CHECKCELL: Data Debugging for Spreadsheets

Daniel W. Barowy Dimitar Gochev Emery D. Berger

School of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{dbarowy,gochev,emery}@cs.umass.edu

Abstract

Testing and static analysis can help root out bugs in programs, but not in data. This paper introduces *data debugging*, an approach that combines program analysis and statistical analysis to *automatically* find potential data errors. Since it is impossible to know a priori whether data are erroneous, data debugging instead locates data that has a disproportionate impact on the computation. Such data is either very important or wrong. Data debugging is especially useful in the context of data-intensive programming environments that intertwine data with programs in the form of queries or formulas.

We present the first data debugging tool, CHECKCELL, an add-in for Microsoft Excel. CHECKCELL identifies cells that have an unusually high impact on the spreadsheet's computations. We show that CHECKCELL is both analytically and empirically fast and effective. We show that it successfully finds injected typographical errors produced by a generative model trained with data entry from 169,112 Mechanical Turk tasks. CHECKCELL is more precise and efficient than standard outlier detection techniques. CHECKCELL also automatically identifies a key flaw in the infamous Reinhart and Rogoff spreadsheet.

1. Introduction

Program correctness has been an important programming language research topic for many years. Techniques to reduce program errors range from testing and runtime assertions to dynamic and static analysis tools that can discover a wide range of bugs. These tools enable programmers to find programming errors and to reduce their impact, improving overall program quality.

Nonetheless, a computation is not likely to be correct if the input data are not correct. The phrase "garbage in, garbage out," long known to programmers, describes the problem of producing incorrect outputs even when the program is known to be correct. Consequently, the automatic detection of incorrect inputs is at least as important as the automatic detection of incorrect programs. Unlike programs, data cannot be easily tested or analyzed for correctness.

Input data errors can arise in a variety of ways [24]:

- **Data entry errors**, including typographical errors and transcription errors from illegible text.
- **Measurement errors**, when the data source itself, such as a disk or a sensor, is faulty or corrupted (unintentionally or not).
- **Data integration errors**, where inconsistencies arise due to the mixing of different data, including unit of measurement mismatches.

By contrast with the proliferation of tools at a programmer's disposal to find programming errors, few tools exist to help find data errors. Traditionally, programmers validate inputs by writing validation routines that mechanically check that inputs match a specification. Precise specifications are difficult to define, but more importantly, this technique fails to capture an entire class of subtle errors: inputs that pass validation but that nonetheless cause unusual program behavior.

Existing automatic approaches to finding data errors include *data cleaning* and *statistical outlier detection*. Data cleaning primarily copes with errors via cross-validation with ground truth data, which may not be present. Statistical outlier detection typically reports data as outliers based on their relationship to a given distribution (e.g., Gaussian). Providing a valid input distribution is at least as difficult as designing a correct validator, but even when the input distibution is known, outlier analysis often is not an appropriate error-finding method. The reason is that it is neither necessary nor sufficient that a data input error be an outlier for it to cause program errors.

Depending on the computation, an input could be an outlier that has no effect (e.g., `MIN()` of a set of inputs containing an erroneously large value), or a non-outlier that affects a computation dramatically (e.g., `IF A1 = 0, "All`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
OOPSLA '14, October 19 - 21 2014, Portland, OR, USA
Copyright 2014 ACM 978-1-4503-2585-1/14/10...$15.00.
http://dx.doi.org/10.1145/2660193.2660207

is Well", "Fire Missiles"). Furthermore, like regular programs, spreadsheets are often a mix of functions that consume and produce both numbers and strings. Traditional outlier analysis is incapable of handling such a wide variety of data types.

Even when the input distribution is known, it is often difficult to automatically decide whether a given input is actually an error. For example, the number 1234 might be correct, or the correct value might be 12.34.

The key insight in this paper is that, with respect to a computation, whether an error is an outlier in the program's *input* distribution is not necessarily relevant. Rather, potential errors can be spotted by their effect on a program's *output* distribution. An important input error causes a program's output to diverge dramatically from that distribution. This statistical approach can be used to rank inputs by the degree to which they drive the anomalousness of the program.

Data Debugging. This paper presents **data debugging**, an automated technique for locating potential data errors. Since it is impossible to know *a priori* whether data are erroneous or not, data debugging does the next best thing: *locating data that have an unusual impact on the computation*. Intuitively, data that have an inordinate impact on the final result are either very important or wrong. By contrast, wrong data whose presence have no particularly unusual effect on the final result do not merit special attention.

Data debugging combines data dependence analysis and statistical analysis to find and rank data errors in proportion to their severity with respect to the result of a computation. Data debugging works by first building a data dependence graph of the computations. It then measures data impact by randomly resampling data items with data chosen from the same group (e.g., a range in a spreadsheet formula) and observing the resulting changes in computations that depend on that data. This nonparametric approach allows data debugging to find errors in both numeric and non-numeric data, without any requirement that data follow any particular statistical distribution.

By calling attention to data with unusual impact, data debugging can provide insights into both the data and the computation and reveal errors.

Spreadsheet Programs. While data errors pose a threat to the correctness of any computation, they are especially problematic in data-intensive programming environments like spreadsheets. In this setting, data correctness can be as important as program correctness. The results produced by the computations—formulas, charts, and other analyses—may be rendered invalid by data errors. These errors can be costly: errors in spreadsheet data have led to losses of millions of dollars [39, 40].

CHECKCELL. We present CHECKCELL, a data debugging tool designed as an add-in for Microsoft Excel and for Google Spreadsheets (Figure 3). Spreadsheets are one of the most

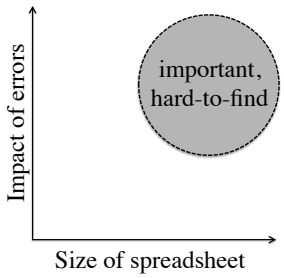


Figure 1. CHECKCELL is designed to find important errors in spreadsheets that would otherwise be too large to audit manually.

	A	B	C	D	E
1	Assignment	Grade		*Homework*	20%
2	HW 1	84		*Quizzes*	30%
3	HW 2	77		*Exams*	50%
4	HW 3	92			
5	HW 4	93		**Final grade**	84.275
6	Quiz 1	87		**Pass/Fail**	Fail
7	Quiz 2	90			
8	Quiz 3	85			
9	Quiz 4	91			
10	Exam 1	84			
11	Exam 2	78			

Figure 2. A typical gradesheet. The formula in E6 is IF(E5 > 85, "Pass", "Fail"). The transposition typo in B11 changes this student's grade from passing to failing. Gaussian outlier analysis fails to detect this error, but CHECKCELL does.

widely-used programming environments, and this domain has recently attracted renewed academic attention [20, 23, 41]. In addition, spreadsheet errors are common, and have led to significant monetary losses in the past, making them an excellent first target for data debugging. CHECKCELL is best suited for large spreadsheets where manual auditing is onerous and error-prone (see Fig. 1).

CHECKCELL highlights all inputs whose presence causes function outputs to be dramatically different than the function output were those outputs excluded. CHECKCELL guides the user through an audit one cell at a time. The order that the audit visits suspected outliers depends on their severity in a total order established by a ranking metric (Section 2).

CHECKCELL is empirically and analytically efficient and effective, as we show in Sections 3 and 4. The current prototype is untuned but analysis time is generally low, taking a median of 2.98 seconds to run on most of the spreadsheets we examine. By employing human workers via Amazon's Mechanical Turk crowdsourcing platform to generate errors, we show that CHECKCELL is effective at finding actual data

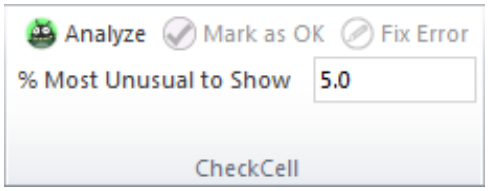


Figure 3. CHECKCELL only requires that a user specify the maximum percentage of spreadsheet inputs to audit. It then guides a user through an audit of highest-ranked error suspects.

entry errors in a random selection of spreadsheets from the EUSES corpus [16]. We also apply CHECKCELL to a real-world spreadsheet, automatically identifying a key flaw in the now-infamous Reinhart-Rogoff spreadsheet [26].

Example Workflow with CHECKCELL

Consider the example spreadsheet in Figure 2, a typical grade sheet for a university course. Grade averages for different curricular activities (homework, quizzes, exams) are weighted according a table and then summed to obtain a final grade. Finally, if the grade crosses a threshold (in this case, `85`), then the student is considered to have passed the course. Otherwise, the student receives a failing grade (here, a "grad fail").

In this example, the error is a transposition of the value in cell `B11` from an `87` to a `78`. Since this grade is an exam, it is weighted more heavily than the grades for homework and quizzes. Note that a two-sided parametric outlier test based on the Gaussian distribution ($\alpha = 0.05$, two standard deviations) does not find this error. This is despite the fact that grades are often normally distributed, and thus the Gaussian distribution should be an appropriate fit. In fact, the error is not even one of the most extreme values, which are actually the values in cells `B3` (`77`) and `B5` (`93`). `78` is not just a valid grade, but in general, a common one. Nonetheless, this error changes this student's final outcome from `Pass` to `Fail`.

CHECKCELL is designed to find precisely this kind of subtle error. First, the user must decide $k\%$, the proportion of input values that they want to inspect ("% Most Unusual to Show"). By default, this value is set to 5%, which is based on our empirical observation that users tend to mistype strings at this rate (See Section 4). After clicking the "Analyze" button, CHECKCELL computes likely errors and ranks them by their hypothesized severity.

Each error is presented to the user one-at-a-time. Upon being presented an error, the user must either mark the cell as correct ("Mark as OK") or fix the error ("Fix Error"). The auditing procedure terminates when either the user has examined at most $k\%$ of the inputs, or when CHECKCELL determines that none of the remaining inputs are likely errors, whichever is smaller. By increasing $k\%$, users may increase accuracy for a greater expenditure in effort. For this example, after a single iteration CHECKCELL finds only this single error, then it terminates.

Contributions

The contributions of this paper are the following:

1. We introduce *data debugging*, an approach aimed at identifying data that has an unusual impact on the final computation, indicating that the data is either extremely important or wrong.
2. We describe novel algorithms to implement data debugging that combine program analysis and nonparametric statistical analysis to identify potential data errors.
3. We present a prototype data debugging tool for spreadsheets, CHECKCELL, and demonstrate its effectiveness at finding errors and identifying highly important data.

Outline

The remainder of this paper is organized as follows. Section 2 describes the algorithms that data debugging employs. Section 3 derives analytical results that demonstrate data debugging's runtime efficiency and effectiveness. Section 4 presents an empirical evaluation of data debugging in the form of CHECKCELL, measuring its runtime performance and its effectiveness at finding errors. Section 5 discusses related work. Section 6 describes directions for future work, and Section 7 concludes.

2. Data Debugging: Algorithms

This section describes data debugging's algorithms in detail. Section 3 includes a formal analysis of its asymptotic performance and statistical effectiveness.

2.1 Dependence Analysis

CHECKCELL's statistical analysis is guided by the structure of the program present in a worksheet. CHECKCELL's first step is to identify the inputs and outputs of those computations. CHECKCELL scans the open Excel workbook and collects all formula strings. Formulas are parsed using an Excel grammar expressed with the FParsec parser combinator library. CHECKCELL uses the Excel formula's syntax tree to extract references to input vectors and other formulas. CHECKCELL resolves references to local, cross-worksheet, and cross-workbook cells.

Spreadsheet programs are always strictly directed acyclic graphs. A formula is a node in a computation tree, and this formula's leaves are input values. Both the root and all the intermediate nodes of the tree are pure functions. Since any cell in a spreadsheet may be used as an input to a formula, formulas may be used as inputs to other formulas. Taken together, these computation trees form a computation forest. The purpose of CHECKCELL is to determine the effect of a particular input on the formulas in the computation forest. CHECKCELL uses techniques similar to past work to identify dependencies in spreadsheets [17].

CHECKCELL's statistical analysis depends on the ability of the analysis to replace input values with other represen-

tative values. When a function has only a scalar argument, namely a single cell or a constant, CHECKCELL does not have enough information to reliably generate other representative values. Therefore, CHECKCELL limits its analysis to vector inputs.

2.2 Impact Analysis

CHECKCELL operates under the premise that the value of a function changes significantly when an erroneous input value is corrected. More precisely, CHECKCELL poses the (null) hypothesis that the removal of a value will *not* cause a large change in function output. CHECKCELL then gathers statistical evidence in an attempt to reject this hypothesis.

Removing an input value requires replacing it with another representative value. Since CHECKCELL never knows the true value of the erroneous input, it must choose from among the only other replacement candidates it can justify, namely other values in the same input vector as the suspected outlier.

Function Classes

CHECKCELL limits its analysis to formula inputs that are justifiably homogeneous, i.e., that input values can be considered as a sample vector drawn from an unknown distribution and that their order does not matter. Our analysis of frequently-used vector functions shows that the most widely-used functions in Excel satisfy this assumption.

CHECKCELL does not directly perturb the inputs to vector functions that do not satisfy the homogeneity requirement. Figure 4 shows the relative frequency of the ten most common vector functions in the EUSES corpus. Of the 5,606 spreadsheets in the EUSES spreadsheet corpus [16], 4,038 contain formulas for a total of 730,765 formulas. Our comprehensive analysis of these spreadsheets showed that vector functions that do not satisfy this assumption, such as HLOOKUP, INDEX, VLOOKUP, and OFFSET, are dominated by homogeneous vector functions, notably SUM. Thus CHECKCELL is useful for a most existing spreadsheets.

Non-Parametric Methods: The Bootstrap

Standard approaches to outlier rejection generally depend on the shape of the distribution. These so-called *parametric* methods require data analysts to parameterize their hypothesis tests with a known parametric form. The normal distribution is most often assumed for outlier rejection. This assumption is justified primarily when a distribution is known to be the result of a summing or averaging of values, since these values will converge in the limit to the normal distribution according to the Central Limit Theorem. Given that CHECKCELL needs to perform statistical tests on *any* function and over unknown data distributions, parametric methods are inappropriate.

Instead, CHECKCELL's input analysis incorporates an adaptation of Efron's bootstrap procedure, a *non-parametric* (distribution-free) statistical method [12]. We use the bootstrap to estimate the distribution of a function output, given only an approximation of the true distribution of inputs (in this case, a sample input vector). This distribution allows one to measure the variability of the test statistic, allowing for reliable inference even when the following conditions hold:

- The sample size is small, *i.e.*, under 30 elements, or
- The distribution is either difficult to compute or is completely unknown.

In particular, CHECKCELL uses an adaptation of Efron's *basic bootstrap* procedure. The procedure works as follows:

1. Draw a random sample, $X_i = (x_0, \ldots, x_{m-1})$, with replacement, from the input vector of interest. This new vector is referred to as a *resample*. Note that m must be the size of the original sample.
2. Compute the function output for sample i, namely $\hat{\theta}_i(X_i)$.
3. Repeat this process n times. In the statistical literature, the number of bootstraps typically is between 1000 and 2500; CHECKCELL uses $n = 1000 \cdot e$, which is approximately 2800 (see Section 3.1).

The resulting distribution $\hat{\theta} = (\hat{\theta}_i, \ldots, \hat{\theta}_{n-1})$ gives an approximation of θ, the true distribution of function outputs. This distribution can now be used for inference, because the bootstrap procedure gives an indication of the variability of θ, i.e., we know which values of θ are unlikely.

Hypothesis test. In order to determine whether an input, x, is likely to be an error, CHECKCELL conditions the output distribution $\hat{\theta}$ on the absence of x in the data. We call this conditional distribution $\hat{\theta}_e$. The conditional distribution approximates the effect of correcting the input error. If the original function output, θ_{orig}, is highly unusual when compared to the $\hat{\theta}_e$, the input x is either a very important input or a likely error. CHECKCELL performs two variants of the hypothesis test, depending on whether the output of the function of interest is numeric or string-valued.

Numeric function outputs. For numeric outputs, the bootstrap distribution is sorted in ascending order, and the quantile function is applied to determine the confidence bound of interest. CHECKCELL uses $\alpha = 0.05$, which is a standard confidence bound in the statistical literature, corresponding to a 95% confidence interval. The original function output is compared with the distribution $\hat{\theta}_e$. If θ_{orig} falls to the left of the 2.5th percentile or to the right of the 97.5th percentile, we reject the null hypothesis and declare x an outlier.

String-valued function outputs. For string-valued function outputs, the bootstrap distribution becomes a multinomial. The multinomial is parameterized by a vector of probabilities, $p_0, \ldots, p_{k-1}$, where k is the number of output categories (in our case, distinct strings), and where $\sum_{i=0}^{k-1} p_i = 1$. CHECKCELL calculates p_i from the observed frequency of category i from $\hat{\theta}_e$. The null hypothesis is then rejected if the probability of observing the original function output, θ_{orig},

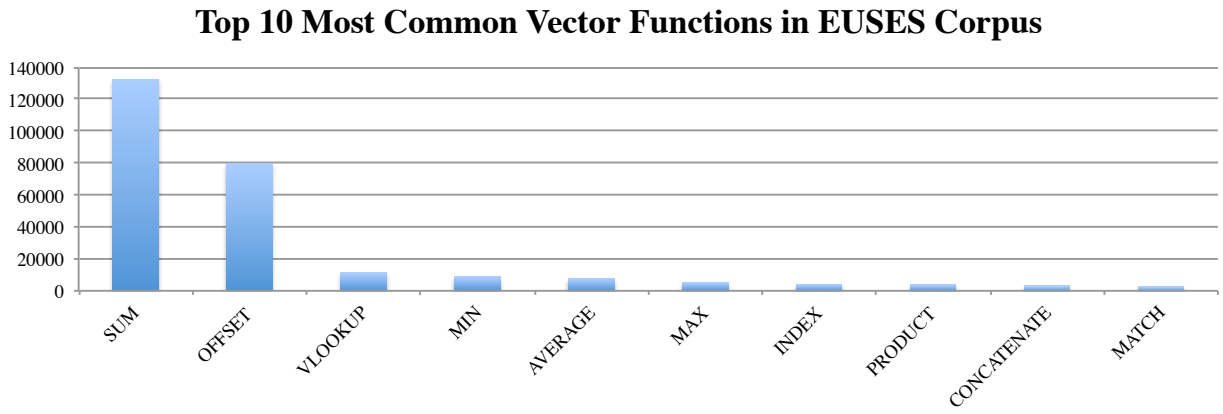


Figure 4. A frequency count of the 10 most common vector functions in the EUSES spreadsheet corpus. The SUM, MIN, AVERAGE, MAX, PRODUCT, MATCH functions are order-independent while OFFSET, VLOOKUP, INDEX, and CONCATENATE are not.

is less than α. The accuracy of the multinomial hypothesis-testing procedure depends on the number of bootstraps, n, since if $n \ll k$ then $\hat{\theta}_e$ is guaranteed to be sparse and incorrect inferences may be drawn. In principle, n can be adjusted such that we are unlikely to observe a $p_i = 0$ when the true value of $p_i = 0 + \epsilon$.

2.3 Impact Scoring

Finally, all inputs that failed at least one hypothesis test are highlighted in red and presented to the user. There are $O(i \cdot f)$ hypothesis tests, one for each (input,output) pair, where i is the total number of inputs in the spreadsheet and f is the number of function outputs. Brightly-colored outputs indicate likely severe outliers while dimly-colored outputs indicate less severe outliers. Input cells that failed no hypothesis test retain their original color (typically black text on a white background). Inputs that do not participate in any computations have no chance of being flagged as potential errors.

CHECKCELL cannot know *a priori* which function outputs are the most important to the end-users. However, inputs that have large effects on large-scale computations are arguably more important to find than inputs that have large effects on small-scale computations. The *total impact* of the error is thus defined as $\sum_{i,f} s_{i,f} \cdot w_f$ where $s_{i,f}$ is the impact score for input i and function f, and where w_f is the weight of function f determined by the size of the computation tree (the number of input leaves) for that function. Weighting helps distinguish between inputs that participate in small computations and those that participate in large ones. The brightness of the highlighting is $(s_{i,f} - s_{min})/(s_{max} - s_{min})$ where 0 is no highlighting and 1 is the brightest highlight.

2.4 Optimizations

CHECKCELL's runtime is $O(i \cdot n)$, or linear in the number of recalculations required (see Section 3), where i is the number of input vectors and n is the number of bootstraps required. Our system uses a configurable default of $n = 1000 \cdot e$ (see Section 3.1).

As n grows larger than m, the length of an input vector, the probability that a given resample will again appear during the bootstrapping procedure increases substantially. CHECKCELL makes use of this fact to save on recalculation cost by caching the output of functions whose input values have been previously calculated.

CHECKCELL calculates a fingerprint for each resample that lets it identify duplicate resamples. Since the inputs to vector functions are order-invariant, CHECKCELL only needs to track the number of appearances of a particular input value in a resample. The fingerprint is a vector of counters, one for each index in the input. CHECKCELL keeps a dictionary of previously-calculated values of $\hat{\theta}_i$, where the key is the aforementioned fingerprint.

For example, given the input vector $(1, 2, 3, 4)$, one possible resample, $X = (x_0, x_1, x_2, x_3)$, is $(1, 4, 4, 3)$. The fingerprint counter would then be $c_0 = 1, c_1 = 0, c_2 = 1, c_3 = 2$. Section 3.2 analyzes the efficiency of this mechanism.

3. Data Debugging: Analysis

This section presents an analysis of data debugging's dominant contributor to the cost of accurate inference: the number of resamples required to perform the bootstrapping method. A mechanism for significantly mitigating this cost is also discussed.

3.1 Number of Resamples

For an input vector of length m and a given value from that vector, x, the probability of randomly selecting a value that is not x is $\frac{m-1}{m}$. The probability of selecting m such values is therefore $(\frac{m-1}{m})^m$. As m grows, we obtain the following identity:

Lemma 3.1. $\lim_{m\to\infty}(\frac{m-1}{m})^m = \frac{1}{e}$

Statistical literature suggests that the number of bootstraps be at least 1000 when the computational cost is tolerable. For efficiency, we perform our bootstrapping procedure once for each input range, and then partition the resulting $\hat{\theta}$ distributions according to the value x of interest. We set $n = 1000 \cdot e$. Lemma 3.1 ensures that, on average, there are 1000 resamples in the bootstrap distribution for $\hat{\theta}_e$.

For i input ranges and a bootstrap size of n, CHECKCELL requires $O(i \cdot n)$ time to analyze a spreadsheet. In practice, the caching feature described in Section 2.4 makes observing even this modest linear cost highly unlikely.

3.2 Efficiency of Caching

For an input vector of length m and a resample X, it must be the case that the sum of the fingerprint counter's values equals m. There are only $f = \binom{2m-1}{m}$ ways to sum to m for a fingerprint vector of length m. There are only f possible fingerprints for an input vector of length m. Input vectors are resampled uniformly randomly, thus the probability of choosing a particular fingerprint is $\frac{1}{f}$. We expect to see a particular fingerprint with a frequency of $\frac{n}{f}$ for a bootstrap of size n. Clearly, for $n > f$, we are likely to observe a repeated fingerprint. As n grows larger than f in the limit, observing a repeated fingerprint is guaranteed.

For example, suppose we have the following vector: ABC. While there are 3^3 possible ways to resample from this vector, a large number of those combinations are not unique when we ignore the ordering of the elements. The complete set of *distinct* order-independent combinations are: AAA, AAB, AAC, ABB, ACC, ABC, BBB, BBC, BCC, CCC. $\binom{2\cdot 3-1}{3} = 10$.

4. Evaluation

We evaluate CHECKCELL across three dimensions: its ability to reduce input errors, its ability to reduce end-user effort in fixing errors, and its execution time. We also use CHECKCELL to examine the now-infamous Reinhart and Rogoff spreadsheet [26, 35, 36].

Our evaluation answers the following questions:

1. Does CHECKCELL identify important data errors?
2. Does using CHECKCELL reduce user effort to identify and correct errors?
3. Is CHECKCELL efficient?

Experimental Methodology

To verify that CHECKCELL is effective at finding important data errors, we run CHECKCELL on a random selection of 61 benchmarks from the EUSES spreadsheet corpus. For each spreadsheet, we randomly select and perturb a single input value with a representative error drawn from an error generator (see Sec 4.1, "Error Generator").

We simulate a user who examines flagged cells as prompted by CHECKCELL. If the simulated user is prompted to inspect a cell that contained a real error, we mark the cell as a true positive and correct the error using the value from the original spreadsheet. If the simulated user is prompted to inspect a cell that did not contain an error, we mark the cell as a false positive.

After CHECKCELL identifies all of the errors at the significance level indicated by the user, all remaining errors are considered to be false negatives. For each error-injected spreadsheet, we compute the *remaining error* and *relative user effort* at the end of the procedure. We repeat this process 100 times for each spreadsheet.

Choice of Baselines. We measure CheckCell's ability to accurately identify errors and the user effort required to find them. Since CHECKCELL is the first fully-automated tool for finding data errors, the baseline for CHECKCELL's effort reduction is the requirement to manually inspect every formula input cell.

To demonstrate CHECKCELL's error-finding performance, we compare CHECKCELL against a variety of alternative error-finding procedures. We report CHECKCELL's performance against the best performing of these methods. We also compare CHECKCELL against a random-flagging procedure to demonstrate that CHECKCELL's results are not simply the result of random chance. We report CHECKCELL's results with its single parameter, *% Most Unusual to Show*, set at 10%. Empirically, this setting provides the best balance of precision and recall. Note that this parameter means that CHECKCELL may report *up to* 10% of the values in the spreadsheet. In practice, this rarely occurs.

Gaussian outlier procedures are fundamentally different from CHECKCELL: Gaussian analysis looks for outliers in the input given a set of inputs, while CHECKCELL looks for outliers in the input given a set of outputs. Furthermore, all Gaussian-based procedures are *parametric*, meaning that they assume data are normally distributed. CHECKCELL is nonparametric, which means that it makes no such assumption about the data's distribution.

Our chosen Gaussian procedure, which we refer to as `NAll`, differs from CHECKCELL in several respects. First, `NAll` flags inputs as likely outliers based on their z-scores, a normalized distance from the mean based on standard deviation. CHECKCELL uses nonparametric tests based on quantiles (for continuous and ordinal data) and histograms (for nominal data). Second, `NAll` considers all of the inputs in the spreadsheet together; all inputs are concatenated into a single input vector. By contrast, CHECKCELL considers inputs one input vector at a time.

Counterintuitively, we found that considering all inputs together boosts the performance of Gaussian methods substantially over those that considered input vectors one at a time. We hypothesize that this change benefits Gaussian procedures because important errors tend to be large in mag-

nitude. By including all inputs, Gaussian procedures can infer more appropriate rejection criteria for the spreadsheet being analyzed.

To keep the comparison straightforward, our evaluation introduces only a single outlier into each spreadsheet (i.e., there is at most one true positive). Furthermore, while input perturbations are drawn from a typo model, we make no effort to ensure that such errors are *important*. This design lets us compare the sensitivity of the two different techniques across two dimensions: (1) the magnitude of the input error, and (2) the magnitude of the output error. Finally, to simplify the comparison, we limited the experiment to input errors for only numerical functions.

It should be noted that limiting the experiment to numerical functions biases the experiment in favor of `NAll`. CHECKCELL's approach is strictly more powerful than Gaussian outlier detection methods since it can work with both numerical and string data. This extra power is needed since Excel is sometimes insensitive to changes in input data type. For example, Excel silently coerces non-numeric inputs into numbers (e.g., `=TRUE+2`). Excel also silently drops nonconforming inputs (`SUM` of a vector of strings and numbers), obscuring the effect of obvious typographical errors.

Since all inputs in this experiment are numerical, measuring the magnitude of an input perturbation is straightfoward. Measuring the magnitude of a spreadsheet's change in outputs is more complicated, as even simple spreadsheets often contain multiple outputs. We use a *total output error* metric to measure the magnitude of an output change relative to other output values in the spreadsheet (see Section 4.1).

After each run, we classify the performance of the two procedures with one of four possible outcomes: (1) `CC10` finds the error, (2) `NAll` finds the error, (3) both procedures find the error, or (4) neither procedure finds the error. Our hypothesis is that `CC10` finds input errors that cause outliers in the output while `NAll` finds errors that cause outliers in the input. More importantly, we hypothesize that CHECKCELL finds a different class of errors, which we term *subtle errors*: small-magnitude input errors that cause large-magnitude output errors. This class of errors is elusive and is therefore most important for automatic tools to be able to find.

Latent Errors. Our benchmarks are drawn from the EUSES spreadsheet corpus, a collection of representative spreadsheets scraped from the Internet. Our experience building an error generator suggests that users make input errors at a rate of roughly 5% per string (see Section 4.1). Thus, it is likely that these spreadsheets already contain errors. Since we do not know whether unusual inputs in unperturbed spreadsheets are correct, we conservatively assume that they are correct. When latent errors are present, our analysis will artifically inflate CHECKCELL's false positive rate (CHECKCELL will have lower precision).

4.1 Error Reduction and User Effort Metrics

To show that CHECKCELL is effective at removing errors, we need a metric that captures the total error of a spreadsheet. To show that CHECKCELL makes users more efficient, we need a metric that captures expended effort. We derive both of these metrics in the following section.

Quantifying User Effort

Without an auditing tool, users must in the worst case inspect all function inputs. An effective tool should reduce the number of inputs a user must manually examine. Let z be the number of cells inspected during the use of the tool ($z \leq m$, the total number of inputs). The *relative effort* of the tool is then defined as effort = z/m.

Quantifying Error

We consider the "correct" (original) spreadsheet to be a vector S of strings. Recall that we assume that the spreadsheet prior to error injection is correct. CHECKCELL may identify latent errors in the EUSES spreadsheets, but because we do not know the ground truth, we conservatively treat such reports as false positives.

We refer to a spreadsheet with errors injected as spreadsheet S_e. Using CHECKCELL leads to a sequence of k corrections, $c_1 \ldots c_k$, rank-ordered by impact. Note that $k \leq n$, the total number of errors injected, since CHECKCELL may not identify all of the errors present.

We apply the corrections in sequence, $c_1 \ldots c_k$, producing a partially-corrected version of the fault-injected spreadsheet S_e, namely the spreadsheet $S_{p,k}$. Spreadsheet $S_{p,0}$ is the spreadsheet with no corrections applied (S_e). Spreadsheet $S_{p,n}$ is the spreadsheet with all n corrections applied (S, when $k = n$).

Because spreadsheets contain both numeric and non-numeric data, we treat them separately and then combine their terms into a total error metric.

Let f be a real-valued function over spreadsheet inputs. Then the *absolute numerical error* of f is:

$$\text{err}_{\mathbb{R}}(f, k) = |f(S_{p,k}) - f(S)|$$

Note that it is possible for a sequence of corrections to temporarily increase the numerical error (i.e., $\text{err}_{\mathbb{R}}(f, k) > \text{err}_{\mathbb{R}}(f, k+1)$), because the effect of multiple errors may combine to reduce total error. Consequently, we normalize numerical errors by the most extreme error observed. Nonetheless, after correcting all n errors, the numerical error is guaranteed to be 0.

The *normalized absolute numerical error* of f is thus:

$$\text{nerr}_{\mathbb{R}}(f, k) = \frac{\text{err}_{\mathbb{R}}(f, k)}{\max\limits_{i \in 0..n} \text{err}_{\mathbb{R}}(f, i)}$$

We treat non-numerical errors (i.e., errors in nominal outputs) by using an indicator function which is 1 if it differs

in value and 0 otherwise. Let g be a *categorical function*. Then the *categorical error* of g is:

$$\text{err}_C(g,k) = \begin{cases} 1 & \text{if } g(S_{p,k}) = g(S) \\ 0 & \text{otherwise.} \end{cases}$$

We then compute the *total error* in a spreadsheet as follows. Let the set of all numeric functions defined in a spreadsheet be F and the set of all categorical functions defined in a spreadsheet be G. Then the total error after k corrections of the spreadsheet is:

$$\text{err}_{tot}(k) = \sum_{f \in F} \text{nerr}_{\mathbb{R}}(f,k) + \sum_{g \in H} \text{err}_C(g,k)$$

$\text{err}_{tot}(k)$ allows us to compute the remaining error over all numeric and string valued functions at step k of the algorithm. Finally, since there may be remaining error at step k, we define the *remaining error* to be:

$$\text{err}_{rem} = \frac{\text{err}_{tot}(n)}{\text{err}_{tot}(0)}$$

This last number expresses the ratio of cells remaining to be fixed. For example, a remaining error of 0.5 means that 50% of the total error remains from the fault-injected spreadsheet. Note that if $k = n$ (we fix all of the errors), then err_{rem} is guaranteed to be 0.

Classifier Accuracy

CHECKCELL's stated purpose is to assist a user in a spreadsheet audit by classifying inputs into one of two categories: errors and non-errors. CHECKCELL cannot distinguish between important errors and important non-errors. Nonetheless, it is informative to examine CHECKCELL's error-finding accuracy using off-the-shelf classifier metrics.

We use *precision* and *recall* for this purpose. Precision and recall are more informative than raw counts for true positives and false positives, because they are normalized by the number of values flagged and by the number of true errors respectively. Nonetheless, since these metrics make use of false positive, false negative, and true negative counts, we explain these counts in terms of CHECKCELL's evaluation. A false positive is when CHECKCELL flags a cell as erroneous that is correct (or an unknown latent error). A false negative is when CHECKCELL fails to flag an injected error as erroneous. A true positive is when CHECKCELL correctly identifies an cell with an injected error.

Precision is defined as:

$$\text{Precision} = \frac{\text{\# true positives}}{\text{\# true positives} + \text{\# false positives}}$$

Recall is defined as:

$$\text{Recall} = \frac{\text{\# true positives}}{\text{\# true positives} + \text{\# false negatives}}$$

Both metrics need to be reported, as they are misleading in isolation. A classifier that produces no false positives will have a precision of 1 ("perfect precision"). Such a classifier may be conservative, missing many true positives. A classifier that flags all errors will have a recall of 1 ("perfect recall'). But perfect recall can be achieved by trivially classifying all inputs as errors. Thus precision and recall are complementary metrics for understanding how often a classification procedure is correct.

As a pessimistic baseline, we also compare CHECKCELL against a procedure that randomly classifies k% of the cells as errors. The expected precision of a random-flagging procedure with one error equals $\frac{1}{\text{\# inputs}}$. The expected recall of that same procedure is k% [43].

Error Generator

In order to inject errors that are representative of the kind of errors that people actually make, we built and trained a classifier by recruiting workers on Amazon's Mechanical Turk to perform data entry tasks. The classifier is designed to spot two kinds of errors: (1) character transpositions and (2) simple typographical errors.

Our input data came from two sources: we randomly sampled formula inputs from 500 spreadsheets in the EUSES corpus (corresponding to 69,112 input strings), and we randomly generated 100,000 additional strings. The additional strings were created to ensure that users were exposed to a wide range of strings, reducing the sparsity in our model. To make it impossible for users to simply cut and paste these strings back into the input field, we rendered strings as images and had 946 workers re-enter the text shown in the image. Workers correctly re-entered 97.14% strings from the first data set and 93.24% from the second data set for a total accuracy of 94.74% (an error rate of 5.26%).

Experimental Results

Distribution of Generated Errors. Of the 6100 error injection experiments, 2836 were numerical only and were thus used for our analysis. The distribution of errors generated and their effects on the output are shown in Figure 5(a). The input error magnitude distribution is reasonably close to a standard Normal distribution (quantiles: 0% = -14.41, 25% = -0.04, 50% =-0.04, 75% = 0.30, 100% = 13.69). The total output error is skewed, as low magnitude errors dominate (quantiles: 0% = 0.00, 25% = 0.03, 50% = 0.08, 75% = 0.25, 100% = 1.00). The vast majority of errors generated were small errors with minimal impact on the computation.

Precision and Recall. Across all benchmark runs, CHECKCELL had a mean precision of 8.0% and a mean recall of 12.1%. `NAll` had a mean precision of 5.9% and a mean recall

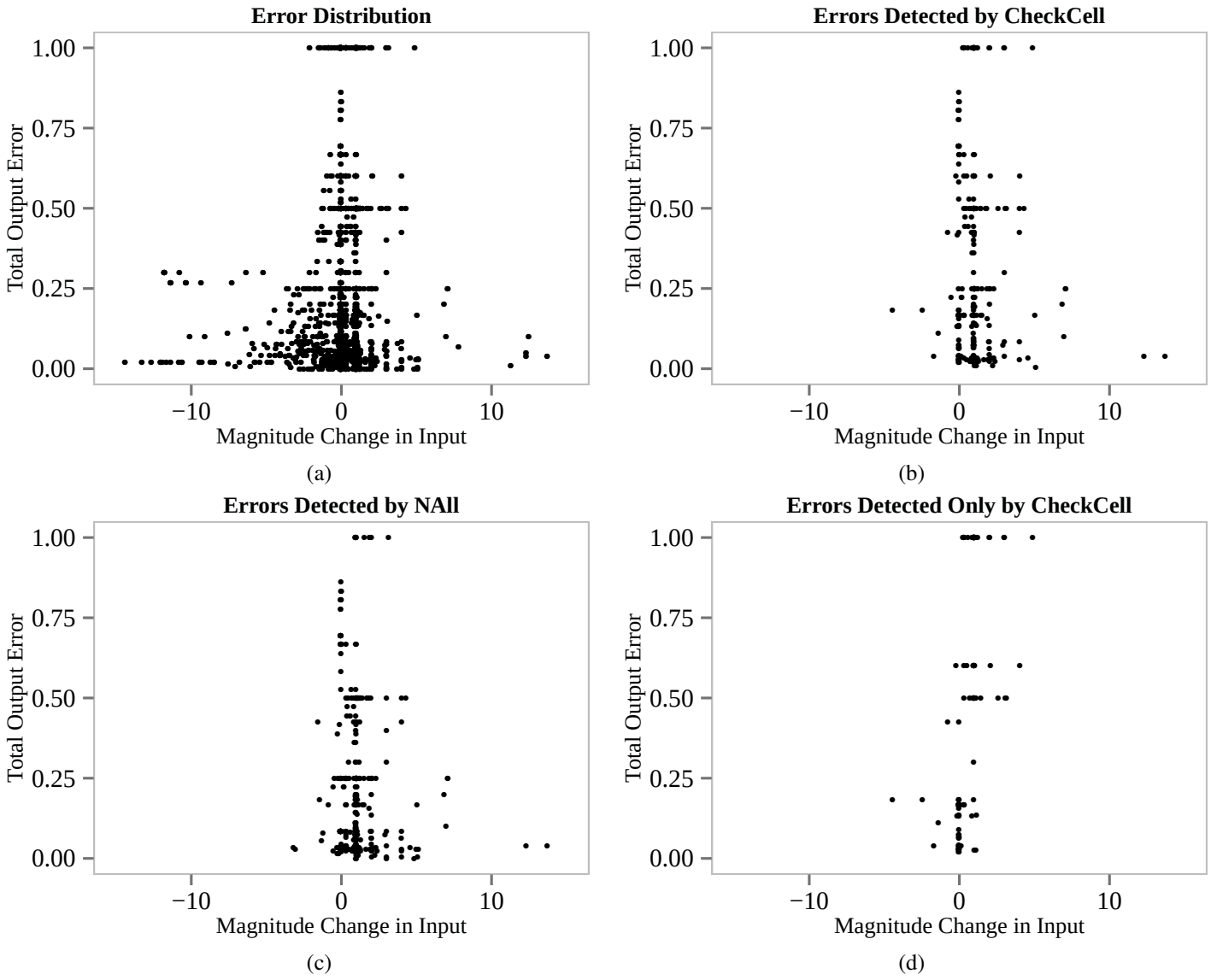


Figure 5. (a) The distribution of input errors. Each point corresponds to a single benchmark run. The change in input magnitude as the result of the error is shown on the x-axis while the change in the spreadsheet's total error is shown on the y-axis. Note that because our typo generator was designed to produce representative errors, they are largely biased toward small-magnitude perturbations. (b) The distribution of errors caught by `CC10`. CHECKCELL favors errors that produce a large effect on the output. (c) The distribution of errors caught by `NAll`. `NAll` favors large input errors in the input. (d) The distribution of errors found by `CC10` but not by `NAll`. These errors tend to be subtle: they have a high impact on the spreadsheet's output and are the result of small magnitude changes in the input.

of 15.8%. A random-answering adversary that expects errors to occur at a rate of 5.26% has a mean expected precision of 3.5% and a mean expected recall of 5.26%. CHECKCELL has higher precision than `NAll`, indicating that it is more discriminating. However, `NAll` has a higher recall, which means it flags more errors than CHECKCELL. Nonetheless, both of these figures are strongly influenced by the presence of a large number of small errors with little impact. The skew is an artifact of our error generator, which does not produce errors uniformly across input and output error magnitudes.

Precision and recall numbers are more informative when we stratify benchmarks by a minimum total output error. Figure 6(a) compares `CC10` and `NAll` mean precision as the minimum total error is increased. Figure 6(b) compares `CC10` and `NAll` mean recall as the minimum total error is increased. `NAll` gains a rapid precision advantage over `CC10` as errors have more of an effect on the computation.

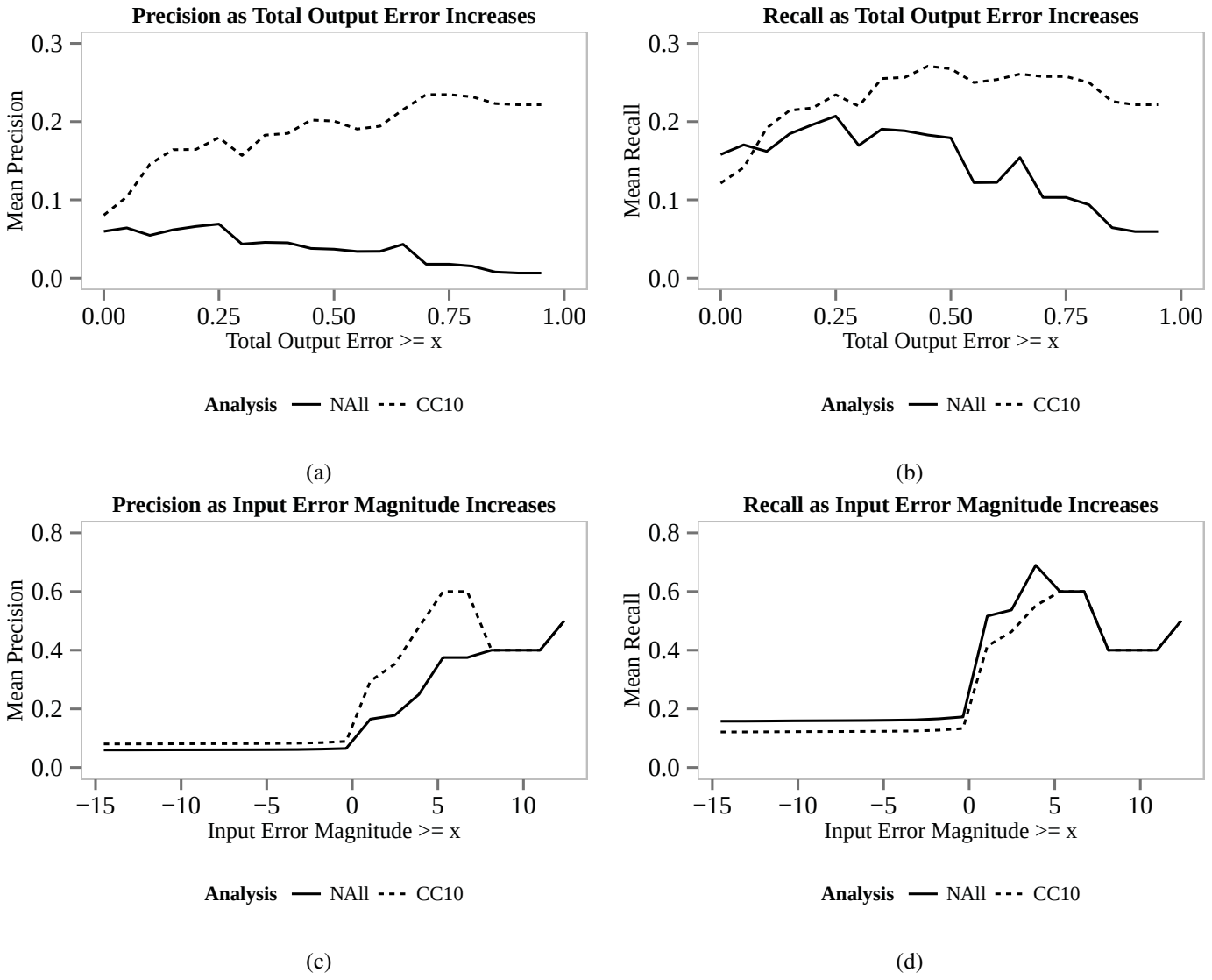


Figure 6. (a) Precision as the minimum total output error is increased. CHECKCELL always has fewer false positives than `NAll`. (b) Recall as the minimum total output error is increased. For errors that cause a small effect, `NAll` returns more false positives, but as errors grow more severe, CHECKCELL returns increasingly relevant errors. (c) Across all input error magnitudes, CHECKCELL is always at least as precise as `NAll`, but usually more. (d) `NAll` is more sensitive to small-magnitude input errors than CHECKCELL.

However, `NAll`'s initial recall advantage over CHECKCELL quickly evaporates as errors grow in importance. When an error is large enough to influence at least a quarter of the total output error, `CC10`'s precision is 17.9% while `NAll`'s is 6.9%. When an error influences at least half of the total error, `CC10`'s precision is 20.0% while `NAll`'s is 3.7%, more than five times higher. These plots mean that as errors grow in importance, CHECKCELL finds them more accurately than `NAll`.

It is also informative to stratify benchmarks by a minimum total input error. Figure 6(c) shows that CHECKCELL is always at least as precise as `NAll`; for a large range of input error magnitudes, it is strictly more precise. Figure 6(d) shows that `NAll` has higher recall (flags more errors) for small magnitude input errors. The difference is not surprising. `NAll` is only sensitive to inputs, while CHECKCELL is only sensitive to outputs. Both procedures are equally precise and sensitive for large magnitude input errors.

Distributions of Detected Errors. `CC10` found 344 errors while `NAll` found 448. 205 errors were found by both procedures. The fact that both `CC10` and `NAll` failed to detect a large number of errors is not surprising given that most errors were inconsequential. The errors missed by both procedures were both small in magnitude (μ =-0.251, median: -0.045,

	Input Error Magnitude			Total Output Error		
	μ	med.	σ	μ	med.	σ
CC10 only	0.59	0.87	1.01	0.49	0.50	0.37
CC10	1.02	0.95	1.49	0.35	0.23	0.32
NAll	0.96	0.95	1.39	0.21	0.08	0.23
NAll only	0.67	0.95	0.98	0.18	0.08	0.23
Undetected	-0.25	-0.04	1.71	0.18	0.07	0.25

Figure 7. Classes of errors detected by CC10 and NAll. *Input error magnitude* is the magnitude of the change in input. *Total output error* is the normalized amount of error in the spreadsheet's output. The errors detected by CHECKCELL (CC10) have a higher total error and are thus more important than the errors detected by NAll. The errors detected *only by* CHECKCELL have a lower input error magnitude, confirming that flagging errors on the basis of input error alone is likely to miss subtle, high-impact errors.

	Df	Sum Sq	Mean Sq	F	Pr(>F)
IEM	1	0.20	0.200	2.600	0.107
AT	1	3.62	3.618	47.00	1.43×10^{-11}
IEM:AT	1	0.00	0.004	0.053	0.817
Residuals	788	60.65	0.077		

Figure 8. Analysis of covariance (ANCOVA) output for the model TOE~IEM*AT. ANCOVA tests whether two populations are significantly different, controlling for confounding variables. TOE is total output error (dependent). AT is the analysis type (independent). IEM is input error magnitude (confound). IEM:AT signifies the interaction term, which should not be significant for valid ANCOVAs. The test shows that on average, CHECKCELL captures 9.8% more output error than NAll for the same IEM (significance: Pr(>F) $= 1.43 \times 10^{-11}$; 9.8% from model coefficients not shown).

$\sigma = 1.712$) and had little effect on the output (μ =0.187, median: 0.074, $\sigma = 0.256$).

CHECKCELL and Gaussian outlier detection find qualitatively different kinds of errors (Figures 5(b) and 5(c)). We expect that the class of errors caught by CHECKCELL will have a large effect on the output, and this is what we observe. CC10 finds errors with a mean total error of 0.350 (median: 0.236, $\sigma = 0.321$) while NAll finds errors with a mean total error of 0.214 (median: 0.083, $\sigma = 0.238$). CC10 favors errors with a high total error and is nearly three times as sensitive as NAll when comparing median total error (NAll is skewed in favor of small total errors).

The effect is even more dramatic when we consider the errors that only CHECKCELL finds: what we call *subtle errors* (see Figure 5(d)). Errors found only by CC10 had a mean total error of 0.491 (median: 0.500, $\sigma = 0.377$) while having only a mean input magnitude change of 0.595 (median: 0.876, $\sigma =$ 1.010).

To put this class of errors in perspective, by flagging a single error that is within 5x of its correct value, CHECKCELL is typically able to eliminate half of the total error of the spreadsheet. The effect of the errors found only by NAll is much smaller by comparison (mean input magnitude change: 0.671, median: 0.952, $\sigma = 0.985$; mean total error: 0.180, median: 0.083, $\sigma = 0.237$).

To show that the difference between the class of errors detected by CHECKCELL and the class of errors detected by CC10 is unlikely to be the result of random chance, we modeled the reduction in total output error (dependent variable; DV) as a function of analysis type (independent variable; IV). We also included input error magnitude as a covariate (a confounding variable; CV) in our model. An analysis of covariance (ANCOVA) rejects the null hypothesis with a p-value of 1.43×10^{-11}, even when accounting for the effect of input error magnitude. Furthermore, there are no significant interactions between the IV and the CV, which means that the test's assumption of the homogeneity of regression slopes is not violated. Thus CHECKCELL reduces total output error by 9.8% more than NAll, an effect that is highly statistically significant.

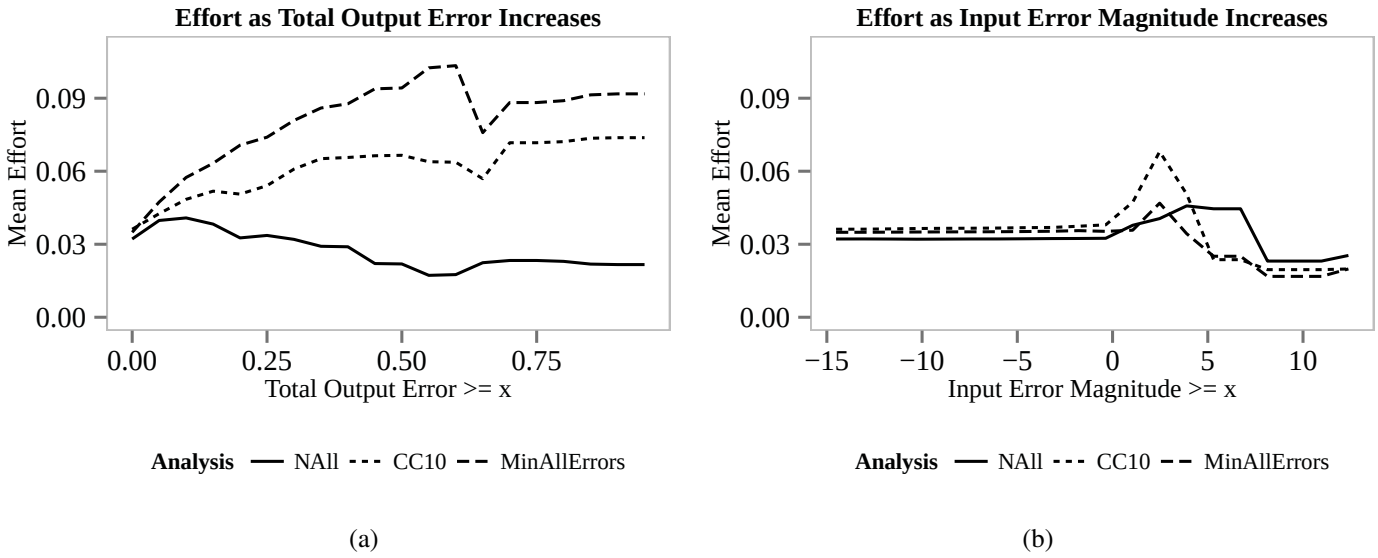
Table 7 summarizes distributions of detected errors, while Table 8 summarizes ANCOVA results.

Effort. CHECKCELL and NAll require comparable effort. Across all benchmarks, CC10 required users to examine 3.6% of a spreadsheet's inputs. NAll required users to examine 3.2% of the inputs, slightly fewer. Again, we analyze both procedures' required effort by stratifying benchmarks by a minimum total error. Figure 9(a) shows that for larger output errors, CHECKCELL typically requires users to inspect between 4% and 7% of the inputs. For the same errors, NAll typically requires users to inspect between 2% and 4% of the inputs. When compared to a user performing a manual audit (100% of the inputs), CHECKCELL saves substantial end-user effort.

NAll's thriftiness comes at a price: it frequently detects nothing at all, saving user effort only by missing important errors. Furthermore, the effort required by CHECKCELL and NAll are the most similar when the majority of the errors are low magnitude, low impact errors. In this region, NAll's recall is slightly higher (Figure 6(b)). Thus, NAll is the most sensitive and requires the greatest user effort for the class of unimportant errors; CHECKCELL behaves in precisely the opposite manner.

Note that both procedures flag fewer than the mean minimum number of inputs required to correctly identify all of the inputs ("MinAllErrors" in Figs. 9(a) and (b)). This means that both tools trade better recall for lower effort. Our evaluation demonstrates that within that tradeoff, CHECKCELL favors errors that cause unusual effects in the output of the program.

Summary. CHECKCELL and Gaussian-based procedures are quite different, and they generally find different sets of errors. CHECKCELL is both more precise than outlier analy-



(a) (b)

Figure 9. (a) For errors that cause a small total error, CHECKCELL requires about the same mean effort as `NAll`. (b) Across input error sizes, mean user effort is roughly similar.

sis, and the errors found by CHECKCELL are more impactful. While CHECKCELL has lower recall than `NAll` across all of our benchmarks, the errors missed by CHECKCELL tend to be inconsequential. For high-impact errors, CHECKCELL clearly dominates `NAll`. Even in a setting where outlier analysis has the greatest possible advantage (numerical functions), CHECKCELL makes better use of a user's limited attention, and focuses user effort on the most important errors. When CHECKCELL's much richer class of non-numeric input and output functions are considered, CHECKCELL is also more useful for finding bugs across a wider range of spreadsheet types.

4.2 Execution Time

Setup. We ran benchmarks on representative end-user hardware: an AMD Phenom X4 running at 8GHz with 8GB of RAM. In all cases, we ran Windows on bare metal, under Windows 8. CHECKCELL was compiled using Microsoft Visual Studio 2012, and runs as an add-in in Microsoft Excel 2010. We also implemented CHECKCELL for Google Spreadsheets, but only report results for the Excel version.

To evaluate CHECKCELL's speed, we measured the time it took to complete its two main tasks, dependence graph construction and outlier analysis, during the experiment run described in the previous section. Performance data was gathered from 100 runs of 61 benchmarks.

Figure 10 reports the performance of data debugging across our spreadsheet suite, ordered by mean total execution time.

Table 1 includes characteristics of these spreadsheets, ordered by the number of formulas each contains. *# Inputs* indicates the total number of inputs to the computation. *Dep. Analysis (μ s)* is the mean time (over 100 runs) to construct the dependence graph. *Outlier Analysis (μ s)* is the mean time (over 100 runs) to run CHECKCELL's outlier inference procedure.

For all but two of the 61 benchmarks, CHECKCELL typically takes 30 seconds or less to complete. Its mean runtime is less than 70 seconds for all spreadsheets. The mean runtime over all spreadsheets is 6.42 seconds, with a median runtime of 2.98 seconds. As our analysis in Section 3 predicts, the time cost of CHECKCELL is largely dominated by the cost of the impact analysis, which is in turn dependent on the number of inputs.

Summary. For nearly every spreadsheet examined, CHECKCELL's runtime is under thirty seconds; we believe this overhead is acceptable for an error detection tool.

4.3 Case Study: The Reinhart and Rogoff Spreadsheet

In 2010, the economists Carmen Reinhart and Kenneth Rogoff, both now at Harvard, presented results of an extensive study of the correlation between indebtedness (debt/GDP) and economic growth (the rate of change of GDP) in 44 countries and over a period of approximately 200 years [35, 36]. The authors argued that there was an apparent "tipping point": when indebtedness crossed 90%, growth rates plummeted. The results of this study were widely used by politicians to justify austerity measures taken to reduce debt loads in countries around the world [26].

Although Reinhart and Rogoff made the original data available that formed the basis of their study, they did not make public the instrument used to perform the actual analysis: an Excel spreadsheet. Herndon, Ash, and Pollin, economists at the University of Massachusetts Amherst, obtained the spreadsheet. They discovered several errors, including the apparently accidental omission of five countries in a range

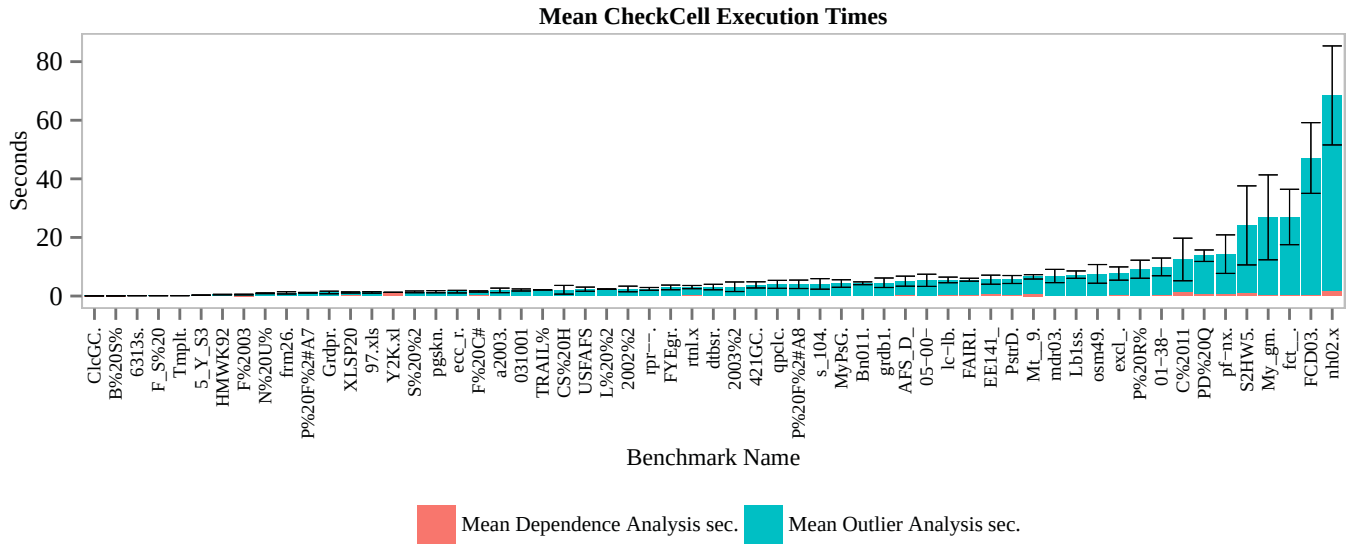


Figure 10. Mean CHECKCELL execution times. For most of the spreadsheets, CHECKCELL completes its analysis in under 30 seconds; for all but two, it completes in under 70 seconds (see Section 4.2). Error bars represent one standard deviation.

of formulas [26]. After correcting for these and other flaws in the spreadsheet, the results invalidate Reinhart-Rogoff's conclusion: no tipping point exists for economic growth as debt levels rise.

While some of the errors in the Reinhart-Rogoff spreadsheet are out of scope for CHECKCELL, we wanted to know whether CHECKCELL would be able to verify any of the other errors or discover new ones. We obtained the Excel spreadsheet directly from Carmen Reinhart and ran CHECKCELL on it. CHECKCELL singled out one cell in bright red, identifying it as a value with an extraordinary impact on the final result. We reported this finding to one of the UMass economists (Michael Ash). He confirmed that this value, a data entry of `10.2` for Norway, indicated a key methodological problem in the spreadsheet. The UMass economists found this flaw by careful manual auditing after their initial analysis of the spreadsheet (emphasis ours) [5]:

> For example, Norway spent only one year (1946) in the 60-90 percent public debt/GDP category over the total 130 years (1880-2009) that Norway appears in the data. Norway's economic growth in this one year was 10.2 percent. **This one extraordinary growth experience contributes fully 5.3 percent (1/19) of the weight for the mean GDP growth in this category even though it constitutes only 0.2 percent (1/445) of the country-years in this category.** Indeed Norway's one year in the 60-90 percent GDP category receives equal weight to, for example, Canada's 23 years in the category, Austria's 35, Italy's 39, and Spain's 47.

This case study demonstrates data debugging's utility not only for detecting errors but also for understanding structural flaws in computations.

5. Related Work

Data Cleaning

Most past work on locating or removing errors in data has focused on *data cleaning* or *scrubbing* in database systems [22, 32]. Standard approaches include statistical outlier analysis for removing noisy data [42], interpolation to fill in missing data (e.g., with averages), and using cross-correlation with other data sources to correct or locate errors [25].

A number of approaches have been developed that allow data cleaning to be expressed programmatically or applied interactively. Programmatic approaches include AJAX, which expresses a data cleaning program as a DAG of transformations from input to output [18]. Data Auditor applies rules and target relations entered by a programmer [19]. A similar domain-specific approach has been employed for data streams to smooth data temporally and isolate it spatially [29]. Potter's Wheel, by Raman and Hellerstein, is an interactive tool that lets users visualize and apply data cleansing transformations [33].

To identify errors, Luebbers et al. describe an interactive data mining approach based on machine learning that builds decision trees from databases. It derives logical rules (e.g., "BRV $=$ 404 $\Rightarrow$ GBM $=$ 901") that hold for most of the database, and marks deviations as errors to be examined by a data quality engineer [31]. Raz et al. describe an approach aimed at arbitrary software that uses Daikon [13] to infer invariants about numerical input data and then report discrepancies as "semantic anomalies" [34]. Data debugging is orthogonal to these approaches: rather than searching for latent relationships in or across data, it measures the interaction of data with the programs that operate on them.

Spreadsheet Errors

Spreadsheets have been one of the most prominent computer applications since their creation in 1979. The most widely used spreadsheet application today is Microsoft Excel. Excel includes rudimentary error detection including errors in formula entry like division by zero, a reference to a non-existient formula or cell, invalid numerical arguments, or accidental mixing of text and numbers. Excel also checks for inconsistency with adjacent formulas and other structural errors, which it highlights with a "squiggly" underline. In addition, Excel provides a formula auditor, which lets users view dependencies flowing into and out of particular formulas.

Past work on detecting errors in spreadsheets has focused on inferring units and relationships (has-a, is-a) from information like structural clues and column headers, and then checking for inconsistencies [1, 3, 9, 14, 15, 30]. For example, XeLda checks if formulas process values with incorrect units or if derived units clash with unit annotations. There also has been considerable work on testing tools for spreadsheets [8, 17, 27, 30, 37, 38].

This work is complementary and orthogonal to CHECKCELL, which works with standard, unannotated spreadsheets and focuses on unusual interactions of data with formulas.

Statistical Outlier Analysis

Techniques to locate outliers date to the earliest days of statistics, when they were developed to make nautical measurements more robust. Widely-used approaches include Chauvenet's criterion, Peirce's criterion, and Grubb's test for outliers [7]. All of these techniques are *parametric*: they require that the data belong to a known distribution, generally the Gaussian (normal). Unfortunately, input data does not necessarily fit a predefined statistical distribution. Moreover, identifying outliers leads to false positives when they do not materially contribute to the result of a computation (i.e., have no impact). By contrast, data debugging only reports data items with a substantial impact on a computation.

Sensitivity Analysis and Uncertainty Quantification

Sensitivity analysis is a method used to determine how varying an input affects a model's range of outputs. Most sensitivity analyses are analytic techniques; however, the one-factor-at-a-time technique, which systematically explores the effect of a single parameter on a system of equations, is similar to data debugging in that it seeks to numerically approximate the effect of an input on an output. Recent research employing techniques from sensitivity analysis in static program analyses seeks to determine whether programs contain "discontinuities" that may indicate a lack of program robustness [2, 10, 21].

Uncertainty quantification draws a relationship between the uncertainty of an input parameter and the uncertainty in the output. Unlike sensitivity analysis, which in the case of OAT can be used as a "black-box" technique, uncertainty quantification requires the analyst to know the functional composition of the model being analyzed.

Data debugging differs from sensitivity analysis and uncertainty quantification in several important respects. First, data debugging is a fully-automated black-box technique that requires no knowledge of a program's structure. Second, unlike sensitivity analysis, data debugging does not vary a parameter through a known range of valid values, which must be parameterized by an analyst. Instead, data debugging infers an empirical input distribution via a nonparametric statistical approach. Finally, the uncertainty of inputs and outputs is irrelevant to CHECKCELL's analysis. CHECKCELL instead seeks to find specific data elements that have an extraordinary effect on program outputs. In essence, sensitivity analysis and uncertainty quantification are aimed at analyzing the model, while data debugging is a technique for analyzing the data itself.

6. Future Work

In future work, we plan to explore applying data debugging to other data-intensive domains, including Hadoop/MapReduce tasks [4, 11], scientific computing environments like R [28], and database management systems, especially those with support for "what-if" queries [6].

We expect all of these domains will require some tailoring of the existing algorithms to their particular context. For databases, we plan to treat as computations both stored procedures and cached queries. While it is straightforward to apply data debugging to databases when queries have no side effects, handling queries that do modify the database will take some care in order to avoid an excessive performance penalty due to copying.

A similar performance concern arises with Hadoop, where the key computation is the relatively costly reduction step. Data debugging will also likely need to take into account features of the R language in order to work effectively in that context. Finally, we are interested in exploring the effectiveness of data debugging in conventional programming language settings.

While CHECKCELL's speed is reasonable in most cases, we are interested in further optimizing it. We are especially interested in developing a version that incrementally updates its impacts on-the-fly. This version would run in the background and detect data with unusual impacts as they are entered, much like modern text entry underlines misspelled words. We believe that having automatic detection of possible data errors on all the time could greatly reduce the risk of data errors.

7. Conclusion

This paper presents data debugging, an approach aimed at finding potential data errors by locating and ranking data items based on their overall impact on a computation. Intuitively, errors that have no impact do not pose a problem,

while values that have an unusual impact on the overall computation are either very important or incorrect.

We present the first data debugging tool, CHECKCELL, which operates on spreadsheets. We evaluate CHECKCELL's performance analytically and empirically, showing that it is reasonably efficient and effective at helping to find data errors. CHECKCELL is available for download at `https://checkcell.org`.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1349784. This work was also supported by a Microsoft Research Software Engineering Innovation Foundation (SEIF) Award. Thanks to Alexandru Toader, our Google Summer of Code student who ported CHECKCELL to Google Spreadsheets. We thank Ben Zorn for the stimulating conversations that led to this work, and also Charlie Curtsinger for the many valuable discussions during the evolution of this project.

References

[1] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A type system for statically detecting spreadsheet errors. In *ASE*, pages 174–183. IEEE Computer Society, 2003.

[2] Y. Ait-Ameur, G. Bel, F. Boniol, S. Pairault, and V. Wiels. Robustness analysis of avionics embedded systems. *SIGPLAN Not.*, 38(7):123–132, June 2003.

[3] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Apache Foundation. Welcome to Apache Hadoop. `http://hadoop.apache.org/`, Nov. 2012.

[5] M. Ash and R. Pollin. Supplemental Technical Critique of Reinhart and Rogoff, "Growth in a Time of Debt". Research brief, Political Economy Research Institute, University of Massachusetts Amherst, Apr. 2013.

[6] A. Balmin, T. Papadimitriou, and Y. Papakonstantinou. Hypothetical queries in an OLAP environment. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 220–231, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[7] V. Barnett and T. Lewis. Outliers in statistical data. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, Chichester: Wiley, 1994, 3rd ed.*, 1, 1994.

[8] J. Carver, M. Fisher, II, and G. Rothermel. An empirical evaluation of a testing and debugging methodology for excel. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 278–287, New York, NY, USA, 2006. ACM.

[9] C. Chambers and M. Erwig. Reasoning about spreadsheets with labels and dimensions. *J. Vis. Lang. Comput.*, 21(5):249–262, Dec. 2010.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, Mar. 1997.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] B. Efron. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1):pp. 1–26, 1979.

[13] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[14] M. Erwig. Software engineering for spreadsheets. *IEEE Softw.*, 26(5):25–30, Sept. 2009.

[15] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *ICSE*, ICSE '05, pages 136–145, New York, NY, USA, 2005. ACM.

[16] M. Fisher and G. Rothermel. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw. Eng. Notes*, July 2005.

[17] M. Fisher, G. Rothermel, T. Creelan, and M. Burnett. Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 13–22. IEEE, 2006.

[18] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: an extensible data cleaning tool. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, page 590, New York, NY, USA, 2000. ACM.

[19] L. Golab, H. Karloff, F. Korn, and D. Srivastava. Data auditor: exploring data quality and semantics using pattern tableaux. *Proc. VLDB Endow.*, 3(1-2):1641–1644, Sept. 2010.

[20] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *POPL*, pages 317–330. ACM, 2011.

[21] D. Hamlet. Continuity in software systems. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 196–200, New York, NY, USA, 2002. ACM.

[22] J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.

[23] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 317–328. ACM, 2011.

[24] J. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[25] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 127–138, New York, NY, USA, 1995. ACM.

[26] T. Herndon, M. Ash, and R. Pollin. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. Working Paper Series 322, Political Economy

Research Institute, University of Massachusetts Amherst, Apr. 2013.

[27] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner. On the empirical evaluation of fault localization techniques for spreadsheets. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 68–82, Berlin, Heidelberg, 2013. Springer-Verlag.

[28] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[29] S. Jeffery, G. Alonso, M. Franklin, W. Hong, and J. Widom. A pipelined framework for online cleaning of sensor data streams. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 140–142, Apr. 2006.

[30] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, Apr. 2011.

[31] D. Luebbers, U. Grimmer, and M. Jarke. Systematic development of data mining-based data quality tools. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 548–559. VLDB Endowment, 2003.

[32] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[33] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 381–390, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[34] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *ICSE*, ICSE '02, pages 302–312, New York, NY, USA, 2002. ACM.

[35] C. M. Reinhart and K. S. Rogoff. Growth in a time of debt. Working Paper 15639, National Bureau of Economic Research, January 2010.

[36] C. M. Reinhart and K. S. Rogoff. Growth in a time of debt. *The American Economic Review*, 100(2):573–78, 2010.

[37] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):110–147, 2001.

[38] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *ICSE 1998*, pages 198–207. IEEE, 1998.

[39] M. Sakal and L. Raković. Errors in building and using electronic tables: Financial consequences and minimisation techiques. *International Journal of Strategic Management and Decision Support Systems in Strategic Management*, 17(3):29–35, 2012.

[40] V. Samar and S. Patni. Controlling the information flow in spreadsheets. *CoRR*, abs/0803.2527, 2008.

[41] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, Apr. 2012.

[42] H. Xiong, G. Pandey, M. Steinbach, and V. Kumar. Enhancing data analysis with noise removal. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):304–319, Mar. 2006.

[43] P. Zhang and W. Su. Statistical inference on recall, precision and average precision under random selection. In *FSKD*, pages 1348–1352. IEEE, 2012.

Benchmark Name	# Inputs	Dep. Analysis (μ s)	Outlier Analysis (μ s)
01-38-PK_tables-figures.xls	170	0.5549288	9.390480198
031001.xls	45	0.0842667	1.983900989
05-00-046.xls	232	0.2372456	5.121365639
2002%20Project%20Reports.xls	85	0.0864263	2.328297928
2003%20Applications%2#A92C1.xls	80	0.1394455	3.033752753
421GradeCalc.xls	20	0.0568004	3.748034307
5_Year_Summary3.xls	17	0.0680315	0.239336643
6313syllabus.xls	5	0.0602589	0.070019999
97.xls	44	0.0763964	1.176864265
AFS_Dec_2002.xls	65	0.4726516	4.594466757
as2003puna.xls	32	0.038661	1.887443405
Bnbjan011.xls	217	0.2617529	4.097381562
Business%20Scenarios%#A88E6.xls	4	0.0190942	0.041386032
CalcGradeCalculator.xls	5	0.0195551	0.03929663
Chemistry%20114%20lab%20web.xls	52	1.4207694	11.05709885
ChiSquare%20Homework.xls	21	0.1257881	2.002615437
databaser.xls	302	0.1874444	2.883269715
ecc_rev.xls	44	0.043359	1.450572499
EE141_s03_grades_clas#A7BDC.xls	69	0.6682386	4.900768485
excel_template.xls	361	0.3944765	7.280832274
factiva_rev_sum.xls	340	0.3629518	26.61497105
FAIRInventory.xls	576	0.5294704	5.030910367
Fall%2003%20grades.xls	18	0.0220087	0.485243888
Fin_St%20June%20for%2#A7FFC.xls	9	0.0948804	0.037443947
Financial%20Compariso#A7ED8.xls	67	0.2627775	1.291007253
FinancialCompilationDec03.xls	348	0.6122993	46.49485957
form26.xls	104	0.1452416	0.94196296
FYEgrades.xls	16	0.07125	2.872127556
gradebook1.xls	112	0.0991529	4.433837156
Gradeprediction.xls	32	0.0388809	1.179147718
HMWK92903.xls	35	0.092238	0.40308102
Lab1assignmentsolutions.xls	71	0.1821385	7.118771288
lc-labrepevalsht.xls	343	0.399497	5.099844378
Listeria%20cross%20co#A7D8F.xls	36	0.2208161	2.150088599
mdr03demo.xls	134	0.231352	6.602564816
Metrics_version_9.xls	101	0.6916134	5.861234578
My_gam.xls	204	0.4057672	26.45568679
MyPsycGrade.xls	43	0.1271021	4.140405027
nih02.xls	370	1.7016567	66.76504172
Nsfcam%20Upgrade%20Es#A7E18.xls	24	0.0355804	0.915177933
osm49.xls	80	0.1032796	7.451600866
PD%20Q4-02.xls	180	0.7858247	12.97192893
pfi-anxa.xls	310	0.9821158	13.3156672
pigskin.xls	38	0.1125059	1.359677528
PosterData.xls	88	0.2789295	5.358120438
Progress%20Report%20a#A8403.xls	98	0.2588778	8.886032269
Project%20Financial%2#A7CEE.xls	8	0.0431702	1.065178219
Project%20Financial%2#A8AD9.xls	46	0.0596221	3.941388616
qpacalculator.xls	35	0.1361675	3.848016248
ratioanal.xls	120	0.4037585	2.65019949
report-financial-part.xls	59	0.1337376	2.296524129
sample_ais104.xls	128	0.1594051	3.974220706
Solution2HW5.xls	340	0.9126632	23.20265487
Summ%20of%20Physical%#A899C.xls	36	0.1387315	1.288594914
Template.xls	13	0.0521189	0.090706512
TRAIL%20INVENTORY%20N#A850A.xls	156	0.1782064	1.932176567
USFAthleticFinancialSummary.xls	45	0.0909246	2.243268007
XLSolverPractice2001.xls	90	0.2117995	1.011882867
Y2K.xls	6	1.1960657	0.059038997

Table 1. Performance statistics for a randomly-selected benchmark suite.