Optimizing Counterfactual-based Analysis of Machine Learning Models Through Databases

Aviv Ben Arie
Intuit Inc.
USA
aviv_benarie@intuit.com

Daniel Deutch
Tel Aviv University
Israel
danielde@post.tau.ac.il

Nave Frost
Tel Aviv University
Israel
navefrost@mail.tau.ac.il

Yair Horesh
Intuit Inc.
USA
yair_horesh@intuit.com

Idan Meyuhas
Tel Aviv University
Israel
idanmeyuhas@mail.tau.ac.il

ABSTRACT

In the context of Machine Learning models, counterfactuals (CFs) are hypothetical perturbations to a given input of the model that would result in a different classification outcome. Multiple lines of recent work have proposed algorithms for finding CFs (hereby referred to as CF Generators) and demonstrated their value in providing insights for model owners. However, obtaining these insights may be computationally expensive, often requiring many invocations of these algorithms with complex constraints. In this work, we complement these efforts by presenting CFDB: a relational, declarative framework for CF-based analysis. Users of CFDB specify analysis tasks as declarative queries over a relational schema tailored for CFs. CFDB then compiles the specification into a series of CF requests, to be fed as input to CF Generators. The main advantage of this approach is that it allows to optimize the tradeoff between CF generation time and quality. Specifically, our optimizations are based on the observation that often, one may satisfy multiple CF requests using the same CFs, thereby reducing the total number of costly CF Generator invocations. We design algorithms that identify when such reuse is possible and optimize the computation accordingly. We experimentally demonstrate the usefulness of our approach and our optimizations, in the context of multiple datasets, multiple previously proposed Counterfactual Generators, and use cases such as assessing model fairness.

1 INTRODUCTION

Counterfactuals (CFs for short) explain the results of Machine Learning classifiers through perturbations to instances that lead to altered classifications. Given the importance of automated decision-making, combined with the complexity of models that make these decisions, such explanations are crucial in facilitating transparency and fairness of decision-making and in helping model owners to debug and improve their models.

Consider for example a loan application classifier, used for recommendations on whether each such application should be approved or rejected. For a rejected application, a counterfactual could indicate that a slight raise to the applicant's income would lead to approval. Another counterfactual may reveal that for an applicant with the same profile except for a different gender, the loan application would have been approved.

A counterfactual concerning a single prediction is useful for providing an explanation to the relevant individual, such as the loan applicant in the above example. If, by contrast, the model owners wish to obtain insights on the model as a whole, they typically need a representative set of CFs, across multiple instances, that fit their analysis goals. For example, the model owners may specifically be interested in CFs for false positives, to reveal the underlying cause for this type of error. They may further wish to restrict attention to CFs that change a particular feature, or a set of features, such as gender and/or race. The results may be telling concerning the fairness of the model. Taking this idea a step further, the work of [30] defines the burden of a group of instances as the average distance from the original instances of CFs obtained for this group. Unexpectedly high burden for a specific group, such as instances with a particular gender or race, may indicate potential unfairness in the model. Model robustness is captured similarly, as the (normalized) distance between instances and their CFs.

CFs are typically characterized according to two aspects. The first is *feasibility*, namely CFs that satisfy some given constraints. These constraints may refer to the modification they induce (for instance, the modified features [27]), or to the new instance that is formed (e.g. it should satisfy some given database constraints [3]). The second is *quality*, which is typically measured as the distance of the CF from the original instance (captured via either $L0$ or other measures, depending on the data type), i.e. the magnitude of change it induced. A CF that is in the proximity to the given instance is typically regarded as more useful than a distant one.

Finding feasible and high-quality CFs is a highly non-trivial task, and the survey of [33] overviews many different approaches. There may be infinitely many counterfactual candidates (all valuations to features), and finding even a single counterfactual for a single instance is in general computationally intractable even for simple white-box models (see e.g. [3]). Heuristics may fail or may be prohibitively slow, for instance, when the data has high dimensionality or the decision boundaries of the model are highly fragmented. A commonly used solution is to restrict the search for CFs to a pool of available instances, e.g. those in the training dataset [27], looking for the instances that are close to the instance for which one wishes to compute counterfactuals. This approach significantly accelerates computation but is inherently non-robust, in the sense that good CFs for a given instance (e.g. ones that are indeed close to the instance, and satisfy some given constraints) may simply not be available in the pool (see e.g. Section 6.3 of [29] for further discussion). The problem is exacerbated because, as explained above, model owners often need to analyze many instances (e.g., representatives of classes of interest [30]) and obtain many CFs for each instance. CF generation at scale is indeed computationally expensive: a CF-based analysis such as the one described above for model fairness may take over 30 minutes (see Section 5).

Main contributions and novelty. To this end, we present in this paper a novel solution for optimizing counterfactual-based analysis. We next list our main contributions and then provide details on each of them.

- We introduce (Section 3.1) CFQL, a *declarative query language for the specification of properties of interest (constraints) that CFs should satisfy, before their generation.* CFQL thereby provides a uniform and generic interface that can interact with a counterfactual generation package of choice (referred to as a CF generator), out of the many existing ones.
- We introduce (Section 3.2) a CFQL *query evaluation algorithm*, namely an algorithm that translates the declarative specification to a sequence of CF generator calls, executes them, and stores the obtained CFs in the database. The algorithm leverages c-tables [14] in a novel way: in the seminal paper by Imielinski and Lipski [14], c-tables were designed to capture incomplete information; here we use them as means for transforming declaratively specified constraints into boolean ones. These constraints are then fed to a CF generator, and the obtained CFs are stored in the database.
- A main motivation for the declarative nature of CFQL is that it provides opportunities for under-the-hood optimizations whose goal is to minimize the number of invocations of the CF generator. This optimization problem was never studied before, to our knowledge, and in Section 4 *we provide the first algorithm that optimizes the number of CF generator calls.* At a high level, the main idea is to construct a graph whose nodes are the compiled constraints and the edges stand for logical implications amongst them. The graph allows to reveal opportunities for re-use: for two constraints c_1, c_2, if c_1 logically implies c_2, then a CF that satisfies c_1 is also guaranteed to satisfy c_2. It is thus sensible to first invoke the oracle with c_1 as input; the output may potentially be useful as a fulfilment of the CF request captured by c_2, depending on the proximity of retrieved CF to the instance. We allow control over the tradeoff between the number of CF oracle calls and the quality (in terms of distance of obtained CFs) via a user-specified *reuse threshold.*
- We have conducted an extensive experimental study, that examines the performance and usefulness of our framework across multiple datasets and use cases. The experimental results show that our solution indeed allows to re-use CFs in an informed way, thereby significantly speeding up CF-based analysis, while also obtaining CFs of high quality. We compare our solution to a baseline that repeatedly calls the CF generator for every given instance and constraint, as done in absence of our framework.

In our technical development, we will focus on the analysis of Machine Learning classifiers, which in turn was the main focus of the vast majority of research on counterfactuals (see [33]). The conceptual contributions nevertheless apply to counterfactual-based analysis beyond classifiers. In Section 4.4 we detail the needed adaptations to support CF-based analysis for other Machine Learning models.

We next provide more details on each component of the solution.

Uniform, Declarative Interface with CFQL*(Section 3.1).* We introduce a simple relational schema, which may store information on models, instances and prior model outputs. Importantly, it includes a dedicated relation that will be used to store the generated CFs. We introduce a query language called CFQL that has an SQL-like syntax but unique semantics: CFQL is used to define queries that specify the desired CFs of interest *before CF Generation.* For instance, a CFQL query may “select" CFs that involve altering the applicant's salary, with respect to instances for which the prediction was false positive. Such a query is interpreted as *requesting the generation* of such CFs (with respect to the instances that are actually stored in the database). These requests are to be fulfilled by a *CF generator*, namely a function that looks for CFs. The generator itself is treated as an oracle, in the sense that it may be implemented by one of the many existing tools for this purpose. More details on the query semantics follow.

From CFQL *Queries to CF Generation via c-tables (Section 3.2).* The space of possible CFs is infinite, and it is impossible to materialize all of them and then select those in which the analyst is interested. To address this challenge, we design a unique semantics for CFQL as outlined next. CFQL queries are not interpreted as selection criteria over existing CFs but rather as specifications of constraints over CFs that are of interest.

For CFQL evaluation, we need to feed the specified constraints to a CF generator, namely an algorithm that looks for CFs for the given model and instances that satisfy the constraints. However, CF generators expect, as input, boolean constraints. This means that the declarative specification of constraints expressed by CFQL queries needs to be compiled into boolean expressions before they can be passed to CF generators. We propose a novel compilation method, giving rise to an operational semantics for CFQL. The main observation that lies at the core of our method is that one can leverage conditional tables (c-tables) [14] for this purpose. Intuitively, c-tables allow to place variables in relation cells whose values are unknown. We design variables that correspond to the contents of CFs, prior to their generation. Query evaluation over tables with such unknowns yields particular *boolean conditions* associated with the query output. These are precisely the boolean expressions that capture the constraints expressed by the query. The expressions are then fed to a CF Generator of choice, thereby looking for CFs that comply with the CFQL query specification.

Efficient Evaluation via Ordering CF Generator Calls (Section 4). The evaluation algorithm outlined so far invokes the CF generator for every instance and constraint. This may be computationally costly. To this end, we develop novel optimizations that reduce the number of CF generator invocations. The idea is that a CF generated in response to an CF generator invocation with a particular instance I and constraint C as input, may also be useful for another constraint C' and possibly for another instance I', in which case we may re-use it instead of invoking the CF generator again. To indeed be useful for I' and C', the CF needs to (1) satisfy C' and (2) be close enough to I', with respect to an appropriate distance metric.

To achieve the former, we design an optimization that takes place after query evaluation yields a c-table, but before invoking the CF generator. The optimization analyzes the boolean expressions, looking for logical implication amongst them: the intuition is that if we find such implication $c_1 \implies c_2$, then we are guaranteed that CFs that satisfy c_1 will also satisfy c_2.

A challenge, then, is that deciding logical implication amongst all pairs of boolean expressions is computationally costly, even if we use a state-of-the-art SAT solver to decide each individual implication. We design a series of optimizations and pruning strategies that significantly reduce the computational cost.

For the latter, namely the proximity of the CF to the instance, we incorporate a user-defined reuse threshold, above which we refrain from re-use. To set the threshold, the analyst may look at a sample of CFs obtained for multiple instances, and observe the density of CFs for instances of interest). `CFDB` then decides on a sequence of CF Generator invocations, executes it in order, and stores the obtained CFs in the database.

Experimental Study (Section 5). We have implemented our solutions and conducted an experimental study that examines the effectiveness of our approach, the execution time they incur, and the trade-off between execution time and obtained CF distances for our optimizations. The experimental results show that our solution is effective in accelerating CF generation if one is willing to allow some deterioration in CF quality. In particular, we replicated the experiment conducted by [30] on analyzing model fairness by generating a pool of CFs. We show that we can reach the same conclusions as [30], i.e. achieve CFs that are of high enough quality to analyze fairness, significantly faster than via direct use of the CF generators. We show that our solution is effective in exploiting re-use: it consistently achieves a significant reduction in execution time for a variety of CF-based analysis goals, different Machine Learning models, and datasets.

2 PRELIMINARIES

We start by recalling (1) the notion of counterfactuals for Machine Learning models, (2) a database schema for storing them, and (3) the notion of c-tables [14] which will serve as a tool in our solution.

2.1 Counterfactuals in Machine Learning

Let M be a Machine Learning model and let $f_1, ..., f_n$ be the model features. An *instance* I is a valuation for the features, namely a vector $(x_1, ..., x_n)$. Given an instance I as input, the model M outputs a *prediction* $M(I)$, which may e.g., be a label of accept/reject. A *counterfactual (CF)* for M and I is an instance I' such that $M(I') \neq M(I)$. A *CF Generator* is an algorithm that, when given access to a Machine Learning model M and an instance I, aims to generate CFs for M and I. Many different CF Generators have been proposed, including solutions [36] that look for CFs that minimize the $L0$ or $L1$ distance from the original instance, solutions [26] that aim to retrieve diverse sets of CFs, solutions that allow analysts to impose constraints on the generated CFs in different ways [3, 9, 17, 27, 29], and others [9, 19]. See Section 6 for an overview.

Our framework treats the ML model as well as the algorithm that generates CFs as black boxes and they may thus be arbitrarily complex. For ease of presentation, our running example will be shown in the context of decision trees.

Example 2.1. Figure 2 depicts two decision trees used for classifying loan applications, based on two numeric features: the requested loan *amount* and the customer's *income*, and a boolean feature *home* capturing whether the loan requester is a homeowner. The classification result is an accept/reject label, color-coded in the tree as green/red tree leaves resp. Consider an instance I standing for a customer with $income = 80$, $amount = 100$ and $home = rent$. I is classified as "reject" by both trees. For the tree in Figure 2(a), a CF for I is an instance I' where *income* and *home* are kept intact and the *amount* is set to 90 (or less). For the tree in Figure 2(b), CFs for I may be achieved by either modifying *income* to 101 or more, or by modifying *home* to "own".

2.2 A Relational Schema for Counterfactuals

We store CFs in a relational database. The schema is flexible but must include at least the following relations (additional relations may be introduced if needed, see Figure 1). The `Instances` relation includes instances fed to the model, their features (which vary based on the application domain), and optionally the ground truth of their label (set to NULL if unavailable). The `Predictions` relation has a `PredictionId` attribute serving as the key, and an `InstanceId` attribute that is a foreign key referring to the corresponding attribute in the `Instances` relation. Other attributes, such as a foreign key to a `Classifiers` relation identifying the classifier that performed each prediction, as well as the assigned `Label`, may be included (but are not mandatory for our model). Additionally, the schema includes a `CFs` relation that will store CFs returned by black-box generators. It has a `CfId` key attribute and an attribute for each of the instance's features. Finally, the `Prediction-CFs` relation connects predictions with CFs, including foreign keys to both relations. As we show below, `Predictions` and `CFs` have unique and distinct roles in the semantics of `CFQL`.

Example 2.2. Figure 1 shows an example of a Counterfactual Database. The `Instances` relation includes 4 instances of loan applicants, for which we store their income, the requested loan amount, the homeownership status, and the ground truth (i.e., whether the loan application should be accepted/rejected) where available. The `Predictions` relation includes predictions for each instance, outputted by the two decision trees of Figure 2, identified by ClassifierId 1 and 2. Geared towards storing CFs, the schema of the `CFs` relation includes a CF identifier as well as the instances' attributes (Income, Amount, Home). Disregard for now the current contents of the `CFs` and the `Predictions-CFs` relations, which will be discussed later.

2.3 Conditional Tables

We next recall the notion of c-tables. In the following, let Vars be a set of variables and let Const be a set of constants.

Definition 2.3 (Boolean condition). A boolean condition is defined recursively: it is either (1) an expression of the form $x\theta y$ or $x\theta c$ for $\theta \in \{=, \neq, \leq, <, \geq, >\}$, where $x, y \in$ Vars and $c \in$ Const or (2) an expression of the form $cond1 \wedge cond2$ or $cond1 \vee cond2$ or $\neg cond1$ where $cond1$ and $cond2$ are conditions.

We are now ready to recall a simplified variant of the notion of c-tables [1] from [14].

Definition 2.4 (conditional table). A conditional table (c-table for short) is a pair (T, φ), where:

- T is a table in which each cell may include either a constant from Const or a variable from Vars;

[1]The simplification is that c-tables in their full generality also allow placing global conditions over the variables, which we do not need here.

Instances

InstanceId	Income	Amount	Home	GroundTruth
1	20	5	rent	rejected
2	60	20	rent	accepted
3	20	15	rent	rejected
4	80	100	own	accepted

Predictions

PredictionId	ClassifierId	InstanceId	Label
1	1	1	rejected
2	1	2	accepted
3	1	3	rejected
4	1	4	rejected
5	2	1	rejected
6	2	2	rejected
7	2	3	rejected
8	2	4	accepted

Classifiers

ClassifierId	ClassifierType
1	Decision Tree
2	Decision Tree

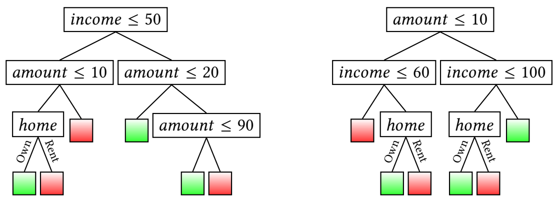
Prediction_CFs

PredictionId	CfId
4	1
6	2
4	3
6	4

CFs

CfId	Income	Amount	Home
1	81	90	own
2	101	20	rent
3	80	90	own
4	101	19	rent

Figure 1: Database



(a) Tree 1 **(b) Tree 2**

Figure 2: Decision Tree Models

- φ maps each tuple t in T to a local condition φ_t.

C-tables form a compact representation of a set of possible worlds, obtained by assigning values to variables and then discarding all tuples t for which φ_t is assigned to $false$. An important property of c-tables, that will be useful in our context, is that they are closed under query evaluation. Namely, given an SPJUD (Select-Project-Join-Union-Difference) query and a database D of c-tables, one may compute in polynomial time a database D' of c-tables whose possible worlds are precisely the query results over all possible worlds of D. Examples appear in the following sections.

```
SELECT P.PredictionId
FROM Instances AS I,
     Predictions AS P
WHERE I.InstanceId = P.InstanceId
  AND I.GroundTruth = 'accepted'
  AND P.Label ='rejected'
```

(a) $Q_{Pred_{FN}}$

```
SELECT C.CfId, C.Income, C.Amount, C.Home
FROM CFs AS C, Prediction_CFs AS PC,
     Instances AS I
WHERE C.CfId = PC.CfId
  AND PC.PredictionId = pred.PredictionId
  AND I.InstanceId = pred.InstanceId
  AND C.Home = I.Home AND C.Income > I.Income
```

(b) $Q_{CF_{(Home,Income)}}$

Figure 3: Example Queries

3 THE FRAMEWORK

We detail the syntax of CFQL, and introduce a first evaluation algorithm for CFQL queries, thereby defining operational semantics. Optimizations are introduced in the next section. For ease of presentation, we will focus on binary classification, yet our framework can be generalized to multi-class problems, as well as to other types of ML models beyond classification (see Section 4.4).

3.1 CFQL syntax

We next introduce CFQL and illustrate it via an example. Queries have the following syntax:

`FOREACH pred in` Q_{Pred} `GENERATE` Q_{CF}

- Q_{Pred} is an SQL query used to select identifiers of predictions of interest. It may use any relation as an auxiliary, except for `CFs` and `Prediction-CFs`, and its output is required to be a relation with a single column, named `PredictionId`, whose contents is a subset of the identifiers appearing in `Predictions.PredictionId`.
- Q_{CF} is a query over the `CFs` relation and possibly other relations. Q_{CF} projects only over attributes of `CFs`. Its selection predicate is a boolean combination of terms of the form `R1.x` θ `R2.y`, `R1.x` θ `Q'` or `(R1.x,R2.y,...) IN Q'` where $R1, R2$ are relation names, x, y are attributes, θ is a comparison operator as in Def. 2.3, and Q' is a query not involving `CFs` or `Prediction-CFs` and having the correct arity for its use as a sub-query according to the standard SQL semantics. No self-joins are allowed, and in particular in a term of the form `R1.x` θ `R2.y`, $R1$ and $R2$ should be distinct (a self-join that is created through multiple such conditions is also disallowed). Q_{CF} may not include any other construct and is intuitively a self-join-free SPJ query with respect to `CFs`, possibly with some nested queries referring only to the other relations. Last, Q_{CF} may use

the keyword *pred* as part of its specification. It will be replaced, in each iteration, by a tuple from `Predictions` corresponding to each of the selected identifiers.

Example 3.1. Figure 3a includes an example of a Q_{Pred} query. It selects (the identifiers of) false negative predictions, namely cases where the classifier has assigned a “rejected” label whereas the ground truth for the classified instance is “accepted”. Figure 3b is an example of a Q_{CF} query. Note its use of *pred*: for each prediction identifier selected by Q_{Pred}, the Q_{CF} query will be evaluated with each occurrence of *pred* being replaced by the corresponding tuple from `Predictions`. This query (called in the sequel $Q_{CF_{(Home,Income)}}$) specifies interest in CFs where the home status stays intact, and the applicant’s `Income` increases. Similarly, one may introduce a different query (called in the sequel $Q_{CF_{(Home,Amount)}}$) where we replace the last row by “AND C.Amount < I.Amount”. Both queries will be shown useful in the examples that follow.

As demonstrated, CFQL provides a novel uniform and declarative interface for the specification of CFs of interest: the user writes queries as if the CFs were already stored in the database. We next explain how to compile this specification into a sequence of invocations of a CF generator, prompted to retrieve the requested CFs. In Section 4, the compilation process will further be optimized to reduce the number of invocations.

3.2 Basic Evaluation Framework

Given a query $Q = (Q_{Pred}, Q_{CF})$ and a partial CF database where the `CFs` and `Prediction-CFs` relations are not instantiated, the basic algorithm partially evaluates the query, as follows. It first evaluates Q_{Pred} to identify predictions that the analyst is interested in. Then, to account for the unique treatment of CFs by CFQL (namely that it is used to specify constraints over them prior to generation, rather than querying existing data), our algorithm uses c-tables in a unique way: it starts by generating tuples with placeholders (variables) for the relevant CFs in the `CFs` table, and then evaluates Q_{CF} using the algorithm of [14], originally designed for querying incomplete information. This algorithm computes the conditions over the placed variables which are necessary and sufficient for the CFs to quality according to the criteria imposed by Q_{CF}. We uniquely treat these conditions as constraints, passing them as input to a CF generator. The retrieved CFs are then used to replace the variables in `CFs`.

We next detail the computation, using the following example:

`FOREACH pred in` $Q_{Pred_{FN}}$ `GENERATE` $Q_{CF_{(Home,Income)}}$

Where $Q_{Pred_{FN}}$ and $Q_{CF_{(Home,Income)}}$ are as in Figure 3.

Step 1. Evaluate Q_{Pred} over D in the standard manner, and let $Pred_0, ..., Pred_k$ be the selected prediction identifiers. For each $Pred_i$, create a fresh identifier and call it CF_i. Create an instance of `Prediction-CFs` including the tuples $\{(Pred_i, CF_i)\}$ for $i = 0, ..., k$. Create an instance of the `CFs` relation which is a c-table including the tuples $\{(CF_i, y_{CF_i,attr_1}, ..., y_{CF_i,attr_n})\}$, where each $y_{CF_i,attr_j}$ is a fresh variable. All tuples are associated with the condition *True*.

Example 3.2. $Q_{Pred_{FN}}$ selects all identifiers of false negative predictions. For the database in Figure 1, these predictions are identified by 4 and 6. For each of these two tuples, we generate a fresh CF identifier and introduce corresponding tuples to `Prediction-CFs`, i.e. $(4, 1)$ and $(6, 2)$. For each of these two anticipated CFs, we generate abstract tuples in `CFs` with variables serving as placeholders for attribute values that will be determined by the CF generator. For instance, for the first CF we have $(1, y_{1,Income}, y_{1,Amount}, y_{1,Home})$, where 1 is the CF identifier and the variable $y_{1,a}$ is a placeholder for the value of attribute a. The condition accompanying the two tuples in the c-tables is set to *True*. We follow similarly for the second CF. The resulting relations are shown at the top of Figure 4.

Step 2. Let D' be the database including all relations from D as well as the `CFs` and `Predictions-CFs` relations obtained in step 1. Evaluate Q_{CF} over D' using the algorithm of [14] for c-tables.

Example 3.3. $Q_{CF_{(Home,Income)}}$ is evaluated over the database consisting of the `Instances`, `Classifiers` and `Predictions` relations from Figure 1, and the `Prediction-CFs` and `CFs` relations appearing in Figure 4. The result is shown in the bottom part of Figure 4. Note that the selection criteria in $Q_{CF_{(Home,Income)}}$ require that the home status does not change, and that the income increases. For our example database, this translates into the condition $(y_{1,Home} = own) \wedge (y_{1,Income} > 80)$ for the first tuple and $(y_{2,Home} = rent) \wedge (y_{2,Income} > 60)$ for the second tuple.

We next show another example of a query serving as Q_{CF}, which will be useful in the sequel.

Example 3.4. The query $Q_{CF_{(Home,Amount)}}$ resembles the query $Q_{CF_{(Home,Income)}}$, except for the last condition which is replaced by $C.Amount < I.amount$. Its result resembles the result in Example 3.3, except for the conditions that which are: $(y_{3,Home} = own) \wedge (y_{3,Amount} < 100)$ and $(y_{4,Home} = rent) \wedge (y_{4,Amount} < 20)$.

Step 3. Let T_{cond} be the c-table obtained as the result of Step 2. For each tuple t_i in T_{cond}, let $cond_i$ be the boolean condition annotating it. Replace, in $cond_i$, each variable $y_{i,attr}$ by $attr$. Further retrieve the model C_i and instance x_i that t_i refers to by joining with `Prediction-CFs`, `Predictions` and `Instances`. Invoke the counterfactual generator with C_i as the model, x_i as the instance, and $cond_i$ as the constraint.

Example 3.5. Reconsider the T_{cond} relation shown in Figure 4, which consists of two tuples. The first refers to CfId 1, which in turn corresponds to the instance 4 and the classifier 1. We invoke the CF generator for this input, with the condition annotating the tuple used as a constraint. Namely, the CF generator is asked to generate a CF with home status “own” and income greater than 80. We repeat for the second tuple in the relation.

Step 4. Replace each variable occurring in T_{cond} by the corresponding value from the generated CFs.

Example 3.6. Two CFs that qualify according to the specification in $Q_{CF_{(Home,Income)}}$ are `[Income=81, Amount=90, Home=own]` and `[Income=101, Amount=20, Home=rent]`. Indeed, these instances satisfy the constraint and are classified as “accept" by the two trees of Figure 2. The CFs are stored in `CFs` and identified by CfId 1 and 2. The analyst may choose to store CFs yielded by multiple CFQL queries for further processing. For example, the relation `CFs` in Figure 1 stores the two CFs mentioned above, as well as CFs yielded for $Q_{CF_{(Home,Amount)}}$ from Example 3.4 (identified by CfId 3 and 4).

4 EFFICIENT EVALUATION

So far, we have introduced a declarative language for specifying CF-based analysis and an evaluation algorithm that compiles the specification into a sequence of CF generator calls. We next

`Prediction-CFs`

PredictionId	CfId
4	1
6	2

`CFs`

CfId	Income	Amount	Home	Cond
1	$y_{1,Income}$	$y_{1,Amount}$	$y_{1,Home}$	True
2	$y_{2,Income}$	$y_{2,Amount}$	$y_{2,Home}$	True

T_{cond}: **Query result over the c-tables (after steps 1 and 2)**

CfId	Income	Amount	Home	Cond
1	$y_{1,Income}$	$y_{1,Amount}$	$y_{1,Home}$	$(y_{1,Home} = own) \wedge (y_{1,Income} > 80)$
2	$y_{2,Income}$	$y_{2,Amount}$	$y_{2,Home}$	$(y_{2,Home} = rent) \wedge (y_{2,Income} > 60)$

Figure 4: Part of the database, and the result after steps 1 and 2.

propose optimizations that enable the re-use of CFs, thereby allowing the reduction of the number of CF generator calls and consequently the overall analysis time. Revisiting the evaluation framework presented in Section 3, steps (1) and (2) thus stay intact. The optimization takes place after step (2), where, for a set of queries $\{(Q^i_{Pred}, Q^i_{CF})\}^n_{i=1}$, we have already obtained a database of c-tables containing conditions $\{C_{i,j}\}_{i\in[n],j\in Q^i_{Pred}(D)}$. For convenience, we assume that prediction identifiers are serial and define: (1) $Condition(i, j) = C_{i,j}$ is the condition obtained for the i'th query and the j'th instance, and (2) $Instance(i, j) = x_{i,j}$ is the instance for which prediction was made. In the rest of this section, we will assume that the model as well as the target labels are the same across all conditions that we handle; cross-label optimizations are deferred for future work.

4.1 Implication Graph

The main idea of the optimization is to identify implications among the computed conditions. Intuitively, implications guarantee that we can reuse an already obtained CF without re-invoking the CF generator. Note that the conditions $C_{i,j}$ are in fact boolean expressions over variables that correspond to attributes of the `CFs` relation. Each attribute is associated (in the schema) with an active domain, dictating the set of values that the corresponding variable may take. We then define implication (denoted using "$\implies$") over these boolean expressions in the standard logical sense, accounting also for the active domain (if, e.g., the domain of y_{Home} is $\{rent, own\}$ then $y_{Home} \neq rent \implies y_{Home} = own$ and vice versa).

We then introduce an implication graph, where nodes intuitively correspond to tuples in the c-table yielded by steps (1) and (2) of our evaluation framework and edges correspond to logical implication between the conditions annotating these tuples.

Definition 4.1 (Implication graph). Given a set of CFQL queries $\{(Q^i_{Pred}, Q^i_{CF})\}^n_{i=1}$ and a database D, let $\{p_{i,j}\}^{r_i}_{j=1}$ be the prediction identifiers selected by Q^i_{Pred} and let $\{C_{i,j}\}^{r_i}_{j=1}$ be the conditions annotating their respective tuples in the c-table `CFs`.

The *implication graph* $\mathcal{G} = (V, E)$ is a directed graph where:

$$V = \{(i, j)\}_{i\in[n],j\in[r_i]}$$
$$E = \left\{((i_1, j_1), (i_2, j_2)) \mid C_{i_1,j_1} \implies C_{i_2,j_2}\right\}$$

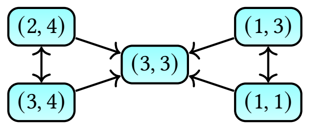
Example 4.2. Consider the following 3 queries over the database in Fig. 1. First, $Q_1 = (Q_{Pred_1}, Q_{CF_{(Home,Income)}})$, where the query $Q_{CF_{(Home,Income)}}$ is as in Figure 3b, and Q_{Pred_1} selects the predictions of instances 1 and 3. Second, $Q_2 = (Q_{Pred_2}, Q_{CF_2})$, where Q_{CF_2} resembles $Q_{CF_{(Home,Income)}}$ except for the conditions in lines 8–9. These are replaced in Q_{CF_2} by

"`(C. Home ≠ rent) OR (C.Income > I.Income)`".

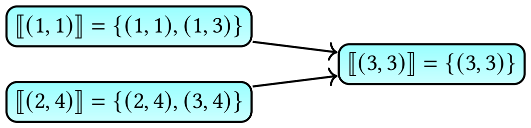
Q_{Pred_2} selects the prediction of instance 4. Last, $Q_3 = (Q_{Pred_3}, Q_{CF_3})$, where Q_{CF_3} again resembles $Q_{CF_{(Home,Income)}}$ except for lines 8–9, replaced by "`(C. Home = own) OR (C.Income > I.Income)`", and Q_{Pred_3} selects the predictions of instances 3 and 4. The following conditions are obtained ($C_{i,j}$ is obtained for the query Q_i and the prediction j):

$$C_{1,1} = (y_{Home} = rent) \wedge (y_{Income} > 20)$$
$$C_{1,3} = (y_{Home} = rent) \wedge (y_{Income} > 20)$$
$$C_{2,4} = (y_{Home} \neq rent) \vee (y_{Income} > 80)$$
$$C_{3,3} = (y_{Home} = own) \vee (y_{Income} > 20)$$
$$C_{3,4} = (y_{Home} = own) \vee (y_{Income} > 80)$$

Figure 5a depicts the implication graph $\mathcal{G} = (V, E)$ for these three queries. For example, $((1, 1), (3, 3)) \in E$ since $C_{1,1} \implies C_{3,3}$.



(a) Implication graph



(b) Its condensation

Figure 5: Example of Implication Graphs

Naive graph generation. For two boolean expressions C_1 and C_2, it holds that $(C_1 \implies C_2) \iff (C_1 \wedge \neg C_2)$ is unsatisfiable. Hence, a naive approach for generating the implication graph is to iterate over all condition pairs $(C_{i_1,j_1}, C_{i_2,j_2})$ and to use a SAT solver to test the satisfiability of $C_{i_1,j_1} \wedge \neg C_{i_2,j_2}$. If this expression is unsatisfiable, we add an edge from (i_1, j_1) to (i_2, j_2).

Atom-based graph generation. As we experimentally show below, the naive graph generation algorithm does not scale. We next propose multiple optimizations that significantly speed up graph generation.

Algorithm 1 is an efficient atom-based solution for constructing the implication graph. It starts with the initialization of two empty maps. Then (in line 4), it iterates over all queries and their

Algorithm 1: Atom-based graph construction

```
Input  : Queries $Q^1_{Gen}, \ldots Q^n_{Gen}$
         Predictions $p_{i_1}, \ldots p_{i_{r_i}}$ for each query $Q^i_{Gen}$
Output : Implication graph $\mathcal{G}$
/* Preprocessing */
$M$ = empty map ; // Map atomic propositions to their conditions
$B$ = empty map ; // Map variables to query and prediction indices
foreach $i \in [n], j \in [r_i]$ do
    $C$ = Condition$(i, j)$ ;
    foreach $a \in atoms(C)$ do
        $B[\text{variable}(a)] = B[\text{variable}(a)] \cup \{a\}$;
        $M[a] = M[a] \cup \{(i, j)\}$;
/* Graph construction */
$\mathcal{G} = (\{(i, j)\}_{i \in [n], j \in [r_i]}, \emptyset)$;
$eval = \emptyset$ ; // Set of evaluated expression pairs
foreach $bucket \in buckets(B)$ do
    foreach $(a_1, a_2) \in bucket \times bucket$ do
        if $a_1 \implies a_2$ then
            foreach $(i_1, j_1), (i_2, j_2) \in M[a_1] \times M[a_2]$ do
                if $i_1 \neq i_2$ or $j_1 \neq j_2$ then
                    if $((i_1, j_1), (i_2, j_2)) \in eval$ then
                        continue;
                    $C_{i_1,j_1}$ = Condition$(i_1, j_1)$ ;
                    $C_{i_2,j_2}$ = Condition$(i_2, j_2)$ ;
                    if solver$(C_{i_1,j_1} \wedge \neg C_{i_2,j_2})$ == unsat then
                        $E[\mathcal{G}] = E[\mathcal{G}] \cup \{((i_1, j_1), (i_2, j_2))\}$;
                    $eval = eval \cup \{((i_1, j_1), (i_2, j_2))\}$;
return $\mathcal{G}$
```

predictions. For each (query,prediction) pair, the algorithm retrieves its corresponding condition C in line 5, and in line 6, it iterates over the atomic propositions of C (denoted as atoms(C)). In lines 7 and 8 it updates both maps: B buckets each atomic proposition a according to the variable occurring in it (denoted by variable(a)) and M maps each atomic proposition to the set of indices of the representing conditions containing it. Then, the algorithm constructs an implication graph. To do so, lines 10 and 11 initialize the graph and a set of condition pairs that were already evaluated. Lines 12 and 13 iterate over pairs of atomic propositions that were bucketed together in B. If an implication is found amongst these atomic propositions, the algorithm iterates over all pairs of conditions containing them. For each pair that has not been evaluated yet (Lines 17 - 18) the algorithm retrieves the representing conditions in lines 19 and 20, and in line 21 it tests for implication. If it holds, then in line 22, the algorithm adds a corresponding edge to the graph. Once the pair evaluation has completed, it is added to the set of evaluated pairs in line 23. When the iteration over all atomic proposition pairs sharing a bucket is over, the algorithm returns the graph and terminates.

Example 4.3. Recall the five conditions in Example 4.2. Constructing the implication graph (figure 5a) using the naive approach requires $5 \cdot 4 = 20$ calls to the SAT solver. Using Algorithm 1, we e.g. observe that implication from Condition$(1, 1)$ to Condition$(2, 4)$ does not need to be checked. In total, we make only 16 calls to the SAT solver.

4.2 Equivalence de-duplication

Our second optimization identifies and merges nodes in the implication graph that capture equivalent conditions. This allows to avoid redundant CF generator invocations. We say that two conditions C_{i_1,j_1}, C_{i_2,j_2} are equivalent, denoted $C_{i_1,j_1} \sim C_{i_2,j_2}$, if $C_{i_1,j_1} \implies C_{i_2,j_2}$ and $C_{i_2,j_2} \implies C_{i_1,j_1}$. This yields equivalence classes: $[\![(i, j)]\!] \overset{\text{def}}{=} \{(i_1, j_1) \mid C_{i_1,j_1} \sim C_{i,j}\}$.

Then, we introduce a condensation of the implication graph. This is a graph $\mathcal{G}' = (V', E')$ where nodes are associated with equivalence classes, and there is an edge from $v'_1 \in V'$ to $v'_2 \in V'$ if $C_{i_1,j_1} \implies C_{i_2,j_2}$ for some $(i_1, j_1), (i_2, j_2)$ in the equivalence classes of v'_1 and v'_2 respectively (this is uniquely defined due to the equivalence relation).

Some conditions are equivalent simply because they are identical; these may be identified via hashing and then merged, even before generating the implication graph. We refer to this as *pre-construction de-duplication.*

Example 4.4. Recall the conditions from Example 4.2. By hashing the conditions we may identify that $C_{1,1}$ and $C_{1,3}$ are identical, and thus their corresponding nodes can be merged ($(1, 1)$ and $(1, 3)$). Thus, we remain with 4 (query, prediction) pairs. Combining pre-construction deduplication with naive graph generation yields $4 \cdot 3 = 12$ calls to the SAT solver; combining it with the optimized atom-based approach (Algorithm 1) yields 10 calls to the SAT solver.

After the implication graph is constructed, we are able to identify further intricate equivalences, namely conditions that are not identical as strings but are logically equivalent. These are strongly connected components in the graph.

Example 4.5. Continuing our running example, we further identify, after the graph is constructed, that $(2, 4)$ and $(3, 4)$ form a strongly connected component in the graph (Home is a binary feature, thus $y_{Home} \neq rent \iff y_{Home} = own$ and thus $C_{2,4} \iff C_{3,4}$). Figure 5b shows the condensation of G, where each equivalence class is contracted to a single node. For example, $([\![(2, 4)]\!], [\![(3, 3)]\!]) \in E'$ since $C_{2,4} \implies C_{3,3}$ and $C_{3,4} \implies C_{3,3}$.

4.3 Batch Evaluation

We will next leverage (the condensation of) the implication graph to sequentially invoke the CF generator in a way that facilitates reuse. In what follows, a *valid CF* for a given condition C is a CF that satisfies C and leads to the desired label (recall that for this optimization, we assume the label to be the same across all (query, prediction) pairs).

Algorithm 2 balances between the desiderata of re-use and short distance, as follows. It is given the condensed implication graph and a re-use threshold. Intuitively, if a CF was generated for an instance I and its distance (for a given metric) from some I' is below the threshold, we can re-use it for I'. The algorithm iterates (line 3) over the equivalence classes according to the topological sort defined by $\mathcal{G}'$. For each equivalence class, it iterates over all (query, prediction) pairs, retrieving for each pair its corresponding condition C (line 5) and original instance x (line 6). If there exists a qualifying CF in the pool whose distance is smaller than the threshold (lines 8, 11 and 12), then we use it, otherwise we invoke the CF generator to obtain valid CFs (lines 9 and 13). To allow further control over the quality/execution time tradeoff in case of very large pools, we also allow the option of restricting the search to a randomly selected subset (of configurable size) of the pool. In lines 18 - 20 we propagate the computed CFs to neighboring nodes, and in line 21 the obtained CFs are returned.

Complexity Analysis. The condensation graph is built in $O(|V|+|E|)$; then, the algorithm makes $O(|V|^2)$ calls to the distance computation function d.

Algorithm 2: Batch CFs Generation

```
Input  : Condensed implication graph $\mathcal{G}'$, Distance
         metric $d$, Threshold $t$, CF Generator $Gen$
Output : CF for each (query, prediction) pair in $\mathcal{G}'$
1  $CFs$ = empty map ;        // Stores all valid CFs
2  $Ans$ = empty map ;        // Stores closest CFs
3  foreach $v \in topological\_sort(\mathcal{G}')$ do
4      foreach $(i,j) \in v$ do
5          $C_{i,j} = Condition(i,j)$ ;
6          $x_{i,j} = Instance(i,j)$ ;
7          /* Collect valid CFs */
8          if $CFs[v].size = 0$ then
9              $CFs[v] = Gen(x_{i,j}, C_{i,j})$;
10         else
11             $cf = \arg\min_{cf \in CFs[v]} d(x_{i,j}, cf)$;
12             if $d(x_{i,j}, cf) > t$ then
13                 $CFs[v] = CFs[v] \cup Gen(x_{i,j}, C_{i,j})$;
14     foreach $(i,j) \in v$ do
15         /* Select closest CF */
16         if $CFs[v].size > 0$ then
17             $Ans[(i,j)] = \arg\min_{cf \in CFs[v]} d(x_{i,j}, cf)$ ;
18     /* Update valid CFs */
19     foreach $u$ s.t. $(v,u) \in \mathcal{G}'$ do
20         $CFs[u] = CFs[u] \cup CFs[v]$ ;
21 return $Ans$;
```

Example 4.6. Consider the condensation in Figure 5b. The order $[[\![(1,1)]\!], [\![(2,4)]\!], [\![(3,3)]\!]]$ is a valid topological order. Algorithm 2 loops over the nodes in this order. It first iterates over the equivalence class of $[\![(1,1)]\!]$, i.e., $[(1,1),(1,3)]$. It calls the CF generator for $Instance(1,1)$ with the condition $C_{1,1}$ and obtains a CF (denote it as cf_1). The algorithm updates the valid CFs of its equivalence class, i.e., $CFs[[\![(1,1)]\!]] = \{cf_1\}$. If cf_1 qualifies for $(1,3)$ in terms of distance, then we re-use it instead of invoking the CF generator for $Instance(1,3)$ with the condition $c_{1,3}$. Next, cf_1 will be propagated to the neighbors of $[\![(1,1)]\!]$, i.e., $[\![(3,3)]\!]$. The algorithm proceeds to treat the equivalence classes $[\![(2,4)]\!]$ and $[\![(3,3)]\!]$.

4.4 Going Beyond Classifiers

Our technical development and examples have so far focused on counterfactual-based analysis of *classifiers*. While indeed classifiers have been the main focus of attention in the context of CFs, some works have studied CFs for other models, including regression [32]. For classification, CFs are perturbations to the instance that lead to a change of label. A generalized notion of CFs can be defined as perturbations that lead to a change in the *model output*, which may be numeric, e.g. for regression. In the latter case, one may further introduce a threshold and aim for CFs that lead to a change whose magnitude is above the threshold. Our solution may be easily integrated with a CF generator for such models as well, as follows.

Note that the database schema (see Section 2.2 and the example in Figure 1) we have introduced for CFs is flexible, and the only relations that play a unique role in CFQL evaluation are `CFs` and `Prediction-CFs`. Other relations can be removed/replaced by others without changing the semantics (naturally, queries should be designed by the analyst according to the actual schema). For instance, for regression, the analyst may replace the `Classifiers` relation from Figure 1 by a `Models` relation, whose attributes are named accordingly; in the `Predictions` relation, they may replace the *label* attribute by a *result* attribute whose type is numeric.

In terms of our algorithmic solution, there are multiple ways in which it could be applied to regression models. It may be used as is, in which case constraints are formulated in CFQL and other desiderata such as the magnitude of change are coded as part of the CF generator. All of our optimizations still apply. Alternatively, if a CF generator for regressors further exposes an interface through which the magnitude of change could be set, then the framework can easily allow analysts to control it through CFQL as well, as follows. We may introduce a dedicated "result" attribute to `CFs`, that would capture the numeric model result over a CF instance; CFQL queries can then refer to this attribute, expressing e.g. the distance between the result of the model over the instance (stored in the `Instance` relation).

5 EXPERIMENTAL STUDY

We next present an experimental study, examining the usability and efficiency of our solutions. The source code is available in [10].

5.1 Experimental Setup

To our knowledge, there is no standard benchmark for CF analysis. We have thus created such a benchmark, based on datasets typically used as benchmarks in Machine Learning research.

Datasets: We have used (1) the Bank Marketing dataset of [25] containing 45K instances of clients and 16 features. The task is to predict whether the client will subscribe. (2) The Adult Income dataset from the UCI Machine Learning repository [5], pre-processed as in [26] and [13] to yield 26K instances and 8 features. The task is to predict whether the income of a given individual exceeds \$50K. (3) The COMPAS dataset [1] contains records about defendants and their estimated likelihood of re-offending. This dataset is commonly used in context of fairness analysis. We preprocess the dataset in the same way as done in previous works [1, 11, 26]. It includes five key features: age, gender, race, prior offense count, and the degree of criminal charge, pertaining to bail applicants. The objective is to decide which of the applicants are likely to re-offense within the next two years. Each dataset is randomly divided to train (70%) and test (30%) sets. We use a target encoding for the categorical columns.

Models: We used (1) `Logistic Regression` with L2 regularization; (2) A `Random Forest` with 10 trees and a maximal depth of 5 and (3) A `Neural Network` with the architecture used in [26], including two hidden layers of 64 neurons each and using ReLU as the activation function. All models achieved 88-90%, 81-83%, and 64-68% accuracy for the Bank Marketing, Adult Income, and COMPAS datasets respectively.

CF Generators: We have used the following implementations as CF generators: (1) *Cec*, the implementation described in [3]; its input is a distance function to optimize. We have used the "tabular distance" proposed in [30] (Section 3.1 there); (2) *Growing Spheres* implemented in [34] and described in [17]. We adapted the code (originally designed for numeric data) to fit tabular formats by introducing dedicated encoders, and added support for constraints through a component that filters out all (sampled) points that do not satisfy them. Other parameters of the generator need to be set. As default, we have set the initial sphere radius, reduction factor and sample size to 2, 0.5 and 200 respectively; (3) DiCE (Diverse Explanations): the CF generator proposed in [26], aims to generate a diverse set of CFs. The implementation of DiCE only supports constraints in the form of a conjunction of atomic conditions, and so we have only used it where applicable. We have used their `Random` mode, setting the number of requested CFs to 3.

`CFQL` *Queries:* We have designed `CFQL` queries of various flavors, representing multiple usage scenarios. For lack of space, the formal queries are given in [10] and we only informally describe them in Table 1. For Q_{CF} we write queries of 10 structures, with two different instantiations of each of them, corresponding to `loosely and densely interconnected` graphs. The `loosely (densely) interconnected` graph includes 13 (resp. 22) percent of all potential edges. For Q_{Pred}, we write 4 queries that select True Positives/Negatives and False Positives/Negatives.

Query	Description
$Q1$	Impose no constraints
$Q2$	Modify at least one feature in a given set
$Q3$	Modify all features in a given set
$Q4$	Keep intact at least one feature from a given set
$Q5$	Do not modify any feature from a given set
$Q6$	Either keep a given feature f_1 intact, or modify both f_1 and f_2
$Q7$	Either increase f_1 or keep f_2 intact
$Q8$	Modify f_1 in some way; modify f_2 to one of its k smallest values in the dataset
$Q9$	Modify f_1 to be larger than the average observed value and smaller than half of the maximum value
$Q10$	Modify f_1 to the least frequent observed value in the dataset

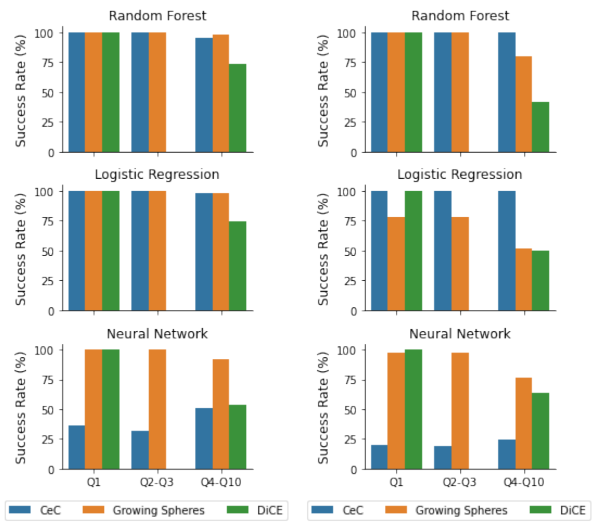
Table 1: Queries Benchmark

Baselines: We compare our solution to a baseline that invokes a CF generator (the same one used as Oracle for CFQL in the experiment) for every instance, without re-use of resulting CFs. This allows us to examine the effect of our optimizations that facilitate CF re-use on the quality of obtained CFs and on the number of oracle invocations and overall execution time. We also compare our optimized generation of the implication graph to approaches that do not incorporate some or all of our proposed optimizations.

Implementation and Measurements: We have implemented our optimizations in Python 3.8, with z3 as a SAT solver engine and SQLite3 as the database. To our knowledge, there is no public general-purpose implementation of c-tables; this is orthogonal to our effort, and we have thus manually crafted the c-tables corresponding to our queries. The computational challenge here stems from the hardness of CF generation, and by contrast, the databases that we examine are very small. We therefore only measure execution times for the optimizations in Section 4 and for CF generation (with and without the optimizations). The time incurred by the basic evaluation steps from Section 3.2 (not included in our experiments) is expected to be negligible in comparison. We also measure the tradeoff between CF quality and the number of CF Generator invocations (the bottleneck of a CF-based analysis). All experiments were run on an ASUS laptop with 8GB of RAM and an Intel(R) Core(TM) i5 CPU 2.5Ghz.

5.2 Usability

We start with a discussion on the usability of the approach: do the engines succeed in generating qualifying CFs? How much time does the CF generation take?

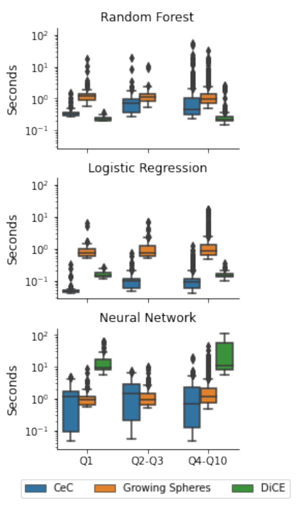


(a) COMPAS dataset **(b) Bank Marketing dataset**

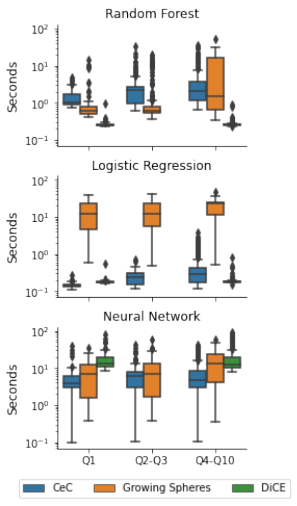
Figure 6: CF generation success rate

Figures 6 and 7 show the success rate and the distribution of execution times (in log scale), for 3 groups of queries, and for the COMPAS and Bank datasets (similar trends are observed for the Adult dataset; results are omitted for lack of space). Queries are grouped according to how constraining they are: (1) a query imposing no constraints ($Q1$), (2) queries that only impose disequality constraints over features ($Q2$ and $Q3$) and (3) queries that impose at least one equality/range constraint ($Q4$–$Q10$). Since the second group includes types of constraints that DiCE does not support, the Figures do not include numbers for DiCE and this group. In this experiment, we run the basic evaluation framework (Section 3); the effect of optimizations is extensively investigated in the subsequent experiments.

All algorithms exhibit a decline in success rates and increased execution times, as the queries become more complex. A particular performance drop is noticed for Q4-Q10 in context of Neural Networks. This drop is more significant (and the overall average success rate is lower) in the context of the Bank dataset which has more categorical features rendering constraints that are more difficult to satisfy. The CeC algorithm achieves good



(a) COMPAS dataset



(b) Bank Marketing dataset

Figure 7: CF generation execution time

performance for Random Forest and Logistic Regression, but fails more often (and is relatively slow when it succeeds) for Neural Networks. This may be attributed to the approach implemented in CeC, aiming to differentiate the gradients of instances and subsequently project them onto feasible instances with valid encodings; this approach is more challenging for Neural Networks than for Random Forests.

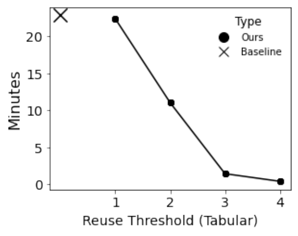
The main conclusion that we draw from this experiment is that in the presence of constraints, CF generation may be quite costly and/or fail, calling for optimizations across multiple invocations.

5.3 Use Case: Accelerating Fairness Computation

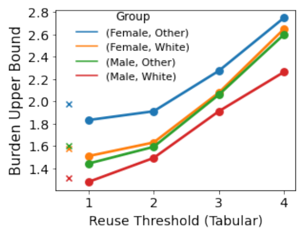
We use our framework to reproduce and extend the fairness experiment of [30]. The authors of [30] partitioned the instances based on their values in a pre-specified feature set (e.g. race and gender). The *burden* of an instance is then defined as the (tabular) distance from its closest counterfactual. Intuitively, the burden reflects the magnitude of needed change for the individual. The burden of a group is the average burden over all its instances. Intuitively, if the burden for a particular group is significantly larger than for another, then it may indicate the unfairness of the classifier with respect to this group.

To cast this experiment in our framework, we formulate CFQL queries to compute the burden with respect to the two attributes of race and gender and with respect to instances with unfavorable outcomes, from the UCI Adult dataset. The queries (whose formal form is deferred to [10] for lack of space) sample 503 instances from each group (503 is the size of the smallest group) to account for data imbalance, and use the Tabular distance function [30], restricting the search to randomly chosen 100 CFs from the pool for each instance. Our queries further ask for CFs that do not change the race and gender attributes.

Optimizing the analysis. Figure 8 compares the execution time and analysis quality of our solution, to the baseline approach of directly computing the CFs for all instances used for burden estimation (via invocations of the CeC CF generator). Note that since CF generators are not guaranteed to find the closest CF for a given instance (this is an NP-hard problem, see e.g. [3]),



(a) Execution time



(b) Burden upper bound

Figure 8: Burden calculation time and obtained burden values for the Adult Income dataset and two sensitive features Gender **and** Race**, using CeC as the CF generator**

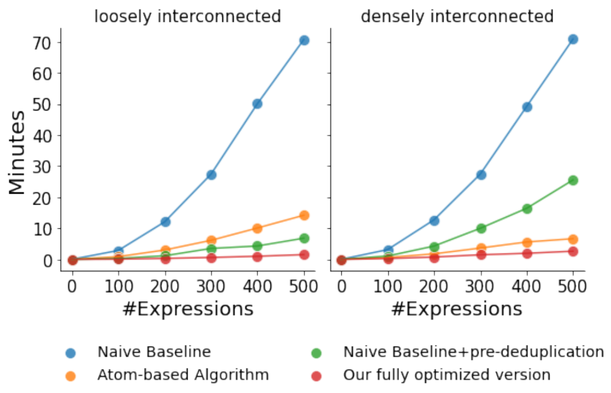


Figure 9: Implication graph construction time for varying number of expressions

the burden computed in the analysis (via our framework as well as via [30]) is in fact *an upper bound* for the actual burden. Our main motivation for CF reuse was reducing the number of CF Generator invocations. Yet, the hardness of finding optimal CFs means that CF reuse also has the potential of improving CF quality: a CF computed for some instance I may in fact be even closer to I' than the CF computed for I' itself.

Figure 8(a) shows the execution time of the baseline (approx. 25 minutes) compared to the execution time of our solution, for different values of the reuse threshold. Recall that the reuse threshold allows to control the tradeoff between execution time and obtained CF quality (see below an explanation for how to set the threshold.) Note that for a threshold value of 2 or more, the execution time using our framework is significantly faster than the baseline. In terms of quality, Figure 8(b) shows the computed burden for our approach (for different reuse thresholds) and the baseline. Recall that the computed burden value serves as an upper bound to the actual burden, and here we observe that for threshold value of 1, the upper bounds computed by our solution for every group are even better than the upper bounds computed by the baseline; when the threshold is set to 2 the bounds are similar to those computed by the baseline. The relative order of groups is further preserved, allowing to draw the same conclusions with respect to fairness, namely that the burden for the class (female, other) is the highest among the classes. Combining the experimental results, we conclude that for a reuse threshold of 1, we achieve results of better quality than the baseline, with similar execution time; for a threshold of 2 we improve the execution time and achieve similar quality; and for a threshold of 3 or 4 we further improve in terms of execution time, albeit with some degradation in the analysis quality.

Choosing the threshold. We have observed in hindsight that a threshold of 2 is a good choice. We will next illustrate how the threshold for CF re-use can be determined in advance. In our use case, we sample 100 data points (20% of the data points) from each combination of sensitive attributes. We then search for the closest data point with a favorable label that satisfies the constraints. The average distance from the closest data point thereby obtained for each group can serve as an upper bound for the true burden, for the sampled predictions. These distances yield an estimation of the proximity of available CFs which can guide the choice of threshold. Here, we obtain average distances of 1.66-2.24 for the different groups, implying that a reasonable choice for the threshold is 2.

5.4 Performance and Effect of Optimizations

We now study the performance and effect of our optimizations presented in Section 4. We use the full pool of available CFs for search. For lack of space, we show results for the Bank Marketing dataset, the Random Forest model, the CeC algorithm, and False Negative predictions. Some of the graphs are also shown for the COMPAS dataset, while others (exhibiting similar trends to another graph that is presented) are deferred to [10] for lack of space. Throughout the analysis, we distinguish between the loosely and densely interconnected instantiations of queries (see Section 5.1). In all plots, except for that of the implication graph construction, we compare against the baseline of invoking the CF generator (CeC) to find CFs for each instance, without re-use. The implication graph construction has no counterparts in prior work (to our knowledge), and so we compare to variants that do not include (subsets of) our optimizations, to examine their effect.

Implication Graph Construction. Figure 9 shows the construction time of the implication graph, given the boolean conditions from the corresponding c-tables, using the different algorithms presented above. The naive baseline that checks implications for all pairs of expressions requires over an hour for 500 expressions. In contrast, our optimizations reduce the graph generation time to 1 minute for the loosely connected graph and 2.5 minutes for the densely interconnected graph.

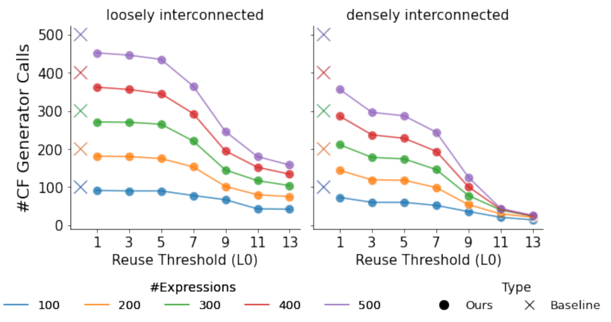


Figure 10: Number of CF Generator calls

CF Generator calls. Figure 10 compares the number of calls to the CF Generator using the batch evaluation algorithm (Algorithm 2) across multiple L0 reuse thresholds.

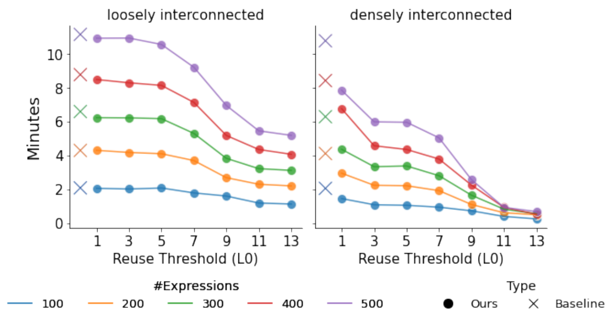


Figure 11: CF generation time

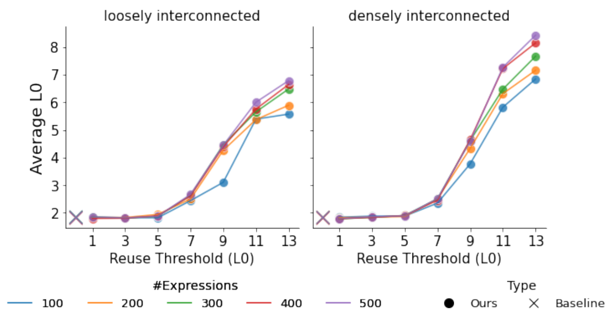
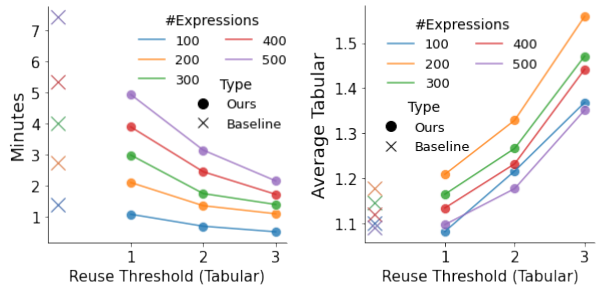


Figure 12: CF Quality (batch evaluation algorithm)



(a) CF generation time

(b) CF Quality

Figure 13: CF generation time and obtained distance (tabular) for the COMPAS Dataset

The baseline invokes the CF generator for every instance, and thus its number of invocations equals the number of expressions. Our framework allows to significantly reduce this number by increasing the threshold (see the discussion above on use cases, for how the threshold may be set in practice). For instance, when the threshold is set to 11, we reduce the number of CF generator calls from 500 to 180 for the loosely connected graph and to 44 for the densely interconnected graph. If we are interested in CFs of better quality (see below for the observed effect of the threshold on the CFs quality), we can , e.g., set the threshold to 7 and further reduce the number of CF generator calls from 500 to 365 for the loosely connected settings and to 244 for the densely interconnected graph. The reduction in the number of CF Generator calls is reflected by a similar reduction in the overall CF generation time (see Figure 11): for example, for 500 expressions and the densely interconnected graph, we reduce the generation time from 11 minutes to 5 minutes (to which one should add 2.5 minutes of constructing the implication graph). If we allow a threshold of 11, then the CF generation time is down to around 1 minute. For the loosely interconnected graph, the gain is significant but less so. For example, we can reduce the generation time from 11 minutes to a total of 9 (5.5) minutes by setting the threshold to 7 (11 resp.).

CF Quality. Figure 12 shows the obtained average distance between the CFs and their corresponding instance, for different thresholds. Observe that the CFs obtained in practice are often significantly closer to the original instance than the threshold allows. Up to a threshold of 5, the average distance is very similar to that of the baseline that does not apply reuse, and then it grows. In both settings, there is a very small difference in the average obtained L0 for different expression numbers. For 500 expressions, the L0 metric, and a threshold of 7, we obtain an average distance of approximately 2.5 for both graphs. For a threshold of 11, we obtain an average distance of approximately 6 (7.2) for the loosely (densely) interconnected graph.

Results for the COMPAS Dataset. Figure 13 presents the CF generation time and obtained CF quality, again as functions of the reuse threshold, for the COMPAS dataset. The dataset has less features than the Bank dataset, and so only reuse thresholds of up to 3 are of interest. The results reaffirm the conclusions we have derived above on the effectiveness of optimizations: we obtain faster execution times with a moderate decrease in quality.

6 RELATED WORK

Our work is the first, to our knowledge, to propose a generic optimization technique for counterfactual-based analysis that spans across multiple instances and constraints. In contrast, previous work has mostly focused on algorithms for finding CFs. These may be integrated into our framework as CF generators, and benefit from our optimizations. Other works have focused on efficient data structures and representations for the CFs themselves; these are orthogonal to our efforts, and potentially may be combined with our solution. We next provide details on some of these works.

Approaches and Algorithms for finding CFs. CFs, which have been discussed in fields other than ML for decades [18], have been shown to be intuitive for comprehension by humans [4] and useful as explanations [6]. Hundreds of different approaches have been proposed to generate CFs that explain ML predictions (see [11, 15, 35] for surveys). For a given prediction, different types of CFs with different merits are of interest. [36] aims at generating *minimal-distance CFs*, i.e., minimal changes to the input instance that lead to a modified prediction. DiCE [26] generates *diverse CF sets*, i.e., CFs involving perturbations to different features. Intuitively, diversification leads to more insights and a better chance of finding a useful, actionable recourse. The work of [21] uses a modified variational auto-encoder tuned to optimize for the feasibility of CFs. The work of [19] uses *class prototypes*, that guide CF generation towards the average encoding of the opposite class via an additional component introduced in the loss function. FACE [27] employs a graph-based approach, looking for feasible CFs, intuitively meaning that they are achievable via actions that “make sense". MACE [37] is a model-agnostic RL-based method for finding counterfactuals. There further exist many model-specific algorithms, leveraging characteristics of the model type to improve generation speed or quality [2, 31]. These and other solutions focus on counterfactual generation for a given instance. In contrast, in this work we focus on facilitating and optimizing CF-based analysis that involves looking for CFs for multiple instances. These works are thus complementary to our framework and optimizations, in the sense that they may be integrated into our framework as CF Generator oracles.

Relational Frameworks for Counterfactuals. In a recent work [29], the authors propose a relational framework called GeCo that supports generating, storing and analyzing CFs for ML models. They also allow the specification of constraints using a dedicated language. Unlike our framework, GeCo focuses on data structures that allow to optimize CF generation, storage and analysis for every *given* instance (and constraints, where available), whereas our focus is on optimization across instances and constraints, to avoid re-generation of CFs and to allow for their re-use. We thus view the work on GeCo as complementary to ours, and it may be integrated with our framework in two possible ways: (1) using GeCo as a black-box CF generator in our framework and (2) using the efficient representation of CFs in GeCo as our underlying data model. Another relational framework [7] focuses on the use of CFs for actionable recommendations for individuals. In contrast, our work focuses on leveraging CFs for general insights on the model, often involving multiple CFs generated over multiple instances. Last, we note that in the relational context, CFs have also been studied as a means of explaining query results [22, 23]. This is orthogonal to our work, where the CFs are defined with respect to ML predictions. A preliminary version of CFDB, that did not include optimizations, was demonstrated in a conference [24].

Other approaches for ML model analysis. Different approaches for analyzing and explaining ML models and their predictions, beyond counterfactuals, have been proposed. See e.g. [8, 12] for surveys. Notably, model agnostic attribution-based feature importance tools such as [20, 28] assign a score to the model features according to their contribution to the prediction. These are complementary to the CF approach (see e.g. [16]).

7 CONCLUSIONS

We have presented a novel relational framework for managing counterfactual explanations for Machine Learning model predictions. Analysts using the framework write queries that specify CFs of interest, which are compiled to an optimized sequence of invocations of black-box Counterfactual Generators. We have demonstrated the usefulness of the solution for multiple analysis tasks, including the evaluation of model fairness based on the properties of its CFs. Our framework is generic, allowing integration with any CF generator which in turn may be applied to any model. Given the significant interest in CF generation in the ML community, new solutions will likely be developed; these may be easily integrated in our framework.

Acknowledgments. This research has been partially funded by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 804302) and was (partially) supported by the Intuit University Collaboration Program grant.

REFERENCES

[1] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine bias: There's software used across the country to predict future criminals. And it's biased against blacks. *ProPublica* (May 2016). www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing

[2] André Artelt and Barbara Hammer. 2019. On the computation of counterfactual explanations – A survey. https://doi.org/10.48550/ARXIV.1911.07749

[3] Daniel Deutch and Nave Frost. 2019. Constraints-Based Explanations of Classifications. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 530–541. https://doi.org/10.1109/ICDE.2019.00054

[4] Jonathan Dodge, Q Vera Liao, Yunfeng Zhang, Rachel KE Bellamy, and Casey Dugan. 2019. Explaining models: an empirical study of how explanations impact fairness judgment. In *Proceedings of the 24th international conference on intelligent user interfaces*. 275–285.

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

[6] Carlos Fernandez, Foster J. Provost, and Xintian Han. 2020. Explaining Data-Driven Decisions made by AI Systems: The Counterfactual Approach. *CoRR* abs/2001.07417 (2020). arXiv:2001.07417 https://arxiv.org/abs/2001.07417

[7] Nave Frost, Naama Boer, Daniel Deutch, and Tova Milo. 2020. Personal Insights for Altering Decisions of Tree-based Ensembles over Time. *Proc. VLDB Endow.* 13, 6 (2020), 798–811.

[8] Krishna Gade, Sahin Geyik, Krishnaram Kenthapadi, Varun Mithal, and Ankur Taly. 2020. Explainable AI in Industry: Practical Challenges and Lessons Learned. In *Companion Proceedings of the Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 303–304. https://doi.org/10.1145/3366424.3383110

[9] Sainyam Galhotra, Romila Pradhan, and Babak Salimi. 2021. Explaining Black-Box Algorithms Using Probabilistic Contrastive Counterfactuals. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 577–590. https://doi.org/10.1145/3448016.3458455

[10] Project github. 2023. https://github.com/idanme45/CFDB.

[11] Riccardo Guidotti. 2022. Counterfactual explanations and how to find them: literature review and benchmarking. *Data Min. Knowl. Disc.* (2022).

[12] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)* 51, 5 (2018), 1–42.

[13] Zhu Haojun. 2016. Predicting Earning Potential using the Adult Dataset. https://rpubs.com/H_Zhu/235617.

[14] Tomasz Imieliński and Witold Lipski. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (sep 1984), 761–791. https://doi.org/10.1145/1634.1886

[15] Mark T Keane, Eoin M Kenny, Eoin Delaney, and Barry Smyth. 2021. If Only We Had Better Counterfactual Explanations: Five Key Deficits to Rectify in the Evaluation of Counterfactual XAI Techniques.

[16] Ramaravind Kommiya Mothilal, Divyat Mahajan, Chenhao Tan, and Amit Sharma. 2021. Towards Unifying Feature Attribution and Counterfactual Explanations: Different Means to the Same End. *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society* (Jul 2021). https://doi.org/10.1145/3461702.3462597

[17] Thibault Laugel, Marie-Jeanne Lesot, Christophe Marsala, Xavier Renard, and Marcin Detyniecki. 2018. Comparison-Based Inverse Classification for Interpretability in Machine Learning. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Theory and Foundations*, Jesús Medina, Manuel Ojeda-Aciego, José Luis Verdegay, David A. Pelta, Inma P. Cabrera, Bernadette Bouchon-Meunier, and Ronald R. Yager (Eds.). Springer International Publishing, Cham, 100–111.

[18] David Lewis. 1973. Counterfactuals and comparative possibility. In *Ifs*. Springer, 57–85.

[19] Arnaud Van Looveren and Janis Klaise. 2021. Interpretable Counterfactual Explanations Guided by Prototypes. In *ECML PKDD (Lecture Notes in Computer Science, Vol. 12976)*. Springer, 650–665.

[20] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4768–4777.

[21] Divyat Mahajan, Chenhao Tan, and Amit Sharma. 2019. Preserving Causal Constraints in Counterfactual Explanations for Machine Learning Classifiers. *CoRR* abs/1912.03277 (2019). arXiv:1912.03277 http://arxiv.org/abs/1912.03277

[22] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F Moore, and Dan Suciu. 2010. The complexity of causality and responsibility for query answers and non-answers. *arXiv preprint arXiv:1009.2021* (2010).

[23] Peter Menzies and Helen Beebee. 2001. Counterfactual theories of causation. (2001).

[24] Idan Meyuhas, Aviv Ben Arie, Yair Horesh, and Daniel Deutch. 2022. CFDB: Machine Learning Model Analysis via Databases of CounterFactuals. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2401–2404. https://doi.org/10.1145/3514221.3520162

[25] Sérgio Moro, Paulo Cortez, and Paulo Rita. 2014. A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems* 62 (2014), 22–31. https://doi.org/10.1016/j.dss.2014.03.001

[26] Ramaravind K. Mothilal, Amit Sharma, and Chenhao Tan. 2020. Explaining Machine Learning Classifiers through Diverse Counterfactual Explanations. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency (FAT* '20)*. New York, NY, USA, 607–617.

[27] Rafael Poyiadzi, Kacper Sokol, Raul Santos-Rodriguez, Tijl De Bie, and Peter Flach. 2020. FACE: Feasible and actionable counterfactual explanations. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. 344–350.

[28] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *KDD*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 1135–1144.

[29] Maximilian Schleich, Zixuan Geng, Yihong Zhang, and Dan Suciu. 2021. GeCo: Quality Counterfactual Explanations in Real Time. *Proc. VLDB Endow.* 14, 9 (2021), 1681–1693.

[30] 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.* Association for Computing Machinery, New York, NY, USA, 166–172. https://doi.org/10.1145/3375627.3375812

[31] Kacper Sokol and Peter Flach. 2019. Desiderata for Interpretability: Explaining Decision Tree Predictions with Counterfactuals. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 10035–10036. https://doi.org/10.1609/aaai.v33i01.330110035

[32] Thomas Spooner, Danial Dervovic, Jason Long, Jon Shepard, Jiahao Chen, and Daniele Magazzeni. 2021. Counterfactual Explanations for Arbitrary Regression Models. arXiv:2106.15212 [cs.LG]

[33] Ilia Stepin, Jose M. Alonso, Alejandro Catala, and Martín Pereira-Fariña. 2021. A Survey of Contrastive and Counterfactual Explanation Generation Methods for Explainable Artificial Intelligence. *IEEE Access* 9 (2021), 11974–12001. https://doi.org/10.1109/ACCESS.2021.3051315

[34] Laugel Thibault. 2018. https://github.com/thibaultlaugel/growingspheres.

[35] Sahil Verma, Varich Boonsanong, Minh Hoang, Keegan E. Hines, John P. Dickerson, and Chirag Shah. 2022. Counterfactual Explanations and Algorithmic Recourses for Machine Learning: A Review. arXiv:2010.10596 [cs.LG]

[36] 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.

[37] Wenzhuo Yang, Jia Li, Caiming Xiong, and Steven C. H. Hoi. 2022. MACE: An Efficient Model-Agnostic Framework for Counterfactual Explanation. arXiv:2205.15540 [cs.AI]