Constraint-based Explanation and Repair of Filter-based Transformations

Dolan Antenucci
University of Michigan
Ann Arbor, MI 48109
dol@umich.edu

Michael Cafarella
University of Michigan
Ann Arbor, MI 48109
michjc@umich.edu

ABSTRACT

Data analysts often need to transform an existing dataset, such as with filtering, into a new dataset for downstream analysis. Even the most trivial of mistakes in this phase can introduce bias and lead to the formation of invalid conclusions. For example, consider a researcher identifying subjects for trials of a new statin drug. She might identify patients with a high dietary cholesterol intake as a population likely to benefit from the drug, however, selection of these individuals could bias the test population to those with a generally unhealthy lifestyle, thereby compromising the analysis. Reducing the potential for bias in the dataset transformation process can minimize the need to later engage in the tedious, time-consuming process of trying to eliminate bias while preserving the target dataset.

We propose a novel interaction model for *explain-and-repair* data transformation systems, in which users interactively define constraints for transformation code and the resultant data. The system satisfies these constraints as far as possible, and provides an explanation for any problems encountered. We present an algorithm that yields filter-based transformation code satisfying user constraints. We implemented and evaluated a prototype of this architecture, Emeril, using both synthetic and real-world datasets. Our approach finds solutions 34% more often and 77% more quickly than the previous state-of-the-art solution.

PVLDB Reference Format:
Dolan Antenucci, Michael Cafarella. Constraint-based Explanation and Repair of Filter-based Transformations. *PVLDB*, 11 (9): 947-960, 2018.
DOI: https://doi.org/10.14778/3213880.3213886

1. INTRODUCTION

A common task in data analysis pipelines is to transform an existing dataset into a new dataset for downstream analysis. Such transformations may include filtering, aggregations, and data wrangling. Much effort has gone into improving this process for users via interactive interfaces [17, 28], transformations by example [23, 25, 43, 44, 45], and more [13, 15, 48].

Proceedings of the VLDB Endowment, Vol. 11, No. 9

DOI: https://doi.org/10.14778/3213880.3213886

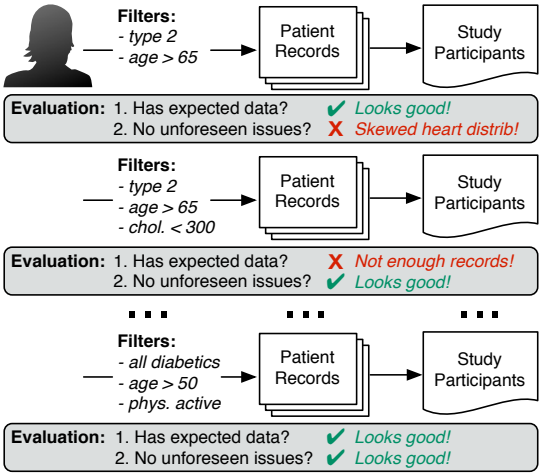

Figure 1: An illustration of Example 1, showing the time-consuming and tedious process of repairing data transformations.

Even the most trivial of transformations can introduce undesirable bias and invalidate conclusions. For example, a medical researcher testing a new statin drug might filter a patient database in an effort to identify people likely to benefit from the test drug. Filtering for subjects with high dietary cholesterol intake may seem reasonable, but it may bias the test population to those with a generally unhealthy lifestyle, compromising the analysis. While an analyst with a functional knowledge of statistics would likely identify this bias, similar mistakes still appear in the literature [7, 18, 37].

Eliminating any bias identified in the dataset can be time-consuming and tedious. The best transformation code does not introduce any undesirable bias and adheres closely to the analyst's original plan. Consider the following example, illustrated in Figure 1:

Example 1. *Janet is a researcher who is identifying subjects for a study of a new diabetes medication. She believes the medication will be most effective for type 2 diabetics, so she* ***adds a filter for*** `type2 = True`. *In the past, she had problems with follow-up in middle-aged participants, so she* ***filters for*** `age > 65`. *After discussing the plan with her colleague, she realizes that older people have worse cardiovascular health than is typical, thereby giving a misleading picture of the drug's effectiveness, so she* ***filters for*** `cholesterol < 300`. *The resulting set of patients is too small for her study, so she* ***removes*** `age > 65`. *The new*

result still has issues with the distribution of cardiovascular health, so she ***removes*** `cholesterol < 300` *and* ***adds*** `exercises = True`. *The result of these changes yields too few patients again, so she* ***removes*** `type2 = True` *and* ***adds*** `diabetic = True`. *The results are closer to what she wants, so she* ***re-adds*** `age > 65`. *Now the patient counts are again too low, so she* ***relaxes the age filter to*** `age > 50`.

Janet had to perform six rounds of filtering and time-consuming data analysis to obtain her desired dataset.

In an ideal environment, Janet would describe her target dataset, and an external system would find a dataset that most closely matches her goal—while helping her avoid any undesirable bias. Consider Janet using an *explain-and-repair* data transformation system:

EXAMPLE 2. *Once again, Janet is identifying subjects for a study. She describes her ideal dataset to the system* `(type2 = True, age > 65, COUNT(subjects) > 500)`*, and the system responds with a transformed dataset and a list of columns whose distributions have changed (e.g., prevalence of cardiovascular disease is skewed). Janet clicks "Do Not Allow" for the cardiovascular disease bias warning, and the system responds with a new dataset, updated transformation code, and a new list of column distribution changes. Janet sees no undesirable bias in the new list of distribution changes and happily accepts the system's result.*

System Goals — We propose an explain-and-repair data transformation system. The system takes the following input: **(1)** An existing dataset to transform; **(2)** User-provided *signal-based* constraints that indicate the desired characteristics of the transformed dataset; **(3)** User-provided *code-based* constraints that indicate the desired characteristics of the transformation code; and **(4)** User-provided *aggregate-based* constraints that indicate the desired aggregate conditions on the transformed dataset.

The system has the following desiderata: **(1)** Finds a set of transformations and the data that the code produces, which best match the user's constraints; **(2)** Explains potentially undesirable bias to the user, requests feedback, and uses this to create a new result; and **(3)** Responds with a result within a reasonable timeframe (i.e., several minutes).

For the current work, we assume all transformations are in the form of filters (i.e., relational selections); thus, some transformations are not possible (e.g., folding or extraction), but for many applications—such as research in economics [31, 33, 36, 46, 49] and medicine [16, 19, 30, 40, 41] that use curated datasets (several of which we test in our experiments)—filters are the only transformations needed.

Technical Challenge — Using a large pool of candidate transformations, the explain-and-repair system must find the set of transformations that best match the user's constraints. This amounts to a combinatorial search problem, but one with a costly objective function: the code must be evaluated with any code-based constraints, the transformations must be applied to the input dataset, and the resulting dataset must be evaluated with any signal-based constraints.

A somewhat similar challenge is faced in *how-to* querying [35], in which the goal is to find the dataset that best satisfies a set of constraints, but this goal does not include determining what transformation code to use. Extending how-to querying work in a straightforward manner, so that the resulting dataset and the transformation code are both considered, causes the problem to quickly become intractable, even for very small datasets (as detailed in Section 3).

Table 1: Notation used to describe our user model.

	Description
$\mathcal{D}$	Raw input dataset
$\mathcal{T}$	Desired output dataset
$\mathcal{C}$	Set of candidate transformations
O^*	The ideal transformation program that yields $\mathcal{T}$
$\mathcal{U}_i$	User constraints at query cycle i
$\mathcal{W}_i$	User preference weights for $\mathcal{U}_i$
$\mathcal{R}_i$	Generated output dataset that best matches $\mathcal{U}_i$
O_i	Transformation program that yields $\mathcal{R}_i$
$\mathcal{P}_i$	Set of system-identified problems for $\mathcal{R}_i$

Our Approach — Inspired by integer programming solutions to combinatorial optimization [39], we represent the problem as that of constrained optimization, which we solve using nonlinear programming. In our optimization model, we create binary variables for each transformation, which are then used to estimate a column distribution for each signal-based constraint. The model's objective function sets the binary variables to minimize the sum of squared differences between the estimated signals and their associated constraints, and maximizes *code-similarity* between the code-based constraints and the chosen transformations.

Contributions — Our contributions are as follows:

- We propose a novel interaction model for constraint-based explanation and repair of data transformations, finding data that best matches user constraints without any surprise issues (Section 2).
- We present an algorithm for finding transformations that closely match a set of constraints. The algorithm models this as a constrained-optimization problem, which it solves using nonlinear programming (Section 4).
- We built a prototype system, EMERIL, and evaluated it with synthetic and real-world data, showing that our approach is 77% faster and finds 34% more solutions than the previous state-of-the-art solution (Section 6).

2. USER MODEL

In this section, we present a user model for explain-and-repair data transformation systems. The terminology introduced in this section is summarized in Table 1.

2.1 Overview

When formulating a ***transformation program***, an analyst has a ***raw input dataset*** that she wants to transform into a ***desired output dataset***. She has an initial set of goals for her desired output dataset, which may include:

- Which items it should contain (e.g., *type 2 diabetics*);
- The distribution of particular items (e.g., *a Gaussian-distributed prevalence of cardiovascular disease*);
- The number of data items (e.g., *at least 500 subjects*);
- Which transformations to use or avoid (e.g., *no gender filtering*).

This information, or set of ***user constraints***, drives the creation of the transformation program, which yields a ***generated output dataset***. Unfortunately, finding a transformation program that satisfies all of the constraints *can be a challenge for an analyst*, so she will likely revise her goals at least once. Even after several revisions, without careful inspection or foresight, the generated output dataset can unknowingly differ from the analyst's desired result.

Figure 2 summarizes how a user can build a transformation program in conjunction with an explain-and-repair system. Janet (from Example 2) starts by providing her raw dataset and initial constraints to the explain-and-repair system (**Step 1**). The system responds with a generated output dataset, transformation program, and a warning that the distribution for cardiovascular disease prevalence has changed (**Step 2**). Since the system treats the constraints as *soft constraints*, they may be violated or imperfectly true in the output data. Janet responds with a constraint that indicates the cardiovascular distribution should be Gaussian (**Step 3**), and the system responds with a new transformation program and generated dataset (**Step 4**). These ***query cycles***, in which the user specifies constraints and receives a result, continue until the user is happy with her result.

Background — For our user model, we drew inspiration from a common formula for research in the social and life science domains: some process—often orchestrated by a government agency—collects and updates a well-curated dataset that captures some general observations (e.g., health details of citizens, or economic statistics), and a researcher uses a subset of the data to draw conclusions. Examples of this can be found throughout economics [31, 33, 36, 46, 49], medicine [16, 19, 30, 40, 41], and more. Since our current work focuses on filter-based transformations, there are some use cases we do not support, such as grouping-based transformations [26, 27] and non-declarative, or ordered, transformations [26], which we leave for future work.

In Section 2.3, we introduce the different constraint types a user can provide. We selected these constraints with reference to our past work in economics [11], and to our review of publications matching our target use cases, which allowed us to anticipate where problems are likely to arise. For instance, a TargetDistrib constraint would help an economist ensure an unbiased range of ages when filtering for certain transactions to evaluate the response of spending to income changes [21]. Likewise, a TargetCount constraint could ensure that a minimum number of consumers are represented to avoid sampling bias.

2.2 Data Transformation Primitives

Raw Input Dataset — This is the dataset that the user wants to transform into a desired output dataset, both of which we assume are relational datasets:

Definition 1 (Raw input dataset). *An input dataset is a set $\mathcal{D}$, where each $d \in \mathcal{D}$ is a tuple of values $(v_1, v_2, ..., v_k)$ with an associated tuple of column names $(c_1, c_2, ..., c_k)$.*

Definition 2 (Desired output dataset). *Dataset $\mathcal{T}$ is the output dataset a user desires from applying a transformation program O^* to raw input dataset $\mathcal{D}$.*

Transformation Program — This is the means of producing a generated output dataset. We limit this paper's focus to only filter-based transformations, or *predicates*:

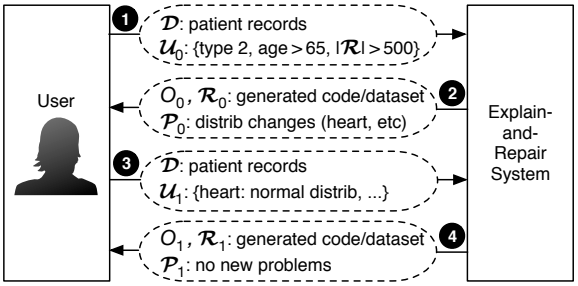

Figure 2: The user model for an explain-and-repair data transformation system, showing Janet's usage in Example 2.

Definition 3 (Predicate). *Predicate $p = (c, o, v)$ is a tuple representing a filter-based transformation such that operator o and value v form a binary test on column c in $\mathcal{D}$, indicating the included rows in a generated output $\mathcal{R}_i$.*

Definition 4 (Candidate Transformations). *$\mathcal{C}$ is the set of all potential predicates applicable to $\mathcal{D}$.*

Definition 5 (Transformation program). *The set $O_i \subseteq \mathcal{C}$ is a set of transformations. The conjunction of O_i applied to $\mathcal{D}$ produces a generated output dataset $\mathcal{R}_i$.*

Definition 6 (Ideal transformation program). *The set $O^* \subseteq \mathcal{C}$ is the transformation program that produces the desired output dataset $\mathcal{T}$.*

Generated Output Dataset — This is the result of applying a transformation program to a raw input dataset:

Definition 7 (Generated output dataset). *Dataset $\mathcal{R}_i$ is the generated output dataset from applying transformation program O_i to $\mathcal{D}$.*

Given dataset and transformation program similarity functions $S_d()$ and $S_c()$, we can now formalize the ***data transformation problem***:

Problem 1. *Given raw input dataset $\mathcal{D}$ and a set of candidate transformations $\mathcal{C}$, find a transformation program $O_i \subseteq \mathcal{C}$ that maximizes $S_d(\mathcal{R}, \mathcal{T})$ and $S_c(O_i, O^*)$.*

The similarity functions $S_d()$ and $S_c()$ can be defined in a number of ways, and Sections 4.2 and 4.5 explain how our algorithm defines them.

2.3 System Input

When using an explain-and-repair system, users provide an input dataset $\mathcal{D}$ and a set of user constraints.

User Constraints — This is a collection of constraints that define the user's desired output dataset at query cycle i:

Definition 8 (User Constraints). *$\mathcal{U}_i$ is a set of user constraints $\mathcal{U}_i = \{U_a, U_s, U_x, U_t\}$, such that:*

1. *U_a is a set of TargetDistrib constraints.*
2. *U_s is a set of DesiredPred constraints.*
3. *U_x is a set of NoPred constraints.*
4. *U_t is a set of TupleCount constraints.*

TargetDistrib Constraints — These constraints are used to indicate a desired distribution for a column in the output dataset (e.g., Janet would specify that the *prevalence of cardiovascular disease* should have a Gaussian distribution).

Definition 9 (TargetDistrib constraint). *A TargetDistrib constraint is a tuple $u = (c, s)$ such that:*

1. *Column name c indicates the name of a column in $\mathcal{D}$.*
2. *Density estimation signal s indicates the desired distribution of column c.*
3. *For probability density function f, $f(\mathcal{R}_i[c]) \simeq s$.*

When specifying density estimation signal s, users have two options: for continuous values, users provide a signal. In past work [9], we provided a web-based tool for "drawing" signals; something similar could be used to assist users. For multinomial values, users provide a set of desired counts or percentages (e.g., *gender*: 60% male, 40% female).

Users can also provide *code-based constraints*, which indicate desired characteristics of the resulting transformation code. These can also be inferred from a user's initial transformation code, which the explain-and-repair system *repairs* by adjusting or removing predicates to best satisfy the entire set of constraints.

DesiredPred Constraints — These constraints are used to indicate when a predicate should be used to generate a result. For example, Janet may require that her study include only females, thus *(gender = 'female')* would be used.

Definition 10 (DesiredPred constraint). *Predicate u is a DesiredPred constraint indicating $\exists\ p \in O_i$ s.t. $p \simeq u$.*

NoPred Constraints — These constraints are used to indicate when a particular column should *not* be used to generate a result. For instance, Janet may require that all levels of patient income be included in her study, so she would specify *income_level* as a NoPred constraint.

Definition 11 (NoPred constraint). *Column name u is a NoPred constraint indicating a column in $\mathcal{D}$, where $\forall p = (c, o, v) \in O_i, c \neq u$.*

TupleCount Constraints — These constraints indicate a required minimum or maximum number of data items in the generated output dataset. For instance, Janet would specify that she wants a minimum of 500 study subjects.

Definition 12 (TupleCount constraints). *Tuple $u = (m, n)$ is a TupleCount constraint, where $m = \min \mid \max$ and $|\mathcal{R}_i| \leq n$ if $m = \min$, otherwise, $|\mathcal{R}_i| \geq n$.*

Constraint Preference Weights — It is often impossible to satisfy all of the user constraints simultaneously, so the explain-and-repair system treats them as soft constraints. To help the system prioritize which to relax first, users can specify a set of *preference weights* for their constraints (If not provided, uniform weighting is assumed):

Definition 13 (Preference Weights). *Set $\mathcal{W}_i$ is a set of sets, where $W_{ij} \in \mathcal{W}_i$ defines the preference weights for one of the user constraint types in $\mathcal{U}_i$, and $\sum \mathcal{W}_i = 1$.*

2.4 System Output and Evaluating Results

The explain-and-repair system output at a given query cycle i includes generated output dataset $\mathcal{R}_i$, transformation program O_i, and a set $\mathcal{P}_i$ of *system-identified problems*.

System-Identified Problems — For a given result, the system identifies any distribution changes amongst columns in $\mathcal{R}_i$, notifying the user that these may be potential problems. For instance, Janet is notified that the distribution of *cardiovascular disease prevalence* has changed.

Algorithm 1 Tiresias-based approach to our problem

Input: $\mathcal{D}, \mathcal{C}, \mathcal{U}$
Result: constrained-optimization problem formulation

1: $binTuples = \{b_1 : (...),\ ...\}$; $tuplePreds = \{t_1 : (...),\ ...\}$
2: $tuples = \{0, 0, ...\}$; $preds = \{0, 0, ...\}$
3: $t_i = (!p_1 \vee tp[i,1]) \wedge (!p_2 \vee tp[i,2]) \wedge ... \wedge (!p_k \vee tp[i,k])$
   where $t_i = tuples[i], p_k = preds[k], tp = tuplePreds$
4: $binCounts[b] = \sum_{t \in binTuples[b]} tuples[t]$
5: minimize $\sum_{b \in binIds} \text{abs}(binCounts[b] - targetCounts[b])$

Definition 14 (System-identified problem). *Tuple $p = (c, s_1, s_2)$ is a system-identified problem, where c is the name of a column in $\mathcal{D}$, and for a density estimation function f, $s_1 = f(\mathcal{D}[c])$, $s_2 = f(\mathcal{R}_i[c])$, and $s_1 \neq s_2$.*

Evaluating Results — When evaluating $\mathcal{P}_i$, a user will indicate if any distribution changes are undesired by providing a TargetDistrib signal-based constraint for each problem. Similarly, if the transformation program O_i has any issues, the user will provide a DesiredPred or NoPred code-based constraint for each problem. If output dataset $\mathcal{R}_i$ has too many or too few records, the user can provide a TupleCount aggregate-based constraint.

3. REVERSE DATA MANAGEMENT

Finding a transformation program that best matches a set of user constraints is an example of a *reverse data management* problem [34]. The most similar area of research from this domain is *how-to* querying, which aims to modify an existing dataset to satisfy select constraints. Meliou and Suciu developed Tiresias [35], which solves general how-to queries using linear programming; Users describe with a declarative query language their data, constraints, and desired actions (modify a row value, add tuples, etc.), and Tiresias finds the best set of actions that creates the desired dataset.

Tiresias works well for how-to querying, but it does not solve our particular problem. Both Tiresias and an explain-and-repair system aim to produce a dataset that matches a set of constraints. However, an explain-and-repair system must also find a transformation program that produces the dataset, while Tiresias does not. With a few alterations, we can adapt Tiresias for application to our problem. Consider Algorithm 1, whose inputs are a raw input dataset $\mathcal{D}$, a set of candidate transformations $\mathcal{C}$, and user constraints $\mathcal{U}$. We first create mappings between tuples, bins (bins are from a histogram on the user constraint signal), and predicates (line 1). Then we create binary variables to indicate if a tuple or predicate is used in a solution (line 2). Next we create constraints that define whether each tuple is used in a solution (line 3) and what the estimated counts are for each bin (line 4). Finally, we define the objective function for the constrained-optimization solver (line 5).

Unfortunately, representing just one of the logical statements in line 8 as a constraint in the optimization problem actually requires *several* constraints and variables. For n tuples and p predicates, the number of constraints and variables are each $O(np)$. While these grow linearly with the input size, constrained-optimization solvers can only handle a finite number of constraints and variables. For this particular approach, we found that with any more than 500 tuples and 100 predicates, the solver Tiresias uses (GLPK [32]) was unable to find a solution after three hours, at which point we terminated the program.

4. FINDING DATA TRANSFORMATIONS

In this section, we propose a novel algorithm for finding a data transformation program that best matches a user's constraints. Unlike a Tiresias-based solution, our approach scales well with the number of tuples in a dataset.

4.1 Overview

The core idea of our approach is to represent our problem as that of constrained-optimization, in which we model the impact each candidate transformation has on matching the user's constraints and use a constrained-optimization solver to find the set of transformations that maximize this match.

Algorithm 2 shows the steps that are needed. The user provides raw input dataset $\mathcal{D}$, user constraints $\mathcal{U}_i$, and an optional preference weights $\mathcal{W}_i$. The first step is to analyze the input dataset to generate candidate predicates and precompute some information used in the problem formulation (line 1). This work can be done offline, as we discuss in Section 5. We then formulate the problem as a constrained-optimization problem (line 2), find a solution using nonlinear programming (lines 3 – 4), and analyze the result for potential problems (line 5). This is the core of our paper's contribution, and we discuss it in detail below.

Before we discuss our approach, readers will benefit from a basic understanding of constrained optimization: the optimization solvers take as input a set of constants, variables, and constraints, along with an objective function. The constants dictate the initial conditions, and the solver finds values for the variables that maximize the objective function, subject to the provided constraints.

4.2 Modeling as an Optimization Problem

To start, we will assume that predicates are independent of each other and that we are matching a single TargetDistrib constraint (e.g., Janet specifying that the distribution of *cardiovascular disease prevalence* should be Gaussian and centered on an average level). If two predicates are independent, then the probability of a row being included when both predicates are applied to a dataset is the same as the product of their marginal probabilities (i.e., $P(A, B) = P(A)P(B)$). Later we will relax these two assumptions.

Target Bounds — For a TargetDistrib constraint with column c and signal s, the goal is to maximize the similarity between s and the distribution of c in the generated output dataset $\mathcal{R}_i$. In order to avoid having the solver calculate probability densities for every candidate solution (or having to perform time-consuming signal comparisons), our algorithm discretizes s into *bins* to create a *target histogram*. This allows for faster distribution matching with only minimal impact on accuracy.

Our algorithm then discretizes the original distribution of c in $\mathcal{D}$ as a set of percentages and finds the maximum number of data items in each bin that satisfies the target histogram. This defines the *target counts*. For example, if *cardiovascular disease prevalence* originally has a uniform distribution of (200, 200, 200), then histogram $(b_1, b_2, b_3) = (25\%, 50\%, 25\%)$ would yield target counts of (100, 200, 100) (dividing the histogram values by the largest bin (b_2), thus taking 25/50 of b_1 and b_3, and 50/50 of b_2). Lastly, our algorithm converts the target counts into *target bounds* by creating an interval around the counts with a *slack* parameter ϵ, allowing for an ϵ margin of error when finding solutions.

Algorithm 2 Overview of our algorithm's solution finding

Input: $\mathcal{D}, \mathcal{U}_i, \mathcal{W}_i$

1: $\mathcal{C}, pp$ = analyzeDataset($\mathcal{D}$)
2: $model$ = formulateProblem($\mathcal{D}, \mathcal{U}_i, \mathcal{C}, pp$)
3: $solverResult$ = runOpSolver($model$)
4: $O_i, \mathcal{R}_i$ = processSolverResult($solverResult$)
5: $\mathcal{P}_i$ = identifyProblems($\mathcal{R}_i, \mathcal{U}_i, \mathcal{W}_i$)
6: return $O_i, \mathcal{R}_i, \mathcal{P}_i$

Model Variables — When formulating the optimization problem, our algorithm creates a set of binary variables, *preds*, with one for each predicate $p \in \mathcal{C}$. These are used to indicate whether a predicate is included in the transformation program. An additional set of variables, *bc*, is created to store the estimated bin counts from applying a candidate solution to $\mathcal{D}$. The constants defined by our algorithm include the precomputed bin probabilities for each predicate (see Section 5), the target counts, the target bounds, and the probabilities for each bin in the target histogram.

Model Constraints — Our algorithm creates a constraint that estimates the bin counts from applying a candidate solution on $\mathcal{D}$. The intuition underlying this procedure is that the probability of a row being included by a set of predicates equals the probability of the bin multiplied by the product of the predicates conditioned on the bin. Thus, the estimated count for a particular bin is the product of this joint probability and the size of the dataset. For example, if there are 200 records in $\mathcal{D}$, bin b_1 contains 25% of them, and two predicates are applied with b_1 probabilities of 0.1 and 0.6, then the estimated count for $b_1 = 200 * 0.25 * 0.1 * 0.6$, or 3 records. This probabilistic approach allows the solver to estimate the result of applying a set of predicates without having to decide whether particular tuples are included in a resulting dataset as a Tiresias-based approach would.

Equation 1 displays how the count is calculated for bin b, where $bc[b]$ is the estimated bin count, $bp[b]$ is the bin probability (or percentage of $\mathcal{D}$ in bin b), $bpp[i, b]$ is the bin probability for predicate variable $preds[i]$, and c is a small constant (e.g., 1×10^{-6}) used to avoid a log of zero. To clarify this with an example, consider a dataset separated into two bins with probabilities $bp = (0.8, 0.2)$ and with one candidate predicate with bin probabilities $bpp[1] = (0.3, 0.05)$. The counts for the first bin $bc[1] = |\mathcal{D}| * 0.8 * 0.3$.

$$bc[b] = |\mathcal{D}| * bp[b] * \exp\Big(\sum_{i=1}^{|preds|} preds[i] * \log(bpp[i, b] + c)\Big) \quad (1)$$

Our algorithm also adds constraints that limit the estimated counts to within the target bounds and ensure that at least one predicate is chosen.

Objective Function — Since our goal is to maximize similarity between the desired output dataset and the generated output dataset, our algorithm adds the following objective function to our optimization model, which finds the set of predicates that minimizes the sum of differences between the target and estimated bin counts:

$$\text{minimize} \sum_{b=1}^{|bins|} \text{abs}(tc[b] - bc[b]) \quad (2)$$

This approach fixes the core problem with a Tiresias-based approach, but its assumption of independent predi-

Algorithm 3 Prioritizing of dependence information

Input: $\mathcal{D}, \mathcal{U}_i, pp, pdp, \epsilon, randExplore$

```
1: depScores = getDepScores(pp)
2: binCounts = getBinCounts(D, U_i)
3: T = getTargetBounds(D, U_i)
4: for all pids, s ∈ depScores do
5:     estCounts = (∏_{p∈pids} pp[p]) * binCounts
6:     if withinBounds(T, estCounts) then
7:         delete depScores[(p_1, p_2)]
8:     end if
9: end for
10: numRand = randExplore * pdp * |depScores|
11: numDep = (1 − randExplore) * pdp * |depScores|
12: finalDepScores = depScores[0 : numDep]
13: delete depScores[0 : numDep]
14: finalDepScores += randomSample(depScores, numRand)
15: return finalDepScores
```

cates is a key weakness that makes it unsuitable for many datasets. In the next section, we remove this assumption.

4.3 Adding Dependence Information

In the previous section, we describe how our algorithm uses a probabilistic approach to estimating the impact of including a predicate in a candidate solution. However, if two predicates are included in a candidate solution that have some dependency between them (i.e., their matching rows overlap more often than would be expected by chance), then the estimated bin counts (Equation 1) will be incorrect.

To remedy this problem, our algorithm includes dependence information of the form $D = \{\mathbf{d}_1, \ldots, \mathbf{d}_m\}$, where $\mathbf{d}_i = \{p_1, p_2, \ldots, p_q \mid 0 < p_j \leq |\mathcal{C}|\}$. It then uses D to create additional constants, variables, and constraints in its optimization problem, which allows the solver to correctly estimate the bin counts for a candidate transformation program that has dependent predicates.

Additional Variables — For each $\mathbf{d}_i \in D$, our algorithm creates a binary variable in the optimization model, which, if true, causes bin counts to be estimated as if $\forall p \in \mathbf{d}_i$ are included in a candidate solution. These binary variables are combined with the previously created *preds* variables, creating $preds'$, which has $|\mathcal{C}|+|D|$ binary variables for which the solver is finding values.

Our algorithm creates additional constants for the bin probabilities of each dependent predicates set, similar to those added for the individual predicates. Additionally, a set of constants, *pred_sets*, is added to the optimization model, where each $pred\_set \in pred\_sets$ links a synthetic predicate variable with its associated predicates.

Additional Constraints — Our algorithm imposes the following restriction in order to avoid misestimations of the bin counts for a candidate solution with dependent predicates: If the synthetic variable for a dependent predicates set $\mathbf{d}_i$ is true, then $\forall p \in \mathbf{d}_i : preds'[p] = \text{false}$. Likewise, if $\exists p \in \mathbf{d}_i : preds'[p] = \text{true}$, then the synthetic variable for dependent predicates set $\mathbf{d}_i$ must be false. Equation 3 summarizes this constraint:

$$\forall pred\_set \in pred\_sets : \sum_{i \in pred\_set} preds'[i] \leq 1 \qquad (3)$$

For example, assume that p_{1001} is a synthetic predicate that represents the set of dependent predicates (p_1, p_2). Our constraint uses $pred\_set = (p_1, p_2, p_{1001})$ to enforce that only one of these predicates can be used in a candidate solution (otherwise their $preds'[i]$ sum is larger than one).

Algorithm 4 Predicate similarity function

Input: p_1, p_2

```
1: score = 0.0
2: if p_1 == p_2 then
3:     score = 1.0
4: else if p_1.column == p_2.column then
5:     score += 0.5
6:     if (isRange(p_1) ∨ isRange(p_1)) ∧ hasOverlap(p_1, p_2) then
7:         score += 0.25
8:     end if
9: end if
10: return score
```

4.4 Prioritizing Dependence Information

Our approach to including dependence information fixes the problem of inaccurate estimates from dependent predicates, but another problem remains: adding dependence information to the optimization model leads to more constraints, and finding solutions becomes more difficult.

Our algorithm addresses this problem by using a *filtering with exploration* policy for limiting dependence information in a model (Algorithm 3). Using precomputed dependence scores for each dependent predicates set (described in Section 5), each set is tested to determine whether its estimated counts are within a margin of error (ϵ) of being independent, and, if so, the dependency set is excluded (lines 4 – 9). Next, the top *pdp* percent of the remaining dependency sets are chosen as the final output, with a *randExplore* percent of those replaced by a random selection of dependency sets (lines 10 – 14). This *random exploration* of predicate dependency helps find solutions that are more common but have predicates with relatively low dependency.

4.5 Adding Code-Based Constraints

In addition to signal-based constraints, users can provide code-based DesiredPred or NoPred constraints, where the objective is to include (or exclude) predicates that are similar (or dissimilar) to the constraint. For instance, Janet in Example 2 provides DesiredPred constraints of `(type2 = True)` and `(age > 65)`, indicating that her ideal transformation program would have predicates similar to these. Thus, a transformation program with `(age > 50)` is preferred over one with no age filter, or one that has `(age < 65)`.

To model this, our algorithm creates additional constants, variables, and constraints, along with an updated objective function, so that the solver prefers solutions that match well on both signal- and code-based constraints.

Additional Variables — Our algorithm adds a set to the model to hold the code-distance calculations, which are used in the objective function. Additionally, a constant is added that defines a similarity matrix between the code-based constraints and each of the predicates. This is generated using a simple code-similarity function that rewards matching columns, operators, and similar ranges of values (Algorithm 4).

Additional Constraints — Our algorithm adds one additional constraint, which uses the code-similarity matrix to define the code distances for any predicate variables that are true. Equation 4 defines this constraint, where U_c is the set of DesiredPred constraints, D_c is array of code distances being estimated, and S is the code-similarity matrix:

$$\forall u \in U_c : D_c[u] = \sum_{p \in preds} (p * (1 - S[i, u])) \qquad (4)$$

Updated Objective Function — The updated objective function, which supports both signal- and code-based constraints, is defined in Equation 5, where D_s is an array of signal distances (i.e., the objective function in Equation 2), D_c is the array of code distances from Equation 4, and $\mathcal{W}$ is the user's constraint preference weights:

$$\text{minimize} \sum_{i=1}^{|U_s|} \mathcal{W}[U_s, i] * D_s[i] + \sum_{i=1}^{|U_c|} \mathcal{W}[U_c, i] * D_c[i] \quad (5)$$

Adding Semantic Information — There are a variety of cases in which users would want to include richer information about the input dataset. For instance, an ontology of the dataset's values could indicate that type 2 diabetes is a class of diabetes. Including this external semantic information is beyond the scope of this project, but one possible way would be to redefine our distance metric in Algorithm 4 to reward a higher similarity score to predicates with associated fields. If a user initially filters for type 2 diabetics, our algorithm would then prefer a filter for all diabetics over one for patients with cancer.

4.6 Optimization Solver

Our algorithm assumes that an off-the-shelf constrained-optimization solver is used to process its formulated optimization problem; however, these solvers often have several internal parameters that control their success at finding solutions, which can be an issue. For many problems the default settings work adequately, but for others the solver can reach a local minimum prematurely and conclude that a problem is infeasible or that a solution cannot be found. Our algorithm offers dynamic configuration changing, so that multiple configurations can be used as a workaround.

4.7 Result Generation

After the optimization solver produces a result, our algorithm processes it into a transformation program and generated output dataset, and then checks for potential problems.

Processing the Solver Output — If the solver finds a solution, it returns the chosen predicates to our algorithm, which replaces any synthetic predicates (i.e., each $\mathbf{d}_i \in D$) with the associated set of predicates and adds them with the other predicates to the transformation program O_i. Our algorithm then determines the validity of the solution by applying O_i to input dataset $\mathcal{D}$ to generate output dataset $\mathcal{R}_i$, which is then compared with the user's constraints. For invalid answers, our algorithm first tries to adjust the configuration of the solver (as described in Section 4.6). Second, the slack parameter ϵ is relaxed, changing what defines a valid solution. Finally, the amount of dependence information included in the model can be increased.

Identifying Potential Problems — For a given result, our algorithm iterates through the different columns to compare their distributions before and after applying a transformation program. Pearson correlation is used for this comparison, where any columns with at least a δ percentage change are added to the system-identified problem set $\mathcal{P}_i$.

4.8 Greedy-Hybrid Approach

For certain problems, a greedy heuristic can replace the constrained-optimization solver in our algorithm; Our algorithm has an optional *greedy-hybrid* approach, in which it uses a greedy heuristic to search for a valid solution. If none is found, our algorithm proceeds with the constrained-optimization solver.

The greedy heuristic works as follows: First, it defines a threshold k to control the maximum number of predicates in a solution. It then finds the single predicate that best matches the target counts. Next, it pairs this predicate with all remaining predicates, finding the set of predicates that best match the target counts. This process repeats until k candidate solutions are determined, and then it selects the best of them as the final answer.

The advantage of the greedy heuristic is its relatively quick search for solutions (with $O(k|\mathcal{C}|)$ time complexity), but it requires the best single predicate (found in its first step) to be part of a valid solution. For this reason, the heuristic on its own is not as generalizable as a greedy-hybrid approach (as we show in our experiments in Section 6).

Algorithm 5 Full algorithm for finding solutions

Input: $\mathcal{D}, \mathcal{U}_i, \mathcal{W}_i, pdp, \epsilon, randExplore$

1: $\mathcal{C}, pp = \text{analyzeDataset}(\mathcal{D})$
2: $S = \text{getFilteredDepSets}(\mathcal{D}, \mathcal{U}_i, pp, pdp, randExplore, \epsilon)$
3: $T = \text{getTargetBounds}(\mathcal{U}_i, \epsilon)$
4: $O_i, \mathcal{R}_i = \text{greedySearch}(\mathcal{D}, \mathcal{U}_i, \mathcal{W}, \mathcal{C}, \epsilon)$
5: **if not** isValid$(O_i, \mathcal{R}_i, \mathcal{U}_i)$ **then**
6: $\quad preds' = \text{getModelPreds}(\mathcal{D}, pp, \mathcal{C})$
7: $\quad config, \epsilon' = \text{getSolverConfig}(\mathcal{D}, \mathcal{U}_i)$
8: $\quad model = \text{formulateProblem}(\mathcal{D}, \mathcal{U}_i, pp, T, S, preds', \epsilon')$
9: $\quad solverResult = \text{runOpSolver}(model, config)$
10: $\quad O_i, \mathcal{R}_i = \text{processSolverResult}(solverResult)$
11: **end if**
12: $\mathcal{P}_i = \text{identifyProblems}(\mathcal{R}_i, \mathcal{U}_i, \mathcal{W}_i)$
13: return $O_i, \mathcal{R}_i, \mathcal{P}_i$

4.9 Full Algorithm for Solution Finding

Algorithm 5 summarizes our full algorithm for finding solutions. In line 1, the raw input dataset is analyzed. Next, in lines 2 – 3, the dependent predicate sets are filtered using the method in Algorithm 3 and the target bounds are determined. In lines 4 – 5, the greedy-hybrid approach is used, and if a valid solution is found, the optimization problem is skipped. Otherwise, the optimization problem is formulated (lines 6 – 8), and the optimization solver is used to find a solution (line 9). In line 10, the solver result is processed into the transformation program and generated output dataset. Lastly, the transformed dataset is evaluated for potential problems (line 12), and the result is returned to the user.

4.10 Time Complexity

Our algorithm has two main components that affect its runtime: problem formulation and solving the constrained optimization problem.

Problem Formulation — The main bottleneck in problem formulation is scoring and sorting the dependent predicate sets. Since sets can be up to k in size, for p predicates, roughly p^k sets can exist, so if scoring a single set has $O(s)$ time complexity, the total scoring has $O(s * p^k)$ time complexity. When we include score sorting, time complexity increases to $O(s * p^k + p^k \log(p^k))$. However, we have found that $k = 2$ is good enough for most datasets, so this simplifies to $O(s * p^2 + p^2 \log(p^2))$.

Optimization Solver — The solvers we use employ active set methods for solving constrained-optimization problems. These approaches can have exponential time complexity in

the worse-case input instance, but, in practice, are generally recognized as much better: $O(n^3)$ for dense problems. However, the cost of nonlinear functions and gradients enters into this complexity. Our problem grows primarily as the quantity of dependent predicate sets increases, because a new model constraint is created for each added set. Generally, any more than a few million sets will result in a runtime of over an hour, and that without guarantee of a potential solution.

5. PROTOTYPE SYSTEM

We embody our algorithm in a prototype software system, EMERIL, which was written in over 6,800 lines of predominately Python source code. EMERIL also makes a few decisions regarding data-specific policies, described below.

Emeril Interaction Model — Users interact with EMERIL by either explicitly declaring their constraints, or by providing an SQL query, from which the constraints are inferred. Our current software is a prototype system, so any revisions to queries or rollbacks to previous ones require a full round of solution-finding. Additionally, while some real-world applications may benefit from distinguishing soft and hard constraints, we assume all constraints are soft. Both of these limitations can be addressed in a production system.

Data-Specific Policies — EMERIL uses the constrained-optimization solvers MINOS [38] and SNOPT [22], where SNOPT is preferred for larger datasets. We use AMPL [20], an algebraic modeling library, to interface with these solvers. As discussed in Section 4.6, our algorithm allows for dynamic solver configurations, so we performed a grid search on several test datasets to define a few configurations for each solver.

EMERIL defaults to the following approach for generating candidate predicates: equality predicates are generated for each value in any categorical columns. Then, for numerical columns, a histogram is generated with the column's values, with three predicates generated for each bin edge: a range predicate between it and the next edge, and greater-than and less-than inequality predicates. This yields the set of candidate transformations $\mathcal{C}$.

Since our algorithm uses predicate dependence information to improve its accuracy, EMERIL precomputes this information before processing queries. Using the formula for independence ($P(A,B) = P(A)P(B)$), we measure dependence between two predicates using $d = abs(P(A,B) - P(A)P(B))$. EMERIL parallelizes this step to quickly create the dependence information. This yields the *depScores* variable in Algorithm 3 from Section 4.4.

As part of the dataset analysis, EMERIL precomputes probability distributions for all predicates and sets of dependent predicates: EMERIL first determines the data items that each predicate or set of predicates matches. It then calculates a density estimation signal for each column in input dataset $\mathcal{D}$ to produce the probability distributions. This yields the *pp* variable in Algorithm 5 from Section 4.9.

6. EXPERIMENTS

In this section, we evaluate our three claims about EMERIL:

1. EMERIL performs better than competing methods at finding high-quality data transformation programs. We evaluated this on synthetic datasets with varying levels complexity (Section 6.2), as well as on several popular real-world datasets (Section 6.3).
2. EMERIL performs well at finding solutions when code-based user constraints are specified, all with no meaningful impact on runtime (Section 6.4).
3. Each individual component of EMERIL contributes meaningfully to EMERIL's ability to find transformation programs that best satisfy user constraints (Section 6.5).

6.1 Experimental Setting

In this section, we describe our baseline methods, evaluation metrics, and our experimental configuration.

Baseline Methods — We evaluated EMERIL against three baseline methods:

- **Tiresias** — For this approach, we use the adaption of Tiresias we describe in Section 3. We expect this approach to scale poorly with increasing dataset size.
- **NChooseK** — For this approach, we test all combinations of up to k predicates ($k = 2$), using the set of predicates with the best score as the final answer. We expect this approach to work well for smaller problems but to be prohibitively time-consuming for larger problems.
- **Greedy** — For this approach, we use the greedy search algorithm described in Section 4.8, with $k = 2$. We expect Greedy to perform well only on problems in which the first selected predicate is part of a valid solution.

Evaluation Metrics — Our synthetic datasets and answers have some randomness associated with their generation, so we repeated each experiment for all variations of the datasets. From this, we measured the ***percentage of times a solution is found*** and the ***average runtime***. Runtimes for all systems do not include offline analysis or pre-loading of data from disk. A production system can always pre-load data to improve query times.

Experimental Configuration — We ran our experiments on a 32-core 2.8GHz Opteron 6110 server with 512 GB RAM. We imposed a one hour time limit on the different methods for finding a solution. While users will have different wait time preferences, we feel this is a reasonable timeframe. For NChooseK, if the time limit expired, the best answer found up to that point was used as the final answer.

6.2 Evaluating on Synthetic Data

In this set of experiments, we tested EMERIL and our baseline methods using synthetic datasets with varying size, correlation, and input data complexity. EMERIL used the top 10% of dependency sets (with a 50% *randExplore* setting) for these experiments.

6.2.1 Synthetic Datasets

We generated our synthetic datasets to control for the following properties, which impact discovery of high-quality transformation programs:

- **Input data complexity** — We varied the number of candidate transformations (a rough proxy for input data complexity, since a more varied dataset will yield more candidates) from 1,000 to 5,000, in increments of 1,000.
- **Number of rows** — We varied the number of rows from 100,000 to 1 million, in increments of 100,000.

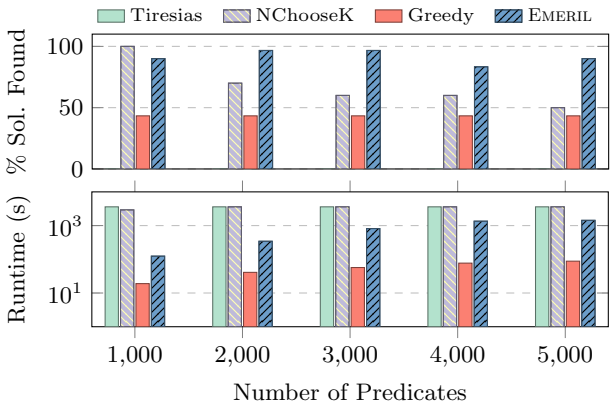

Figure 3: Percent of solutions found and runtime (log scale) for Emeril and our baselines with a varied number of predicates.

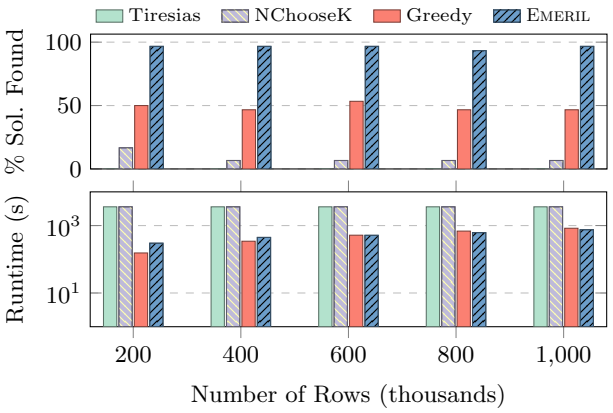

Figure 4: Percent of solutions found and runtime (log scale) for Emeril and our baselines with a varied number of data items.

- **Level of data correlation** — We varied the correlation level between columns from 0.0 to 1.0, in increments of 0.1.

Dataset Generation — We repeated the following process with different random seeds to generate 30 synthetic datasets for each property configuration described above:

1. We chose a target number of rows, columns, and data correlation. When varying a property, we used one of the variations listed above; otherwise, we used 50,000 rows, 100 columns, 2,000 candidate predicates, and an average correlation of 0.2.
2. We generated a relational dataset with the target number of rows and columns by sampling values from a Gaussian distribution. To control the level of correlation between columns, the distribution was defined with a covariance matrix that achieved the target correlation.
3. For candidate predicates, we generated inequality predicates for each unique column value, and then randomly sampled our target number.

Synthetic Answer Generation — The primary use case of Emeril is to match signal-based constraints, so to test this in our experiments without entangling our results with user behavior, we mimicked a user providing a TargetDistrib constraint that specifies a desired parabolic-shaped distribution on a column with a Gaussian distribution. This constraint was chosen to support operations like those we described in Section 1. For example, Janet may see that *age* in her input dataset follows a Gaussian distribution, but wants no middle-aged patients for her experiment.

Since the Greedy baseline success is heavily dependent on the kind of synthetic answer we generate, we want to ensure that some of our synthetic answers are Greedy-compatible and others are not. As a result, we added a "honeypot" column, which serves as a local maximum for Greedy to choose. It is only the true global maximum roughly 50% of the time, so in these cases, Greedy will fail to find a valid answer.

6.2.2 *Varied Input Data Complexity*

Summary: Emeril found a valid solution *34% more often* than our best baseline (NChooseK), in *77% less time*, while Tiresias failed to find a valid solution.

Overview — In this experiment, we vary the number of candidate predicates (a rough proxy for input data complexity) from which a solution is selected, which is the most important factor in problem difficulty.

Results — Figure 3 summarizes the results, which show that Emeril was able to find answers at significantly greater frequency than our baseline methods, and with a reasonable runtime—finding a solution 91% of the time, averaging 14 minutes per run. By contrast, our baseline methods produced mixed results. Tiresias was unable to find any solutions in the allotted time, as predicted for the reasons described in Section 3. Greedy found answers 43% of the time, averaging 1 minute per run. NChooseK found answers 68% of the time, but generally used the full hour to do so—with the notable exception of instances with 1000 predicates, since NChooseK was able to exhaustively search in the hour provided, finding valid solutions 100% of the time. It is worth noting that for small datasets, NChooseK can be an reasonable option: in some earlier experiments, which we did not include due to space limitations, we had 100 candidate predicates and 50,000 rows, and Emeril and NChooseK both found valid solutions 100% of the time (averaging 4 seconds and 77 seconds respectively). While the combinatorial methods produce some promising results, they are not a comprehensive solution for our problem, because they function in limited cases.

6.2.3 *Varied Number of Rows*

Summary: Emeril found valid solutions 103% more than our best baseline (Greedy), while Tiresias fails to produce a valid solution.

Overview — In this experiment, we varied the number of rows in each dataset. For most methods, this number impacts how quickly a candidate answer can be evaluated.

Results — As shown in Figure 4, Emeril found valid solutions 97% of the time and had great runtimes, averaging 8 minutes per run. We designed our algorithm in Section 4 to have a time complexity independent of the number of rows in a dataset, but when using the Greedy-hybrid search (as these results are), the runtime of a Greedy search is added to Emeril's total runtime. The Greedy search on its own averaged 7.8 minutes per run, but found valid solutions just 48% of the time. Tiresias was unable to find a solution for any of the experiment runs, reaching the one hour time limit

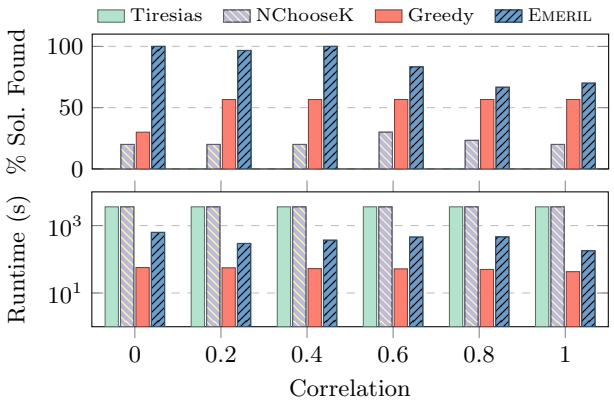

Figure 5: Percent of solutions found and runtime (log scale) for Emeril and our baselines with a varied level of correlation.

Table 2: Real-world datasets used in our experiments.

Dataset	Rows	Cols	Preds	Corr
ATUS [1]	68,077	374	3,363	0.05
Food Facts [5]	65,503	159	1,103	0.18
GTD [2]	170,350	135	1,262	0.02
NFL Play Data [4]	46,129	66	535	0.02
NHANES [3]	9,813	693	9,618	0.13
WDI [6]	409,992	62	1,712	0.78

in all cases. NChooseK struggled with the large number of candidate predicates and increasing number of rows, thus exhausting the one-hour time limit on all runs and finding valid solutions only 10% of the time.

6.2.4 *Varied Level of Data Correlation*

Summary: Emeril found valid solutions 59% more than our best baseline (Greedy), while Tiresias fails to produce a valid solution.

Overview — In this experiment, we varied the level of correlation between columns in the datasets. As this number increases, candidate predicates become more correlated, and since Emeril's constrained optimization model assumes predicate independence when no other information is provided, we expect valid solutions to be more challenging to find as the correlation level increases.

Results — Figure 5 summarizes the results. Across all datasets, Emeril found valid solutions 85% of the time, with a runtime of around 6 minutes per run. For datasets with correlation of 0.5 or below, Emeril found valid solutions 94% of the time, but for those with correlation above 0.5, Emeril found solutions 73% of the time. This lower success rate is because Emeril has a small budget of dependence information in its constrained optimization model (to help keep runtimes low), and when data correlation levels get too high, the synthetic answer's dependence information is generally not included, even though it is needed. While Emeril may struggle with highly correlated datasets, we have found the incidence of highly-correlated datasets to be relatively infrequent in real-world datasets. For instance, only one of the six real-world datasets discussed in this paper has an average correlation above 0.18 (see Table 2).

For our baselines, Tiresias reached the time limit before finding any solutions. Even with the relatively small dataset used in these experiments, the required model becomes too complex for an optimization solver to solve in the time allotted. Greedy was able to find valid solutions 53% of the time, averaging 53 seconds of runtime. NChooseK again struggled with the large number of candidate predicates, exhausting the one-hour time limit on all runs, and only finding valid solutions 23% of the time.

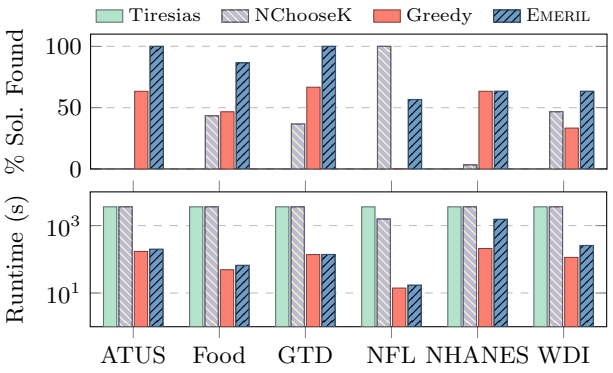

Figure 6: Percent of solutions found and runtime (log scale) for Emeril and our baselines evaluated using real-world datasets.

6.3 Evaluating with Real-World Data

Summary: Emeril finds solutions 44% more often than the best baseline (Greedy), increasing average runtime from 3 minutes to only 14 minutes. Tiresias never finds a solution.

Overview — In these experiments, we evaluated Emeril and our baseline methods using real-world datasets (summarized in Table 2), many of which are used extensively in research (e.g., ATUS and NHANES are US government health surveys, and WDI is the World Bank's Development Indicators). The candidate predicates were created using the method described in Section 5. Emeril used the top 1% of dependency sets (with a 0% *randExplore* setting) for these experiments. We chose slightly different values from the synthetic dataset experiments to account for the larger dataset sizes and less common target answers.

We defined a TargetDistrib constraint for a column in each dataset (e.g., *carbohydrates* was used for Food Facts), which the competing systems try to match. In order to ensure that there was a valid answer to evaluate against, we inserted a synthetic answer using a similar approach to that of our previous experiments: two columns in the data were replaced with synthetic values that match with two predicates we created, so that, when applied together on the dataset, the result matches the signal of the TargetDistrib constraint. While using a synthetic answer does not fully capture a real-world scenario, it allowed us to evaluate our systems on data that has lots of real-life dataset characteristics without adding the additional complexity of arbitrary user preferences.

Results — As shown in Figure 6, Emeril found solutions 71% of the time, averaging 15 minutes per run. For our baseline methods, Tiresias was unable to find any solutions during the allotted time; NChooseK found a valid solution 38% of the time, averaging 54 minutes per run; and Greedy quickly found valid solutions 42% of the time, averaging just 2 minutes per run.

Emeril also has comparatively poor performance on the NFL and NHANES datasets (finding 57% and 63% of the

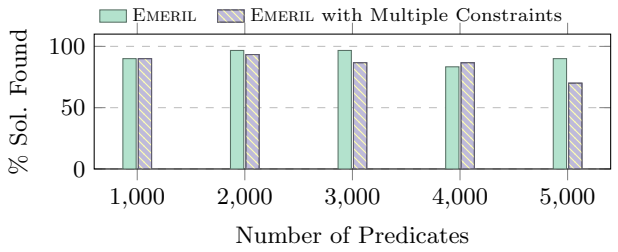

Figure 7: Percent of solutions found by Emeril when matching code-based constraints (i.e., repairing) versus matching just one signal-based constraint.

solutions respectively), but for different reasons: NHANES has the largest number of predicates of any test set, so the 1% limit added a very large amount of dependence information to our algorithm; this caused Emeril to exhaust the 1-hour time limit before finding a solution.

For NFL, by pure chance, the target answers in the datasets have predicates that are almost—but not entirely—independent. That means our naïve approach from Section 4.2, which assumes total independence, would not find the answer, nor does our standard algorithm, which does not rank the relevant predicates very highly; We will have to include a much larger percent of the dependence information in order to find the correct answer. Indeed, we found that when increasing the percentage of dependency sets included in the optimization model from 1% to 20%, the percentage of solutions found for NFL increases from 57% to 67%. Somewhat ironically, NChooseK does very well on the NFL dataset, as it is small enough to exhaustively search all predicate sets—suggesting a potential front-end to our system that runs NChooseK when the input data is sufficiently spare.

Emeril also performs relatively poorly on the WDI dataset, and this can be attributed to its high level of correlation. As we describe in 6.2.4, Emeril will often struggle with highly-correlated datasets.

6.4 Evaluating with Code-Based Constraints

Summary: When the user provides a code-based constraint, and there are multiple solutions that match identically on a signal-based constraint, Emeril prefers the solution with better code similarity 100% of the time, with no meaningful impact on runtime.

Overview — In this experiment, we tested how well Emeril performs when matching both signal- and code-based constraints. As described in Section 2.3, a user may provide an initial transformation program (or explicitly provide a predicate that should be in the answer). The system then tries to *repair* this initial program to satisfy the signal-based constraints, while making minimal changes to the initial program. For example ($age > 30$) would have a decent match with the initial code of ($age \geq 40$), but a poor match with ($gender$ = "female"), thus the former would be preferred if all else is equal.

We assumed the user provided both code- and signal-based constraints. We used the same datasets as in our varied schema complexity experiments (Section 6.2.2), except that we duplicated the synthetic answer with new column names. Thus, two synthetic answers exist, each equally match the signal-based constraint, but one better matches the code-based constraint.

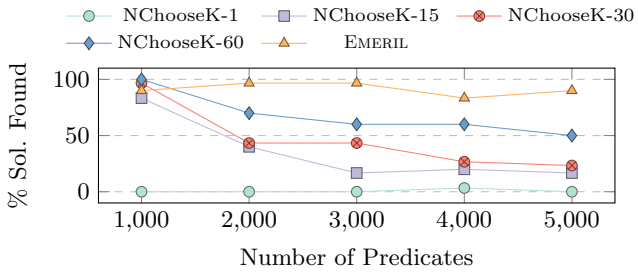

Figure 8: Percent of time a solution found when varying the number of predicates and using either Emeril or NChooseK with varied max runtimes of 1, 15, 30, and 60 minutes.

Results — As Figure 7 shows, including a code-based constraint makes finding solutions slightly more difficult, finding valid solutions 85% of the time instead of 91% when matching just the signal-based constraint. Of the valid solutions found when matching multiple constraints, Emeril preferred the solution with the better code similarity 100% of the time. Finally, total runtime of Emeril was unaffected by adding the additional code-based constraint, as both sets of experiments averaged 14 minutes per run.

6.5 Evaluating Algorithm Components

In this set of experiments, we tested whether each of the various components of Emeril is essential.

6.5.1 *Optimization Solver*

Summary: Using the constrained optimization solver allowed Emeril to perform better than any competing method.

Overview — Using the optimization solver is core to Emeril, and the alternative to using it is to simply use one of our baseline methods (e.g., Greedy or NChooseK). In Section 6.2, we show that these baseline methods did not perform as well as Emeril; however, we can also evaluate a few variations of NChooseK, where the allowed runtime is decreased, thus the best answer within that time limit is used. We tried four variations of NChooseK, allowing time limits of 1, 15, 30, and 60 minutes.

Results — As Figure 8 shows, Emeril found valid solutions 91% of the time (averaging 14 minutes per run). The best version of NChooseK, which used 60 minutes per run, only found valid solutions 68% of the time. Given enough time, NChooseK would find solutions 100% of the time, but as the number of predicates grows (N), or the maximum number of predicates in an answer grows (K), an exhaustive combinatorial search becomes prohibitively slow.

6.5.2 *Using Dependence Information*

Summary: When Emeril used dependence information, it found valid solutions 4x more often than Emeril without it.

Overview — The next component we tested was the use of predicate dependence information in our optimization problem. As described in Section 4.3, this information is needed for most real-world datasets that have some dependencies amongst its data; If the data are completely independent, then no dependence information is needed.

Results — Table 3 shows the results from using Emeril with dependence information (Emeril, *10% Dep Info*) and without it (NoDep) on all of our synthetic datasets. When

Table 3: Ratio of solutions found for EMERIL with and without prioritized dependence information (greedy search disabled).

		5% Dep Info		10 % Dep Info	
Experiment	NODEP	RAND	EMERIL	RAND	EMERIL
Predicates (1k – 5k)	0.00	0.70	**0.77**	0.72	**0.82**
Rows (100k – 1m)	0.00	0.14	**0.85**	0.27	**0.95**
Corr. (0.0 – 1.0)	0.38	0.30	**0.56**	0.24	**0.62**

including dependence information, EMERIL found valid solutions 79% of the time, whereas NODEP only found 16% of valid solutions—a 4x improvement when including dependence information. NODEP is unable to find any valid solutions for the varied-predicates datasets nor the varied-rows datasets. With the varied-correlation datasets, NODEP does great with lower levels of correlation, finding solutions 70% of the time for correlations from 0.0 – 0.5. This is due to the independence assumption holding, at least within our target bounds' margin of error.

6.5.3 Prioritizing of Dependence Information

Summary: EMERIL used with prioritization of dependence information found valid solutions much more often than EMERIL used without it.

Overview — In this experiment, we tested the benefit of EMERIL's prioritization of predicate dependence information in our optimization model. We tested this by disabling our Greedy-hybrid search and replacing EMERIL's method for including dependence information with a process that randomly selected 5% and 10% of the unordered dependence information (RAND, *5%* and *10% Dep Info* respectively). We compared this with EMERIL using the top 5% and 10% of dependence information.

Results — Table 3 summarizes these results. When comparing each method using 5% of the dependence information, EMERIL found solutions much more often than RAND on each of the varied data experiments. RAND still performed relatively well on some experiments; In the predicate-based experiment, as the number of predicates grow, there is a greater chance of a valid answer being included by random, especially if there are multiple valid solutions in a dataset. This is why EMERIL includes a small percentage of randomly-sampled dependence information in its model. In the correlation-based experiment, the independence assumption again allows EMERIL to find some valid solutions without prioritized dependence information. When comparing the two methods using 10% of the dependence information, we see similar results: EMERIL found valid solutions much more often than EMERIL used without it.

7. RELATED WORK

There are several areas of research that are related to our work, which we discuss below.

Data Preparation — Much research has been done on improving the ease of data preparation, or transforming data for some downstream analysis, for users. This includes helping users with data extraction [48, 13, 15], wrangling [43, 44, 45, 23, 25]; and interactive systems [28, 17]. Our work is a subtype of automatic program generation like Trifacta [17] or Foofah [25], but in our case, the user guides program generation by providing constraints around desired outputs, instead of with programming-by-demonstration or concrete output tuples.

Reverse Data Management — In reverse data management [34], an action is taken on an input dataset to achieve a certain effect, such as in *how-to* querying [35], where the goal is to produce an output that satisfies one or more constraints. Tiresias [35] similarly uses constrained optimization, but, as we show in Section 3, applying their approach to our problem becomes impractical with the growth of dataset size or candidate transformation space.

Why-Not Provenance — In *why-so* and *why-not* querying, users want to understand why query results are as they are, such as why particular tuples are included or missing from a result. Methods include finding hypothetical database modifications that yield a desired result [24], examining the query execution graph for a problematic manipulation [14], and modifying the original query to include a desired result [42]. Our work is similar to `ConQueR` [42] in that we suggest query modifications, but our goal is to alter the distribution of a particular column, which likely requires removing tuples in addition to adding them. Further, `ConQueR` assumes inequality predicates when determining how to include why-not tuples, which would further limit its ability to help with our problem.

Query Formulation and Refinement — Query formulation is a well-studied problem in database literature, in which the goal is to help users formulate or revise queries so that generated results better match users' desired results. Past work has used query logs [29] and the data being queried [47] to aid with this formulation. Our work has the similar goal of aiding users with query refinement, except we ask the users for explicit constraints, which we use to formulate a transformation program.

Constraint-Based Querying — Constraint-based querying systems, such as CONQUEST [12] and our work with RACCOONDB [8, 9, 10], provide custom query languages for defining constraints, which are used to find relevant results. Our work and RACCOONDB both use code- and signal-based constraints to find a small subset of items in a large pool of candidates. However, RACCOONDB's constraints are less expressive than EMERIL and assume a PCA-based pipeline that makes transformation program generation substantially easier than in the explain-and-repair task.

8. CONCLUSION AND FUTURE WORK

In this work, we present a novel interaction model and algorithm for explain-and-repair data transformation systems, in which users provide constraints on their desired transformation program and the data it produces, and the system finds the best match for both. We show that our constrained-optimization-based algorithm for matching user constraints outperforms the previous state-of-the-art solution. Future work includes expanding beyond filter-based transformations and allowing for near real-time interactive exploration of the tradeoff between different constraints.

9. ACKNOWLEDGEMENTS

This project is supported by National Science Foundation grant 1054913, a University of Michigan MIDAS grant, and gifts from Yahoo! and Google.

10. REFERENCES

[1] American Time Use Survey - Bureau of Labor Statistics. http://www.bls.gov/tus. Accessed: 2016-06-01.

[2] Global Terrorism Database. http://start.umd.edu/gtd. Accessed: 2016-06-01.

[3] National Health and Nutrition Examination Survey - CDC. http://www.cdc.gov/nchs/nhanes. Accessed: 2016-06-01.

[4] NFL API. http://api.nfl.com/docs. Accessed: 2016-06-01.

[5] Open Food Facts - World. http://world.openfoodfacts.org. Accessed: 2016-06-01.

[6] World Development Indicators - World Bank. http://data.worldbank.org. Accessed: 2016-06-01.

[7] Lets think about cognitive bias. *Nature*, 526(163), Oct 2015.

[8] M. R. Anderson, D. Antenucci, and M. J. Cafarella. Runtime support for human-in-the-loop feature engineering system. *IEEE Data Eng. Bull.*, 39(4):62–84, 2016.

[9] D. Antenucci, M. R. Anderson, and M. Cafarella. A declarative query processing system for nowcasting. *PVLDB*, 10(3):145–156, 2016.

[10] D. Antenucci, M. R. Anderson, P. Zhao, and M. Cafarella. A query system for social media signals. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1326–1329. IEEE, 2016.

[11] D. Antenucci, M. Cafarella, M. C. Levenstein, C. Ré, and M. D. Shapiro. Using social media to measure labor market flows. Working Paper 20010, NBER, 2014.

[12] F. Bonchi, F. Giannotti, C. Lucchese, S. Orlando, R. Perego, and R. Trasarti. A constraint-based querying system for exploratory pattern discovery. *Information Systems*, 34(1):3–27, 2009.

[13] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[14] A. Chapman and H. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 523–534. ACM, 2009.

[15] Z. Chen, M. Cafarella, J. Chen, D. Prevo, and J. Zhuang. Senbazuru: a prototype spreadsheet database management system. *PVLDB*, 6(12):1202–1205, 2013.

[16] G. Dunton, D. Berrigan, R. Ballard-Barbash, B. Graubard, and A. Atienza. Joint associations of physical activity and sedentary behaviors with body mass index: results from a time use survey of us adults. *International journal of obesity*, 33(12):1427, 2009.

[17] T. W. Enterprise. Trifacta wrangler (2015), 2016.

[18] D. Fanelli, R. Costas, and J. P. Ioannidis. Meta-assessment of bias in science. *Proceedings of the National Academy of Sciences*, page 201618569, 2017.

[19] K. M. Flegal, D. Kruszon-Moran, M. D. Carroll, C. D. Fryar, and C. L. Ogden. Trends in obesity among adults in the united states, 2005 to 2014. *Jama*, 315(21):2284–2291, 2016.

[20] R. Fourer, D. M. Gay, and B. Kernighan. *Ampl*, volume 117. Boyd & Fraser Danvers, MA, 1993.

[21] M. Gelman, S. Kariv, M. D. Shapiro, D. Silverman, and S. Tadelis. Harnessing naturally occurring data to measure the response of spending to income. *Science*, 345(6193):212–215, 2014.

[22] P. E. Gill, W. Murray, and M. A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM review*, 47(1):99–131, 2005.

[23] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[24] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[25] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 683–698. ACM, 2017.

[26] A. W. Johnson, A. J. Stimpson, and T. K. Clark. Turning the tide: big plays and psychological momentum in the nfl. In *Sloan Sports Analytics Conference Papers and Proceedings*, 2012.

[27] C. Julia, P. Ducrot, S. Péneau, V. Deschamps, C. Méjean, L. Fézeu, M. Touvier, S. Hercberg, and E. Kesse-Guyot. Discriminating nutritional quality of foods using the 5-color nutrition label in the french food market: consistency with nutritional recommendations. *Nutrition journal*, 14(1):100, 2015.

[28] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[29] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *PLVDB*, 4(1):22–33, 2010.

[30] J. M. Kolodinsky and A. B. Goldstein. Time use and food pattern influences on obesity. *Obesity*, 19(12):2327–2335, 2011.

[31] T. Krieger and D. Meierrieks. Terrorism in the worlds of welfare capitalism. *Journal of Conflict Resolution*, 54(6):902–939, 2010.

[32] A. Makhorin. GLPK (GNU linear programming kit). *http://www.gnu.org/software/glpk/*, 2008.

[33] D. Meierrieks and T. Gries. Causality between terrorism and economic growth. *Journal of Peace Research*, 50(1):91–104, 2013.

[34] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12):1490–1493, 2011.

[35] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2012.

[36] K. Menyah, S. Nazlioglu, and Y. Wolde-Rufael. Financial development, trade openness and economic growth in african countries: New insights from a panel causality approach. *Economic Modelling*, 37:386–394, 2014.

[37] K. Mullane, M. Williams, et al. Bias in research: the rule rather than the exception? *Elsevier*, 2013.

[38] B. A. Murtagh and M. A. Saunders. MINOS 5.4 Users Guide, Report SOL 83-20R, Systems Optimization Laboratory, 1983.

[39] G. L. Nemhauser and L. A. Wolsey. Integer programming and combinatorial optimization. *Wiley, Chichester. GL Nemhauser, MWP Savelsbergh, GS Sigismondi (1992). Constraint Classification for Mixed Integer Programming Formulations. COAL Bulletin*, 20:8–12, 1988.

[40] C. L. Ogden, M. D. Carroll, H. G. Lawman, C. D. Fryar, D. Kruszon-Moran, B. K. Kit, and K. M. Flegal. Trends in obesity prevalence among children and adolescents in the united states, 1988-1994 through 2013-2014. *Jama*, 315(21):2292–2299, 2016.

[41] S. Saydah, K. M. Bullard, Y. Cheng, M. K. Ali, E. W. Gregg, L. Geiss, and G. Imperatore. Trends in cardiovascular disease risk factors by obesity level in adults in the united states, nhanes 1999-2010. *Obesity*, 22(8):1888–1895, 2014.

[42] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 15–26. ACM, 2010.

[43] B. Wu and C. A. Knoblock. An iterative approach to synthesize data transformation programs. In *IJCAI*, pages 1726–1732, 2015.

[44] B. Wu, P. Szekely, and C. A. Knoblock. Learning data transformation rules through examples: Preliminary results. In *Proceedings of the Ninth International Workshop on Information Integration on the Web*, page 8. ACM, 2012.

[45] B. Wu, P. Szekely, and C. A. Knoblock. Minimizing user

effort in transforming data by example. In *Proceedings of the 19th international conference on Intelligent User Interfaces*, pages 317–322. ACM, 2014.

[46] L. Yang and B. McCall. World education finance policies and higher education access: A statistical analysis of world development indicators for 86 countries. *International Journal of Educational Development*, 35:25–36, 2014.

[47] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword query reformulation on structured data. In *ICDE*, 2012.

[48] A. Yates, M. Cafarella, M. Banko, O. Etzioni, M. Broadhead, and S. Soderland. Textrunner: open information extraction on the web. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 25–26. Association for Computational Linguistics, 2007.

[49] S. M. Ziaei. Effects of financial development indicators on energy consumption and co2 emission of european, east asian and oceania countries. *Renewable and Sustainable Energy Reviews*, 42:752–759, 2015.