GeCo: **Quality Counterfactual Explanations in Real Time**

Maximilian Schleich
University of Washington
schleich@cs.washington.edu

Zixuan Geng
University of Washington
zg44@cs.washington.edu

Yihong Zhang
University of Washington
yz489@cs.washington.edu

Dan Suciu
University of Washington
suciu@cs.washington.edu

ABSTRACT

Machine learning is increasingly applied in high-stakes decision making that directly affect people's lives, and this leads to an increased demand for systems to explain their decisions. Explanations often take the form of *counterfactuals*, which consists of conveying to the end user what she/he needs to change in order to improve the outcome. Computing counterfactual explanations is challenging, because of the inherent tension between a rich semantics of the domain, and the need for real time response. In this paper we present GeCo, the first system that can compute plausible and feasible counterfactual explanations in real time. At its core, GeCo relies on a genetic algorithm, which is customized to favor searching counterfactual explanations with the smallest number of changes. To achieve real-time performance, we introduce two novel optimizations: Δ-representation of candidate counterfactuals, and partial evaluation of the classifier. We compare empirically GeCo against five other systems described in the literature, and show that it is the only system that can achieve both high quality explanations and real time answers.

PVLDB Reference Format:
Maximilian Schleich, Zixuan Geng, Yihong Zhang, and Dan Suciu. GeCo: Quality Counterfactual Explanations in Real Time. PVLDB, 14(9): 1681 - 1693, 2021.
doi:10.14778/3461535.3461555

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mjschleich/GeCo.jl.

1 INTRODUCTION

Machine learning is increasingly applied in high-stakes decision making that directly affects people's lives. As a result, there is a huge need to ensure that the models and their predictions are interpretable by their human users. Motivated by this need, there has been a lot of recent interest within the machine learning community in techniques that can explain the outcomes of models. Explanations improve the transparency and interpretability of the underlying model, they increase user's trust in the model predictions, and they are a key facilitator to evaluate the fairness of the model for underrepresented demographics. The ability to explain is no longer a nice-to-have feature, but is increasingly required by law; for example, the GDPR regulations grant users the *right to explanation* to automated decision algorithms [32]. In addition to supporting the end user, explanations can also be used by model developers to debug and monitor their ever more complex models.

In this paper we focus on *local explanations*, which provide post-hoc explanations for one single prediction, and are in contrast to *global explanations*, which aim to explain the entire model (e.g. for debugging purposes). In particular we study *counterfactual explanations*: given an instance $\boldsymbol{x}$, on which the machine learning model predicts a negative, "bad" outcome, the explanation says what needs to change in order to get the positive, "good" outcome, usually represented by a counterfactual example $\boldsymbol{x}_{cf}$. For example, a customer applies for a loan with a bank, the bank denies the loan application, and the customer asks for an explanation; the system responds by indicating what features need to change in order for the loan to be approved, see Fig. 1. In the AI literature, counterfactual explanations are currently considered the most attractive types of local explanations, even for models that are considered "interpretable", such as random forests or generalized linear models [12, 31–33], because they offer actionable feedback to the customers, and have been deemed satisfactory by some legislative bodies, for example they are deemed to satisfy the GDPR requirements [32].

The major challenge in computing a counterfactual explanation is the tension between a rich semantics on the one hand, and the need for real-time, interactive feedback on the other hand. The

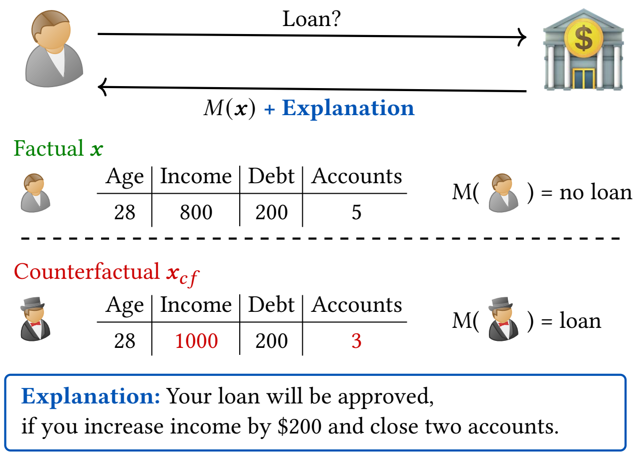
Loan?
$M(\boldsymbol{x})$ + Explanation
Factual $\boldsymbol{x}$
Age | Income | Debt | Accounts
28 | 800 | 200 | 5
M( ) = no loan
Counterfactual $\boldsymbol{x}_{cf}$
Age | Income | Debt | Accounts
28 | 1000 | 200 | 3
M( ) = loan
Explanation: Your loan will be approved, if you increase income by $200 and close two accounts.

Figure 1: Example of a counterfactual explanation scenario.

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461555

	Limited search space	Complete search space
Non-interactive	CERTIFAI [27]	MACE [12]
Interactive	What-If [33] DiCE [17]	GeCo (this paper)

Figure 2: Taxonomy of Counterfactual Explanation Systems

semantics needs to be rich in order to reflect the complexities of the real world. We want $\boldsymbol{x}_{cf}$ to be as close as possible to $\boldsymbol{x}$, but we also want $\boldsymbol{x}_{cf}$ to be *plausible*, meaning that its features should make sense in the real world. We also want the transition from $\boldsymbol{x}$ to $\boldsymbol{x}_{cf}$ to be *feasible*, for example *age* should only increase. The plausibility and feasibility constraints are dictated by laws, societal norms, application-specific requirements, and may even change over time; an explanation system must be able to support constraints with a rich semantics. Moreover, the search space for counterfactuals is huge, because there are often hundreds of features, and each can take values from some large domain. On the other hand, the computation of counterfactuals needs to be done at interactive speed, because the explanation system is eventually incorporated in a user interface. Performance has been identified as the main challenge for deployment of counterfactual explanations in industry [5, 33]. The tension between performance and rich semantics is the main technical challenge in counterfactual explanations. Previous systems either explore a complete search space with rich semantics, or answer at interactive speed, but not both: see Fig. 2 and discussions in Section 6. For example, on one extreme MACE [12] enforces plausibility and feasibility by using a general-purpose constraint language, but the solver often takes many minutes to find a counterfactual. At the other extreme, Google's What-if Tool (WIT) [33] restricts the search space to a fixed dataset of examples, ensuring fast response time, but poor explanations.

In this paper we present GeCo, the first interactive system for counterfactual explanations that supports a complex, real-life semantics of counterfactuals, yet provides answers in real time. At its core, GeCo defines a search space of counterfactuals using a plausibility-feasibility constraint language, PLAF, and a database D. PLAF is used to define constraints like "*age* can only increase", while the database D is used to capture correlations, like "*job-title* and *salary* are correlated". By design, the search space of possible counterfactuals is huge. To search this space, we make a simple observation. A good explanation $\boldsymbol{x}_{cf}$ should differ from $\boldsymbol{x}$ by only a few features; counterfactual examples $\boldsymbol{x}_{cf}$ that require the customer to change too many features are of little interest. Based on this observation, we propose a *genetic algorithm*, which we customize to search the space of counterfactuals by prioritizing those that have fewer changes. Starting from a population consisting of just the given entity $\boldsymbol{x}$, the algorithm repeatedly updates the population by applying the operations *crossover* and *mutation*, and then *selecting* the best counterfactuals for the new generation. It stops when it reaches a sufficient number of examples on which the classifier returns the "good" (desired) outcome. While counterfactual explanations can be applied to various data types, e.g. DiCE can explain image classifications [17], we focus on structured tabular data. Structured data is the predominant data type used in financial services, a key target industry for counterfactual explanations. We also assume that the model and data is static; the impact of updates on the explanations is an challenging direction for future work.

The main performance limitation in GeCo is its innermost loop. By the nature of the genetic algorithm, GeCo needs to repeatedly add and remove counterfactuals to and from the current population, and ends up having to examine thousands of candidates, and apply the classifier $M(\boldsymbol{x}')$ on each of them. We propose two novel optimizations to speedup the inner loop of GeCo: Δ-representation and classifier specialization via partial evaluation. In *Δ-representation* we group the current population by the set of features ΔF by which they differ from $\boldsymbol{x}$, and represent this entire subpopulation in a single relation whose attributes are only ΔF; for example all candidate examples $\boldsymbol{x}'$ that differ from $\boldsymbol{x}$ only by *age* are represented by a single-column table, storing only the modified *age*. This leads to huge memory savings over the naive representation (storing all features of all candidates $\boldsymbol{x}'$), which, in turn, leads to performance improvements. *Partial evaluation* specializes the code of the classifier M to entities $\boldsymbol{x}'$ that differ from $\boldsymbol{x}$ only in ΔF; for example, if M is a decision tree, then normally $M(\boldsymbol{x}')$ inspects all features from the root to a leaf, but if ΔF is *age*, then after partial evaluation it only needs to inspect *age*, for example M can be *age* > 30, or perhaps $30 <$ *age* < 60, because all the other features are constants within this subpopulation. Δ-representation and partial evaluation work well together, and, when combined, allows GeCo to compute the counterfactual in interactive time. At a high level, our optimizations belong to a more general effort that uses database and program optimizations to improve the performance of machine learning tasks (e.g., [3, 10, 14, 20, 25]).

We benchmarked GeCo against four counterfactual explanation systems: MACE [12], DiCE [17], What-If Tool [33], and CERTIFAI [27]. We show that in all cases GeCo produced the best quality explanation (using quality metrics defined in [12]), at interactive speed. We also conduct micro-experiments showing that the two optimizations, Δ-representation and partial evaluation, account for a performance improvement of up to 5×, and thus are critical for an interactive deployment of GeCo.

Discussion Explanation techniques can be broadly categorized as *white-box* or *black-box*. *White-box* explanations are designed for a specific class of models and domains, e.g., Krpyton [19] is specific to neural networks for image classification. These explanations exploit the structural properties of the classifier and typically cannot be applied across domains. While there is no commonly agreed definition of *black-box* explanations, we adopt a strict definition in this paper: the explanation is *black-box* if the classifier M is available only as an oracle that, when given an input $\boldsymbol{x}'$, returns the outcome $M(\boldsymbol{x}')$. Thus, black-box explanations do not enforce any restrictions on the underlying domain and classifier. This is particularly useful when the model is subject to IP restrictions or provided through an API, e.g., the Google Vision API. A black-box classifier makes it difficult for the system to compute an explanation that is *consistent* with the underlying classifier, meaning that $M(\boldsymbol{x}_{cf})$ returns the desired, "good", outcome. As an example, LIME [22] uses a black-box classifier, learns a simple, interpretable model locally, around the input data point $\boldsymbol{x}$, and uses it to obtain an explanation; however, its explanations are not always consistent with the predictions of the original classifier [23, 28].

A key advantage of counterfactual explanations is that they guarantee model consistency and they can be black-box. A black-box model, however, makes it difficult to explore the search space of counterfactuals, because we cannot examine the code of M for hints of how to quickly get from $\boldsymbol{x}$ to a counterfactual $\boldsymbol{x}_{cf}$. For this reason, several previous systems claim to be black box, but require access to the code of the classifier. We classify these systems as *gray-box*: A *gray-box* explanation has access to the code of the model M. Early counterfactual explanation systems are based on gradient descent [18, 32]. They are gray-box, because they need access to the code of M in order to compute its gradient. MACE [12] is also gray-box, since it translates both the classifier logic and the feasibility/plausibility constraints into a logical formula and then solves for counterfactuals via multiple calls to an SMT solver. These systems have similar restrictions as white-box explanations. In contrast, CERTIFAI [27] and Google's What-if Tool (WIT) [33] are fully black-box, but limit the search space (cf. Fig. 2). CERTIFAI is based on a genetic algorithm, but the quality of its explanations are highly sensitive to the computation of the initial population; we discuss CERTIFAI in detail in Sec. 4.7. WIT severely limits its search space to a given dataset.

Our system, GeCo, is designed to work with any black-box model and to explore the large search space of counterfactuals. Yet, if the code of the classifier is available, then we use the partial evaluation optimization to improve the runtime. Thus, GeCo differs from other systems in that it uses access to the code of M not to guide the search, but only to optimize the execution time.

Counterfactual explanations are related to the problem of finding adversarial examples [7, 9, 16, 29], which are small perturbations of the input instance that lead to a different classification. Such examples are used to evaluate the robustness of the classifier. A key difference is that adversarial examples typically have many, almost indistinguishable changes, which are not required to be plausible or feasible. In Sec. 6, we show that techniques for adversarial examples are not directly applicable to the problem of finding quality counterfactual explanations by comparing GeCo with an adaptation of the SimBA algorithm [9].

Contributions In summary, our contributions are as follows.

- We describe GeCo, the first system that computes feasible and plausible explanations in interactive response time.
- We describe the search space of GeCo consisting of a database D and a constraint language PLAF. Section 3.
- We describe a custom genetic algorithm for exploring the space of candidate counterfactuals. Section 4.
- We describe two optimization techniques: Δ-representation and partial evaluation. Section 5.
- We conduct an extensive experimental evaluation of GeCo, and compare it to MACE, Google's What-If Tool, CERTIFAI, and an adaptation of SimBA. Section 6.

2 COUNTERFACTUAL EXPLANATIONS

Consider n features $F_1, \ldots, F_n$, with domains $\text{dom}(F_1), \ldots, \text{dom}(F_n)$. We assume to have a black-box model M, i.e. an oracle that, when given a feature vector $\boldsymbol{x} = (x_1, \ldots, x_n)$, returns a prediction $M(\boldsymbol{x})$. The prediction is a number between 0 and 1, where 1 is the desired, or good outcome, and 0 is the undesired outcome. For simplicity, we will assume that $M(\boldsymbol{x}) > 0.5$ is "good", and everything else is "bad". If the classifier is categorical, then we simply replace its outcomes with the values $\{0, 1\}$. Given an instance $\boldsymbol{x}$ for which $M(\boldsymbol{x})$ is "bad", the goal of the counterfactual explanation is to find a counterfactual instance $\boldsymbol{x}_{cf}$ such that (1) $M(\boldsymbol{x}_{cf})$ is "good", (2) $\boldsymbol{x}_{cf}$ is close to $\boldsymbol{x}$, and (3) $\boldsymbol{x}_{cf}$ is both *feasible* and *plausible*. Formally, a counterfactual explanation is a solution to the following optimization problem:

$$\begin{aligned} &\arg\min_{\boldsymbol{x}_{cf}} \text{dist}(\boldsymbol{x}, \boldsymbol{x}_{cf}) && (1)\\ &\quad s.t.\ M(\boldsymbol{x}_{cf}) > 0.5 \\ &\quad\quad \boldsymbol{x}_{cf} \in \mathcal{P} && \texttt{// } x_{cf} \texttt{ is plausible} \\ &\quad\quad \boldsymbol{x}_{cf} \in \mathcal{F}(\boldsymbol{x}) && \texttt{// } x_{cf} \texttt{ is feasible} \end{aligned}$$

where dist is a distance function. The counterfactual explanation $\boldsymbol{x}_{cf}$ ranges over the space $\text{dom}(F_1) \times \cdots \times \text{dom}(F_n)$, subject to the plausibility and feasibility constraints $\mathcal{P}$ and $\mathcal{F}(\boldsymbol{x})$, which we discuss in the next section. In practice, as we shall explain, GeCo returns not just one, but the top k best counterfactuals $\boldsymbol{x}_{cf}$.

The role of the distance function is to ensure that GeCo finds the *nearest counterfactual* instance that satisfies the constraints. In particular, we are interested in counterfactuals that change the values of only a few features, which helps define concrete actions that the user can perform to achieve the desired outcome. For that purpose, we use the distance function $\text{dist}(\boldsymbol{x}, \boldsymbol{y})$ introduced by MACE [12], which we briefly review here.

We start by defining domain-specific distance functions $\delta_1, \ldots, \delta_n$ for each of the n features, as follows. If $\text{dom}(F_i)$ is categorical, then $\delta_i(x_i, y_i) \overset{\text{def}}{=} 0$ if $x_i = y_i$ and $\delta_i(x_i, y_i) \overset{\text{def}}{=} 1$ otherwise. If $\text{dom}(F_i)$ is a continuous or an ordinal domain, then $\delta_i(x_i, y_i) \overset{\text{def}}{=} |x_i - y_i|/w$, where w is the range of the domain. We note that, when the range is unbounded, or unknown, then alternative normalizations are possible, such as the Median Absolute Distance (MAD) [32], or the standard deviation [33]. We define the ℓ_p-distance between $\boldsymbol{x}, \boldsymbol{y}$ as:

$$\text{dist}_p(\boldsymbol{x}, \boldsymbol{y}) \overset{\text{def}}{=} \left(\sum_i \delta_i^p(x_i, y_i) \right)^{1/p}$$

and adopt the usual convention that the ℓ_0-distance is the number of distinct features: $\text{dist}_0(\boldsymbol{x}, \boldsymbol{y}) \overset{\text{def}}{=} |\{i \mid \delta_i(x_i, y_i) \neq 0\}|$. Finally, we define our distance function as a weighted combination of the ℓ_0, ℓ_1, and ℓ_∞ distances:

$$dist(\boldsymbol{x}, \boldsymbol{y}) = \alpha \cdot \frac{\text{dist}_0(\boldsymbol{x}, \boldsymbol{y})}{n} + \beta \cdot \frac{\text{dist}_1(\boldsymbol{x}, \boldsymbol{y})}{n} + \gamma \cdot \text{dist}_\infty(\boldsymbol{x}, \boldsymbol{y}) \quad (2)$$

where $\alpha, \beta, \gamma \geq 0$ are hyperparameters, which must satisfy $\alpha + \beta + \gamma = 1$. Notice that $0 \leq dist(\boldsymbol{x}, \boldsymbol{y}) \leq 1$. The intuition is that the ℓ_0-norm restricts the number of features that are changed, the ℓ_1 norm accounts for the average change of distance between $\boldsymbol{x}$ and $\boldsymbol{y}$, and the ℓ_∞ norm restricts the maximum change across all features. Although [12] defines the distance function (2), the MACE system hardwires the hyperparameters to $\alpha = 0, \beta = 1, \gamma = 0$; we discuss this, and the setting of different hyperparameters in Sec. 6.

3 THE SEARCH SPACE

The key to computing high quality explanations is to define a complete space of candidates that the system can explore. In GeCo the search space for the counterfactual explanations is defined by two components: a database of entities $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots\}$ and plausibility and feasibility constraint language called PLAF.

The database D can be the training set, a test set, or historical data of past customers for which the system has performed predictions. It is used in two ways. First, GeCo computes the domain of each feature as the active domain in D: $\text{dom}(F_i) \overset{\text{def}}{=} \Pi_{F_i}(D)$. Second, the data analyst can specify *groups* of features with the command:

$$\texttt{GROUP}\ F_{i_1}, F_{i_2}, \ldots$$

and in that case the joint domain of these features is restricted to the combination of values found in D, i.e. $\Pi_{F_{i_1} F_{i_2} \cdots}(D)$. Grouping is useful in several contexts. The first is when the attributes are correlated. For example, if we have a functional dependency like `zip` → `city`, then the data analyst would group `zip`, `city`. For another example of correlation, consider `education` and `income`. They are correlated without satisfying a strict functional dependency; by grouping them, the data analyst ensures that GeCo considers only combinations of values found in the dataset D. As a final example, consider attributes that are the result of one-hot encoding: e.g. `color` may be expanded into `color_red`, `color_green`, `color_blue`. By grouping them together, the analyst restricts GeCo to consider only values $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ that actually occur in D.

The constraint language PLAF allows the data analyst to specify which combination of features of $\boldsymbol{x}_{cf}$ are plausible, and which can be feasibly reached from $\boldsymbol{x}$. PLAF consists of statements of the form:

$$\texttt{PLAF IF}\ \Phi_1\ \texttt{and}\ \Phi_2\ \texttt{and}\ \cdots\ \texttt{THEN}\ \Phi_0 \quad (3)$$

where each Φ_i is an atomic predicate of the form e_1 op e_2 for op $\in \{=, \neq, \leq, <, \geq, >\}$, and each expression e_1, e_2 is over the *current* features, denoted `x.`F_i, and/or *counterfactual* features, denoted `x_cf.`F_i. The `IF` part may be missing.

Example 3.1. Consider the following PLAF specification:

```
GROUP education, income                    (4)
PLAF x_cf.gender = x.gender                (5)
PLAF x_cf.age >= x.age                     (6)
PLAF IF x_cf.education > x.education
     THEN x_cf.age > x.age+4               (7)
```

The first statement says that `education` and `income` are correlated: GeCo will consider only counterfactual values that occur together in the data. Rule (5) says that `gender` cannot change, rule (6) says that `age` can only increase, while rule (7) says that, if we ask the customer to get a higher education degree, then we should also increase the `age` by 4. The last rule (7) is adapted from [17], who have argued for the need to restrict counterfactuals to those that satisfy *causal* constraints.

PLAF has the following restrictions. (1) The groups have to be disjoint: if a feature F_i needs to be part of two groups then they need to be union-ed into a larger group. We denote by $G(F_i)$ the unique group containing F_i (or $G(F_i) = \{F_i\}$ if F_i is not explicitly included in any group). (2) the rules must be acyclic, in the following sense. Every consequent Φ_0 in (3) must be of the form `x_cf.`F_i op e, i.e. must "define" a counterfactual feature F_i, and the following graph must be acyclic: the nodes are the groups $G(F_1), G(F_2), \ldots$, and the edges are $(G(F_j), G(F_i))$ whenever there is a rule (3) that defines `x_cf.`F_i and that also contains `x_cf.`F_j. We briefly illustrate with Example 3.1. The three rules (5)-(7) "define" the features `gender`, `age`, and `age` respectively, and there there is a single edge {`education, income`} → {`age`}, resulting from Rule (7). Therefore, the PLAF program is acyclic.

```
explain (instance $x$, classifier $M$, dataset $D$, PLAF $(\Gamma, C)$)
$C_x$ = ground($x$, C);    DG = feasibleSpace($D, \Gamma, C_x$);
POP = [ $(x, \emptyset)$ ]
POP = mutate(POP, $\Gamma$, DG, $C_x$, $m_{init}$)        //initial population
POP = selectFittest($x$, POP, $M$, $q$)
do {
    CAND = crossover(POP, $C_x$) $\cup$ mutate(POP, $\Gamma$, DG, $C_x$, $m_{mut}$)
    POP = selectFittest($x$, POP $\cup$ CAND, $M$, $q$)
    TOPK = POP[1 : $k$]
} until ( counterfactuals(TOPK, $M$) and TOPK $\cap$ CAND = $\emptyset$ )
return TOPK
```

Algorithm 1: Pseudo-code of GeCo**'s custom genetic algorithm to generate counterfactual explanations.**

The restrictions are not limiting the expressive power of PLAF, but only encourage users to write constraints that GeCo can evaluate efficiently. Indeed, consider any CNF formula $C_1 \wedge C_2 \wedge \cdots$ where each clause C_i has the form $\Phi_1 \vee \Phi_2 \vee \cdots$ If Φ'_i is the negation of Φ_i (e.g. the negation of $e_1 \leq e_2$ is $e_1 > e_2$), then we can write the clause as the PLAF statement $\Phi'_1 \wedge \Phi'_2 \wedge \cdots \Rightarrow (1 = 2)$, where $1 = 2$ stands for `false`. The PLAF program is acyclic because the precedence graph has no edges, but it would force GeCo to search for counterfactuals through rejection sampling only. Instead, by encouraging users to write constraints where each counterfactual feature is defined by a rule as explained above, we reduce the need for, or completely avoid rejection sampling.

4 GECO'S CUSTOM GENETIC ALGORITHM

In this section, we introduce the custom genetic algorithm that GeCo uses to efficiently explore the space of counterfactual explanations. A genetic algorithm is a meta-heuristic for constraint optimization that is based on the process of natural selection. There are four core operations. First, it defines an **initial population** of candidates. Then, it iteratively selects the **fittest** candidates in the population, and generates new candidates via **mutate** and **crossover** on the selected candidates, until convergence.

While there are many optimization algorithms that could be used to solve our optimization problem (defined in Eq (1)), we chose a genetic algorithm for the following reasons: (1) The genetic algorithm is easily customizable to the problem of finding counterfactual explanations; (2) it seamlessly supports the rich semantics of PLAF constraints, which are necessary to ensure that the explanations are feasible and plausible; (3) it does not require any restrictions on the underlying classifier and data, and thus is able to provide black-box explanations; and (4) it returns a diverse set of explanations, which may provide different actions that can lead to the desired outcome.

In GeCo, we customized the core operations of the genetic algorithm based on the following key observation: A good explanation $\boldsymbol{x}_{cf}$ should differ from $\boldsymbol{x}$ by only a few features; counterfactual examples $\boldsymbol{x}_{cf}$ that require the customer to change too many features are of little interest. For this reason, GeCo first explores counterfactuals that change only a single feature group, before exploring increasingly more complex action spaces in subsequent generations.

In the following, we first overview of the genetic algorithm used by GeCo and then provide additional details for the core operations.

4.1 Overview

GeCo's pseudocode is shown in Algorithm 1. The inputs are: an instance $\boldsymbol{x}$, the black-box classifier M, a dataset D, and PLAF program (Γ, C). Here $\Gamma = \{G_1, G_2, \ldots\}$ are the groups of features, and $C = \{C_1, C_2, \ldots\}$ are the PLAF constraints, see Sec. 3. The algorithm has four integer hyperparameters $k, m_{\text{init}}, m_{\text{mut}}, q > 0$, with the following meaning: k represents the number of counterfactuals that the algorithm returns to the user; q is the size of the population that is retained from one generation to the next; and m_{init}, m_{mut} control the number of candidates that are generated for the initial population, and during mutation respectively. We always set $k < q$.

As explained in Sec. 3, the active domain of each attribute are values found in the database D. More generally, for each group $G_i \in \Gamma$, its values must be sampled together from those in the database D. The GeCo algorithm starts by grounding (specializing) the PLAF program C to the entity $\boldsymbol{x}$ (the **ground** function), then calls the **feasibleSpace** operator, which computes for each group G_i a relation DG_i with attributes G_i representing the sample space for the group G_i; we give details in Sec. 4.2.

Next, GeCo computes the initial population. In our customized algorithm, the initial population is obtained simply by applying the mutate operator to the given entity $\boldsymbol{x}$. Throughout the execution of the algorithm, the population is a set of pairs $(\boldsymbol{x}', \Delta')$, where $\boldsymbol{x}'$ is an entity and Δ' is the set of features that were changed from $\boldsymbol{x}$. Throughout this section we call the entities $\boldsymbol{x}'$ in the population *candidates*, or *examples*, but we don't call them counterfactuals, unless $M(\boldsymbol{x}')$ is "good", i.e. $M(\boldsymbol{x}') > 0.5$ (see Sec. 2). In fact, it is possible that none of the candidates in the initial population are classified as good. The goal of GeCo is to find at least k counterfactuals through a sequence of mutation and crossover operation.

The main loop of GeCo's genetic algorithm consists of extending the population with new candidates obtained by mutation and crossover, then keeping only the q fittest for the next generation. The operators **selectFittest**, **mutate**, **crossover** are described in Sec. 4.3, 4.4, and 4.5. The algorithm stops when the top k (from the select q fittest) candidates are all counterfactuals *and* are stable from one generation to the next; the function **counterfactuals** simply tests that all candidates are counterfactuals, by checking[1] that $M(\boldsymbol{x}')$ is "good". We describe now the details of the algorithm.

[1]In our implementation we store the value $M(\boldsymbol{x}')$ together with $\boldsymbol{x}'$, so we don't have to compute it repeatedly. We omit some details for simplicity of the presentation.

4.2 Feasible Space Operators

Throughout the execution of the genetic algorithm, GeCo ensures that all candidates $\boldsymbol{x}'$ satisfy all PLAF constraints C. It achieves this efficiently through three functions: **ground**, **feasibleSpace**, and **actionCascade**. We describe these functions here, and omit their pseudocode (which is straightforward).

The function **ground**$(\boldsymbol{x}, C)$ simply instantiates all features of $\boldsymbol{x}$ with constants, and returns "grounded" constraints $C_{\boldsymbol{x}}$. All candidates $\boldsymbol{x}'$ will need to satisfy these grounded constraints.

Example 4.1. Let C be the three constraints of the PLAF program in Example 3.1. Assume that the instance is:

$$\boldsymbol{x} = (\texttt{gender} = \texttt{female}, \texttt{age} = 22, \texttt{education} = 3, \texttt{income} = 80k)$$

Then the set of grounded rules $C_{\boldsymbol{x}}$ is obtained from the rules (5)-(7). For example, `x_cf.age >= x.age` becomes `age` ≥ 22. The three grounded rules are:

$$\texttt{gender} = \texttt{female} \tag{8}$$

$$\texttt{age} \geq 22 \tag{9}$$

$$\texttt{education} > 3 \Rightarrow \texttt{age} > 26 \tag{10}$$

Every candidate $\boldsymbol{x}'$ must satisfy all three rules.

A naive strategy to generate candidates $\boldsymbol{x}'$ that satisfy $C_{\boldsymbol{x}}$ is through rejection sampling: after each mutation and/or crossover, we verify $C_{\boldsymbol{x}}$, and reject $\boldsymbol{x}'$ if it fails some constraint in $C_{\boldsymbol{x}}$. GeCo improves over this naive approach in two ways. First, it computes for each group $G_i \in \Gamma$ a set of values DG_i that, in isolation, satisfy $C_{\boldsymbol{x}}$; this is precomputed at the beginning of the algorithm by **feasibleSpace**. Second, once it generates candidates $\boldsymbol{x}'$ that differ in multiple feature groups $G_{i_1}, G_{i_2}, \ldots$ from $\boldsymbol{x}$, then it enforces the constraint by possibly applying additional mutation, until all constraints hold: this is done by the function **actionCascade**.

The function **feasibleSpace**$(D, \Gamma, \mathtt{C}_{\boldsymbol{x}})$ computes, for each group $G_i \in \Gamma$, the relation:

$$DG_i \overset{\text{def}}{=} \sigma_{C_{\boldsymbol{x}}^{(i)}} \left(\Pi_{G_i}(D)\right)$$

where $C_{\boldsymbol{x}}^{(i)}$ consists of the conjunction of all rules in $C_{\boldsymbol{x}}$ that refer only to features in the group G_i. We call the relation DG_i the *sample space* for the group G_i. Thus, the selection operator $\sigma_{C_{\boldsymbol{x}}^{(i)}}$ rules out values in the sample space that violate of some PLAF rule.

Example 4.2. Continuing Example 4.1, there are three groups, $G_1 = \{\texttt{gender}\}$, $G_2 = \{\texttt{education}, \texttt{income}\}$, $G_3 = \{\texttt{age}\}$, and their sample spaces are computed as follows:

$$DG_1 \overset{\text{def}}{=} \sigma_{\texttt{gender=female}}(\Pi_{\texttt{gender}}(D))$$

$$DG_2 \overset{\text{def}}{=} \Pi_{\texttt{education,income}}(D))$$

$$DG_3 \overset{\text{def}}{=} \sigma_{\texttt{age}\geq 22}(\Pi_{\texttt{age}}(D))$$

Notice that we could only check the grounded rules (8) and (9). The rule (10) refers to two different groups, and can only be checked by examining features from two groups; this is done by the function **actionCascade**.

The function **actionCascade**$(\boldsymbol{x}', \Delta', \mathtt{C}_{\boldsymbol{x}})$ is called during mutation and crossover, and its role is to enforce the grounded constraints $C_{\boldsymbol{x}}$ on a candidate $\boldsymbol{x}'$ before it is added to the population.

```
mutate (POP, $\Gamma$, DG, $C_x$, $m$)
CAND = $\emptyset$
foreach $(x^*, \Delta^*) \in$ POP do {
    foreach $G \in \Gamma$ s.t. $G \notin \Delta^*$ do {
        $x' = x^*$;  $S$ = sample(DG[$G$], $m$)  //without replacement
        foreach $v \in S$ do {
            $x'.G = v$;    $\Delta' = \Delta^* \cup \{G\}$
            $(x', \Delta')$ = actionCascade($x', \Delta'$, $C_x$)
            CAND $\cup= \{(x', \Delta')\}$
        }
    }
}
return CAND
```

Algorithm 2: Pseudo-code for GeCo**'s mutate operator.**

```
crossover (POP, $C_x$)
CAND = $\emptyset$;    $\{\Delta_1, \ldots, \Delta_r\} = \{\Delta \mid (x, \Delta) \in$ POP$\}$
foreach $(\Delta_i, \Delta_j) \in \{\Delta_1, \ldots, \Delta_r\}$ s.t. $i < j$ do {
    let $(x_i, \Delta_i)$ = best instance in POP with $\Delta = \Delta_i$
    let $(x_j, \Delta_j)$ = best instance in POP with $\Delta = \Delta_j$
    //combine actions from $x_i$ and $x_j$
    $x' = x_i$;    $\Delta' = \Delta_i \cup \Delta_j$
    foreach $G \in \Delta'$ do {
        if $G \in \Delta_i \setminus \Delta_j$ do $x'.G = x_i.G$
        elseif $G \in \Delta_j \setminus \Delta_i$ do $x'.G = x_j.G$
        else $x'.G$ = rand(Bool) ? $x_i.G : x_j.G$
    }
    $(x', \Delta')$ = actionCascade($x', \Delta'$, $C_x$)
    CAND $\cup= \{(x', \Delta')\}$
}
return CAND
```

Algorithm 3: Pseudo-code for GeCo**'s crossover operator.**

Recall from Sec. 3 that the PLAF rules are acyclic. **actionCascade** checks each rule, in topological order of the acyclic graph: if the rule is violated, then it changes the feature defined by that rule to a value that satisfies the condition, and it adds the updated feature (or group) to Δ'. The function returns the updated candidate x', which now satisfies all rules C_x, as well as its set of changed features Δ'. For a simple illustration, referring to Example 4.2, when GeCo considers a new candidate x', it checks the rule (10): if the rule is violated, then it replaces age with a new value from DG_3, subject to the additional condition age > 26, and adds age to Δ'.

4.3 Selecting Fittest Candidates

At each iteration, GeCo's genetic algorithm extends the current population (through mutation and crossover), then retains only the "fittest" q candidates x' for the next generation using the **selectFittest**(x, POP, M, q) function. The function first evaluates the fitness of each candidate x' in POP; then sorts the examples x' by their fitness score; and returns the top q candidates.

We describe here how GeCo computes the fitness of each candidate x'. For that, GeCo takes into account two pieces of information: whether x' is counterfactual, i.e. $M(x') > 0.5$, and the distance from x to x' which is given by the function $dist(x, x')$ defined in Sec. 2 (see Eq. (2)). It combines these two pieces of information into a single numerical value, $score(x')$ defined as:

$$score(x') = \begin{cases} dist(x, x') & \text{if } M(x') > 0.5 \\ (dist(x, x') + 1) + (1 - M(x')) & \text{otherwise} \end{cases} \quad (11)$$

The rationale for this score is that we want every counterfactual candidate to be better than every non-counterfactual candidate. If $M(x') > 0.5$, then x' is counterfactual, in that case it remains to minimize $dist(x, x')$. But if $M(x') \leq 0.5$, then x' is not counterfactual, and we add 1 to $dist(x, x')$ ensuring that this term is larger than any distance $dist(x, x'')$ for any counterfactual x'' (because $dist \leq 1$, see Sec. 2); we also add a penalty $1 - M(x')$, to favor candidates x' that are closer to becoming counterfactuals.

In summary, the function **selectFittest** computes the fitness score (Eq. (11)) for each candidate in the population, then sorts the population in decreasing order by this score, and returns the q fittest candidates. Notice that all counterfactuals examples x' will precede all non-counterfactuals in the sorted order.

4.4 Mutation Operator

Algorithm 2 provides the pseudo-code of the mutation operator. The operator **mutate**(POP, Γ, DG, C_x, m) takes as input the current population POP, the list of feature groups $\Gamma = \{G_1, G_2, \ldots\}$, their associated sample spaces DG $= \{DG_1, DG_2, \ldots\}$, the grounded constraints C_x, and an integer $m > 0$. The function generates, for each candidate x' in the population, m new mutations for each feature group G_i. We briefly explain the pseudo-code. For each candidate $(x^*, \Delta^*) \in$ POP, and for each feature group $G \in \Gamma$ that has not been mutated yet ($G \notin \Delta^*$), we sample m values without replacement from the sample space associated to G, and construct a new candidate x' obtained by changing G to v, for each value v in the sample. As explained earlier (Sec. 4.2), before we insert x' in the population, we must enforce all constraints in C_x, and this is done by the function **actionCascade**.

In GeCo, the mutation operator also generates the initial population. In this case, the current population POP contains only the original instance x with $\Delta = \emptyset$. Thus, GeCo first explores candidates in the initial population that differ from the original instance x only in a change for one group $G \in \Gamma$. This ensures that we prioritize candidates that have few changes. Subsequent calls to the mutation operator than change one additional feature group for each candidate in the current population.

4.5 Crossover Operator

The crossover operator generates new candidates by combining the actions of two instances x_i, x_j in POP. For example, consider candidates x_i and x_j that differ from x in the feature groups {age}, and {education, income} respectively. Then, the crossover operator generates a new candidate x' that changes all three features age, education, income, such that that x'.age = x_i.age and x'.{education, income} = x_j.{education, income}. The pseudo-code of the operator is given by Algorithm 3.

In a conventional genetic algorithm, the candidates for crossover are selected at random. In GeCo, however, we want to combine candidates that (1) change a distinct set of features, and (2) are the best candidates amongst all candidates in POP that change the same set of features. Recall that, for each candidate $\boldsymbol{x}'$ in the population, we also store the set Δ' of feature groups where $\boldsymbol{x}'$ differs from $\boldsymbol{x}$. To achieve our customized crossover, we first collect all distinct sets Δ in POP. Then, we consider any combination of two distinct sets (Δ_i, Δ_j), and find the candidates $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ which have the best fitness score (Eq. (11)) in the subset of POP for which $\Delta = \Delta_i$ and respectively $\Delta = \Delta_j$. Since POP is already sorted by the fitness score, the best candidate with $\Delta = \Delta_i$ is also the first candidate in POP with $\Delta = \Delta_i$. The operator then combines the candidates $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ into a new candidate $\boldsymbol{x}'$, which inherits both the mutations from $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$; if a feature group G was mutated in both $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, then we choose at random to assign to $\boldsymbol{x}'.G$ either $\boldsymbol{x}_i.G$ or $\boldsymbol{x}_j.G$. Finally, we check if $\boldsymbol{x}'$ requires any cascading actions, and apply them in order to satisfy the constraints $C_{\boldsymbol{x}}$. We add $\boldsymbol{x}'$ and the corresponding feature set Δ' to the collection of new candidates CAND.

4.6 Discussion

In this section, we discuss implementation details and potential tradeoffs that we face for the performance optimization of GeCo.

Hyperparameters. We use the following defaults for the hyperparameters: $q = 100, k = 5, m_{\text{init}} = 20, m_{\text{mut}} = 5$. The first parameter means that each generation retains the fittest $q = 100$ candidates. The value $k = 5$ means that the algorithm runs until the top 5 candidates of the population are all counterfactuals (i.e. $M(\boldsymbol{x}) > 0.5$) *and* they remain in the top 5 during two consecutive generations. The initial population consists of 20 candidates $\boldsymbol{x}'$ *per feature group*, and, during mutation, we create 5 new candidates per feature group. These defaults ensure that selected candidates of the initial population will change at least five distinct feature groups.

Representation of Δ. For each candidate $(\boldsymbol{x}', \Delta') \in$ POP, we represent the feature set Δ' as a compact bitset, which allows for efficient bitwise operations.

Sampling Operator. In the mutation operator, we use weighted sampling to sample actions from the sample space DG_i associated to the group G_i. The weight is given by the frequency of each value $v \in DG_i$ in the input dataset D. As a result, GeCo is more likely to sample those actions that frequently occur in the dataset D, which helps to ensure that the generated counterfactual is plausible.

Number of Mutated Candidates. By default the mutation operator mutates every candidate in the current population, which ensures that (1) we explore a large set of candidates, and (2) we sample many values from the feasible space for each group. The downside is that this operation can take a long time, in particular if we return many selected candidates (q is large), and the mutation operator can become a performance bottleneck. In this case, it is possible to mutate only a selected number of candidates. We propose to group the candidates by the set Δ, and then to mutate only the top candidates in each group (just like we only do crossover on the top candidate in each group). This approach ensures that we still explore candidates with the Δ sets as the default, but limits the total number of candidates that are generated.

Large Sample Spaces. If there is a very large sample space DG_i, then it is possible that GeCo does not sample the best action from this space. In this case, we designed *selectiveMutate*, a variant of the mutate operator. Whereas the normal mutation operator mutates candidates by changing feature groups that have not been changed, selective mutation mutates feature groups that were previously changed by sampling actions that decrease the distance between the candidate and the original instance. This ensures that GeCo is more likely to identify the best action in large sample spaces.

4.7 Comparison with CERTIFAI

CERTIFAI [27] also computes counterfactual explanations using a genetic algorithm. The main difference that distinguishes CERTIFAI from GeCo is that CERTIFAI assumes that the initial population is a random sample of only "good" counterfactuals. Once this initial population is computed, the goal of its genetic algorithm is to find better counterfactuals, whose distance to the original instance is smaller. However, it is unclear how to compute the initial population of counterfactuals, and the quality of the final answer of CERTIFAI depends heavily on the choice of this initial population. CERTIFAI also does not emphasize the exploration of counterfactuals that change only few features. In contrast, GeCo starts from only the original instance $\boldsymbol{x}$, whose outcome is "bad", and assumes that some "good" counterfactual is nearby, i.e. with few changed features.

Since CERTIFAI is not publicly available, we implemented our own variant based on the description in the paper. Since it is not clear how the initial population is computed, we consider all instances in the database D that are classified with the good outcome and satisfy all feasibility and plausibility constraints. This is the most efficient way to generate a sample of good counterfactuals, but it has the downside is that the quality of the initial population, and, hence, of the final explanation, varies widely with the instance $\boldsymbol{x}$. In fact, there are cases where D contains no counterfactual that satisfies the feasibility constraints w.r.t. $\boldsymbol{x}$. In Section 6, we show that GeCo is able to compute explanations that are closer to the original instance significantly faster than CERTIFAI.

5 OPTIMIZATIONS

The main performance limitation in GeCo are the repeated calls to **selectFittest** and **mutate**. Between the two operations, GeCo repeatedly adds tens of thousands of candidates to the population, applies the classifier M on each of them, and then removes those that have a low fitness score. In Section 6, we show that selection and mutation account for over 95% of the overall runtime. In order to apply GeCo in interactive settings, it is thus important to optimize the performance of these two operators. In this section, we present two optimizations which significantly improve their performance.

To optimize **mutate**, we present a loss-less, compressed data representation, called Δ-representation, for the candidate population that is generated during the genetic algorithm. In our experiments, the Δ-representation is up to 25× more compact than an equivalent naive listing representation, which translates to a performance improvement of up to 3.9× for mutate.

To optimize **selectFittest**, we draw on techniques from the PL community, in particular *partial evaluation*, to optimize the evaluation of the classifier. The optimizations exploit the fact that we know the values for subsets of the input features before the evaluation, which allows us to translate the model into a specialized equivalent model that pre-evaluates the static components. These optimizations improve the runtime for selectFittest by up to 3.2×.

Age	Edu.	Occup.	Income
28	HS	Service	10,000
30	HS	Service	10,000
32	HS	Service	10,000
22	PhD	Service	10,000
22	BSc	Service	10,000
25	HS	Service	15,000
23	HS	Service	20,000
22	HS	Sales	10,000
22	HS	Student	10,000

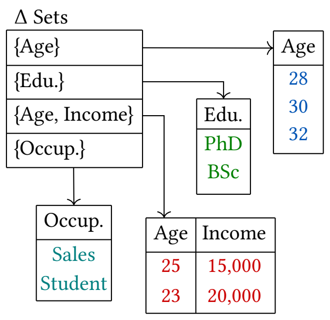

Figure 3: Example candidate population for instance x = (22,HS,Service,10000) using the naive listing representation (left) and the equivalent Δ-representation (right).

The Δ-representation and partial evaluation complement each other, and together decrease the end-to-end runtime of GeCo by a factor of 5.2×. Next, we provide more details for our optimizations.

5.1 Δ-Representation

The naive representation for the candidate population of the genetic algorithm is a listing of the full feature vectors $\boldsymbol{x}'$ for all candidates $(\boldsymbol{x}', \Delta')$. This representation is highly redundant, because most values in $\boldsymbol{x}'$ are equal to the original instance $\boldsymbol{x}$; only the features in Δ' are different. The Δ-*representation* can represent each the candidate $(\boldsymbol{x}', \Delta')$ compactly by storing only the features in Δ'. This is achieve by grouping the candidate population by the set of features Δ', and then representing the entire subpopulation in a single relation $R_{\Delta'}$ whose attributes are only Δ'.

Example 5.1. Figure 3 presents a candidate population for the instance $\boldsymbol{x}$ = (Age=22, Edu=HS, Occup=Service, Income=10000) using (left) the naive listing representation and (right) the equivalent Δ-representation. For simplicity, we highlight the changed values in the listing representation, instead of enumerating all Δ sets,

Most values stored in the listing representation are values from $\boldsymbol{x}$. In contrast, the Δ-representation only represents the values that are different from $\boldsymbol{x}$. For instance, the first three candidates, which change only Age, are represented in a relation with attributes Age only, and without repeating the values for Edu, Occup, Income.

In our implementation, we represent the Δ sets as bitsets, and the Δ-representation is a hashmap of that maps the distinct Δ sets to the corresponding relation R_Δ, which is represented as a DataFrame. We provide wrapper functions so that we can apply standard DataFrame operations directly on the Δ-representation.

The Δ-representation has a significantly lower memory footprint, which can lead to significant performance improvements over the naive representation, because it is more efficient to add candidates to the smaller relations, and it simplifies garbage collection.

There is, however, a potential performance tradeoff for the selection operator, because the classifier M typically assumes as input the full feature vector $\boldsymbol{x}'$. In this case, we copy the values in the Δ-representation to a full instantiation of $\boldsymbol{x}'$. This transformation can be expensive, but, in our experiments, it does not outweigh the speedup for mutation. In the next section, we show how we can use partial evaluation to avoid the construction of the full $\boldsymbol{x}'$.

5.2 Partial Evaluation for Classifiers

We show how to adapt PL techniques, in particular code specialization via partial evaluation, to optimize the evaluation of a given classifier M, and thus speedup the performance of **selectFittest**.

Consider a program $P : X \times Y \rightarrow O$ which maps two inputs (X, Y) into output O. Assume that we know $Y = y$ at compile time. Partial evaluation takes program P and input $Y = y$ and generates a more efficient program $P'_{\langle y \rangle} X \rightarrow O$, which precomputes the static components. Partial evaluation guarantees that $P'_{\langle y \rangle}(x) = P(x, y)$ for all $x \in \text{dom}(X)$. See [11] for more details on partial evaluation.

We next overview how we use partial evaluation in GeCo. Consider a classifier M with features F. During the evaluation of candidate $(\boldsymbol{x}', \Delta')$, we know that the values for all features $F \setminus \Delta'$ are constants taken from the original instance $\boldsymbol{x}$. Thus, we can partially evaluate the classifier M to a simplified classifier $M_{\Delta'}$ that precomputes the static components related to features $F \setminus \Delta'$. Once $M_{\Delta'}$ is generated, we cache the model so that we can apply it for all candidates in the population that change the same feature set Δ'. Note that by using partial evaluation, GeCo no longer explains a black-box, since it requires access to the code of the classifier.

Example 5.2. Consider a decision tree classifier M. For an instance $(\boldsymbol{x}', \Delta')$, $M(\boldsymbol{x}')$ typically evaluates the decisions for all features along a root to leaf path. Since we know the values for the features $F \setminus \Delta'$, we can precompute and fold all nodes in the tree that involve the features $F \setminus \Delta'$. If Δ' is small, then partial evaluation can generate a very simple tree. For instance, if $\Delta' = \{\text{Age}\}$ then $M_{\Delta'}$ only needs to evaluate decisions of the form $age > 30$ or $30 < age < 60$ to classify the input.

In addition to optimizing the evaluation of M, a partially evaluated classifier M_Δ can be directly evaluated over the partial relation R_Δ in the Δ-representation, and thus we mitigate the overhead resulting from the need to construct the full entity for the evaluation.

Partial evaluation has been studied and applied in various domains, e.g., in databases, it has been used to optimize query evaluation (see e.g., [26, 30]). We are, however, not aware of a general-purpose partial evaluator that be applied in GeCo to optimize arbitrary classifiers. Thus, we implemented our own partial evaluator for two model classes: (1) tree-based models, which includes decision trees, random forests, and gradient boosted trees, and (2) neural networks and multi-layered perceptrons.

In the following, we briefly introduce the partial evaluation we use for tree-based models and neural networks.

Tree-based models. We optimize the evaluation of tree-based models in two steps. First, we use existing techniques to turn the model into a more optimized representation for evaluation. Then, we apply partial evaluation on the optimized representation.

Tree-based models face performance bottlenecks during evaluation because, by nature of their representation, they are prone to cache misses and branch misprediction. For this reason, the ML systems community has studied how tree-based models can be represented so that they can be evaluated without random lookups in memory and repeated if-statements (see e.g., [4, 15, 34]). In

Table 1: Key characteristics for each considered dataset.

	Credit	Adult	Allstate	Yelp
Data Points	30K	45K	13.2M	22.4M
Variables	14	12	29	34
Features (one-hot enc.)	14	42	548	764
Feature Groups	14	11	29	34
Constraints / Implications	7 / 2	7 / 1	0 / 0	18 / 2

GeCo, we use the representation proposed by the QuickScorer algorithm [15], which we briefly overview next.

Instead of evaluating a decision tree T following a root to leaf path, QuickScorer evaluates *all* decision nodes in T and keeps track of which leaves cannot be reached whenever a decision fails. Once all decisions are evaluated, the prediction is guaranteed to be given by the first leaf that can be reached. All operations in QuickScorer use efficient, cache-conscious bitwise operations, and avoid branch mispredictions. This makes the QuickScorer very efficient, even if it evaluates many more decisions than the naive evaluation.

For partial evaluation, we exploit the fact that the first phase in QuickScorer computes the decision nodes for each feature independently of all other features. Thus, given a set of fixed feature values, we can precompute all corresponding decisions, and significantly number of decision that are evaluated at runtime.

Neural Networks and MLPs. We consider neural networks and multi-layered perceptrons for structured, tabular data (as opposed to images or text). In this setting, each hidden node N in the first layer of the network is typically a linear model [2, 17]. Given an input vector $\boldsymbol{x}$, parameter vector $\boldsymbol{w}$, and bias term b, the node N thus computes: $y = \sigma(\boldsymbol{x}^\top \boldsymbol{w} + b)$, where σ is an activation function. If we know that some values in $\boldsymbol{x}$ are static, then we can apply partial evaluation for N by precomputing the product between $\boldsymbol{x}$ and $\boldsymbol{w}$ for all static components and adding the partial product to the bias term b. During evaluation, we then only need to compute the product between $\boldsymbol{x}$ and $\boldsymbol{w}$ for the non-static components.

The impact of partial evalaution depends on the network structure. When applied to structured data, the model typically consists of fully-connected layers, in which case we can only partially evaluate the first layer of the network. We can apply partial evaluation to subsequent layers, only if the layers are not fully-connected.

6 EXPERIMENTS

We present the results for our experimental evaluation of GeCo on four real datasets. We conduct the following experiments:

(1) We investigate whether GeCo is able to compute counterfactual explanations for one end-to-end example, and compare the explanation with five existing systems.
(2) We benchmark all considered systems on 5,000 instances, and investigate the tradeoff between the quality of the explanations and the runtime for each system.
(3) We conduct microbenchmarks for GeCo. In particular, we breakdown the runtime into individual components and investigate the impact of the optimizations from Section 5. We also evaluate GeCo's ability to find the optimal explanation using synthetic classifiers.

6.1 Experimental Setup

In this section, we present the considered datasets and systems, as well as the setup used for all our experiments.

Datasets. We consider four real datasets: (1) *Credit* [35] is used predict customer's default on credit card payments in Taiwan; (2) *Adult* [13] is used to predict whether the income of adults exceeds $50K/year using US census data from 1994; (3) *Allstate* is a Kaggle dataset for the Allstate Claim Prediction Challenge [1], used to predict insurance claims based on the characteristics of the insured's vehicle; (4) *Yelp* is based on the public Yelp Dataset Challenge [36] and is used to predict review ratings that users give to businesses.

Table 1 presents key statistics for each dataset. Credit and Adult are from the UCI repository [8] and commonly used to evaluate explanations (e.g., [12, 18, 31]). For all datasets, we one-hot encode the categorical variables. For the evaluation with existing systems, we further apply the same preprocessing that was proposed by the existing system, in order to ensure that our evaluation is fair.

For all datasets, we group the features derived from one-hot encoding in one feature group. In addition, we encode various PLAF constraints with and without implications (cf. Table 1). For instance, we enforce that `Age` and `Education` can only increase, and `MaritalStatus`, `Gender`, and `NativeCountry` cannot change. An example of a constraint with implications is given by Eq. (7). We present a detailed description of all considered PLAF constraints in Appendix A.1 in [24]. Since the existing systems do not support constraints with implications, we do not enforce these constraints in the experiments in Sec. 6.3.

Considered Systems. We benchmark GeCo against five existing systems. (1) *MACE* [12] solves for counterfactuals with multiple runs of an SMT solver. (2) *DiCE* [17] generates counterfactual explanations with a variational auto-encoder. (3) *WIT* is our implementation of the counterfactual reasoning approach in Google's What-if Tool [33]. WIT looks up the closest counterfactual that satisfies the PLAF constraints in database D. We implemented our own version, because the What-if Tool does not support feasibility constraints. (4) *CERT* is our implementation of the genetic algorithm that is used in CERTIFAI [27] (see Sec. 4.7 for details). We reimplemented the algorithm because CERTIFAI is not publicly available. (5) *SimCF* is our adaptation of the SimBA [9] algorithm for adversarial examples to the problem of finding counterfactual explanations. The algorithm randomly selects one feature group, samples five feasible values for this group, and greedily applies the change that returns the best score. This process is repeated until the classifier returns the desired outcome. Since SimCF randomly changes one feature at a time, the explanations may not be consistent across runs.

Evaluation Metrics. We use the following three metrics to evaluate the quality of the explanation: (1) The *consistency* of the explanations, i.e., does the classifier return the good outcome for the counterfactual $\boldsymbol{x}_{cf}$; (2) The *distance* between $\boldsymbol{x}_{cf}$ and the original instance $\boldsymbol{x}$; (3) The *number of features changed* in $\boldsymbol{x}_{cf}$.

For the comparison with existing systems, we use the ℓ_1 norm to aggregate the distances for each feature (i.e., $\beta = 1$, $\alpha = \gamma = 0$ in Eq. (2)), because MACE and DiCE do not support combining norms. We examine other choices of these hyperparameters in Appendix A.4 in [24]. To compare runtimes, we report the average wall-clock time it takes to explain a single instance.

Table 2: Examples of counterfactual explanations by GeCo, MACE, DiCE, and SimCF for one instance in Adult (presenting selected features). MACE and DiCE use different models; we show GeCo's explanation for each model. The neural network does not use CapitalGain and CapitalLoss.

	Age	Education	Occupation	CapitalGain	CapitalLoss	Hours/week	Gender	Prediction
$\boldsymbol{x}$	49	School	Service	0	0	16	F	Bad
			Decision Tree					
GeCo	49	School	Service	**4,787**	0	16	F	**Good**
MACE	49	School	Service	**4,826**	**20**	16	F	**Good**
SimCF	49	School	Service	**7,688**	**1,380**	16	F	**Good**
			Neural Network					
GeCo	**53**	**Masters**	Service	–	–	16	F	**Good**
DiCE	**54**	**PhD**	**BlueCol**	–	–	**40**	**M**	**Good**

GeCo and CERT return multiple counterfactuals for each instance. We only consider the best counterfactual in this evaluation.
Classifiers. We benchmark the systems on tree-based models and multi-layered perceptrons (MLP).

Comparison with Existing Systems. For the comparison of with existing systems we use the classifiers proposed by MACE and DiCE for all systems to ensure a fair comparison. For tree-based models, we attempted to compute explanations for random forest classifiers, but MACE took on average 30 minutes to compute a single explanation, which made it infeasible to compute explanations for many instances. Thus, we consider a single decision tree for this evaluation (computed in scikit-learn, default parameters). DiCE does not support decision trees, since it requires a differentiable classifier. For the neural net, we use the classifier proposed by DiCE, which is a two-layered neural network with 20 (fully connected) hidden units and ReLU activation.

Microbenchmarks. In the micro benchmarks, we consider a random forest with 500 trees and maximum depth of 10 from the Julia MLJ library [6], and the MLPClassifier from the scikit-learn library [21]. We learn two MLP models, a small variant with one hidden layer (100 nodes, the default setting) and a larger variant with two hidden layers (100 nodes each), which was the best network structure we found in a comparison of 10 different structures.
Setup. We implemented GeCo, WIT, CERT, and SimCF in Julia 1.5.2. All experiments are run on an Intel Xeon CPU E7-4890/2.80GHz/64bit with 108GB RAM, Linux 4.16.0, and Ubuntu 16.04.

We use the default hyperparameters for GeCo (c.f. Sec. 4.6) and MACE ($\epsilon = 10^{-3}$). CERT runs for 300 generations, as in the original CERTIFAI implementation. In GeCo, we precompute the active domain of each feature group, which is invariant for all explanations.

6.2 End-to-end Example

We consider one specific instance in the Adult dataset that is classified as "bad" (`Income` <\$50K) and illustrate the differences between the the explanations for each considered system.

Table 2 presents the instance $\boldsymbol{x}$ and the counterfactuals returned by GeCo, MACE, SimCF, and DiCE. WIT and CERT fail to return an explanation, because the Adult dataset has no instance with income >\$50K that also satisfies all PLAF constraints for $\boldsymbol{x}$. MACE and SimCF compute the explanation over a decision tree, and DiCE considers a neural network. We present GeCo's explanation for each model, and argue that they are better than the explanations by MACE, SimCF, and DiCE.

For the decision tree, GeCo is able to find a counterfactual that changes only Capital Gains. In contrast, the explanations by MACE and SimCF require a change in Capital Gains and Capital Loss. Remarkably, their changes in Capital Gains are larger than the one required by GeCo. The neural network does not use the features Capital Gains and Capital Loss. For this model, GeCo proposes an increase in education, which in turn requires an increase in age according to our PLAF constraints. If this change is deemed infeasible, we can update the PLAF constraints and ask GeCo to generate a new counterfactual. In contrast, DiCE changes the values of eight features in total, which is neither feasible nor plausible.

6.3 Quality and Runtime Tradeoff

In this section, we investigate the tradeoff between the quality and the runtime of the explanations for all considered systems on Credit and Adult. As explained in Sec. 6.1, we evaluate the systems using a single decision tree and a neural network.
Takeaways for Evaluation with Decision Trees. Figures 4 shows the results of our evaluation with decision trees on 5,000 instances from Credit and Adult for which the classifier returns the negative outcome. We present the average distance and runtime for each considered system and dataset.

GeCo and MACE are always able to find a feasible and plausible explanation. WIT and CERT, however, fail to find an explanation in 2.1% of the cases for Adult. This is because the two techniques are restricted by the database D, which may not contain an instance that is classified as good and represents feasible and plausible actions. SimCF fails to find explanations in 1.5% and 2.4% of the cases for Adult and respectively Credit.

GeCo's explanations are on average the closest to the original instance. This can be explained by the fact that GeCo is able to find these explanations by changing significantly fewer features. For the Credit dataset, for instance, GeCo can find explanations with 1.27 changes on average, while MACE (the best competitor) changes on average 3.39 features. WIT, CERT, and SimCF change on average 3.97, 3.15, and respectively 2.98 features. We provide further details on the number of features changed by each system in Appendix A.2 in [24].

GeCo is consistently able to compute each explanation in less than 300ms on average. On average, GeCo is 20× faster than MACE. This performance is only matched by WIT and SimCF, which do not return explanations with the same quality as GeCo.

Like GeCo, CERT uses a genetic algorithm, but it takes 5.6× longer to compute explanations that do not have the same quality as GeCo's. Thus, GeCo's custom genetic algorithm, which is designed to explore counterfactuals with few changes, is very effective.
Takeaways for Evaluation with Neural Net. Figure 5 presents the key results for our evaluation on the MLP classifier. We only show the comparison of GeCo and DiCE, because the comparison with WIT, SimCF, and CERT is similar to the one for decision trees.

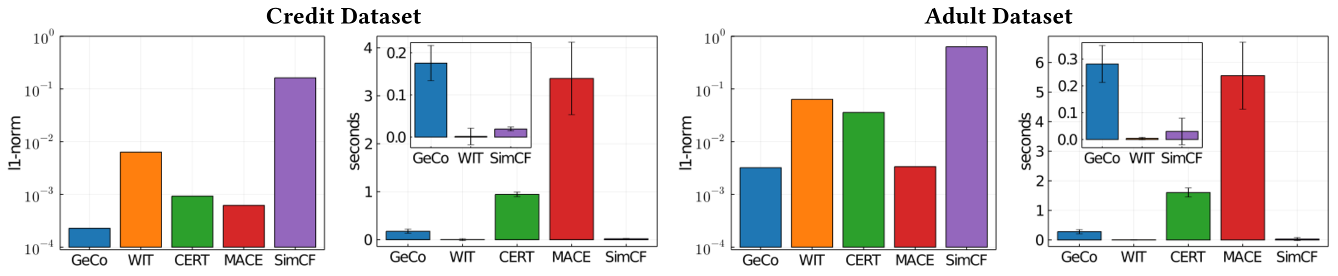

Figure 4: Comparison of the average distance (ℓ_1 norm, $\log_{10}$ scale), and average runtime for the explanations by GeCo**, WIT, CERT, MACE, and SimCF for 5000 instances on the Credit and Adult datasets. Error bars represent one standard deviation.**

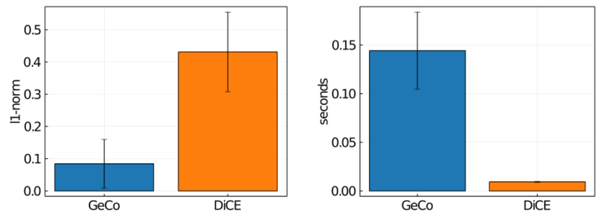

Figure 5: Comparison of the average distance and runtime of GeCo **and DiCE for a neural network on 5000 Adult instances. Error bars represent one standard deviation.**

Table 3: Microbenchmarks results for tree-based models.

	Credit	Adult	Allstate	Yelp
Generations	4.22	4.66	5.26	3.24
Explored Candidates	18.1K	15.3K	62.8K	42.9K
Size of Naive Rep.	62.23K	117.41K	8.03M	7.84M
Size of Δ-Rep.	18.64K	21.88K	412K	131K
Compression	3.3×	5.4×	19.5×	60.1×

MACE requires an extensive conversion of the classifier into a logical formula, which is not supported for the considered model.

Since DiCE computes the counterfactual explanation in one pass over a variational auto-encoder, it is able to compute the explanations very efficiently. In our experiments, DiCE was able to find an explanation on average 15.5× faster than GeCo. The counterfactuals that DiCE generates, however, have poor quality. Whereas GeCo is again able to find explanations with only few required changes, DiCE changes on average 5.3 features. As a result, GeCo's explanation is on average 4.4× closer to the original instance. Therefore, we consider GeCo much more suitable for real-world applications.

6.4 Microbenchmarks

In this section, we present the results for the microbenchmarks.

Breakdown of GeCo**'s Components.** First, we analyze the runtime of each operator presented in Sec. 4, as well as the impact of the Δ-representation and partial evaluation (Sec. 5) on two tree-based models over Allstate and Yelp, as well as the small and large multi-layered perceptrons (MLP) over Yelp. We run GeCo for 5 generations on 1,000 instances that have been classified as bad.

Figure 6 presents the results for this benchmark. Initial population captures the time it takes to compute the feasible space, and to generate and select the fittest candidates for the initial population. The times for selection, crossover, and mutation are accumulated over all generations. For each scenario, we first present the runtime for: (1) without the Δ-representation and partial evaluation enabled, (2) with each optimization individually, and (3) with both.

The results show that the selection and mutation operators are the most time consuming operations of the genetic algorithm. This is not surprising since they operate on tens of thousands of candidates, whereas crossover combines only a few selected candidates.

Partial evaluation of the classifier is effective for the random forest model. For Allstate, it decreases the runtime of the selection operator by up to 3.2×, which translates into an overall speedup of 1.7×. For MLPs, the optimization is less effective, because we can only partially evaluate the first layer. In fact, the overhead of partial evaluation results in a slowdown for the larger variant.

The Δ-representation decreases the runtime of the mutation operator by 3.9× for Allstate and 4.7× for Yelp (random forest). This speedup is due to the compression achieved by the Δ-representation (see below). If the classifier is not partially evaluated, then there is a tradeoff in the runtime for the selection operator, because it requires the materialization of the full feature vector. This materialization increases the runtime of selection by up to 1.6× (Yelp, MLP small).

The best performance is achieved if the Δ-representation and partial evaluation of the classifier are used together. In this case, there is a significant runtime speedup for both the mutation operator and selection operators. Overall, this can lead to a performance improvement of 5× (Yelp, random forest).

Validating Explanation Quality. To evaluate whether GeCo is able to find good explanations, we design synthetic classifiers for which the optimal explanation is known. Each classifier is a conjunction of unary threshold conditions, and the outcome is positive iff all conditions are satisfied. Given an instance that fails all conditions, the number of conditions is equal to the number of features that the counterfactual needs to change. We present further details on the synthetic classifiers in Appendix A.3 in [24].

Figure 7 presents the results of our evaluation on 100 Credit instances which fail all conditions for all classifiers. We consider classifiers with up to 12 conditions, which is the maximum number of features that can be changed. We add features in the decreasing order of their domain sizes, which is the most challenging order for GeCo; we consider a different order in Appendix A.3.

GeCo always finds a valid counterfactual explanation, even if we require changing all 12 features. The runtime is linear with respect to the number of features changed, and proportional to

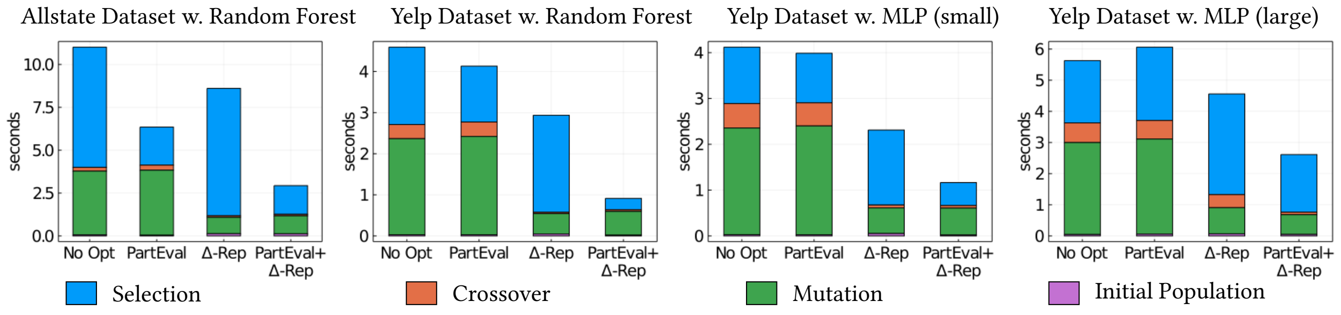

Figure 6: Breakdown of GeCo**'s runtime into the main operators on Allstate and Yelp using random forest and MLP classifiers. We present the runtimes (1) without the Δ-representation and partial evaluation, (2) with partial evaluation, (3) with the Δ-representation, and (4) with both optimizations. The runtime is averaged over 1,000 instances.**

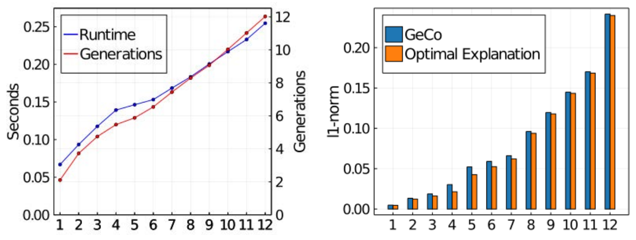

Figure 7: Evaluation of GeCo **over 100 Credit instances with synthetic classifiers that require 1-12 feature changes. (left) Average runtime and number of generations; (right) Average distance of** GeCo **compared to the optimal explanation.**

number of generations of the genetic algorithm. The distance of the explanations is always close to the distance of the optimal explanation. In Appendix A.3, we show that, by sampling more values during mutation, we can further decrease the distance gap to the optimal explanation with minor performance degradation.

Number of Generations and Explored Candidates. Table 3 shows for each dataset how many generations GeCo needed on average to converge, and how many candidates it explored. The majority (up to 97%) of the candidates were generated by mutate.

Compression by Δ-representation. Table 3 compares the sizes for the naive listing representation and the Δ-representation. We measure size in terms of the number of represented values, and take the average over all generations for the size of the candidate population after the mutate and crossover operations. Overall, the Δ-representation can represent the candidate population up to 60× more compactly than the naive listing representation.

Effect of Constraints. We evaluate the impact of the PLAF constraints on GeCo's runtime. The constraints without implications significantly restrict the search space of feasible counterfactuals. Thus, including these constraints improves the performance by 19.3% for Adult and 32.2% for Credit. The constraints with implications, however, may introduce some overhead since they need to be checked dynamically using action cascading. For example, the overhead is 19.2% for Adult and 32.3% for Credit. Yet, the version with all constraints is still faster by 9.8% for Credit and 18.7% for Adult over the version without any constraints. Finally, the grouping of features also restricts the search space. For Adult, GeCo is 1.4× faster using the feature groups than without.

7 CONCLUSIONS

We described GeCo, the first interactive system for counterfactual explanations that supports a complex, real-life semantics of counterfactuals, yet provides answers in real time. GeCo defines a rich search space for counterfactuals, by considering both a dataset of example instances, and a general-purpose constraint language. It uses a genetic algorithm to search for counterfactuals, which is customized to favor counterfactuals that require the smallest number of changes. We described two powerful optimization techniques that speed up the inner loop of the genetic algorithm: Δ-representation and partial evaluation. We demonstrated that, among five other systems reported in the literature, GeCo is the only one that can both compute quality explanations and find them in real time.

This work opens up several directions for future work. First, counterfactual explanations are subject to updates to the underlying data and classifier. We plan to explore how we can generate explanations that are robust to small changes in the data distribution or classifier. This is related to the more general problem of robust machine learning. Second, GeCo requires that the PLAF constraints are provided by a domain expert. We plan to explore how we can leverage constraints and dependencies in databases to generate these constraints automatically. Third, GeCo currently assumes that the input to the model is the raw data. In practice, however, the model input is typically the output of extensive feature engineering. We plan to explore how GeCo can generate explanations for the feature engineered data, but then return the corresponding raw data values to the user. For structured relational data, the feature engineering involves aggregating the raw data, in which case we would have to connect GeCo with techniques on explaining aggregate queries that have been developed in the database community. Fourth, counterfactual explanations expose values from the database, which may lead to privacy issues. We plan to explore how to return explanation that satisfy both privacy and legislative requirements. Finally, we have implemented partial evaluation manually, and it is only supported for random forests and simple neural network classifiers. We plan to extend this optimization to other models by leveraging work from the compilers community.

ACKNOWLEDGMENTS

This work was partially supported by NSF IIS 1907997, NSF IIS 1954222, and a generous gift from RelationalAI.

REFERENCES

[1] Allstate. 2011. Allstate Claim Prediction Challenge. https://www.kaggle.com/c/ClaimPredictionChallenge

[2] Sercan Arik and Tomas Pfister. 2019. Tabnet: Attentive interpretable tabular learning. *arXiv preprint arXiv:1908.07442* (2019).

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.

[4] Nima Asadi, Jimmy Lin, and Arjen P De Vries. 2013. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering* 26, 9 (2013), 2281–2292.

[5] Umang Bhatt, Alice Xiang, Shubham Sharma, Adrian Weller, Ankur Taly, Yunhan Jia, Joydeep Ghosh, Ruchir Puri, José MF Moura, and Peter Eckersley. 2020. Explainable machine learning in deployment. In *FAT**. 648–657.

[6] Anthony D. Blaom, Franz Kiraly, Thibaut Lienart, Yiannis Simillides, Diego Arenas, and Sebastian J. Vollmer. 2020. MLJ: A Julia package for composable machine learning. *arXiv preprint 2007.12285* (2020). http://arxiv.org/abs/2007.12285

[7] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *IEEE SP*. 39–57.

[8] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[9] Chuan Guo, Jacob Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Weinberger. 2019. Simple black-box adversarial attacks. In *International Conference on Machine Learning*. PMLR, 2484–2493.

[10] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2019. Declarative Recursive Computation on an RDBMS: Or, Why You Should Use a Database for Distributed Machine Learning. *PVLDB* 12, 7 (March 2019), 822–835.

[11] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.

[12] Amir-Hossein Karimi, Gilles Barthe, Borja Balle, and Isabel Valera. 2020. Model-agnostic counterfactual explanations for consequential decisions. In *AISTATS*. 895–905.

[13] Ron Kohavi. 1996. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid.. In *KDD*, Vol. 96. 202–207.

[14] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2015. Learning generalized linear models over normalized data. In *SIGMOD*. 1969–1984.

[15] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2015. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *SIGIR*. 73–82.

[16] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *ICLR*. OpenReview.net. https://openreview.net/forum?id=rJzIBfZAb

[17] Divyat Mahajan, Chenhao Tan, and Amit Sharma. 2019. Preserving Causal Constraints in Counterfactual Explanations for Machine Learning Classifiers. In *CausalML @ NeurIPS*.

[18] Ramaravind K Mothilal, Amit Sharma, and Chenhao Tan. 2020. Explaining machine learning classifiers through diverse counterfactual explanations. In *FAT**. 607–617.

[19] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2019. Incremental and Approximate Inference for Faster Occlusion-Based Deep CNN Explanations. In *SIGMOD*. 1589–1606.

[20] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16.

[21] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, and et al. 2011. Scikit-learn: Machine Learning in Python. *J. Machine Learning Research* 12 (2011), 2825–2830.

[22] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *SIGKDD*. 1135–1144.

[23] Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 1, 5 (2019), 206–215.

[24] Maximilian Schleich, Zixuan Geng, Yihong Zhang, and Dan Suciu. 2021. GeCo: Quality Counterfactual Explanations in Real Time. *arXiv preprint 2101.01292* (2021). http://arxiv.org/abs/2101.01292

[25] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *SIGMOD*. 1642–1659.

[26] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.* 43, 1, Article 4 (2018), 45 pages.

[27] Shubham Sharma, Jette Henderson, and Joydeep Ghosh. 2020. CERTIFAI: A Common Framework to Provide Explanations and Analyse the Fairness and Robustness of Black-box Models. In *AIES*. 166–172.

[28] Dylan Slack, Sophie Hilgard, Emily Jia, Sameer Singh, and Himabindu Lakkaraju. 2020. Fooling lime and shap: Adversarial attacks on post hoc explanation methods. In *AIES*. 180–186.

[29] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *ICLR*. http://arxiv.org/abs/1312.6199

[30] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. 307–322.

[31] Berk Ustun, Alexander Spangher, and Yang Liu. 2019. Actionable recourse in linear classification. In *FAT**. 10–19.

[32] Sandra Wachter, Brent Mittelstadt, and Chris Russell. 2017. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *Harv. JL & Tech.* 31 (2017), 841.

[33] James Wexler, Mahima Pushkarna, Tolga Bolukbasi, Martin Wattenberg, Fernanda Viégas, and Jimbo Wilson. 2019. The what-if tool: Interactive probing of machine learning models. *IEEE transactions on visualization and computer graphics* 26, 1 (2019), 56–65.

[34] Ting Ye, Hucheng Zhou, Will Y Zou, Bin Gao, and Ruofei Zhang. 2018. Rapid-scorer: fast tree ensemble evaluation by maximizing compactness in data level parallelization. In *SIGKDD*. 941–950.

[35] I-Cheng Yeh and Che-hui Lien. 2009. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications* 36, 2 (2009), 2473–2480.

[36] Yelp. 2017. Yelp Dataset Challenge. https://www.yelp.com/dataset/challenge/