Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling

Charles Sutton
University of Edinburgh and The Alan Turing Institute
Edinburgh, UK
csutton@inf.ed.ac.uk

Timothy Hobson
The Alan Turing Institute
London, UK
thobson@turing.ac.uk

James Geddes
The Alan Turing Institute
London, UK
jgeddes@turing.ac.uk

Rich Caruana
Microsoft Research
Redmond, WA
rcaruana@microsoft.com

ABSTRACT

Many analyses in data science are not one-off projects, but are repeated over multiple data samples, such as once per month, once per quarter, and so on. For example, if a data scientist performs an analysis in 2017 that saves a significant amount of money, then she will likely to be asked to perform the same analysis on data from 2018. But more data analyses means more effort spent in data wrangling. We introduce the data diff problem, which attempts to turn this problem into an opportunity. Comparing the repeated data samples against each other, inconsistencies may be indicative of underlying issues in data quality. By analogy to text `diff`, the data diff problem is to find a "patch", that is, transformation in a specified domain-specific language, that transforms the data samples so that they are identically distributed. We present a prototype tool for data diff that formalizes the problem as a bipartite matching problem, calibrating its parameters using a bootstrap procedure. The tool is evaluated quantitatively and through a case study on an open government data set.

CCS CONCEPTS

• **Information systems → Data mining**;

KEYWORDS

automl; data mining; data wrangling; sequential analysis

ACM Reference Format:
Charles Sutton, Timothy Hobson, James Geddes, and Rich Caruana. 2018. Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling. In *KDD 2018: 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3220057

1 INTRODUCTION

Data wrangling is the process of transforming raw data into a clean sample that can be used for analysis, data mining, and machine learning. Wrangling includes steps such as structuring data into tables, standardizing formats, and addressing data quality issues such as typographical errors, changes in units of measure, coding of missing values, and so on. Individually, each of these transformations are quite simple, but the wrangling process as a whole becomes expensive and tedious, for two reasons. The first, which we might call *death by a thousand wranglings*, is usually there is not just one issue in the data to fix, but many, and each requires separate data wrangling effort. The second, an *Anna Karenina principle*, is that every dirty data set is dirty is its own way. In our experience, it is uncommon for data cleaning and preprocessing workflows to be transferable across data sets.[1] For these reasons, data wrangling is often cited as taking up a large proportion of the time required for a data science or machine learning project [5, 16].

Often data scientists will write a series of data preparation scripts [16, 18], but because of the Anna Karenina principle, the analyst is aware that these scripts are likely to be one-offs, and so the scripts are brittle and closely tied to details of the data source and format. It is therefore unfortunate that there is in fact a large class of data analyses that are *not* "one offs" but are repeated over time. For example, suppose a data scientist creates a report analysing hospital admissions. If the report is useful, then it may well be requested again, but computed on a data sample from the next year. More generally, many data sets arrive in batches, which we call *data samples*, at periodic intervals, e.g., every month, every quarter, every year and so on. But even when created by the same organization, with great care, repeated data samples are often not exactly alike, because of changes to what columns are available, how data is coded, and so on. Each difference between the data samples means more data wrangling effort.

The "data diff" task that we propose is an attempt to transform the difficulty of repeated data samples into an opportunity.[2] The opportunity is that if the data samples are expected to be identically distributed, or at least in the same format, we can perform a first pass

[1] A similar point is made in advice to UK government data scientists here: https://ukgovdatascience.github.io/rap_companion/exemplar.html (see Section 4.2)
[2] "data diff" was proposed by R. Caruana as an AutoML challenge task in a talk on "Open Research Problems in AutoML" at the 2015 ICML AutoML Workshop in Lille, France.

of data cleaning by comparing the latest data sample to the earlier ones. Differences discovered in this comparison may represent underlying data quality issues, or formatting differences that are likely to break data preparation scripts, which warrant further investigation.

More specifically, the data diff tool is analogous to the Unix `diff` tool for text files, which examines two text files and reports a small change, called a patch, that will transform one file to another. However, running text diff on a pair of data sets, e.g., as comma-separated value files, is not likely to produce useful output—in most cases, we know the rows in the new data set are different, but we want to test whether the data distribution and format are the same. Instead, in data diff, a patch between two data sets D_1 and D_2 is a simple transformation that will make D_1 appear as if it were sampled from the same population as D_2. Transformations can include permutation of columns, linear transformations of columns, and so on. We introduce a first method for the data diff as a structure learning problem, in which we search for a transformation that minimizes a measure difference between distributions, with a penalty term to favour simpler, more interpretable diffs.

We present a case study of applying data diff to an open government data set. In the case study we have multiple data samples that were generated by the same organization over different time spans. There are significant changes in the format across the samples that would be highly likely to break most data preparation scripts. The data diff tool is able to detect these differences and explain them to an analyst, thus eliminating the tedious debugging time that would be required to discover them manually.

More broadly, the potential applications of a data diff tool range far beyond data wrangling. Detecting changes in distributions is one of the most fundamental tasks in data analysis. As an example, consider the deployment of machine learning models. We might need to deploy the same classifier across multiple sites, for example, multiple hospitals for a clinical application, multiple factories for a manufacturing application, and so on. But the classifier has been trained on data of a particular format, and it is highly unlikely that every site will format the input data for the classifier correctly the first time. A data diff tool can provide guidance on fixing errors in input format. As another example, deployed machine learning models commonly experience changes over time in the distribution of test examples, perhaps due to concept drift [17], or due to feedback effects in a large production system [21]. When the distribution of run-time samples deviates too far from the original training distribution, a data diff tool can provide an interpretable summary of what has changed. Although many statistical tests have been developed to detect differences in distributions, the data diff problem is unique in that it aims at a "diff", a compact, interpretable summary of the difference. Data diff aims beyond detecting differences in distributions, to providing information to help analysts understand and respond to those differences.

2 MOTIVATION

This is a relatively new research area with little prior literature, so we first present examples to motivate the need for such a capability. In our experience, on real projects that span multiple years, we have never found that a complex data set can be generated a year or two later that has the exact same structure, coding and statistics as prior samples. Modern data sets are so complex, and are collected from so many different sources, that it is very difficult to create a new sample of data that is exactly like prior samples. For example, a few years ago one of the authors trained a model to predict 30-day readmission using hospital data collected from 2011-2013. There were about 100,000 patient records per year and 4,000 features per patient. Because there is seasonality in the data and some patients occur multiple times in the records, we used data from 2011 as training data, 2012 as validation data, and 2013 as test data. The model performed well and was released at the hospital in 2014.

In 2015, we decided to update the model using a new table of 100,000 records and 4,000 columns collected in 2014, For this new model, we used data from 2011+2012 for training, 2013 for validation, and the new 2014 sample for testing. Before training a new model, however, we ran a primitive data diff tool on the new sample of data from 2014 to compare it to the previous three years' data, which had all been prepared a year earlier. The new data looked good, except that the statistics for columns 562–568 looked different in the new table compared to the preceding three years. The hospital reviewed the SQL scripts used to prepare the new data and found a subtle bug that explained the differences in these columns. These differences would have silently reduced the accuracy of the new model, requiring time-consuming debugging, if they had not been detected by the primitive data diff tool.

To highlight some of the "mundane complexity" that arises in "living, breathing" data sets, here are quotes from email exchanges for projects in our recent experience. We label each quote with the type of wrangling issue that it describes:

- *Changes in units of measure:* "... Just want to verify format. Before we had a separate file with age in two columns. I think one was for age in years, and the other was age in days? Looks like you added these two columns for age at the end of the new files as new cols 3996 and 3997, right?"
- *Missing columns:* "... The matrix you uploaded has 3755 columns (not including the ID column), but the data we've been using has 3982 columns. Are we missing 227 columns in the new data? Or maybe I'm doing something wrong at my end."
- *Changes to distribution within a column:* "... The model wasn't working so I spent some time debugging. Looks like we suddenly have 323 columns with only one unique value in each of them. Most are -1's, but a few are all 0's. Any idea how long ago that happened?"
- *New data violates integrity conditions:* "... I sorted that ages from highest to lowest. In the new data we 2000 people over 100, and 1400 over 120. That doesn't sound right. The ages are all integers. Maybe months are getting mixed with years?"
- *Anomalies in new data*: "... Looks like there's a problem with column 41: it has ROC = 0.56, but all other columns have ROC above 0.75. Looks like there are many cases with predicted values of 0.0000 in column 41. Any idea what went wrong with that column?"
- *Changes in column order:* "... Got the new matrix. I think I should reorder the columns to put them back into the order we originally used. Where should the last two columns go? If we use col 0 to

refer to the targets, so now col 1 is the 1st column of predictions and cols 199 and 200 are the two new columns, then I think the new cols should move between the predictions in column 158 and 159, creating new columns 159 and 160. Have I got that right? And the new columns are in the correct order for scaling?"

Although our current data diff tool would not have resolved all of these problems, it would have detected them, often providing specific enough information to make identifying the true problem much easier. The most important stage in data debugging is to know you have a problem in the first place.

3 PROBLEM STATEMENT

Data diff is the general problem of providing a concise, interpretable summary of the differences between two data sets. We call this summary a *patch*, by analogy to the Unix `diff` utility. By reading the patch, an analyst can quickly understand the differences in format and distribution between the data sets. The analyst can then investigate whether the differences in the patch are expected, whether they reflect interesting changes in the population, or whether they reflect data quality issues, and act accordingly.

For text files, the Unix `patch` command applies a patch to transform a text file. In data diff, patches can have two potential purposes: to *inform* or to *transform*. In some cases—just as in Unix—the data diff patches are executable, and can transform the new data to be compatible with the older data. For example, suppose the new data set introduces additional columns; the patch would simply remove these. In other cases, such as if a column has radically changed distribution, there is no sensible way to transform the new data to be backwards compatible. Then the patch will still indicate that the column has changed distribution, but will not be directly executable. Instead the patch serves the purpose of informing the analyst of the change across the data sets.

To formalize the data diff problem, it is easiest to take the transformation viewpoint, even though the patches will in the end be used for both purposes. Specifically, given two data tables D_1 and D_2, the formal data diff problem is to identify a patch Π, which is simply a function on data sets, so that D_1 and $\Pi(D_2)$ are encoded in the same format and drawn from the same distribution. Note that we do not assume that the data in D_1 and D_2 are paired in any way, or that they have the same number of rows or columns. For example, D_1 and D_2 might be two data sets of different size that are drawn from the same distribution, except that D_2 contains a new column that does not appear in D_1, and that column 73 in data set D_2 is measured in Celsius instead of Fahrenheit as in D_1. Then executing $\Pi(D_2)$ will remove the additional column and linearly transform column 73 appropriately.

Clearly, the set of allowable patches must be restricted, or Π may simply remove all rows from D_1 and insert all rows from D_2. Instead, the set of allowable transformations is restricted to those that can be described by an expression defined in a *transformation language*. The transformation language is designed both to describe typical ways that sequences of data samples tend to diverge over time, and also to be easily read by an analyst. Ideally, the transformations should be specific, so that patches do more than simply detect a change, but also explain what type of change occurred. It is easy to imagine a broad variety of transformations which would be important to detect, such as changes in numbers of columns and column order; changes in number, date, and time format; changes in currencies and units of measure; changes in the marginal distribution of columns; changes in statistical dependences between columns; and so on. The idea of using a transformation language is inspired by previous work in data wrangling [9, 16], but for our purposes, the language must be designed specifically for the data diff problem.

Because detecting changes in distributions is a fundamental problem, a wide variety of methods have been proposed in the statistics literature, depending on whether the expected change is abrupt or gradual, and whether we know when the change might have taken place. These methods include two-sample tests, change-point methods, and sequential analysis [3, 4, 8, 11, 22, 24]. Data diff is different from all of these, because it is about how to proceed further when the two data sets *were not* drawn from the same distribution, but this discrepancy is due to issues that can be described by a compact, interpretable transformation, such as differences in data formats, units of measure, coding of missing values, and so on. Essentially, the difference is that data diff aims to produce a diff.

4 THE DATA DIFF SYSTEM

In this section, we describe a prototype of a system for tackling the data diff problem.[3] Although the framework could be applied to a broad variety of settings, including unstructured and semistructured data, complex string transformations, and so on, we present an initial data diff implementation that focuses on a simple set of column transformations that we have observed repeatedly in our practical experience (Section 4.1). We measure the quality of a proposed patch by a nonparametric measure of distance between distributions, which we formalize as a linear assignment problem (Section 4.2). The objective function contains a penalty term to encourage the method to return simpler patches. The weights for this penalty are selected using a bootstrap procedure to match a false positive rate given by the analyst (Section 4.3).

4.1 Transformation Language

The transformation language $\mathcal{L}$ defines the set of patches that we consider in our system. There is a large potential design space here. In our prototype, we use a fairly simple transformation language, chosen specifically by reflecting on our own experiences, which we described in Section 2.

To set notation, a patch $\Pi(D)$ is a function that maps data tables to data tables. We assume that the input data D is a table with columns $A = \{A_1, \ldots A_M\}$. Each row in D is a tuple $t_i = (A_1[t_i], \ldots, A_M[t_i])$ for $i \in \{1 \ldots N\}$. We use the notation $D[i]$ to refer to a single column, or occasionally, for a index set $I \subseteq \{1 \ldots N\}$, we use $D[I]$ to denote the appropriate projection of D. Each attribute A_i has a domain $\mathcal{X}_i$, which for this paper we take to be either real-valued or a finite set, so each tuple t_i is a member of the set $\mathcal{X} = \mathcal{X}_1 \times \ldots \times \mathcal{X}_M$. A patch Π is a function that transforms one table to another, but importantly, a patch can change the schema of D, e.g., by transposing columns, or converting a column from one type to another. So if we denote a table $D = (A, t_{1:N})$, then a patch defines a transformation $\Pi : (A, t_1 \ldots t_N) \mapsto (A', \pi(t_1) \ldots \pi(t_N))$, where

[3]The prototype system will be released publicly after completion of the review process.

$\pi : \mathcal{X} \to \mathcal{X}$ acts on individual tuples. In other words, patches can add or remove attributes and can perform arbitrary transformations on one tuple at a time. Most patches have parameters $a_0 \dots a_J$ beyond the data table D, for example, which column to delete. To handle this in our notation, we will define a class of patches via a generator function `fn`$(a_0 \dots a_J)$, which is a higher-order function that returns a patch Π that can be applied to a specific data table.

We begin by defining classes of *elementary patches*:

- *Permute column* patches have the form `permute`(ℓ), where ℓ is a permutation of $\{1 \dots M\}$. This patch has the effect of permuting the columns of the data set.
- *Insert column* patches have the form `insert`(i, C), where i is an integer in $\{0 \dots M\}$. This patch inserts the column C into D just after the column with index i, where we take `insert`$(0, C)$ to mean that the new column should be inserted as the first column.
- *Delete column* patches have the form `delete`(i), and removes the column A_i.
- *Replace column* patches have the form `replace`(i, C), where i is an integer in $\{1 \dots M\}$, and replace the current cells in column $D[i]$ with the values in C. This patch is useful where we can detect that the distribution of a column has changed, e.g. with a standard two-sample test, but the library contains no transformation that will repair this discrepancy.
- *Recode column* patches modify the values of a categorical column. The patch `recode`$(i,\ \{(v_1, \hat{v}_1), \dots, (v_n, \hat{v}_n)\})$ can be applied whenever column i is categorical, i.e. $n := |\mathcal{X}_i|$ is finite. The meta-parameter is a set of pairs $(v, \hat{v}) \in \mathcal{X}_i \times \hat{\mathcal{X}}_i$ in which every v in $\mathcal{X}_i$ is represented and the $\hat{v}$ are unique (so $|\mathcal{X}_i| \leq |\hat{\mathcal{X}}_i|$). The effect is to replace value v, whenever it appears in column i, with value $\hat{v}$. One particularly important use of this is for missing data, which is often coded by special values, such as "NA", -1, 0, 999, which are easy to inadvertently change across data samples. This has been a common source of discrepancies in our experience, and others report this as well [18].
- *Linear transform* patches have the form `linear`$(i,\ a,\ b)$ and can be applied when A_i is a real-valued attribute. The effect is to multiply every cell in the column by a and add b. This transformation is especially useful for detecting inadvertent changes in units of measure.
- The *identity* patch `identity()` makes no change to the data set.

Finally, we introduce composite patches `compose`$(\Pi_1 \dots \Pi_K)$, whose effect is to return the function composition $\Pi = \Pi_1 \circ \cdots \circ \Pi_K$ of the given sequence of patches $\Pi_1 \dots \Pi_K$. A *univariate patch* is one that affects only a single column. So all of the elementary patch types are univariate, except for `permute`, and any composition of univariate patches over the same column is also univariate.

As we have described them, all patches in the language are fully executable. For example the `insert()` patch specifies the precise data to be inserted, and so on. This simplifies the description of the algorithm. But as we have noted, when columns are missing or radically change distribution in D_2, it is probably not sensible for a data diff tool to try to guess the values of an entire data column for D_2 so that it would have matched D_1. So in practice, when our prototype detects that a patch of type `insert()` or `replace()` are necessary, the patch displayed to the user specifes what column is affected, but does not include the column C that is included in our formalism.

4.2 Optimizing over Patches

The main algorithmic challenge of data diff is the search through the space of possible patches. In this section we describe the objective function that `datadiff` uses to rank possible patches, and the optimization algorithm that it uses to minimize this objective.

4.2.1 Objective Function. We wish to return a patch Π so that $\Pi(D_2)$ and D_1 appear to have been sampled independently from the same distribution. This notion can be formalized by the optimization problem

$$\min_{\Pi \in \mathcal{L}} \mathcal{D}(D_1, \Pi(D_2)) + \Omega(\Pi), \tag{1}$$

where $\mathcal{D}()$ is a distance measure between data sets and $\Omega()$ is a penalty function to discourage patches that are too complex. For the distance measure $\mathcal{D}()$, we sum, over all columns, a column-wise distance measure that depends on the attribute type. For real-valued columns C_1 and C_2, we use the Kolmogorov-Smirnov (KS) statistic

$$\mathrm{KS}(C_1, C_2) = \max_x |F_1(x) - F_2(x)|\,, \tag{2}$$

where F_1 and F_2 are the empirical cumulative distribution functions of C_1 and C_2, respectively. We choose the KS distance because of its simplicity, but one could consider a large number of possible measures, such as Euclidean distances between vectors of moments or maximum mean discrepancy (MMD) [8].

For categorical columns we use the total variation (TV) statistic:

$$\mathrm{TV}(C_1, C_2) = \sum_x |D_1(x) - D_2(x)|\,, \tag{3}$$

where now D_1 and D_2 are the empirical discrete distribution functions (*not* the cumulative distribution functions). When the two columns C_1 and C_2 are of different types we arbitrarily assign a distance of 1.

Putting these definitions together, we can define a column-wise distance measure:

$$\mathcal{D}_c\,(C_1, C_2) = \begin{cases} \mathrm{KS}(C_1, C_2) & \text{if both } C_1 \text{ and } C_2 \text{ continuous,} \\ \mathrm{TV}(C_1, C_2) & \text{if both } C_1 \text{ and } C_2 \text{ discrete,} \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

Then our distance measure $\mathcal{D}()$ over data sets becomes

$$\mathcal{D}(D_1, D_2) = \sum_{i=1}^{M_1} \mathcal{D}_c\,(D_1[i], D_2[i])\,. \tag{5}$$

If D_1 and D_2 do not have the same number of columns, then we take $\mathcal{D}(D_1, D_2)$ to be ∞.

The penalty $\Omega()$ is necessary because patches can overfit. For example, suppose D_1 and D_2 are both generated from standard multivariate Gaussians. Without the penalty, the optimal diff is likely to multiply every column of D_2 by a small constant so that the column means of $\Pi(D_2)$ are exactly those of D_1. The penalty discourages this behaviour. We find that a simple choice for $\Omega(\Pi)$ is effective, basically the number of elementary patches in Π, weighted by patch type. More formally, for each type t of elementary patch in the language, we introduce as a parameter a nonnegative weight $\lambda[t]$. We define `type`(Π) to be the type of an elementary patch, and

Algorithm 1 Main datadiff algorithm

```
1: procedure DataDiff($D_1, D_2, \lambda$)
2:     ▷ Optimize over univariate patches
3:     for $i \in 1 \ldots M_1$ do
4:         for $j \in 1 \ldots M_2$ do
5:             $\Pi_{ij} \leftarrow$ BestUnivariatePatch($D_1[i], D_2[j]$)
6:             $C_{ij} \leftarrow \mathcal{D}(D_1, \Pi_{ij}(D_2)) + \Omega(\Pi_{ij})$
7:     ▷ Optimize over permute patches
8:     $\Sigma \leftarrow$ MinBipartiteMatching($C$)
9:     ▷ Construct the patch to be returned
10:    $\ell \leftarrow ()$
11:    Append permute($\Sigma$) to $\ell$ unless $\Sigma$ is identity
12:    for $(i, j) \in \Sigma$ do
13:        Append $\Pi_{ij}$ to $\ell$ unless type($\Pi_{ij}$) = identity
14:    for $i \in 1 \ldots M_1$ do
15:        if $(i, \emptyset) \in \Sigma$ then
16:            Append insert($i$, $D_1[i]$) to $\ell$
17:    for $j \in 1 \ldots M_2$ do
18:        if $(\emptyset, j) \in \Sigma$ then
19:            Append delete($j$) to $\ell$
20:    return compose($\ell$)
```

$\mathcal{P}(\Pi)$ to be the set of elementary patches contained in a given patch Π. Then we can define

$$\Omega(\Pi) = \sum_{\Pi_i \in \mathcal{P}(\Pi)} \lambda[\texttt{type}(\Pi_i)] \tag{6}$$

4.2.2 Optimization Algorithm. The problem (1) is a difficult combinatorial optimization problem, and strategies for solving it may well depend on specific features of the transformation language $\mathcal{L}$. In datadiff we notice that, except for permute, all of the elementary patch types are univariate patches. Therefore our optimization algorithm will first compute the best univariate patch over pairs of columns from D_1 and D_2, and then optimize over possible permutations of the columns.

This is described in Algorithm 1. First, for all pairs of columns i from D_1 and j from D_2 (lines 3–6), we compute the optimal patch assuming that a permutation patch has already aligned columns $D_1[i]$ and $D_2[j]$. The function BestUnivariatePatch optimizes the univariate data diff problem, i.e., it optimizes (1) for the pair of data sets $(D_1[i], D_2[j])$. We describe the details of this function in a moment. Now that we have constructed the cost matrix C, finding the optimal permute patch according to (1) corresponds to a minimum weight assignment problem, i.e., a complete bipartite matching, which we solve in line 8. We can represent the returned matching as the set of pairs of columns that are aligned. Our implementation uses the Hungarian algorithm, and we use an implementation which produces matchings that are complete in the sense that they contains as many edges as possible. However, if C is not square, i.e., if D_1 and D_2 have different numbers of columns, then some columns will be unmatched in one of the data sets. The returned matching uses a special token $\emptyset$ to indicate unmatched columns, i.e. $(i, \emptyset)$ indicates that column i in D_1 was unmatched, etc.

Finally, in lines 10–20, we construct the patch to be returned. First we construct the permute patch corresponding to Σ, and then for each pair of columns that are aligned between the two data sets, we include the univariate patch that best matches those two columns. This has been previously computed from the calls to BestUnivariatePatch. Columns that were unmatched in Σ have insert or delete patches added as appropriate.

Now we describe the optimization over univariate patches. The function BestUnivariatePatch takes two columns C_1 and C_2 as input, and returns a patch Π_{12} that minimizes (1) for those two columns in isolation. The only three patch types that this function needs to handle are recode, linear and identity, as other patch types are handled by the main datadiff algorithm. We can assume that the optimal patch within this class is elementary, because recode and linear are mutually exclusive, and there is no need to compose a patch with the identity patch. So the BestUnivariatePatch optimizes (1) in turn for the best recode and linear patches, then computes the objective for the identity patch, and finally returns the minimum of the three. There is one additional case to be handled, that in which columns C_1 and C_2 have different types, one being real-valued and the other categorical. The return value in that case is a replace patch, representing an irreconcilable difference, with an associated penalty chosen large enough to reflect that this outcome is a last resort.

To finish the description of BestUnivariatePatch, we need to describe how to optimize over the parameters of recode and linear patches. First, for linear patches, the optimization problem is only two-dimensional, so we simply minimize (1) with respect to the parameters of the linear function using a bisection method. For recode patches, we observe that the mapping between old values and new values can be expressed as a bipartite matching, so we again use the Hungarian algorithm to find the best parameters.

4.2.3 Correctness. We can formalize how this algorithm optimizes (1). First, we define a canonical form over patches $\Pi(D)$, where D has M columns. A patch in *canonical form* is a composite patch $\Pi = (\Pi_0, \Pi_1, \ldots \Pi_M)$, where Π_0 is a patch of type permute, and each of the patches Π_m, for $m \in 1 \ldots M$, is a univariate elementary patch whose first argument is column index m. Two patches Π_1 and Π_2 are semantically equivalent if they define the same function over data sets, that is, $\Pi_1(D) = \Pi_2(D)$ for all D. Then we have (proof in online supplementary material):

Proposition 4.1. *For every patch Π, there exists a semantically equivalent patch $\tilde{\Pi}$ in canonical form.*

The patch language $\mathcal{L}$ is ambiguous because of the presence of insert, delete, and replace. To resolve this ambiguity, we make the assumption (A1) that patches returned from DataDiff do not include insert, delete, and replace unless they are necessary to produce a patch with finite cost. That is, if $M_1 < M_2$, then DataDiff (D_1, D_2) contains exactly $M_2 - M_1$ insert patches, and so on. This corresponds to setting a very large penalty $\lambda[\texttt{insert}]$, etc., so as to overwhelm the effect of $\mathcal{D}_c$ (). Let $\mathcal{P}(M_1, M_2)$ be the set of patches that satisfy this assumption. Then we can show that

Proposition 4.2. *Algorithm 1 returns the canonical patch Π^* that minimizes $\mathcal{D}()$ under assumption (A1), that is,*

$$\Pi^* = \underset{\Pi \in \mathcal{P}(M_1, M_2)}{\operatorname{argmin}} \ \mathcal{D}(D_1, \Pi(D_2)) + \Omega(\Pi). \tag{7}$$

Proof. Let d^* denote the optimizing value of (7), and $\mathcal{U}(i)$ denote the set of all elementary univariate patches Π_i for which column $D_2[i]$ is a valid argument.

First suppose that $M_1 = M_2 = M$. Then optimizing over patches in canonical form yields

$$d^* = \min_{\sigma} \sum_{m=1}^{M} \min_{\Pi_m \in \mathcal{U}(m)} \mathcal{D}_c \left(D_1[\sigma^{-1}(m)], \Pi_m(D_2[m]) \right) + \Omega(\mathtt{type}(\Pi_m)), \tag{8}$$

where σ ranges over all permutations over $\{1 \dots M\}$. By definition, the function BestUnivariatePatch computes the inner minimization, i.e., after line 6, for all $i, j \in \{1 \dots M\}$, we have

$$C_{ij} = \min_{\tilde{\Pi} \in \mathcal{U}(j)} \mathcal{D}_c \left(D_1[i], \tilde{\Pi}(D_2[j]) \right) + \Omega(\mathtt{type}(\tilde{\Pi})). \tag{9}$$

Then line 8 computes the minimization $\Sigma = \operatorname{argmin}_\sigma C_{i,\sigma(i)}$. This completes the proof for $M_1 = M_2$. If $M_1 \neq M_2$, then first suppose $M_1 < M_2$. Optimizing (1) over canonical patches that satisfy (A1) is equivalent to optimizing

$$\min_{I \subseteq \{1 \dots M_2\}, |I| = M_1} d^*(D_1, D_2[I]), \tag{10}$$

where d^* is defined as in (8). This step follows because all patches in $\mathcal{P}(M_1, M_2)$ contain exactly $M_2 - M_1$ `delete` patches, so the penalty for those patches can be ignored in the optimization. Now line 8 again optimizes (10) over I, as after this line, we have $I = \{j \mid (\emptyset, j) \in \Sigma\}$. Finally, the proof for $M_2 < M_1$ is similar. □

4.3 Calibrating Patch Complexity

The complexity of the returned patches is controlled by the penalty weights on each patch type. The best choice of weights may well depend on the number of rows, whether D_1 contains many columns with similar distributions, and so on. In this section we propose a simple and generic procedure for choosing the penalty weights.

In an analogy to hypothesis testing, we allow the analyst to choose a false positive rate (FPR) α, and we aim to set the penalty weights such that if D_1 and D_2 were drawn from the same distribution, we return a non-identity patch with probability at most α. An obvious, but not entirely helpful, attempt to meet this goal is to perform a standard two sample test at the beginning of the `datadiff` algorithm, and return an `identity` patch if the test fails to reject the null hypothesis of equality of distributions. This proposal is not fully helpful because, although it would prevent us from incorrectly returning a diff when D_1 and D_2 are entirely similar, it does not prevent the system from returning a complicated diff when a simpler one would be almost as good. It is for this reason that we use the regularised formulation of (1) instead.

Instead, we select the penalty weights using a slightly different, but related criterion. For each elementary patch type t (except for `identity`), we aim to set the penalties so that under the null hypothesis that D_1 and D_2 are identically distributed, the composite patch returned by `datadiff` contains an elementary patch of type t with probability at most α. This measure for false positive rate, which we might call "partial match FPR for patch type t" (partial FPR), provides a signal for controlling the penalty weight of each patch individually. Choosing the penalty weights for a given false positive rate α is difficult because there are many interactions within the data diff procedure, for example, increasing the penalty for including a `permute` patch will affect all of the other patches that are selected. Therefore, we set the penalty weights using a bootstrap-type procedure [6].

Specifically, we draw R pairs of data sets of size (M_1, M_2) by sampling independently from the data set D_1 with replacement. We run DataDiff(D_{r1}, D_{r2}) individually for each sampled pair of data sets, and compute the partial FPR for each patch type over the R samples. We tune the penalty weights for each patch type using a simple greedy search until the empirical partial FPRs for all patch types are as close as possible to α.

5 EVALUATION

Evaluating a data diff system is complicated by the fact that one of its main goals is to inform human analysts. Our final goal is to reduce the amount of analysts' time required to carry out the end-to-end analysis task, as well as the quality of the final analysis, but this can be difficult to measure, possibly requiring an extensive study with human analysts. Instead, as a more preliminary evaluation, performance of the `datadiff` system was evaluated through a series of experiments involving synthetically corrupted data, for which we can measure whether `datadiff` recovers the true corruption. Nine of the most popular data sets from the UCI machine learning repository [19] were used for this purpose. Each experiment involved generating a *corruption patch* (either elementary or composite), of a given type, by selecting its parameters at random. For instance, in the case of `linear` patches, the shift parameter b was selected by sampling from a $U[-2, 2]$ distribution and multiplying by the standard deviation of the (randomly-selected, numerical) attribute. The corruption was applied to one half of a random partition of the data, before running `datadiff` on the two halves. The experiments were repeated $R = 100$ times, with ten different randomly-chosen corruptions of each corruption type, and ten different random splits of the data for each sampled corruption.

We define a collection of metrics to quantify the fidelity of the `datadiff` output. The first two, precision p_t and recall r_t, are parameterised by patch type t, and measure the algorithm's performance at selecting transformations of the correct type. For each repetition i, let Π_i^* be the patch that represents the true corruption of the data, and Π_i be the corresponding output of `datadiff`. As before, we define $\mathtt{type}(\mathcal{P}(\Pi))$ for the set of elementary types in patch Π. Then precision and recall are defined as

$$p_t = \frac{\sum_{i=1}^{R} \delta\{t \in \mathtt{type}(\mathcal{P}(\Pi_i^*)) \text{ and } t \in \mathtt{type}(\mathcal{P}(\Pi_i))\}}{\sum_{i=1}^{R} \delta\{t \in \mathtt{type}(\mathcal{P}(\Pi_i))\}} \tag{11}$$

$$r_t = \frac{\sum_{i=1}^{R} \delta\{t \in \mathtt{type}(\mathcal{P}(\Pi_i^*)) \text{ and } t \in \mathtt{type}(\mathcal{P}(\Pi_i))\}}{\sum_{i=1}^{R} \delta\{t \in \mathtt{type}(\mathcal{P}(\Pi_i^*))\}}, \tag{12}$$

where the function $\delta\{\cdot\}$ is 1 if the given condition is true, and 0 otherwise. Precision measures how often `datadiff` is correct when it reports that a particular patch type is present, and recall measures how often a patch of type t is detected when it is in fact present.

The remaining three metrics measure the accuracy of `datadiff` in identifying the parameters of the corruption, in cases where the type is correctly identified. For univariate patch types, we define the column accuracy to be the frequency with which, when an elementary patch of the correct type appears in the result, the column index is also correct. For patch types with real-valued parameters (currently only `linear`), we report the root-mean-square error (RMSE) over the parameter values. For `recode` patches, we report parameter accuracy, which counts the frequency with which the set of recoding pairs in the result is identical to that in the corruption. This is a strict, exact match measure of the percentage of cases in which the entire matching between categorical values is correct, and does not give any credit to partially correct matches. Finally, if the corruption and result contain elementary patches $\texttt{permute}(\ell^*)$ and $\texttt{permute}(\ell)$, then we measure parameter accuracy as the Hamming distance between ℓ^* and ℓ expressed as a sequence of integers.

For all of the experiments in this section, the penalty weights were chosen by the calibration procedure described in Section 4.3. The calibration procedure was run once to achieve a target partial FPR $\alpha = 0.05$ on average across the nine data sets. (This is designed to reflect a situation in which the analyst calibrates `datadiff` on D_1 just before running it.) The resulting penalty weights were then used for `datadiff` for all R repetitions of the experiments.

The synthetic experiments were divided into four groups according to the complexity of the corruption. Each of the corruptions in the first group was an elementary, univariate patch (Table 1). In the second group a column transposition was added (Table 2). The third group consisted of corruptions composed of two univariate patches (Table 3), and the fourth added to this a column transposition (Table 4). In general, we see that the precision, recall, and accuracy at identifying columns are over 0.85 and often over 0.90, even for the more complex cases in 4 where there are multiple univariate corruptions and a permutation. The only exception is that the `linear` patches have lower precision, e.g., as low as 0.70 in 2. This is to be expected because our generating procedure can sometimes sample true corruptions with very small magnitude, e.g. multiplying a column by 1.01.

False positive rate. In the above experiments, there was always some true corruption to find in the pair of data sets. Now we evaluate the *total false positive rate (tFPR)*, which is how often `datadiff` returns a non-identity patch when the two data sets are in fact identically distributed. Note that this is not exactly the same as the FPR measure used in the calibration procedure, as the calibration FPR is a partial match procedure for each patch type. We evaluate the tFPR of `datadiff` for the calibrated parameter values that were used in the previous set of experiments. For each of the nine UCI data sets, we sample $R = 100$ bootstrap samples of identically distributed pairs, and measure the false positive patches returned by `datadiff`. The median tFPR over the nine UCI data sets was 0.03, which is close to the target value.

6 CASE STUDY

We present a case study to illustrate how the output of `datadiff` can be useful in real-world data analysis. The UK's Office of Communications (Ofcom) publishes data annually on the performance of residential fixed-line broadband services[4]. These data exhibit many of the transformation types handled by `datadiff`. For illustration, Figure 1 shows the first ten attributes from years 2013 to 2015. We see that new columns were introduced in each year, the original column order is not preserved, column names change over time, as do the encodings in certain categorical columns. In total, there are 29 attributes and 1450 rows in the 2013 data, 31 attributes and 1971 rows in 2014, and 36 attributes and 2802 rows in 2015.

Prior to running `datadiff` on the broadband data, appropriate patches to reconcile differences in adjacent years were identified manually. We call these *target patches*. We focus on adjacent years to make the patches easier to read. We choose the penalty weights $\lambda[t]$, for each patch type t, to generate patches of small complexity when executed on the broadband data. A degree of preprocessing was done to manually strip stray characters from numeric data.

[4]https://data.gov.uk/dataset/uk-fixed-line-broadband-performance

Patch Type	Type Precision	Type Recall	Column accuracy	Parameter RMSE	Parameter accuracy
Linear	0.787	0.885	0.888	1.550	—
Recode	0.862	0.977	0.935	—	0.314
Insert	1.000	0.989	0.964	—	—
Delete	1.000	0.980	0.937	—	—

Table 1: Performance of `datadiff` at recognizing known corruptions, on single univariate corruptions.

Patch Type	Type Precision	Type Recall	Column accuracy	Parameter RMSE	Parameter accuracy
Linear	0.703	0.757	0.856	2.143	—
Recode	0.830	0.973	0.935	—	0.306
Insert	1.000	1.000	0.978	—	—
Delete	1.000	1.000	0.954	—	—
Permute	1.000	0.931	—	—	0.395

Table 2: Performance of `datadiff`, when the true corruption contains both a column transposition and a univariate corruption.

Patch Type	Type Precision	Type Recall	Column accuracy	Parameter RMSE	Parameter accuracy
Linear	0.941	0.938	0.725	1.559	—
Recode	0.984	0.975	0.924	—	0.289
Insert	1.000	1.000	0.709	—	—
Delete	1.000	1.000	0.795	—	—

Table 3: Performance of `datadiff`, when the true corruption contains two univariate corruptions.

Patch Type	Type Precision	Type Recall	Column accuracy	Parameter RMSE	Parameter accuracy
Linear	0.940	0.801	0.724	2.373	—
Recode	0.972	0.976	0.907	—	0.315
Insert	1.000	1.000	0.789	—	—
Delete	1.000	1.000	0.877	—	—
Permute	1.000	0.944	—	—	0.717

Table 4: Performance of `datadiff`, when the true corruption contains two univariate corruptions and a column transposition.

ID	Distance band	Urban/ rural	Market	ISP	Headline speed	Technology	Download speed (Mbit/s) 24-hour	Download speed (Mbit/s) Max	Download speed (Mbit/s) 8-10pm weekday
1302	2563-5000	Urban	3	Virgin	60	Cable	58.597	62.673	56.677
1318	1367-1635	Urban	3	TalkTalk	20	ADSL2+	9.325	11.362	9.162
1319	1992-2563	Urban	3	Sky	20	ADSL2+	5.369	5.726	5.399
1320	1367-1635	Urban	3	Virgin	60	Cable	60.544	62.675	59.84

(a) 2013

Id	Distance band	Distance band used for weighting	Urban/ rural	Market	ISP	Technology	LLU	Headline speed	Download speed (Mbit/s) 24-hour
1302	2563-5000		Urban	3	Virgin	Cable	Cable	60	46.984
1308	0-385		Urban	3	BT	FTTC	Non-LLU	80	45.564
1318	1367-1635	1299-1680	Urban	3	TalkTalk	ADSL	LLU	20	6.950
1319	1992-2563	1680-2274	Urban	3	Sky	ADSL	LLU	20	5.180

(b) 2014

unit_id	WT_national	WT_ISP	Valid_panel	Pack_number	Tech	Plusnet_upload	LLUvsNon	MarketClass	URBAN2
3640	0.46	1.1	Valid panelist	BT 80	FTTC 76		Y	3	Urban
814409	0	1.09	Valid panelist	Sky 80	FTTC 76		Y	3	Urban
662382	0	5.48	Valid panelist	Plusnet 40	FTTC 38	20	Y	3	Urban
675946	1.5	1.04	Valid panelist	BT 20	ADSL2		Y	3	Urban

(c) 2015

Figure 1: UK home broadband performance data samples

First, in Figure 2, we show the patches that transform the 2014 data to meet the 2013 format, including both the manually-generated target patch and the result from `datadiff`. The first column in Figure 2 lists the elementary constituents of the true target patch. Two columns are deleted and the *Technology* column is recoded before being transposed with *Headline speed*. In fact the recoding is not a valid patch in our transformation language $\mathcal{L}$, since both categories `FTTC` and `FTTP` are mapped onto the single `FTTx` category. The second column shows the result returned by `datadiff`. Every element of the target patch is represented in the result, and the meta-parameters are identified correctly in all cases except for the `recode` patch which, as already noted, cannot be captured by the simplified patch language in the current prototype. The result additionally contains a pair of `linear` patches, highlighting an important point: not all transformations are due to inconsistencies in the data format. The `linear` patches do not represent changes in measurement unit, but are instead consequences of technological improvements over time (i.e. reduced packet loss) which affect the scale but not the shape of the distribution. Transformations of this sort indicate to the analyst interesting changes in the underlying distributions.

The transformation required to reconcile differences between the 2014 and 2015 data is considerably more complex, so we present only a partial listing of the target and output patches in Figure 3. The permutation is expressed, for brevity, in two-line notation using column indices and only the reorderings involving the first ten columns are shown. In total, 24 of the 36 attributes are permuted by the target patch. In the presence of such complexity the accuracy of the `datadiff` algorithm is degraded. The *Valid_panel* column, whose domain contains only two elements, is erroneously recoded onto the domain of the *Urban/rural* attribute. Attribute *WT_national* is replaced, rather than permuted, although its neighbour *WT_ISP* is correctly moved from column 3 to 30. The *MarketClass* column is successfully recoded and permuted from position 9 to 5, but the other `recode` patches in the result do not perform accurately. Even so, the `datadiff` output is still useful to highlight to the analyst that a large permutation in columns has occurred. Also, the `datadiff` output correctly identifies several columns that would need to be deleted, such as *Plusnet_upload* and *Pack_number*, as well as several columns whose value has been recoded, such as *LLUvsNon* and *MarketClass*. Finally, there is wide variation between the permutations in the target and result. In this example we see that even when the `datadiff` output is not fully accurate, it still can serve a red flag to the analyst that there are multiple data quality issues and provide some strong hints for addressing them.

7 RELATED WORK

We are not aware of related work specifically on the data diff problem. Perhaps the closest related work is in the databases, programming languages, and user interfaces literature on improved tools for *data wrangling*. Data wrangling is the process of transforming raw data into a format that is suitable for data analysis, including tasks such as standardizing the format and data cleaning. In the research literature, data wrangling tools have been proposed for defining transformations on columns [9, 16] and for parsing data from unstructured sources [7, 10]. Additionally data warehouses provide extract-transform-load tools for data cleaning [2], including most of the transformations that we consider in our prototype. Other work considers interactive data wrangling and exploration [13, 20]. Another approach to addressing data quality issues is automatic *data cleaning* [1, 15], but we are unaware of previous work that considers the "two-sample cleaning problem" in the way that datadiff does. The specific problem of finding the best permute patch is

```
delete(Distance band used for weighting)
delete(LLU)
recode(Technology, {(ADSL, DSL), (Cable, Cable),
                    (FTTC, FTTx), (FTTP, FTTx) })
permute(Technology, Headline speed)
```

```
delete(Distance band used for weighting)
delete(LLU)
recode(Technology, {(ADSL, DSL), (Cable, ADSL2+),
                    (FTTC, Cable), (FTTP, FTTx) })
linear(Packet loss 24-hour, 384, 0)
linear(Packet loss 8-10pm weekday, 141, 0)
permute(Technology, Headline speed)
```

Figure 2: Target "gold standard" patch (left) and datadiff output (right) for broadband 2013/14

```
delete(Valid_panel)
delete(Plusnet_upload)
recode(Pack_number)
recode(Tech)
recode(LLUvsNon)
recode(MarketClass)
recode(URBAN2)
...
```

$$\texttt{permute}\begin{pmatrix} 2 & 3 & 5 & 6 & 9 & 10 & \dots \\ 31 & 30 & 6 & 7 & 5 & 4 & \end{pmatrix}$$

```
delete(Pack_number)
delete(Plusnet_upload)
replace(WT_national)
recode(Valid_panel)
recode(LLUvsNon)
recode(MarketClass)
recode(URBAN2)
...
```

$$\texttt{permute}\begin{pmatrix} 3 & 6 & 9 & 11 & \dots \\ 30 & 9 & 5 & 3 & \end{pmatrix}$$

Figure 3: Target "gold standard" patch (left) and datadiff output (right) for broadband 2014/15

sometimes called table matching; Gretton et al. [8] present a demonstration of the Hungarian algorithm for this problem. As discussed earlier, *detecting changes in distributions* has been a long-studied problem in statistics, including changes between two data samples [4, 8], changes at unknown points in time [3, 11, 22] and detecting gradual changes [22, 24]. Data diff goes beyond this work by providing an interpretable explanation to an analyst of precisely in what way the data samples differ. Finally, data diff is a problem in the general area of *automating machine learning (AutoML)* [12, 14, 23], which aims to build learning systems that handle all of the steps of machine learning, including preprocessing data, selecting models, and building and evaluating models, without human intervention.

8 CONCLUSIONS

We have introduced the data diff problem, which aims to help the common situation of data analyses over repeated data samples, e.g., every month, every quarter, and so on. Although repeated data samples require repeated data wrangling effort, the idea behind data diff is to use this as an opportunity, because the repeated data samples can be used to detect differences in formatting and in distribution that can reflect underlying data quality issues. We present an algorithm for data diff that casts the problem as a bipartite matching problem, and presented a bootstrap-style procedure for calibrating the parameters of the method. For future work, there is an opportunity for a rich variety of improved algorithms for the data diff problem, richer transformation languages of potential patches, and improved scoring functions for patches that could be trained using machine learning methods, e.g., that take column names or other metadata into account.

ACKNOWLEDGMENTS

The authors would like to acknowledge the funding provided by the UK Government's Defence & Security Programme in support of the Alan Turing Institute. The authors also thank Zoubin Ghahramani and Chris Williams for helpful discussions.

REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment* 9, 12 (2016), 993–1004.
[2] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record* 26, 1 (March 1997), 65–74. https://doi.org/10.1145/248603.248616
[3] Jie Chen and Arjun K Gupta. 2012. *Parametric Statistical Change Point Analysis: With Applications to Genetics, Medicine, and Finance* (2nd ed.). Springer.
[4] William Jay Conover. 1980. Practical nonparametric statistics. (1980).
[5] T. Dasu and T. Johnson. 2003. *Exploratory Data Mining and Data Cleaning.* John Wiley & Sons, Inc, New York, NY.
[6] Bradley Efron and Robert J Tibshirani. 1994. *An introduction to the bootstrap.* CRC press.
[7] Kathleen Fisher, David Walker, Kenny Q Zhu, and Peter White. 2008. From dirt to shovels: fully automatic tool generation from ad hoc data. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 421–434.
[8] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. A Kernel Two-Sample Test. *Journal of Machine Learning Research* 13 (Mar 2012), 723âĹŠ773.
[9] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Symposium on Principles of Programming Languages (POPL).*
[10] Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Programming Language Design and Implementation (PLDI).*
[11] Fredrik Gustafsson. 2000. *Adaptive filtering and change detection.* John Wiley & Sons.
[12] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, Alexander Statnikov, Sébastien Treguer, and Evelyne Viegas. 2016. A brief Review of the ChaLearn AutoML Challenge: Any-time Any-dataset Learning without Human Intervention. In *Proceedings of the Workshop on Automatic Machine Learning (Proceedings of Machine Learning Research)*, Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.), Vol. 64. PMLR, New York, New York, USA, 21–30.
[13] Jeffrey Heer, Joseph M Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation. In *Conference on Innovative Data Systems Research (CIDR).*
[14] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2016. *Workshop on Automatic Machine Learning.* Proceedings of Machine Learning Research, Vol. 64. PMLR.

[15] Ihab F. Ilyas and Xu Chu. 2015. Trends in Cleaning Relational Data: Consistency and Deduplication. *Foundations and Trends® in Databases* 5, 4 (2015), 281–393.

[16] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *ACM Human Factors in Computing Systems (CHI)*.

[17] Ralf Klinkenberg and Thorsten Joachims. 2000. Detecting Concept Drift with Support Vector Machines. In *ICML*. 487–494.

[18] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. 2017. BoostClean: Automated Error Detection and Repair for Machine Learning. *CoRR* abs/1711.01299 (2017).

[19] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

[20] Tomas Petricek. 2017. Data exploration through dot-driven development. In *European Conference on Object-Oriented Programming (ECOOP)*.

[21] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural Information Processing Systems*. 2503–2511.

[22] Alexander Tartakovsky, Igor Nikiforov, and Michele Basseville. 2014. *Sequential analysis: Hypothesis testing and changepoint detection*. CRC Press.

[23] Jan van Rijn, Bernd Bischl, Luís Torgo, Bo Gao, Venkatesh Umaashankar, Simon Fischer, Patrick Winter, Bernd Wiswedel, Michael Berthold, and Joaquin Vanschoren. 2013. OpenML: A Collaborative Science Platform. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery (ECML-PKDD)*, Vol. 8190. 645–649.

[24] Abraham Wald. 1973. *Sequential analysis*. Courier Corporation.