude). Expected: shortcut-reliant models invert decisions. S2 Spurious removal: Train: α_e = +α. Test: α_e = 0. Expected: shortcut signal disappears; causal-only models remain stable. S3 Covariate shift in u: Test uses u ~ N(μ_e, Σ_e) with μ_e/

Σ_e different from training; specify shift type (mean/cov/rotation). S4 Missingness shift: Change missingness mechanism and/or rate between train/test. Include MCAR rate and MNAR rule if used. S5 Knowledge contamination: Corrupt a subset of constraints (flip sign, widen/tighten bounds, wrong scope/hardness) and vary severity.

D4. Renderer contract (R_m) and modality-specific guidance

A renderer maps the abstract feature vector x into a modality-specific observation x^(m). The renderer MUST preserve the intended causal mechanism (y depends on x_c) while exposing spurious cues derived from x_s that can be shifted across environments.

```
x^(m) = R_m(x, e; seed)
```

Renderer requirements: R_m MUST be fully determined by the configuration and seed (no hidden randomness). R_m SHOULD provide a mapping from observable elements to sources (causal vs spurious) for explainability evaluation. If renderers include nuisance variability (e.g., blur, synonym choice), it MUST be controlled and logged via the seed.

Recommended renderer sketches: Tabular (baseline): x^(tab) = x. Image-like: causal features control shape (size/position/orientation), while spurious features control background (color/texture). Text-like: templates encode causal relations; spurious tokens (rare word/emoji/punctuation) correlate with y only in training.

Example (text-like renderer template, conceptual)
Causal clause: "Entity A has property P = {x_c1:.2f}."
Decision rule: "If P > 0 then label is positive." (or a more complex f)
Spurious token injection (train only):
if y==1: append "however" else append "therefore" (correlation controlled by α_e)

D5. Knowledge derivation rules (K_from_generator)

Because the generator is known, KID-Synth SHOULD derive a knowledge base K directly from the generator specification. This ensures that constraint definitions, budgets, and ground truth are aligned.

Range constraints: bounds on y or on output probabilities are derived from generation/clipping rules. Monotonicity constraints: if f is monotone in selected latent/causal dimensions, derive $\partial\hat{y}/\partial x\_k \geq 0$ (or ≤

0). Cross-environment invariance: encode that causal relations remain stable across e; evaluate via worst-env metrics and stratified audits. Logic constraints (text-like): derive implications and consistency rules from the template grammar. Budgets: optionally define per-constraint violation budgets ε_j so that users specify acceptable violation rather than tuning λ manually.

D6. Reproducibility checklist (MUST report for KID-Synth)

The following checklist is designed to be copy-pasted into a paper's appendix or reproducibility statement. A KID-Synth result SHOULD be considered incomplete if any MUST item is missing.

Checklist item	Req.	Provided? (☐/☑)	Notes
Config object (Table F1) with all required fields	MUST	☐	Includes seed, env split, dims, α_e, noise, f/g, renderer.
Exact definitions of f and g (with parameters)	MUST	☐	State linear/nonlinear/interaction family and coefficients.
A matrix construction (and seed if random)	MUST	☐	Report whether A is orthonormal/sparse and how generated.
Scenario set (S1–S5) and per-scenario parameter changes	MUST	☐	Explicitly state train vs test settings.
Renderer specification and mapping to causal/spurious sources	SHOULD	☐	Needed for explanation evaluation.
Ground truth export: S_c, S_s and any renderer mappings	MUST	☐	Used for Precision@k/Recall@k and faithfulness tests.
Knowledge base K (IDs, hardness, scope, weights, priorities, relaxation)	MUST	☐	Include budgets ε_j if used.
Evaluation metrics and k for attribution (e.g., k=5/10)	MUST	☐	Report both performance and compliance metrics.
Conformance tests results (hard satisfaction, audit availability, conflicts, budgets)	SHOULD	☐	Can be summarized as pass/fail with notes.
Compute budget (samples per env, number of runs/seeds)	SHOULD	☐	Helps interpret stability and variance.

Checklist D1. KID-Synth reproducibility checklist (fill in).

D7. Minimal reference implementation skeleton (pseudo-API)

The framework does not mandate a specific codebase, but the following pseudo-API clarifies the expected artifacts: Dataset D, KnowledgeBase K, and GroundTruth GT. Implementations MAY serialize Config and K in YAML/JSON for governance.

```
@dataclass
class KIDSynthConfig:
    seed: int
    train_envs: list[int]
    test_envs: list[int]
    n_per_env: int
    d: int; d_c: int; d_s: int; d_n: int
    alpha: dict[int, float]       # α_e per environment
    noise: dict[str, float]       # σ_y, σ_c, σ_s, σ_n
    f: dict                       # family + params
    g: dict                       # type
    renderer: dict                # modality + params
    scenarios: list[str]          # e.g., ["S1", "S4"]
    contamination: dict | None    # S5
    missingness: dict | None      # S4

def generate_kid_synth(cfg: KIDSynthConfig) -> (D, K, GT):
    # 1) sample u per env
    # 2) generate y = f(u) + ε
    # 3) generate x_c, x_s, x_n and compose x
    # 4) render x^(m) = R_m(x,e)
    # 5) apply missingness/noise; record metadata m
    # 6) derive K from generator spec; export GT
    return D, K, GT
```

D8. Recommended default suite (for method papers)

To reduce degrees of freedom, we recommend reporting at least three task families (linear, nonlinear, interaction) under S1 and S2, and adding S4 and S5 as stress tests. The table below provides a minimal default configuration that can be scaled.

Component	Default
Environments	train_envs=[1,2,3], test_envs=[4,5]
Samples	n_per_env=5000 (repeat with 3 seeds)
Dims	d=10, d_c=5, d_s=2, d_n=20
Spurious strength	α=1.0 (train); test uses -1.0 (S1) or 0 (S2)
Noise	σ_y=0.1, σ_c=0.1, σ_s=0.1, σ_n=1.0
f families	linear; nonlinear (MLP or sin); interaction (u1*u2+u3)
Renderers	tabular (required); optionally image-like and text-like
Knowledge	range + (optional) monotonicity; include budgets ε_j if using dual updates

Table F2. Minimal recommended KID-Synth configuration (suggested defaults).

References

1. a, b *Geirhos R, Jacobsen J-H, Michaelis C, Zemel R, Brendel W, Bethge M, Wichmann FA (2020). "Shortcut Learning in Deep Neural Networks." Nat Mach Intell. **2**(11):665–673. doi:10.1038/s42256-020-00257-z.*
2. a, b *Quiñonero-Candela J, Sugiyama M, Schwaighofer A, Lawrence ND (2008). Dataset Shift in Machine Learning. Cambridge, MA: The MIT Press.*
3. a, b, c, d *Sagawa S, Koh PW, Hashimoto TB, Liang P (2020). "Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization." International Conference on Learning Representations (ICLR). https://openreview.net/forum?id=ryxGuJrFvS.*
4. a, b, c *Doshi-Velez F, Kim B (2017). "Towards a Rigorous Science of Interpretable Machine Learning."*
5. a, b, c *Rudin C (2019). "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead." Nat Mach Intell. **1**:206–215. doi:10.1038/s42256-019-0048-x.*
6. a, b *Pearl J (2009). Causality: Models, Reasoning, and Inference. 2nd edition. Cambridge: Cambridge University Press.*
7. ^ *Karpatne A, Atluri G, Faghmous JH, Steinbach M, Banerjee A, Ganguly A, Shekhar S, Samatova N, Kumar V (2017). "Theory-Guided Data Science: A New Paradigm for Scientific Discovery from Data." IEEE Trans Kno*

wl Data Eng. ***29****(10):2318–2331. doi:10.1109/TKDE.2017.2720168.*

8. ^*Raissi M, Perdikaris P, Karniadakis GE (2019). "Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations." J Comput Phys.* ***378****:686–707. doi:10.1016/j.jcp.2018.10.045.*
9. a, b*Ganchev K, Graça J, Gillenwater J, Taskar B (2010). "Posterior Regularization for Structured Latent Variable Models." J Mach Learn Res.* ***11****(67):2001–2049.*
10. a, b*Arjovsky M, Bottou L, Gulrajani I, Lopez-Paz D (2019). "Invariant Risk Minimization."*
11. ^*Xu Z, Chen Y, Pan M, Chen H, Das M, Yang H, Tong H (2023). "Kernel Ridge Regression-Based Graph Dataset Distillation." Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23). doi:10.1145/3580305.3599398.*
12. a, b*Gebru T, Morgenstern J, Vecchione B, Vaughan JW, Wallach H, Daumé III H, Crawford K (2021). "Datasheets for Datasets." Commun ACM.* ***64****(12):86–92. doi:10.1145/3458723.*
13. a, b*Mitchell M, Wu S, Zaldivar A, Barnes P, Vasserman L, Hutchinson B, Spitzer E, Raji ID, Gebru T (2019). "Model Cards for Model Reporting." Proceedings of the Conference on Fairness, Accountability, and Transparency (FAccT '19). doi:10.1145/3287560.3287596.*
14. a, b*Bender EM, Friedman B (2018). "Data Statements for Natural Language Processing: Toward Mitigating System Bias and Enabling Better Science." Trans Assoc Comput Linguist.* ***6****:587–604. doi:10.1162/tacl_a_00041.*
15. a, b*Pushkarna M, Zaldivar A, Kjartansson O (2022). "Data Cards: Purposeful and Transparent Dataset Documentation for Responsible AI." Proceedings of the ACM Conference on Fairness, Accountability, and Transparency (FAccT '22). doi:10.1145/3531146.3533231.*
16. a, b*Holland S, Hosny A, Newman S, Joseph J, Chmielinski K (2018). "The Dataset Nutrition Label: A Framework to Drive Higher Data Quality Standards."*
17. a, b, c*Pineau J, Vincent-Lamarre P, Sinha K, Larivière V, Beygelzimer A, d'Alché-Buc F, Fox E, Larochelle H (2021). "Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program)." J Mach Learn Res.* ***22****(164):1–20.*
18. ^*Xu J, Zhang Z, Friedman T, Liang Y, Van den Broeck G (2018). "A Semantic Loss Function for Deep Learning with Symbolic Knowledge." Proceedings of the 35th International Conference on Machine Learning (ICML), PMLR 80.*
19. a, b*Hokamp C, Liu Q (2017). "Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search." Proceedings of ACL 2017. doi:10.18653/v1/P17-1141.*

20. a, b *Cotter A, Jiang H, Sridharan K (2019). "Two-Player Games for Efficient Non-Convex Constrained Optimi zation." Algorithmic Learning Theory (ALT), Proceedings of Machine Learning Research 98:300–332.*
21. a, b *Adebayo J, Gilmer J, Muelly M, Goodfellow I, Hardt M, Kim B (2018). "Sanity Checks for Saliency Maps." N eurIPS.*
22. a, b *Gulrajani I, Lopez-Paz D (2021). "In Search of Lost Domain Generalization." International Conference on Learning Representations (ICLR).*

Supplementary data: available at https://doi.org/10.32388/TMC25F

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.