Outlier Detection in Heterogeneous Datasets using Automatic Tuple Expansion

Clément Pit--Claudel, Zelda Mariet, Rachael Harding, Sam Madden
MIT CSAIL
Email: {cpitcla,zmariet,rhardin,madden}@csail.mit.edu

***Abstract*—Rapidly developing areas of information technology are generating massive amounts of data. Human errors, sensor failures, and other unforeseen circumstances unfortunately tend to undermine the quality and consistency of these datasets by introducing outliers – data points that exhibit surprising behavior when compared to the rest of the data. Characterizing, locating, and in some cases eliminating these outliers offers interesting insight about the data under scrutiny and reinforces the confidence that one may have in conclusions drawn from otherwise noisy datasets.**

In this paper, we describe a tuple expansion procedure which reconstructs rich information from semantically poor SQL data types such as strings, integers, and floating point numbers. We then use this procedure as the foundation of a new user-guided outlier detection framework, *dBoost*, which relies on inference and statistical modeling of heterogeneous data to flag suspicious fields in database tuples. We show that this novel approach achieves good classification performance, both in traditional numerical datasets and in highly non-numerical contexts such as mostly textual datasets. Our implementation is publicly available, under version 3 of the GNU General Public License.

I. Introduction

Sensor glitches, data-entry errors, and malicious activities are a few examples of events that can lead to the appearance of outliers in a dataset. If undetected, these values can skew statistics, support invalid conclusions, slow database operations, and cause otherwise avoidable expenses. On the other hand, careful analysis of these values can yield new insight about the data, prevent undesirable events, and generally improve the reliability of the data [1].

Previous literature has generally defined an outlier as "an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism" [9], and has suggested a number of ways to detect and in some cases eliminate suspicious values. Previous approaches to outlier detection include modeling numerical data using Gaussian Mixture Models [15], [17], [18], Histogram modeling [8], [20], and k-nearest neighbors [16].

Little work, however, has focused on developing generic methods for user-guided outlier detection that work with widely diverse and heterogeneous data stored in typical relational database management systems. The relative inexpressivity of basic SQL types – integers, strings, and doubles in particular – might be to blame: strings, for example, can be used to store information as diverse as city names, email addresses, or phone numbers; typically it is the job of application logic to encode the semantically rich domains of these values as one of the basic SQL data types. This paucity of semantic information leaves outlier detection algorithms that run inside of the database with very little information to work with.

This paper presents *dBoost*, a new approach for detecting outliers in highly heterogeneous datasets. Our tool systematically expands the limited space of SQL types to derive richer information. This is done automatically, by applying a library of possible transformations ("expansions") to the values of each column in a table to find rules that are consistent with the bulk of the values that appear in the column. For example, integers can expanded into dates and times by considering them as Unix timestamps, and into sets of booleans by considering them as bit vectors. If an expansion of an integer column to a the day of week from a date found that most of the dates reconstructed from an integer column fall on the same day of the week, then values falling on other days might be flagged as suspicious. As in this example, these expansions can be used to detect outliers that are difficult or impossible to detect using raw data and provide detailed reports to the user.

Hence, our main contribution is a method that automatically applies expansion rules (type-dependent transformations) to create a set of derived attributes for every tuple. These derived attributes are then processed by several outlier detection models to efficiently learn soft constraints about the individual attributes, and to detect soft functional dependencies *between* derived attributes, enabling multidimensional models to detect of a broad class of data inconsistencies. By extracting numerical or highly structured attributes from unstructured data (for example, by extracting casing information from strings), our method makes it significantly easier to detect inconsistencies that would have escaped sophisticated outlier detection systems.

We designed our system to be both fast and memory efficient; it proceeds in three online passes over the data, keeping no more information than strictly necessary (in general, no more than a few dozen values per field in the database schema). The architecture is parallelizable, the analysis can be distributed over multiple computation nodes, and information can be kept from one run to the next so as to eliminate the first and possibly the second pass.

In summary, this paper makes the following contributions:

1) We present tuple expansion, a novel method used to reconstruct structured and semantically rich information from raw data using a user-extensible library of type-dependent expansion rules.
2) We find correlations in non-numerical data and use a novel attribute-based partitioning scheme to find outliers using an efficient histogram model.

3) We integrate these techniques in an easy-to-use, user-extensible toolkit for outlier detection, *dBoost*, and evaluate its effectiveness using several synthetic and real-world datasets.

The rest of this paper is structured as follows: In Section II we present an overview of the *dBoost* framework. We then detail our tuple expansion method in Section III, before applying it to outlier detection in Section IV. We evaluate our tool on synthetic and real-world datasets in Section V, and we describe related work in Section VI. Finally, we conclude in Section VII.

II. dBoost overview

The overall design of the dBoost system can be seen in Figure 1. The first step is to perform *tuple expansion*, where additional semantically rich candidate features are added on to each tuple. Examples of features include the length of a string, the parity of an integer, or the range of dates an integer column can represent when it is interpreted as a Unix time stamp. This process is described in Section III.

These expanded tuples are then analyzed in order to obtain simple statistical information, and to detect soft functional dependencies between different fields. The expanded tuples are then used to train one of three data models (Histograms, Gaussian, or Mixtures), with the help of the statistics and correlation hints gathered at the previous stage.

Finally, the trained model is used to classify tuples into regular records and outliers; these tuples can be the ones the model was trained with, or future inputs to the database system.

From a high level view, our pipeline is implemented as a three-pass streaming algorithm, requiring no memory beyond that required to train the individual models.

The different components of our system are summarized as follows and described in detail in the following sections:

1) Tuple expansion – Tuples are expanded using knowledge about the database schema and field types (Section III).
2) Statistical analysis – The expanded data is analysed to gather basic statistics, along with correlation information. These statistics are used for modeling and outlier detection (Section IV-A).
3) Data modeling – We apply various machine-learning algorithms (Histograms, Gaussian, and Mixtures) to build models of the data (Section IV-B).
4) Outlier detection – Using the models built in the previous stage and user-provided sensitivity thresholds, we report outliers identified by the models trained during the previous stage (Section IV-C).

Table I illustrates these ideas on a very small dataset that includes a transaction ID, a registration date (Reg. date), and a social security number (SSN). We read the data row-by-row, and expand the registration date and SSN values into additional columns. The particular expansion rules are based on the type of each values: Reg. date is an `INT`, so it gets expanded, among others, into a year and a weekday. SSN is a `STRING`, and gets expanded into among others a length and a copy of the string with numbers stripped out and replaced by `<num>`.

Tuple expansions are not materialized in the database, but rather fed into the model one-by-one as the engine processes each row. After being processed, the expanded tuples are discarded. Thus, tuples must be expanded before each stage of the engine's pipeline.

III. Tuple Expansion

> *What if you need to store a date and time value with subsecond resolution? MySQL currently does not have an appropriate data type for this, but you can use your own storage format: you can use the BIGINT data type and store the value as a timestamp in microseconds, or you can use a DOUBLE and store the fractional part of the second after the decimal point.*
>
> *High-Performance MySQL*, 3rd edition (2012), p. 127

Data stored in databases often has rich semantics encompassing dates and times, highly stuctured datatypes such as phone numbers, addresses, and GIS data. The semantics of plain SQL, however, are not expressive enough to properly store these rich data types. Programmers are thus forced to revert to simpler types, relying on application logic to parse the data.[1]

Unfortunately, structural information on the data is what tasks such as outlier detection would most benefit from. For example, the day of the week in a date may be relevant information in a banking application in which transactions are only completed on weekdays, but unless the programmer explicitly duplicates this information in a separate column, it remains inaccessible to an automated outlier detector.

To reconstruct this lost information we expand database tuples by enriching each field with a collection of extracted features. The particular rules that are used to expand each field are selected from an extensible library provided with our tool, based on the field's data type. Figure 2 lists some of the rules that our library provides for three common data types. These rules serve as a starting point for tuple expansion, but we expect users of our toolbox to expand this set of rules to add insight specific to their application domain. Expressing rules as simple functions mapping a value of a given type to a tuple of features makes it possible to express soft constraints about the data easily: instead of specifying hard constraints, users state that a certain way of looking at attributes should be consistent across the rows of the table. In addition, since expansions are never materialized in the database, users are free to experiment with various rules.

The rules presented in Figure 2 are fairly general rules, likely to apply to be suitable to many datasets. For example, the `strip numbers` rule takes a string, and replaces any sequence of digits by the string `<num>`. Such a rule is useful to check that a column is formatted consistently, allowing for variations only in the numbers: a database of legislative citations, for example, could contain entries such as `S.933`, `H.R.21`, `P.L.107-155`, or `P.L.88-352`; applying the `strip`

[1] A number of database systems let user define their own data types, but these are not often used, and migrating an entire database to a different data representation is generally a costly and time-consuming process.

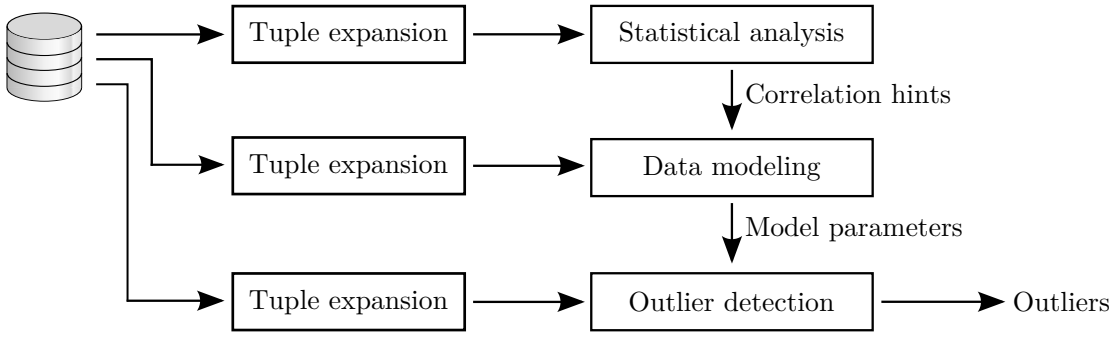


Fig. 1: The *dBoost* pipeline

Original Data			Expansions of Reg. Date (not materialized)			Expansions of SSN (not materialized)		
XID	Reg. Date	SSN	Year	Weekday	...	Length	Strip Numbers	...
1	1416497422	783-345-2351	2014	Thursday	...	12	<num>-<num>-<num>	...
2	1418201134	773-746	2014	Wednesday	...	**8**	**<num>-<num>**	...
3	1420359855	773-289-5552	2015	Sunday	...	12	<num>-<num>-<num>	...
4	1421575392	849-843-2729	2015	Sunday	...	12	<num>-<num>-<num>	...
5	01302015	773-387-9201	**1970**	Friday	...	12	<num>-<num>-<num>	...
6	1424866716	821-322-1857	2015	Wednesday	...	12	<num>-<num>-<num>	...
7	1425059692	822-971-1892	2015	Friday	...	12	<num>-<num>-<num>	...

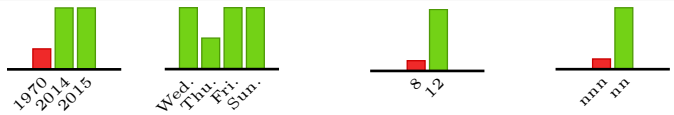


TABLE I: An example dataset showing outliers based on a histogram model. The rows detected as outliers are highlighted in red. The row with `XID=2` is flagged due to its "Length" and "Strip Numbers" expansions: as seen in the corresponding histograms, the values `8` and `<num>-<num>` are seen few enough times in the data that they are flagged as suspicious. The row with `XID=5` is also flagged as an outlier due to its incorrect registration date: indeed, the histogram analyzing the years shows that 1970 does not occur in the database frequently.

```
string:
"32-G414"        →  length:      7
                    signature:   NdNdPdLuNdNdNd
                    uppercase:   True (1)
                    lowercase:   False (0)
                    email ok:    False (0)
                    stripped:    <num>-G<num>
                    title case:  True (1)

int:
1418222134       →  date:        (2014,12,10)
                    time:        (14,35)
                    weekday:     Wed (2)
                    weekend:     False (0)
                    binary:      0b10101...010
                    mod-10:      4

float:
1418222134.325   →  intpart:     1418222134
                    fracpart:    0.325
                    millis:      325
                    date, ...:   ...
```

Fig. 2: Selected tuple expansion rules.

`numbers` rule to this data yields `S.<num>`, `H.R.<num>`, `P.L.<num>-<num>`, or `P.L.<num>-<num>`; this reduces the whole dataset to a small number of patterns that can be easily analyzed using data models.

The `signature` rule is equally interesting: to extract the signature of a string, our tools replace each character by the name of its Unicode class: uppercase Latin letters are `Lu`, lowercase Latin letters are `Ll`, digits are `Nd`, and punctuation signs are among others `Po`, `Pe`, `Ps`, and `Pd`. Such a rule is useful to check the consistency of various fixed-width formats such as ISO 8061 timestamps, birth dates, or phone numbers. As a more detailed example, a column in a database could be used to record references to particular sections, subsections, and paragraphs in a text: entries would be in the form `§901.04(a)`, `§853.02(d)`, or `§910.45`. Extracting the signature of these entries would yield `PoNdNdNdPoNdNdPsLlPe` for the first two entries, and `PoNdNdNdPoNdNd` for the last one. Just like before, this reduces the dataset to a small set of patterns, making it easy to detect discrepancies.

IV. Data Modeling and Outlier Detection

Once a tuple is expanded, it is fed into a statistical analyzer which consolidates statistics on the data as well as correlations for use in the data models and subsequent outlier detection. This section provides details for each stage of this process.

A. Statistical Analysis

The first stage in our engine is analyzing the expanded tuples. This phase collects simple statistics including average, variance, standard deviation, and approximate cardinality on each column of the table and estimates which sets of columns are correlated.

These statistics have three purposes. First, they are used to detect univariate outliers, for example values that are several standard deviations from the mean in numerical attributes, or that have never occurred before in low cardinality attributes. Second, they are used to determine which columns in the table are correlated. Third, these statistics precompute parameters required by certain data models, thus speeding up the training phase of the models.

We focus on two inter-column correlation strategies:

- For mostly non-numerical datasets, we use a cardinality-based measure, flagging groups of expanded columns as correlated when their joint cardinality is below a user-specified threshold. When two columns are correlated (e.g., when one is computed directly from the other), the number of distinct pairs in the columns is similar to the number of distinct items in either column. On the other hand, when two columns are independent, the number of distinct pairs is close to the product of the number of distinct values in each column.

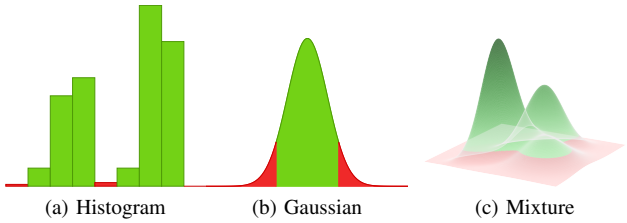
- For mostly-numerical datasets, we use Pearson's product-moment correlation. It relies on the Pearson correlation coefficient, which measures linear dependencies between two vectors.
  Given two column vectors X and Y, Pearson's coefficient R is given by the following formula:

$$R = \frac{\mathrm{Covar}(X,Y)}{\sqrt{\mathrm{Var}(X)\,\mathrm{Var}(Y)}} \tag{1}$$

  R's value always lies between -1 and 1. An R value close to 0 indicates little or no correlation, while values close to $+1$ or -1 indicate strong positive or negative correlations, respectively. Pairs of columns with a value of R above a user-specified threshold are added to a list of correlation hints, for use by the models.

It is debatable whether correlations between expanded tuple fields from the same original value lead to better outlier detection. On the one hand such dependencies may provide valuable insight about the data (e.g., an event that occurs every Monday of May and every Thursday of June). On the other hand, taking these subtuple correlations into account vastly increases the size of the search space, and may add spurious hits to the results. Experimentally, we found that disregarding intra-field correlations made the entire process faster and more robust, and did not hurt accuracy on our test sets.

All aforementioned statistics and correlation hints can be computed using a single pass over the data: the expanded tuples are analyzed one row at a time, and the final statistics and correlations are computed after the last tuple has been processed. This contrasts with more advanced approaches to the detection of correlations and soft functional dependencies, such as the one used in CORDS [11]. Our simpler approaches yields a lower specificity, but still achieves good classification results, in part because each model only uses correlation hints as a guideline for interesting groups of columns to analyze. An excessive number of hints can thus affect performance, but does not significantly diminish the quality of the results. On the other hand, missing a correlation causes models to not analyze the corresponding group of columns, and thus to fail to uncover potential outliers. As with the other parts of our system, the correlation detector used in the statistical analysis phase is modular and could be replaced by any other scheme, including CORDS.



(a) Histogram (b) Gaussian (c) Mixture

Fig. 3: Simple visualization of the outlier detection strategy employed by each model. Possible outlier values are shown in red.

The results of the analysis pass are available to all models used at later stages in the tool.

B. Data modeling

The current implementation of *dBoost* contains three data modeling strategies. First, histograms provide a way to study discrete heterogeneous distributions. Although histograms are widely used to capture statistics in databases, we present a simple pruning heuristic to limit the number and size of histograms that we construct for outlier detection purposes. We also provide standard statistical Gaussians and mixture models that handle continuous data well, but assume that it follows a uni- or multi-modal Gaussian distribution. The effectiveness of these basic models comes from the rich information exposed by the expanded tuples, which allows them to be competitive with more complex models.

In addition to uniformly modeling of entire datasets, dBoost is capable of automatically clustering data using expanded attributes, generating different models for different subsets of the data. We call this novel approach to outlier detection *meta-modeling*.

The next four subsections describe these ideas in more detail.

1) Histogram-based Modeling: The histogram model (Figure 3a) does not make any assumption about the data under study. Instead, it counts the occurrences of each unique value in each column of the expanded tuple and for each set of potentially correlated sub-tuples (as suggested by the analysis module). These counts, accumulated over the entire dataset, provide a *de facto* distribution of the data in each field and set of correlated fields. This makes histograms a powerful model for non-numerical and heterogeneous data.

To limit memory usage, and to speed up the modeling phase, we discard histograms as soon as they reach a certain size – say, 16 bins. Discarding histograms when their number of bins reaches a fixed threshold is just one of a number of heuristics that could be implemented here; the idea is that a profusion of different values, all repeating infrequently, is unlikely to provide valuable insight as far as outlier detection is concerned (as an extreme example, the histogram of an attribute with no repeated values would only have one value per bin, and would not yield any insight about the data). With this discarding heuristic applied, histograms are quick to generate and extremely memory efficient.

Histograms also have the valuable property of treating sets of fields (obtained via correlation analysis) and single fields in the exact same way, thus permitting to model single columns or groups of attributes indifferently. Finally, because they make no assumption about the data they manipulate (aside from the requirement that it be of small cardinality), histograms are able to accurately describe a broad class of discrete distributions.

2) Simple Gaussian Modeling: Univariate Gaussians (Figure 3b) are a widely used statistical model for data. They treat each value x_i of the expanded tuples as random sample drawn from a normal distribution $\mathcal{N}(\mu_i, \sigma_i)$.

The model's parameters (a pair (μ, σ) for each numerical column) are computed as each column's mean and standard deviation. In the common case where the dataset has not significantly changed between the analysis and the modeling passes, the information obtained during the statistical analysis pass is sufficient to derive these parameters.

Despite its simplicity, this model presents the attractive property of requiring extremely little memory – on the order of the size of one expanded tuple.

3) Mixture Modeling: Multivariate Gaussian Mixture models (Figure 3c) are another standard statistical model. They take advantage of the correlation hints supplied by the statistical analysis pass to model sub-tuples of the expanded tuples as samples from multivariate Gaussian mixtures (GMMs), creating one model per group of correlated columns.

For example, if the statistical analysis phase outlines a pair of fields (f_1, f_2) as good candidate for joint modeling, then the Mixture modeling strategy will learn a particular GMM to model this correlation. Pairs of values (X_1, X_2) are here assumed to have been produced by random sampling off a distribution of the form

$$\sum_{j=1}^{N} \pi_j \mathcal{N}(\mu_j, \Sigma_j)$$

where N is the number of individual components that the GMM comprises (N is a user-defined value in our implementation, but abundant literature exists on the subject of choosing N [19] [3]), and π_j, μ_j and Σ_j are parameters of the GMM learned as part of the modeling pass [7].

Unlike simple Gaussian models, the expectation maximization algorithm used in inferring the optimal model parameters for Gaussian mixtures does require retaining some data in memory. Still, most of the fields obtained after expanding each tuple are discarded after the relevant ones are extracted for learning purposes; in most cases we expect the set of values retained to be much smaller than the set of all attributes, thus limiting the memory usage.

In addition, when dealing with large amounts of data, it is possible – and indeed, preferable – to train the Mixture model on a randomly sampled subset of the data before running the full analysis. This approach is particularly relevant when using the Mixture model, but can be applied to all models to shorten the learning phase when dealing with very large datasets.

4) Meta-modeling through attribute-based partitioning: The models presented above treat attributes and sets of correlated attributes as a whole. In some cases, however, it is possible to identify sub-populations of tuples by scrutinizing certain expanded attributes of the data; these sub-populations can then be studied separately, yielding more insight and better outlier classification performance.

As an example, consider the case of an airline adjusting status levels for its frequent fliers, using the number of flights for each passenger as well as their status level. A non-partitioned analysis may not return any interesting information, but a partitioned analysis could single out passengers in lower status levels traveling significantly more than average, or passengers with higher status traveling rarely. This would work even if statuses were stored as textual data, with no indication of their relative rankings.

The general approach, given a dataset and a pre-existing model, therefore consists of extracting sets of attributes based on correlation hints provided by earlier stages of the pipeline, and dividing each group of attributes between a single key (in the example, the status level) and one or more sub-population attributes (in the example, the number of flights). One instance of the selected model is then built for each value of the key. For example, if the statistical analysis phase highlights a correlation between columns A (status levels) and B (number of flights), and column A contains values $a_1, \ldots, a_n$ (`bronze`, `silver`, `gold`, ...), then we distribute the pairs (A, B) into n partitions based on the value of A; values of B in each of these partitions are then modeled independently (in the example, this yields a different model of flights count for each status level).

This type of approach is useful when the distribution for an attribute or set of attributes is multi-modal. A high-level non-partitioned analysis will reveal values that fall in none of the classes; a partitioned approach, on the other hand, may more easily reveal discrepancies by suppressing interference between each class.

In addition to providing better classification accuracy, partitioning may lead to better runtime performance by diminishing the size of the dataset covered by each model. These benefits are especially important when model construction performance does not scale linearly, and when data volumes are too large to be analyzed on a single machine.

Finally, attribute-based partitioning allows for previously impossible analysis. Assuming for example that a dataset with two columns has 4 classes identified by the value in the first column, each with 10 distinct expected values in the second column, a generic histogram-based analysis would discard the histogram for the pair of values as having too many buckets (40). A partitioned analysis, on the other hand, would allow

the construction of four histograms, each with 10 regular bins and potentially a few outliers.

In our prototype implementation, we focused on partitioning applied to the discrete histogram case; the technique, however, generalizes to all the models presented above.

C. Outlier Detection

Models, once properly trained, are used for classification and detection of outliers – either in incoming `INSERT` operations on a running system, or in existing rows (possibly but not necessarily the ones used during the model training phase).

Given that databases can contain tables with tens or hundreds of columns, simply flagging a row as an outlier is insufficient: users cannot be expected to painstakingly analyze each outlying row. Instead, *dBoost* automatically indicates which values in the row caused it to be flagged as an outlier.

The inter-column correlations are also taken into account during modeling: if the statistical analysis phase detected a correlation between two columns a and b, each tuple t will be augmented by an additional field that contains the corresponding pair (t_a, t_b). This field is treated as a single, multidimensional value, and is analyzed similarly to the other values by the models.

As for statistical modeling, the heuristics that we employ for simple Gaussian and mixture modeling are not new; our contribution rests in the heuristics that we use for histograms (IV-C1), and in the description of *meta-modeling*.

1) Histogram Modeling: The histogram-based modeling strategy proceeds in two phases to detect outliers.

First, after running through the learning phase, it decides for each histogram whether that histogram is "peaked" (i.e. showing a few strong modes) enough to be used to detect outliers. The aim of this phase is to discard histograms where most bins have a similar number of values, and are thus not useful for outlier detection. In practice, we use a simple statistical test to determine whether a histogram is sufficiently modal: if the number of elements that fall into the most populated ("top") bins is less than some user-specified proportion, the histogram is discarded. Finding how many bins to include in the set of top bins is the most challenging part, and for this paper we explored two thresholding strategies (Figure 4):

- *Distribution-independent* – Given a histogram with N bins, we count only the values in the top bin if $1 \leq N \leq 3$, in the top 2 bins if $4 \leq N \leq 5$, and in the top 3 bins for $3 \leq N \leq 16$ (histograms with $N > 16$ bins were previously discarded). This method is stable when the set of bins is static (week days, booleans, ...), but it is sensitive to the addition of removal of bins.
- *Distribution-dependent* – We sort the bins in increasing order of bin size b_i, and find the index $i_{\max}$ such that the ratio $r = {}^{b_{i+1}}/_{b_i}$ is maximal (this calculation is safe, because the bin sizes are non-zero integers). If that ratio is under a user-defined threshold, we reject the histogram; otherwise, we consider bins $i_{\max}..\texttt{end}$ to be "top" bins.

Figure 4 shows various types of histograms, and lists the conclusions that each of these two approaches yield.

After identifying a relevant set of histograms (this operation only needs to run once, at the very beginning of the last pass), we proceed to the actual detection phase. We classify an expanded tuple X as an outlier if any of its values (or set of values, as grouped according to the correlation hints previously obtained) x_a verifies:

$$h_a(x_a) \leq \epsilon \sum_k h_a(k) \tag{2}$$

where $h_a(x)$ designates the number of tuples with value x for field a, and ϵ is a user-chosen sensitivity parameter.

In this model, identifying and reporting the outlying attributes is simply a matter of remembering which values x_a failed test (2).

2) Simple Gaussian Modeling: The simple Gaussian model measures how much each value differs from the mean computed in the preceding pass. Given a tolerance parameter θ, a row is deemed an outlier if at least one of its attributes a has a value v_a such that

$$|v_a - \mu_a| \geq \theta \cdot \sigma_a \tag{3}$$

where μ_a and σ_a are the model's parameters for column a, as described in Section IV-B2.

In this model, detecting which values are responsible for the outlier flag is simply a matter of keeping track of which attributes satisfy Equation (3). The simple Gaussian model does not take correlation hints into account, and thus reports only single-attribute outliers.

3) Mixture Modeling: In the Mixture model, the likelihood of each (possibly multidimensional) field is evaluated using the corresponding GMMThis model operates under the assumption that data is accurately modeled by the chosen number of components in the GMM, and in particular that each non-outlying data point is well modeled by one of the Gaussians of the GMM.

This makes it possible to assign a Gaussian component to each tuple, and then flag as outliers the tuples that are not sufficiently well explained by their corresponding Gaussian (see [18]). Given a tuple t and its corresponding Gaussian c, this means rejecting t if

$$\pi_c \cdot \Pr(\textbf{dist}(t, \mu_c) > d_0) \leq \theta \tag{4}$$

where θ is a user-defined parameter between 0 and 1, and d_0 is the Mahalanobis distance of t to the Gaussian.

As in the Gaussian Model, providing the user with a list of attributes that caused the row to be flagged as an outlier is simply a matter of tracking correlations that satisfied equation (4).

4) Partition-based modeling: In the partition-based case, outliers are detected by the underlying models. To classify a given expanded tuple, each group of correlated attributes is divided between a one-attribute key and a group sub-population attributes. This group of attributes is then passed

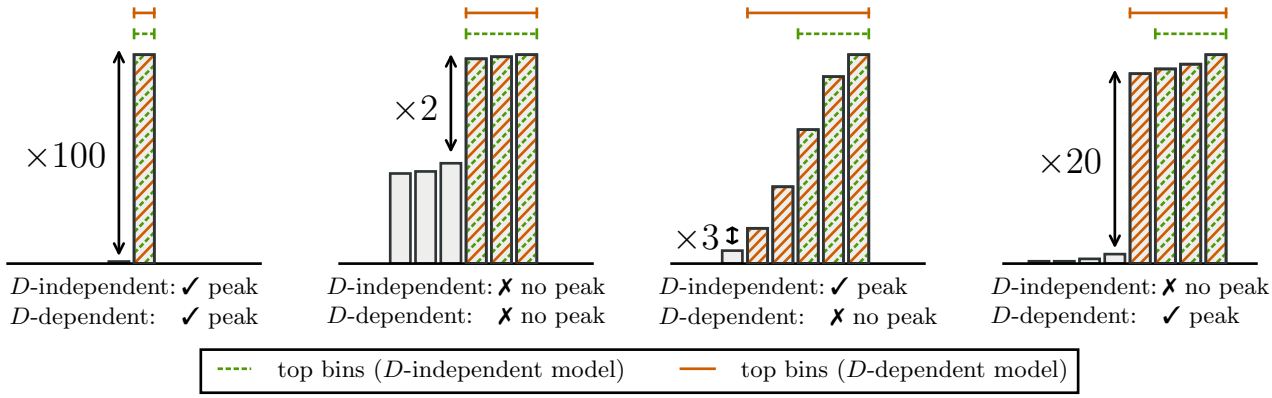


Fig. 4: Sample histograms, and corresponding decisions with distribution-dependent (D-independent) and distribution-independent (D-independent) thresholds. Each figure shows a sorted histogram, with the top bins hatched (in dotted green in the distribution-independent case, and in solid orange in the distribution-dependent case). The vertical arrows show the small value $r = 3$ in the distribution-dependent case. The weaknesses of the distribution-independent-model show in the third and fourth plots: in the third one the distribution-dependent strategy correctly rejects because of the small r; in the fourth the distribution-independent strategy yields an incorrect threshold.

to the underlying model corresponding to the given value of the key, and the whole original tuple is reported as an outlier if any of its groups of sub-population attributes is marked as such by the underlying models.

V. Evaluation

Sam: Can we compare to some kind of "baseline" – i.e., how well would these statistical methods do without access to expanded data?

We implemented *dBoost* as a library that runs on top of a standard relational database or set of structured data files. Our code is publicly available on GitHub under version 3 of the GNU Public License[2]. The program is made of two parts: a library, and a number of data acquisition front-ends (CSV and SQL are currently supported). The library provides functions for each of the phases previously described, and includes a collection of expansion rules from SectionIII.

This section presents the results of running our tool on the following real and synthetic datasets.

- *Synthetic datasets*
  Fizz-Buzz A mixed textual-numerical dataset in which each record contains two entries: a number, and either "Fizz" if the number is divisible by 3, "Buzz" if the number is divisible by 5, "FizzBuzz" if it is divisible by both, and the number itself (as a string) otherwise. Outliers appear when the second column does not respect these rules; this can be a misplaced "Fizz", a missing "Buzz", or even a totally different string (e.g. "Woof!").
  Web logins A series of three non-numeric datasets in which entries contain the login time and connection location for different users. Each user has different connection habits, leading to different types of outliers.
- *Real-world datasets*
  CSAIL Directory A publicly-accessible directory of researchers, in which each record may include a first and a last name, a phone number, an office number, an email, and a job title. Outliers are hard to define mathematically in this case, and we instead demonstrate how the ideas exposed in previous sections of the paper come together to allow for efficient detection of unusual values.
  Intel lab data A publicly-available numerical dataset of temperature, light, humidity and voltage measurements. Outliers are due mostly to sensor glitches.

These datasets showcase the power of our methods, both in terms of classification power and expressiveness and succinctness when adding new rules to the system[3]. Where relevant we include performance measurements. These numbers intend to demonstrate that our approach is computationally reasonable and that our models scale linearly given a fixed training size. We show in Section V-E how our prototype requires on the order of a few minutes to process a million elements using a high-level single-threaded scripting language. A production-ready implementation would run one or two orders of magnitude faster by taking advantage of the inherent parallelizability of the models, using an efficient on-disk representation of the data, and relying on a lower-level language with an efficient optimizing compiler.

The following subsections describe each of the test sets and associated results in greater detail.

A. Soft constraint specifications: Fizz-Buzz

We start with an extremely simple example, highlighting how easy our system makes it to encode and check data integrity constraints. The Fizz-Buzz programming exercise is based on a children's game and frequently found in programming interviews. The synthetic dataset we generated obeys the following

[2] https://github.com/cpitclaudel/dBoost

[3] Indeed, the set of rules used for tuple expansion is user-configurable, and new rules can be easily added; thus, specific knowledge about the data can be taught to the system by users, expressing some soft form of data integrity constraints.

rules: for each record x, y, x is a number between 0 and 1000, and y is "Fizz" if $x \mod 3 = 0$, "Buzz" if $x \mod 5 = 0$, "FizzBuzz" if $x \mod 15 = 0$, and x otherwise. In our synthetic dataset we introduced three outliers: `(25, "Fizz")`, `(28, "Woof!")`, `(30, "Buzz")`. Each demonstrates a different error, namely swapping "Fizz" and "Buzz", producing entirely incorrect output, and failing to recognize that a number is divisible by both 3 and 5.

A traditional way of checking that all tuples verify the production rule outlined above is to encode this rule itself as a database integrity constraint. This requires encoding the full complexity of the exercise in the rule, and would require manual adjustments if the rules were to change. Instead, a user might want to specify the bare minimum for the system to infer the rules; in this case, it is sufficient to add one extraction rule, mapping integers to two booleans denoting whether they are divisible by 3 or 5. Such a rule could be written like this:

```
@rule
def fizzbuzz(x: int) -> ("div 3", "div 5"):
  return (x % 3 == 0, x % 5 == 0)
```

Running the discrete statistical analyzer on the synthetic datasets suggests that the two columns are correlated, and using the histogram model flags the aforementioned outliers. The output of the program for the `(30, "Buzz")` line, for example, is similar to:

```
30 Buzz
> Values (30, 'Buzz') do not
  match features ('div 3', 'strip numbers')
• histogram for ('div 3', 'strip numbers'):
  [532] ▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨ (False, '<num>')
  [133] ▨▨▨▨▨▨ (False, 'Buzz')
  [  1] ▯ (False, 'Fizz')
  [  1] ▯ (False, 'Woof!')
  [  1] ▯ (True, 'Buzz')
  [267] ▨▨▨▨▨▨▨▨▨▨▨▨ (True, 'Fizz')
  [ 66] ▨▨ (True, 'FizzBuzz')
```

Using the partitioned histogram model produces similar output:

```
30 Buzz
> Values (30, 'Buzz') do not
  match features ('div 3', 'strip numbers')
• histogram for ('strip numbers',) if 'div 3' = True:
  [  1] ▯ ('Buzz',)
  [267] ▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨ ('Fizz',)
  [ 66] ▨▨▨▨▨▨ ('FizzBuzz',)
... if 'div 3' = False:
  [532] ▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨ ('<num>',)
  [133] ▨▨▨▨▨▨ ('Buzz',)
  [  1] ▯ ('Fizz',)
  [  1] ▯ ('Woof!',)
```

B. *Logins: a more realistic partitioned dataset*

Our web activity synthetic datasets are comprised of two columns: a Unix timestamp stored as an `INT`, and a country. Each dataset is supposed to track the connections of a registered user on a website; such a dataset could be obtained by selecting the relevant rows out of a large table listing all connections of all users. Each user exhibits a different connection pattern:

- One user always connects from the same country; values that do not match this country are outliers.
- The second connects from one country during the week, and from another during the week-end; outliers in this case are connections from a country that doesn't match the country for that day of the week.
- The third user connects from a set of three countries, with no discernible pattern. This should not return any outliers.

The datasets are randomly generated sets of 2000 connections, listed in no particular order. The target outlier rate is 5 % in each generated dataset.

As in the *Fizz-Buzz* example, numerical models are useless here, whereas histogram-based models produce good results. In the first case, a histogram-based model with no correlation analysis is sufficient to flag the outliers (analysis time: 0.08 s, training time: 0.14 s, total runtime: 0.38 s)[4]. In the second case, the discrete statistical analysis phase singles out interesting pairs of correlated columns, including `(date # day of week, country)` and `(date # is weekend, country)`. A histogram-based model is sufficient to successfully flag outliers, without resorting to partitioning.

Mixing data from two or more users, however, shows the limits of the non-partitioned histogram approach. If we only look at two-columns correlations the individual behavior patterns become less apparent, and if we look at three-column correlations the histograms become too large and spurious hits start to appear due to the many discrete correlations hints returned by the analyzer. The partitioned histograms model, on the other hand, can handle the three-users without particular difficulties, by highlighting (among others) the triplet `(user, date # is weekend, country)` (analysis time: 0.14 s, training time: 0.19 s, total runtime: 0.56 s).

C. *CSAIL Directory*

The CSAIL directory is an online directory of about 1000 faculty, staff and students in the MIT Computer Science and Artificial Intelligence Laboratory[5]. Each entry contains a person's name, phone number, office number, email address, and position.

Some entries, such as a phone number, may be missing from the directory. Nonetheless, we expect our framework to be useful in flagging discrepancies between different records. Since the notion of what constitutes an outlier here is imprecise at best, we also expect the tool to allow the user to explore different sets of parameters. To illustrate the process, we present the results returned by two iterations of the tool in the next subsection, each with increasingly strict limits on the number of outliers returned. Because the CSAIL test set is exclusively textual, we use the histogram model for evaluation; continuous models would not fare as well, since only part of the expanded

[4] All runtime results were obtained using a 4 core i7-4810MQ CPU @ 2.80GHz and 32GB of RAM.

[5] https://www.csail.mit.edu/peoplesearch

tuples are numeric. We also manually annotated the dataset for outliers to determine the accuracy of our system.

1) Initial run: low specificity filtering: The search for outliers is initiated with parameters $\theta = 0.8, \epsilon = 0.2$ (analysis time: 0.36 s, training time: 0.11 s, total runtime: 0.60 s). Correlation detection is disabled for these experiments.

This invocation produces a long list of outliers; a small subset of these is presented below. For privacy reasons, names, phone numbers, office numbers, and emails have been omitted or anonymized in the following listings.

```
Hacker, Alyssa, 32-D968,
  aph@CSAIL.MIT.EDU, Postdoctoral Associate
> Value 'aph@CSAIL.MIT.EDU' doesn't match feature '
    lower case'

Bitdiddle, Ben, NE47-989,
  bbitdid@mit.edu, Graduate Student
> Value 'NE47-989' doesn't match feature 'signature'

Lu-ater, Eva, 32-G972,
  eva@csail.mit.edu, Research Scientist
> Value 'Lu-ater' doesn't match feature 'title case'

Tweakit, _, 32-G699,
  twktem@mit.edu, Administrative Assistant
> Value '_' doesn't match feature 'empty'
```

In total, 451 entries contain outliers, out of a total of 1000. Office numbers are often flagged, as well as names and email addresses. By changing the input parameters to $\theta = 0.8, \epsilon = 0.05$, most of the outliers due to office numbers disappear due to the lower sensitivity. `Hacker, Alyssa` disappears from the list, since e-mails with inconsistent capitalization occur frequently enough in the database that they are not considered outliers at sensitivity level $\epsilon = 0.05$. After tuning these parameters, we are left with 68 outliers.

In addition to identifying outliers, *dBoost* is equipped with tools that provide the user with additional feedback on why features were identified as outliers.

```
Bitdiddle, Ben, NE47-989,
  bbitdid@csail.mit.edu, Graduate Student
 > Value 'NE47-223' doesn't match feature 'signature'
 • histogram for ('signature',):
   [266] <empty>
   [  1] Lu,Lu,Nd,Nd,Pd,Nd,Nd,Nd
   [  1] Lu,Nd,Nd,Pd,Nd,Nd,Nd
   [  2] Nd,Nd,Lu,Pd,Nd,Nd,Nd
   [485] Nd,Nd,Pd,Lu,Nd,Nd,Nd
   [ 51] Nd,Nd,Pd,Lu,Nd,Nd,Nd,Lu
   [155] Nd,Nd,Pd,Nd,Nd,Nd
   [ 36] Nd,Nd,Pd,Nd,Nd,Nd,Lu
   [  3] Nd,Nd,Pd,Nd,Nd,Nd,Nd
   [  1] Nd,Pd,Nd,Nd,Nd

Lu-ater, Eva, 32-G972,
  eva@csail.mit.edu, Research Scientist
> Value 'Lu-ater' doesn't match feature 'title case'
• histogram for ('title case',):
  [ 15] False
  [986] True

Tweakit, _, 32-G699,
  twktem@mit.edu, Administrative Assistant ...
> Value '_' doesn't match feature 'empty'
• histogram for ('empty',):
  [1000] False
  [   1] True
```

Our tool highlights the incorrect field, and prints the corresponding histogram. The bin in which the suspicious value falls is also highlighted. The `signature` case is particularly interesting: recall that to extract the signature of a string, our tools replace each character by the name of its Unicode class; hence the string `NE47-989` is converted to `Lu,Lu,Nd,Nd,Pd,Nd,Nd,Nd` (two letters, two numbers, one dash, three numbers), which does not fall in any of the dominant bins (the most frequent case, `Nd,Nd,Pd,Lu,Nd,Nd,Nd`, describes office numbers like `32-G804`, the predominant form of office numbering in the Stata Center).

Manual inspection of the results reveal that most of the outliers reported are actually bad inputs. There are, however, a number of false positives, such as:

```
DeFect, Cy, 32-D597,
  cydf@csail.mit.edu, Graduate Student
> Value 'DeFect' doesn't match feature 'title case'
• histogram for ('title case',):
  [ 15] False
  [986] True
```

The case of `DeFect` is correct, but our tool notes that it does not adhere to the casing standard derived from other tuples, and thus reports it.

We compared *dBoost*'s output to a manually annotated version of the CSAIL directory to analyze its accuracy; the results are shown in Figure 5.

D. Intel Lab Data

We also evaluated our outlier detection framework on sensor data from the publicly available Intel Lab Data set[6]. The Intel Lab Data contains data collected from 54 sensors spread throughout the Intel Berkeley Research Lab. Each data entry contains information including temperature, humidity, light and voltage taken from a Mica2Dot sensor and weatherboard. The dataset contains a total of approximately 2.3 million measurements.

The Intel lab dataset has known outliers from faulty sensor readings due to periods of critically low voltage. During these periods, the sensors go haywire and produce faulty measurements. For example, the temperature may be registered as over 120 degrees Celsius, which is obviously abnormal behavior in a human environment such as where the sensors were deployed.

We analyzed a sample of 1000 data points selected at random from the sensor data; due to the numerical nature of this data, the Simple Gaussian and Mixture models are better-suited to analyzing it than the Histogram model.

We also compare the results of our models to Local Outlier Factors, a common outlier detection methodology, in this section.

[6] http://db.csail.mit.edu/labdata/labdata.html

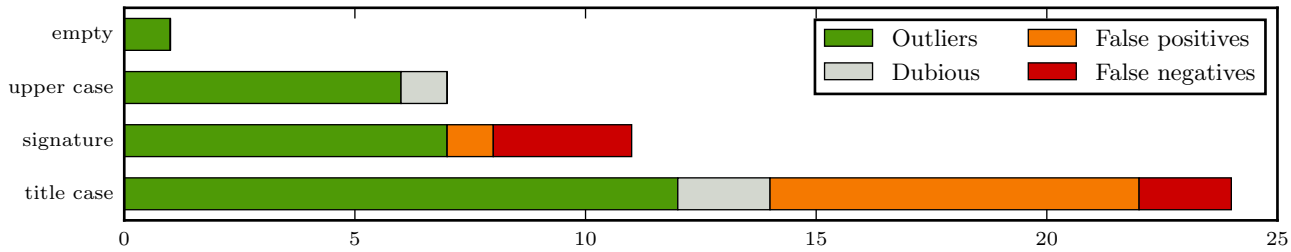


Fig. 5: Accuracy of *dBoost* on the CSAIL dataset, evaluated by comparison to manual annotation of outliers in the directory. Outliers were detected using $\theta = 0.8$ and $\epsilon = 0.025$; the tuples are sorted according to why they are – or are flagged as – outliers. The false positives in the last category are due to names whose proper capitalization is not title case. Green represents region of agreement between the output of *dBoost* and manual annotation, red shows outliers in the manual annotation that *dBoost* did not find, orange shows records that *dBoost* flagged as outliers that the manual annotation did not, and grey indicates values that could not manually be categorized with complete certainty as either outlying or non-outlying.

1) Simple Gaussian Model: The results from running the sensor data set through the Simple Gaussian model are shown in Figure 7a. The data is plotted in light green, and the outliers are marked by dark red crosses.

In this experiment we flag the entries with column values that fall outside 1.5 standard deviations of the mean of that particular column as outliers. This model runs relatively fast, as no correlations are computed (analysis time: 0.03 s, training time: 0 s, total runtime: 0.12 s).

2) Mixture Model: We set the statistical threshold to 0.7, which produces two correlations between temperature and humidity and between temperature and voltage. Figure 7b shows the results when using a single Gaussian component. Points flagged as outliers have a likelihood of less than 7.5% of being produced by the Gaussian generated by the model (analysis time: 0.03 s, training time: 0.34 s, total runtime: 0.73 s).

This model is able to detect values with high temperature and low voltage as outliers.

Figure 7c shows the results obtained using the Mixture model with two components, using the same 1000 randomly selected data points. Flagged values have a likelihood of less than 7.5% under their dominant Gaussian (analysis time: 0.03 s, training time: 0.35 s, total runtime: 0.78 s). When using two Gaussians, the points clustered around the temperature 120 degrees Celsius are no longer detected as outliers: they are modeled by their own Gaussian (although this Gaussian's weight is smaller than its counterpart). This model highlights the points within normal sensor operation that have outlying results.

3) Local Outlier Factors: In this section we compare the results of our Gaussian and Mixture models to Local Outlier Factors (LOF) [5], a frequently used method for outlier detection. LOF measures the degree to which a data point is an outlier by comparing each data point's reachability to those of its k nearest neighbors. The higher the LOF, the more isolated the data point relative to its local neighborhood and therefore the more likely the point is to be an outlier.

One downside of LOF compared to *dBoost* is that it can only evaluate two-dimensional data. The original algorithm also has significant computation complexity in order to calculate the distance to the nearest neighbors of each data point. One benefit of LOF, however, is that the algorithm returns a continuous value that indicates the degree to which a point is an outlier, as opposed to a binary value.

Figure 7d shows the outliers detected by LOF when $k = 2$. We observe that contrary to the Gaussian and Mixture models, the outliers detected by LOF are scattered throughout the data. The outliers are not necessarily the points one would intuitively assume are outliers. This is because points that are within the normal range of the data will be selected as outliers if they are far enough away from the other points nearest to them. We find that LOF is not as useful at pointing out the tagged outliers in the sensor data set.

E. Scalability

We measure the total runtime of our system, including the data modeling and outlier detection phases for the Simple Gaussian, Mixtures with 2 Gaussians, and Histograms. We used the Intel sensor data set from Section V-D to evaulate the Gaussian and Mixture models, and the CSAIL directory from Section V-C to evaluate the Histograms. We use random sampling to provide training sets of 1 thousand and 10 thousand elements from the Intel dataset to build the data models. We test them on all 2+ million elements in the dataset. To provide a more comprehensive study of the scalability of the Histogram model, we replicated the rows of the CSAIL directory to increase the training and test set sizes.

In Figure 6, we show the runtime of our prototype. Each line shows a model trained with a different training set size. As shown in the figure, runtime scales linearly as the test set size increases. Applying additional optimizations such as using a lower-level language or enabling parallism would improve runtime performance to production-ready levels.

VI. Related Work

There has been substantial research in how to build models to detect outliers [2], including how to detect outliers in high-dimensional data by searching the subspaces of the data [23][12]. However many of these algorithms are complex and can require substantial computation to determine whether a new data point lies outside the data.

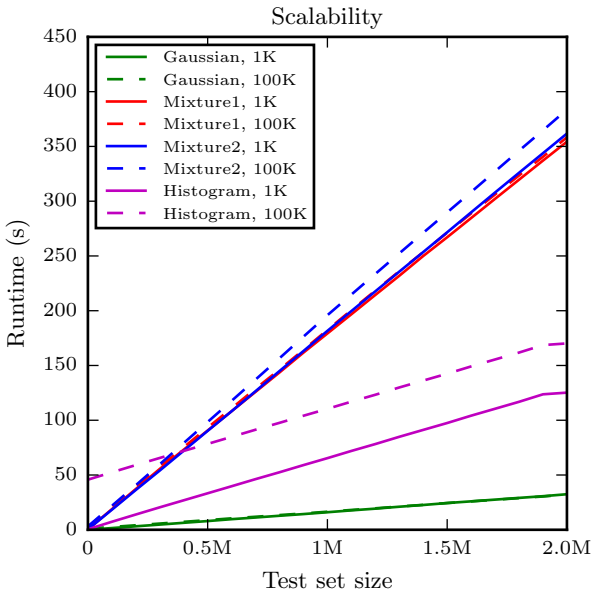


Fig. 6: The scalability of the Gaussian and Mixture Models produced with different training sample sizes (listed next to the model in the legend) as the test set size increases.

Several algorithms exist in the data mining community to determine outliers when doing data analysis. Local outlier factor measures the degree to which a data point is an outlier [5]. Other techniques include k-nearest neighbor [16] and cluster analysis.

Research has been done to attempt to explain why outliers exist given properties of the original data [22]. Unlike our tool, Scorpion starts with user-defined outliers and works backwards to find potential explanations as to why the data points are outliers.

Statistical methods have been used to detect dependencies between columns of relational databases for the purpose of informing the query optimizer of potential data dependencies [11]. These methods require only a small sample of the data to detect functional dependencies with high probability of correctness. The relatively low computation required by these algorithms makes them more amenable to detecting data anomalies in real time. However, these methods are better suited for numerical data [10]

Gaussian Mixture Models have been used for outlier detection in multiple contexts [15], [17], [18].

Histograms are used in conjunction with local outlier factors to detect outliers [8], [20], in cases of numerical or categorical data.

To the extent of our knowledge, the literature regarding outlier detection on non-numerical data is much less extensive. Some common approaches include identifying outliers using a similarity measure [6], Probabilistic Suffix Trees [21] and sequence alignment [4].

Some specialized work has focused on inferring domain-specific rules on highly specific data such as a sequence of UNIX commands [13], [14]. By contrast, we take on a general-purpose approach that is capable of dealing with data as diverse as a set of names and office numbers to real-valued sensor data. Additionally, we analyze data without any additional information on its structure.

Overall, we differ from previous approaches in that we are capable of analyzing a very wide range of data and do not use predefined rules for outlier detection – although adding user-defined rules is possible in our framework.

VII. Conclusion

In this paper we presented *dBoost*, a toolkit that leverages tuple expansion to detect outliers in both numerical and heterogeneous data sets. We demonstrated that well-known machine-learning strategies could be used to flag spurious numerical and to a lesser extent non-numerical data. We also demonstrated that simple correlation modeling is useful in inferring data dependencies and improving the accuracy of outlier detection procedures. We discussed histogram-based models, and showed that they provided a useful tool in analyzing mostly textual data.

We showed that our toolkit performs well on real-world problems, including identifying potentially wrong entries in a people directory and flagging erroneous values generated by faulty sensors. Our toolkit and its source code are available for public use under a permissive license, with the hope of allowing database users to formulate their own type-based rules and find discrepancies in their own data. Once a large library of rules is developed, we anticipate new challenges such as how to select expansion rules for efficient analysis.

FIXME do we really want an appendix? FIXME subplots should not have legends

References

[1] S. Achour and M. Rinard. Energy-efficient approximate computation in topaz. Technical report, MIT CSAIL, 08 2014.

[2] C. C. Aggarwal. *Outlier Analysis*. Springer, 2013.

[3] H. Akaike. A new look at the statistical model identification. In *IEEE Transactions on Automatic Control*, pages 716–723, 1974.

[4] L. Bouarfa and J. Dankelman. Workflow mining and outlier detection from clinical activity logs. *Journal of Biomedical Informatics*, 45(6):1185–1190, 2012.

[5] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. LOF: Identifying Density-based Local Outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 93–104, New York, NY, USA, 2000. ACM.

[6] S. Budalakoti, A. N. Srivastava, R. Akella, and E. Turkov. Anomaly Detection in Large Sets of High-Dimensional Symbol Sequences. 2006.

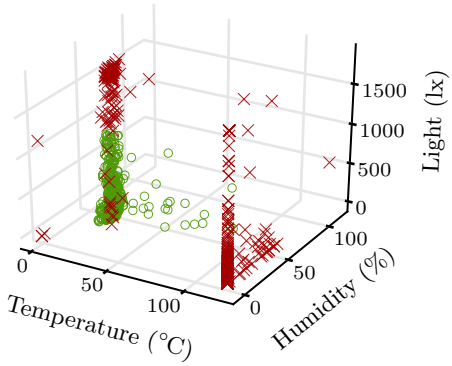
[7] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

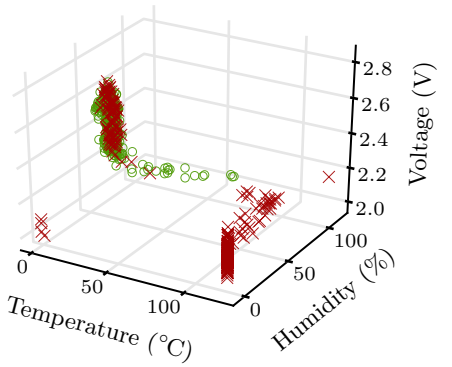
[8] M. Gebski and R. Wong. An efficient histogram method for outlier detection. In R. Kotagiri, P. Krishna, M. Mohania, and E. Nantajeewarawat, editors, *Advances in Databases: Concepts, Systems and Applications*, volume 4443 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin Heidelberg, 2007.

[9] D. Hawkins. *Identification of Outliers*. Springer, 1980.

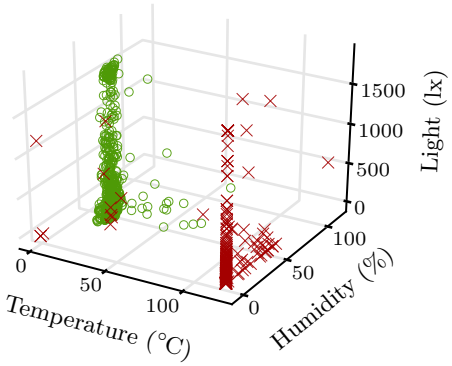
[10] V. Hodge and J. Austin. A Survey of Outlier Detection Methodologies. *Artif. Intell. Rev.*, 22(2):85–126, Oct. 2004.

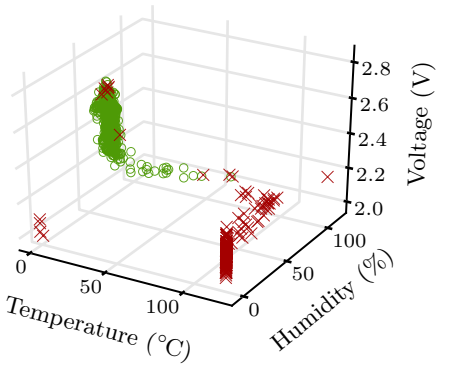
[11] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*, page 647, New York, New York, USA, June 2004. ACM Press.



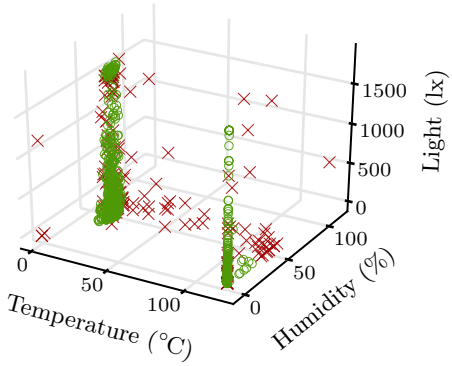


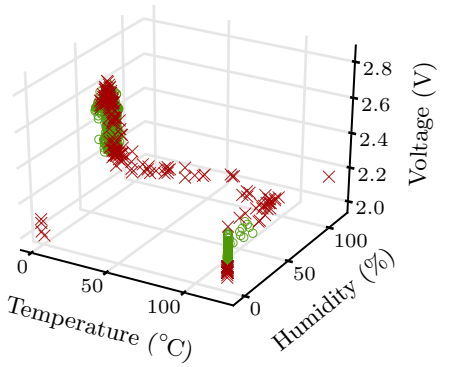
(a) Simple Gaussian, $\theta = 1.5$



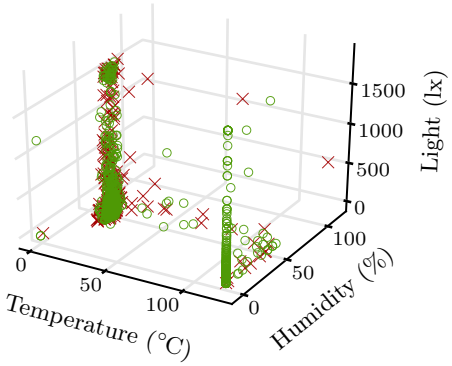


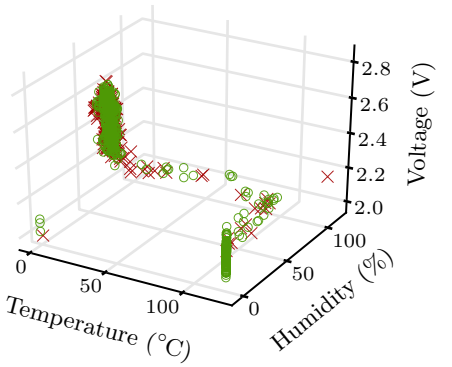
(b) GMM, $n = 1$, $\theta = 0.1$





(c) GMM, $n = 2$, $\theta = 0.05$





(d) Local Outlier Factor, $k = 2$

Fig. 7: A comparison of various modeling approach on the Intel dataset. Red crosses indicate detected outliers. Figure 7a shows the results of simple uni-dimensional Gaussian modeling (Subsection IV-B2). Figure 7b shows what happens when correlations are taken into account, using a multidimensional GMM with a single Gaussian (Subsection IV-B3). Figure 7c shows the results of using two Gaussians instead of one to model the data. Note how the secondary data blobs, detected as outliers in the first two cases, are in the third case identified as valid data. Figure 7d shows the results of the Local Outlier Factor Algorithm on the same dataset.

[12] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek. LoOP. In *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*, page 1649, New York, New York, USA, Nov. 2009. ACM Press.

[13] T. Lane and C. Brodley. An Application of Machine Learning to Anomaly Detection. (COAST TR 97-03), February 1997.

[14] T. Lane and C. E. Brodley. Sequence matching and learning in anomaly detection for computer security. In *Proceedings of AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[15] W. Lu and I. Traoré. A New Unsupervised Anomaly Detection Framework for Detecting Network Attacks in Real-Time. In Y. Desmedt, H. Wang, Y. Mu, and Y. Li, editors, *CANS*, volume 3810 of *Lecture Notes in Computer Science*, pages 96–109. Springer, 2005.

[16] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. *SIGMOD Rec.*, 29(2):427–438, May 2000.

[17] S. Roberts and L. Tarassenko. A Probabilistic Resource Allocating Network for Novelty Detection. *Neural Comput.*, 6(2):270–284, Mar. 1994.

[18] S. J. Roberts. Novelty Detection using Extreme Value Statistics. In *IEE Proceedings on Vision, Image and Signal Processing*, pages 124–129, 1999.

[19] G. Schwartz. Estimating the dimension of a model. In *The Annals of Statistics*, pages 461–464, 1978.

[20] B. Sheng, Q. Li, W. Mao, and W. Jin. Outlier detection in sensor networks. In *Proceedings of the 8th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '07, pages 219–228, New York, NY, USA, 2007. ACM.

[21] P. Sun, S. Chawla, and B. Arunasalam. Mining for outliers in sequential databases. In *ICDM*, pages 94–106, 2006.

[22] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553–564, June 2013.

[23] J. Zhang, M. Lou, T. Wang Ling, and H. Wang. HOS-Miner: A System for Detecting Outlying Subspaces of High-dimensional Data. In *Proceedings of the 30th VLDB Conference*, pages 1265–1268, Toronto, 2004.