RI13002, 2 January 2013 Computer Science

IBM Research Report

Combining Machine Learning and Combinatorial Search in Program Repair

Divya Gopinath
University of Texas at Austin

Sarfraz Khurshid
University of Texas at Austin

Diptikalyan Saha
IBM Research - India

Satish Chandra
IBM Research - India

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com).. Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Abstract

We consider the problem of automatically generating repair suggestions for a defective database program that behaves incorrectly due to an error in the WHERE condition of a SELECT statement.

A common setting in database programs is that the output is incorrect only for part of the data, e.g., for certain key values. In this paper, we use techniques from machine learning to take advantage of the information revealed by the defect-free data. Our basic approach is to learn a decision tree from *correct* behavior—including correct behavior on the defect-inducing data—of the SELECT statement. This decision tree can give valuable hints, if not directly the correct WHERE condition.

Our novelty is in the crucial step of determining the correct behavior of the defect-inducing data. We do this using a combination of SAT-based search and prediction generated by support vector machines (SVMs). Our insight is that SVMs can learn from the behavior of the defect-free data to predict the behavior of defect-inducing data with high accuracy, with SAT-based search bridging over any deficit in the accuracy efficiently.

We implemented this approach and present experimental results using a suite of programs and data sets obtained from real-world applications.

1 Introduction

A majority of enterprise software systems are database-centric programs. Defects in such programs, specifically in database manipulating statements, are expensive to fix and can require much human effort in understanding the interplay between traditional imperative code and database-centric logic. Automated tools to help diagnose these defects, and furthermore, to assist with fixing them can make a substantial reduction in the cost of developing and maintaining database-centric programs.

1.1 Problem Context and Examples

Our specific focus is on SAP ERP systems, in which database-centric programming is carried out in a proprietary language called ABAP. ABAP contains SQL-like commands, but it mixes imperative code and SQL's declarative syntax. We introduce the essential constructs of ABAP that are relevant for this paper using a small example (Figure 1).

The meaning of this ABAP code segment is straightforward. At line 1, it reads all rows from a database called `OrderTab` into an internal table called `itab`. The `SORT` statement sorts this internal table by `CstId`, which is the key. The `DEL` statement at line 4 removes from `itab` those rows that match the condition described in the statement. The `LOOP` at line 5 iterates over `itab`. When it encounters a new `CstId`—that is when `AT NEW` at line 6 is true— it resets an accumulator called `amount`, and it prints the accumulated amount when the last of that `CstId` has been visited; this is done when `AT END` on line 10 is true. (`AT NEW` and `AT END` help with key-wise aggregation akin to the SQL GROUP-BY construct.)

```
1   SELECT CstId Price Year from OrderTab INTO itab
2
3   SORT itab by CstId
4   DEL from itab where Year <= 2009 and Price > 5
5   LOOP AT itab INTO wa
6     AT NEW CstId
7       amount=0
8     ENDAT
9     amount = amount + wa.Price
10    AT END CstId
11      WRITE wa.CstId amount
12    ENDAT
13  ENDLOOP
```

Figure 1: A sample ABAP code segment.

CstID	Price	Year	
1	20	2012	+
1	16	2011	+
1	12	2001	-
1	10	2002	-
1	15	2011	+
2	7	2005	**-**
2	13	2007	-
2	15	2010	+
2	10	2011	+
3	4	2012	+
3	3	2009	+
3	9	2001	-

Continued on right . . .

Table 1: A sample input to program in Figure 1. The last column with '+'/'-' is not a part of the input table.

Suppose the program in Figure 1 is run on the database in Table 1. The rows marked '+' are retained in `itab` after the `DEL` statement. The output of the program is, which unfortunately differs from the expect output, also shown:

ID	Amount	Expected Amount
1	51	51
2	25	**32**
3	7	7

The bug arises from an error in the condition of the `DEL` statement, which causes one row for `CustId` 2 to be incorrectly deleted (shown by a bold '**-**').

We can think of the `DEL` statement as a (equivalent) SELECT statement: `SELECT * FROM itab WHERE Year > 2009 OR Price <= 5`. We call such a defect a *selection* bug, because the bug is due to an incorrect WHERE condition in a SELECT statement. The problem is to find an alternate WHERE condition for the faulty SELECT statement (whose location is assumed to be known), so that the entire output, corresponding to each of the keys, is correct.

Selection bugs are commonplace in ABAP programs. In fact, many database statements in ABAP program allow a selection condition, and therefore, are vulnerable to a selection bug. For example, the ABAP `READ` statement matches rows based on a condition, but returns only the first matched row (if any). Based on our experience working with practitioners in IBM Global Business Services, about 25% of the ABAP code level defects have to do with selection. Therefore, techniques that can help in fixing defective selection statements are of much value.

Note that in our setting of debugging ABAP programs, the process starts with the *end user* of this software filing a bug report, citing a deviation of actual output from the expected output on given input data. Thus, the expected output of the program is already known to the programmer (or the maintainer). As we shall see, the challenge here is in determining the correct behavior of the defective SELECT statement from the expected output of the entire program, and in determining an alternate WHERE condition for the selection that would match the correct behavior.

1.2 Repair Based on Machine Learning

We assume, as is true in this example, that each row of the output depends on only one key. We define *passing keys* as those keys for which the corresponding output is correct, and *failing keys* for which the output is incorrect. In the example above, `CstId` 1 and 3 are passing keys, and `CstId` 2 is a failing key. We define *passing rows* as rows that correspond to a passing key, and *failing rows* as rows that correspond to a failing key. Note that a passing row does not mean the row passed the WHERE condition and appears in the immediate output of the SELECT; rather, it means the decision taken by the WHERE condition is correct, whether it decided to select or not select the row. The WHERE statement is assumed to take correct decisions for all passing rows. In rare

cases the final output may be coincidentally correct despite an incorrect decision by the WHERE condition.

Our motivation for exploring machine learning based approach comes from the observation that the defective WHERE condition does work correctly for the passing keys.

In machine learning, a common setting—known as *supervised learning*—is as follows. We are given labeled training data, which means each element of a set of tuples is assigned a positive or a negative label, based on some unknown function. The learner is supposed to discover a function—a *classifier*—that produces the same labels on the training data, and moreover, can correctly assign labels to test data given to it in the future. In Section 2, we provide background on some of the pertinent machine learning constructs that we refer to in this paper.

Coming back to ABAP programs, whether or not the SELECT statement selected a row in the table can be interpreted as assignment of a positive ('+') or a negative label ('-'). Table 1 showed such labels for the SELECT statement in the program shown in Figure 1. The rows corresponding to the passing keys, along with their labels, is our training data.

Since our focus is on repairing the WHERE condition, the *decision tree* learning [20] method from machine learning method is applicable. Specifically, the classifier learned from the labeled samples is a decision tree, in which each node contains a predicate on some attribute, and leaf nodes contain positive or negative signs. An as-yet-unlabeled row can be assigned a label based on the decision tree.

In our setting, a WHERE condition—a formula corresponding to the decision tree—learned solely on the basis of labeled training data is unlikely to assign correct labels to the test data, i.e. the rows corresponding to the failing keys. In our example, a learner could learn the condition $Year > 2008$ from the training data, but that condition is clearly incorrect for the test data. In fact, the learner would be wholly justified in producing the *same* defective condition, because that condition is the one that produced the labels for the passing rows! Intuitively, the defective WHERE condition worked correctly on the passing rows because they did not exhibit a certain *corner case*, which the failing rows did. Therefore, a necessary step in our approach is to *figure out the correct labels for the failing rows* as well, before we can apply decision tree learning.

1.3 Overview of Our Approach

Our approach divides the problem into two phases. The first phase determines a correct labeling for the failing rows. The second phase uses the complete and correct labeling of both passing and failing rows (computed by the first phase) to determine the correct WHERE clause. See Figure 2.

1.3.1 Basic Approach

We first explain the basic approach, as shown in Figure 2(a), in which the correct label assignment (Phase 1) is computed using symbolic execution, which amounts to a combinatorial search for the missing labels.

Phase 1. For each failing key:
(a) Assign a symbolic label (a boolean) to each row of the failing key to indicate whether it is to be retained or deleted.
(b) Execute the rest of the program symbolically up to the output statement.

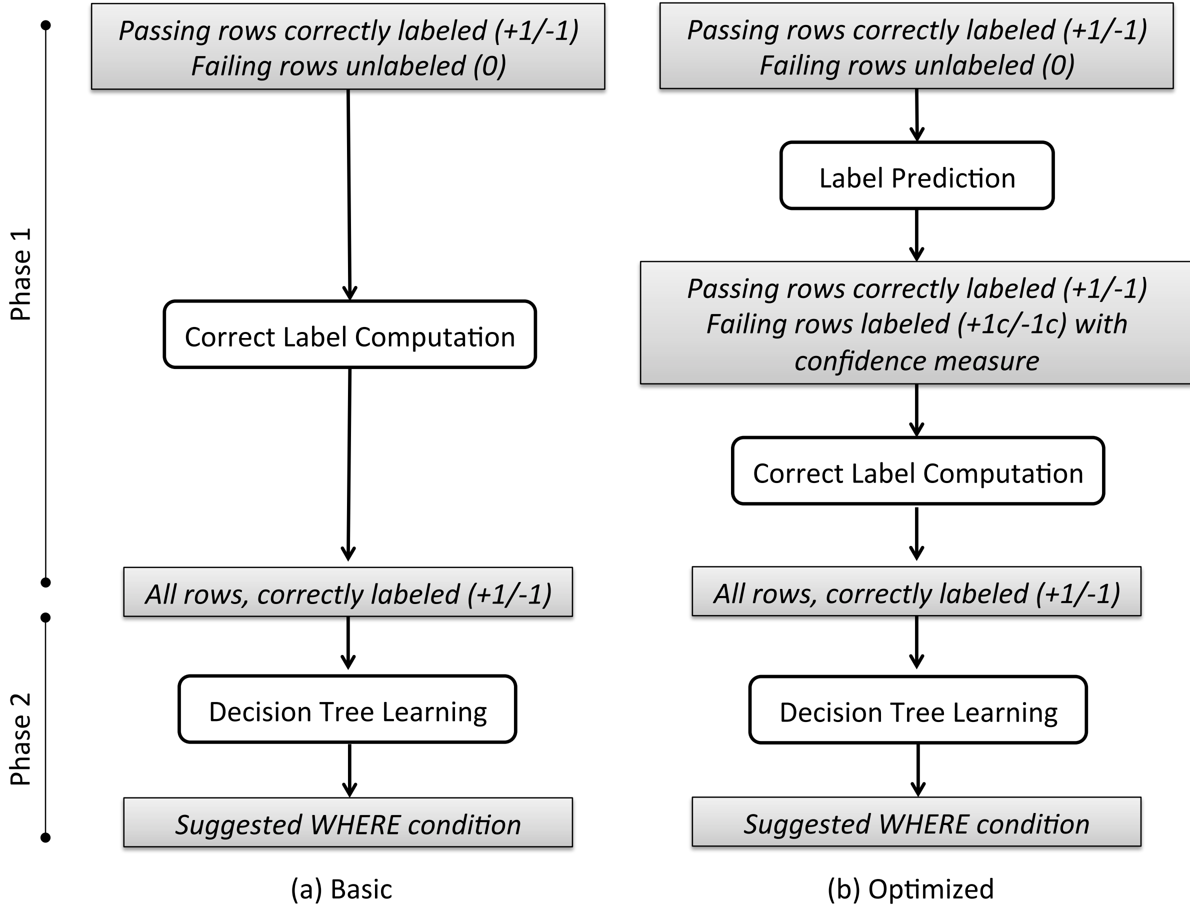

Figure 2: An overview of our approach

(c) Use the expected output to determine a correct labeling by constraint solving.
Phase 2. Use decision-tree learning algorithm such as ID3 to generate a suggested WHERE condition.

As a quick illustration, in Phase 1, (a) the rows corresponding to `CstId` 2 will be assigned boolean variables b_0, b_1, b_2, and b_3. (b) The constraint that would be generated is $(b_0 ~?~ 7 : 0) + (b_1 ~?~ 13 : 0) + (b_2 ~?~ 15 : 0) + (b_3 ~?~ 10 : 0) = 32$. (c) Upon constraint solving, b_0, b_1, and b_3 are true and b_2 false. The corrected labeling would be as shown in Table 1 but with the bold **-** replaced by a +.

Phase 2 starts with the corrected labeling. In decision-tree learning first it would realize that partitioning the rows of the table on the basis of $Year \leq 2008$ gives the maximum *information gain* (see Section 2.1.) In the partition for which $Year \leq 2008$, the maximum information gain is obtained on the basis of $Price \leq 8$, at which point, all positives are perfectly separated from the negatives. In the partition for which $Year > 2008$, all rows are positive regardless of the price. This decision tree is (written as conjuctions of clauses on paths from root to +ve leaf nodes, and disjunction over such paths): $Year > 2008 \vee (Year \leq 2008 \wedge Price \leq 8)$. By DeMorgan's laws, this simplifies to the following condition: $Year > 2008 \vee Price \leq 8$

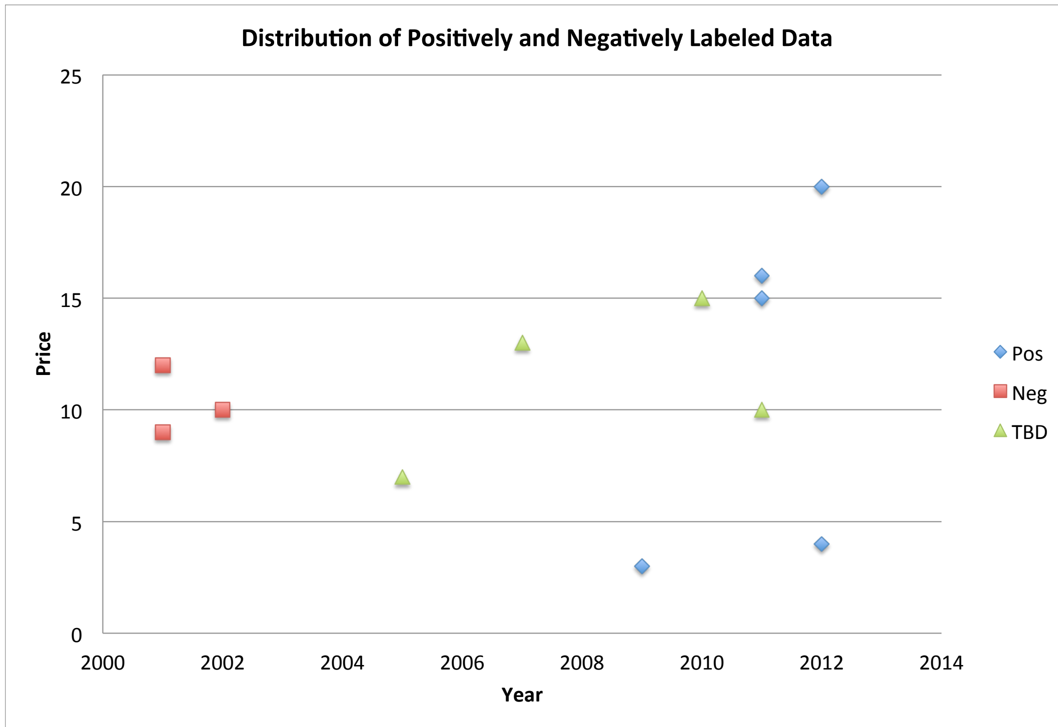

Figure 3: Distribution of data in Table 1

Comparing this to the previous incorrect WHERE condition, we see that while the learned WHERE clause for `Year` is slightly but gratuitously different, for `Price` it is crucially different.

1.3.2 Optimized Approach

The basic approach has two problems. First, depending on the number of rows for failing keys, the amount of symbolic state can be considerable, requiring more resources to come to a correct labeling. Second, Phase 1 may have multiple solutions, and Phase 2 might not be able to generate a good WHERE condition with an arbitrary one; this would require repeating Phase 2 with several different correct labelings obtained from Phase 1.

Here we show a technique, again inspired by machine learning, to address both of these concerns. Our idea is to learn from passing rows (which carry correct labels) likely—but not necessarily perfect—labels for the failing rows. See Figure 2(b). Label learning is used as a pass before correct label identification. Our approach rests on the observation that WHERE clauses are typically generic, and would act the same way for classes of tuples. Therefore, "closeby" tuples are likely to be labeled similarly.

In Figure 3, data of passing keys in Table 1 is shown with diamonds for positively labeled data and squares for negatively labeled data. For the failing key, `CstId` 2, data is shown with a triangle (unlabeled). Assuming that points that are spatially close are likely to be labeled similarly, an

assignment of a positive label to the two unlabeled points on the right can be done with relatively high confidence. The two unlabeled data points in the middle of the chart could go either way, so an assignment of a negative label to the two points in the middle can be done only with low confidence. Support vector machines (see Section 2.2) [24] can be used to assign such labels based on a geometric notion of proximity. Table 2 shows a sample assignment of predicted labels to the failing rows.

CstID	Price	Year	Predicted Label	Confidence
2	7	2005	-	0.3
2	13	2007	-	0.2
2	15	2010	+	0.9
2	10	2011	+	0.9

Table 2: Predicted label assignment for the failing rows

We can use this predicted labeling to reduce the amount of symbolic computation in the basic algorithm. The combinatorial search for correct labels now works in an outer loop. In the first iteration, we assume that the predicted labeling is correct, and attempt to validate it by executing the rest of the program with no symbolic state. In this example, this validation fails. In the next iteration, we remove the least confident label—which is for the tuple 13, 2007—and make it symbolic, remaining labels are as in Table 2. In this case, b_1 is the only symbolic variable. But the constraint $b_1 ? 13 : 0 + 15 + 10 = 32$ is still not satisfiable. In the next iteration, we select next least confident label—which is for the tuple 7, 2005—and make it symbolic as well. This time, the constraint is $b_0 ? 7 : 0 + b_1 ? 13 : 0 + 15 + 10 = 32$ for which the correct solution is found. It turns out that the tuple 7, 2005 needs to be positively labeled, unlike the way it appear visually in Figure 3. Notice that we needed at most two symbolic variables as opposed to four in the basic algorithm. For larger problems, in which one failing key may have number of rows, this optimization could even make the difference between being able to handle the symbolic part computationally or not.

To illustrate the second problem, consider a slightly different table that includes a weight column. Suppose $Weight \geq 30$ is an additional necessary (but not sufficient) condition for a record to be selected, and that a correct WHERE condition ought to have it. In all the passing key data, all positively labeled rows respect this condition. The predicted labels for a failing key might be as shown in Table 3; predications are based on all three attributes (price, year and weight).

CstID	Price	Year	Weight	Predicted Label	Confidence
2	15	2010	20	+	0.4
2	15	2010	35	+	0.9
2	10	2011	40	+	0.9

Table 3: Predicted labels for the failing key, revised table

Suppose the correctness of output for failing key 2 requires *Price* to add up to 25. There are two ways in which this can happen: either select the first and the third rows, or the second and the third rows. Indeed, a constraint-solver implementing correct-label-computation would come up with two possibilities.

If Phase 2 arbitrarily chooses the row (2,15,2010,20), in which weight is less than 30, the decision tree learning algorithm—one that strives to correctly label all training data—would not be able to add the $Weight \geq 30$ clause in the condition *compactly*. This would lead to a more complex

than necessary condition, one which may be overfitted to this example. The correct choice would have been (2, 15, 2010, 35).

This is where closeness helps again. Among the two possibilities of selecting either row 1 or row 2, row 2 would be closer to the other positively labeled training data, and this reflects in the confidence value of row 2 compared to that of row 1. The confidence values are again based on geometric proximity, now in 3-d space.

This seemingly contrived situation is in fact common when selecting from a join of two tables (see Section 3.4 and examples in Section 4.)

1.4 Contributions

Our contributions are the following:

1. We describe a new approach for repairing the WHERE condition of a SELECT statement in a database program. The approach is based on the observation that standard decision-tree learning can be used to arrive at a repair suggestion once the *correct* behavior of the WHERE is known for the failing keys as well.

2. We give a new way of combining machine learning and combinatorial search in determining the correct labels for the failing keys. The learning part takes advantage of the known behavior of the passing keys, whereas, the combinatorial part makes up for cases in which the knowledge for passing keys does not extend perfectly to the failing keys.

3. We present an evaluation of the proposed approach on a suite of programs drawn from an industrial setting. These programs are excerpts of real programs, and the data sets we use come from real data: this is crucial because the effective of our approach cannot be guaged based on synthetic data, which may not be representative of distributions found in real data. Although limited in its scope, the evaluation indicates promise in the approach.

2 Background

In this section, we provide the essential background on machine learning concepts that we use in this work.

2.1 Decision Tree Learning

Given complete and correct labeling on rows of a table such as Table 1, we wish to learn a decision tree which can accurately classify both the data already seen (training data) as well as new, unseen data (test data). For the purpose of this brief introduction, we consider a slightly modified table as below. Suppose *Year* is either high (H) or low (L), and *Price* is one of high (H), medium (M) or low (L). The labels are as follows:

ID	1	1	1	1	1	2	2	2	2	3	3	3
Price	H	H	H	M	H	L	M	H	M	L	L	M
Year	H	H	L	L	H	L	L	H	H	H	H	L
Label	+	+	-	-	+	+	-	+	+	+	+	-

To build a *compact* decision tree, which attribute do we choose as the root of the tree: is it better to first test for *Year*, or *Price* (or *ID*)? If we choose to decide first on the basis of *Year*, our decision would split the data as follows (shown by double vertical line):

Year	H	H	H	H	H	H	H	L	L	L	L	L
ID	1	1	1	2	2	3	3	1	1	2	2	3
Price	H	H	H	H	M	L	L	H	M	L	M	M
Label	+	+	+	+	+	+	+	-	-	+	-	-

Whereas, if we choose to decide first on the basis of *Price*, our decision would split the data as follows:

Price	H	H	H	H	H	M	M	M	M	L	L	L
ID	1	1	1	1	2	1	2	2	3	2	3	3
Year	H	H	L	H	H	L	L	H	L	L	H	H
Label	+	+	-	+	+	-	-	+	-	+	+	+

Intuitively, splitting first on the basis of *Year* seems to be advantageous, as *Year* = H directly determines a large number of labels to be +. *Information gain* [20] formalizes this intuition:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Val(A)} \frac{|S_v|}{S} Entropy(S_v)$$

where $Entropy(S) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$. Here, S is the collection of tuples, A is an attribute (such as *Year*), $Val(A)$ is the set of values in attribute A, S_v is the subset of S with tuples in which attribute A is v. $p_{\oplus}$ is the fraction of positive samples in the set S, and $p_{\ominus}$ is the fraction of negative samples in the set S. $\log 0$ is assumed to be 0.

The entropy of the entire table is (based on 8 positive and 4 negative samples): $-\frac{8}{12} \log_2(\frac{8}{12}) - \frac{4}{12} \log_2(\frac{4}{12}) = 0.92$. Splitting on *Year* gives two sets $S_{Y=H}$ and $S_{Y=L}$ as shown above by the double line column. After computing entropies of these smaller sets—note that the entropy of the set $S_{Y=H}$ is 0—and using the formula above, the gain works out to 0.62. The same computation for *Price* gives 0.35.

The standard decision tree learning algorithm, ID3 [21], is a greedy algorithm that chooses the attribute on which it is going to achieve the highest information gain. It proceeds recursively along the subsets created by the split on the decision attribute, until it obtains sets with uniform labels.

In our example, the algorithm first chooses *Year* to split the set and then works recursively on $S_{Y=H}$ and $S_{Y=L}$, selecting the next attribute on which to decide. For $S_{Y=H}$, no further work is necessary as labels are uniformly +. For the other set, *Price* = L gives the decisive next split. The final decision tree is (+ leaf nodes):

$$Year = H \vee (Year \neq H \wedge Price = L)$$

Since we often have continuous attributes rather than categorical ones, we use the standard technique of finding boundaries at which target labeling changes. In the example of Table 1, if we sort *Price* values in ascending order, 8 and 14 are two such boundaries. The information gain of splitting at 14 would be higher so we could choose to have $Price \leq 14$ and $Price > 14$ as two categories; choosing three categories would also work fine.

Decision tree learning is a vast area, and significant literature exists on how to deal with such problems as overfitting, and dealing with incomplete or noisy data. Discussion of these concerns is outside the scope of this paper.

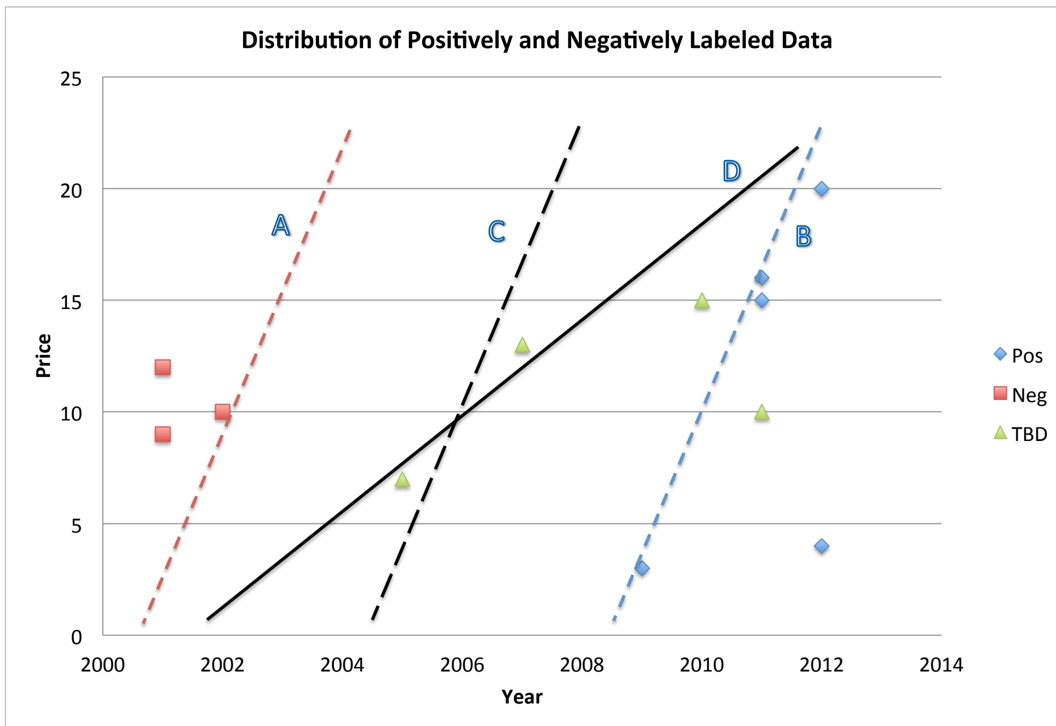

Figure 4: Figure 3 with separating lines

2.2 Support Vector Machines

Support Vector Machine (SVM) [24, 2] is a technique based on mathematical optimization. It is based on the principle of finding the *maximum-margin separating hyperplane* that separates positively labeled data from the negatively labeled data. We explain this intuitively using Figure 4, which presents the same data as Figure 3. Lines labeled A and B both are "hyperplanes" that separate positively labeled points from the negatively labeled points. Once such as separator is found, an unlabeled point can be assigned the same label as the label of the points on its side of the line. The notion of a separating line is a separating hyperplane in higher dimensions.

Clearly, neither of the two lines A and B are particularly satisfactory for the purpose of classifying as yet unlabeled points; they would be prone to mislabeling. For example, a point immediately left of line B would be classified negative, which is incorrect. The line C is the one that creates as much margin as possible between lines A and B, and SVMs find such a line. It is expected that such a line would have the best classification behavior, based on what can be inferred from the training data. Lines A and B are called *support vectors*, which lend their name to the technique.

Mathematically, let w be a vector that is normal to the seperating hyperplane and let b be its offset. For each posivitely labeled point x_i, we require $w.x_i \geq b+1$ and for each negatively labeled point x_i, we require $w.x_i \leq b-1$. The optimization problem is to minimize $\|w\|$.

SVMs create a classifier based on a *geometric* interpretation of proximity of points in the space.

This may not always coincide with the domain-specific measure of proximity, which is outside the purview of SVM (unless explicit captured by synthetic attributes.) In the example above, line D is the one that creates the *correct* separating line, meaning that line would assign the correct labels to the unlabeled points as per the requirements of the database program. D cannot be found by an SVM based on the given training data.

Fortunately, SVMs provide a numerical measure of prediction confidence. Labels assigned to points close to the separating hyperplane that an SVM discovers (*i.e.* line C in this example) would be assigned a lower confidence, and labels assigned to points far from the separating hyperplane would be assigned a higher confidence. As illustrated in the introductory example and Table 2, our premise is that the higher confidence labels assigned on the basis of geometric proximity are usually correct even in terms of the semantic proximity. This premise should be true if there is sufficient amount of training data.

SVMs are considerably more general that what the above brief treatment reveals. It is not necessary for the positively and negatively labeled training data to be perfectly linearly separable. Furthermore, SVMs can look for complex non-linear separators using *kernels*. Vast amount of literature exists to how to make SVMs more efficient both in performance and in prediction accuracy.

3 Details of Our Approach

In this section, we describe the details of the label learning component, the SAT-based label assignment, the decision tree component, and then finally how these components operated together in our approach.

3.1 Label Assignment with SVM

Recall from Figure 2 that this component takes as input (correctly labeled) passing rows and assigns labels to the failing rows, along with confidence measures.

To use an SVM, each passing row from the input tuples containing an n-dimensional tuple is mapped to a point in n-dimensional space. It is assigned a positive or negative label depending on which way the WHERE condition evaluates on that tuple. (Recall that the WHERE condition works correctly on passing rows, so these labels are correct.) This is the training data on which SVM learns a classifier. Next, the classification capability of the SVM is used to compute labels, with confidence measure, for each failing row mapped similarly to a point in n-dimensional space. The output is a real number for each row, where the sign indicates the predicted label and magnitude the confidence.

The application of SVMs to the problem at hand requires several steps of data conditioning. The main issue is that SVMs prefer to view data as numerical values for the purpose of distance computation. Relational database tables seldom contain data in this form. We discuss some of the problems and our solutions.

Nominal Attributes The table could contain *nominal* attributes, which are compared for equality, but not for order. For example, a *State* attribute 2-letter state abbreviation is a nominal attribute. For nominal data, we introduce fresh columns, one for each distinct value of the nominal attribute that

appears in tables. For *State*, we might introduce boolean attributes such as *State=AK* to *State=WY* and hide the original *State* attribute.

At other times, data that looks like non-numeric data might need to be treated numerically. For example, if ranges over dates are significant, then dates have to be mapped to a numeric interval.

Key Attributes Keys are usually nominal data in that value-based proximity of two keys is not meaningful. In joined tables created by Cartesian cross product, one will have two distinct key attributes initially, one coming from each of the two tables. Since it would require too many additional attribute to "de-nomimalize" both of these attributes, we instead include an additional boolean attribute that denotes the equality of these two keys (as it is common to have key equality comparison in SELECT statements for a natural join.)

Scaling It is typical in use of SVMs to scale data to a normalized [0.0,1.0] range for each attribute. In case the range of data for a certain attribute is very large due a few outlier values, care is needed to prevent lower values being scaled down to too close to zero.

We used SVM Light [12] in our experiments, with default parameters. We did not require non-linear kernels in our experiments so far, but in general, the need could arise.

3.2 Label Computation Using SAT

Input to this component is a set of rows, a conservative estimate of which of the attributes may contribute to the output, and the expected output. Typically, this set of rows corresponds to one failing key at a time.

The output is one or more subsets of the set of rows given to the component. Each subset respects the property that if the program is run from immediately after the select statement with only those rows, the output for the corresponding key would be correct.

We use the Alloy tool-set [10] to model the data and control-flow of the ABAP code. Alloy is a language to model and reason about systems using declarative first-order logic constraints on sets and relations. It supports all set-based operations, including transitive closure of relations. It comes with a model finder to find satisfying solutions.

Encoding of the data or state of an ABAP program in relational logic is done as follows. ABAP types typically represent tuples of fields. These are represented as relations with arity equal to the number of fields. Local variables are represented as singleton sets. Control-flow-graph is unrolled a finite number of times to yield a DAG. The control flow from one statement to the other is represented as relational implications while branch points are represented as disjunctions.

Suppose $\sigma_0 S_0; \sigma_1 S_1; \ldots S_k; \sigma_k$, represents the execution of an ABAP program, where σ_0 and σ_1 are the states of the program before and after the execution of the (faulty) database statement S_0, and σ_k is the state after executing $S_0, \ldots, S_k$.

Given σ'_k, the expected output, we use Alloy to compute σ'_1 such that execution of the sequence $S_1; \ldots; S_k$ on σ'_1 produces the output σ'_k. Since S_0 is a SELECT statement, we have an additional constraint that $\sigma'_1 \subseteq \sigma_0$. σ'_1 gives the desired labeling.

Statements $S_1 \ldots S_k$ could be either database or imperative statements. We illustrate the translation of a (correct) ABAP code fragment in Alloy. Here is the ABAP fragment:

```
1  SELECT Price Year from t where Year > 2010
2  LOOP AT t INTO wa
3    sum = sum + wa.Price
4  ENDLOOP
5  assert sum == 30
```

The following Alloy code models it. We assume a maximum of 3 rows with year > 2010.

```
1  sig Table {  tuples: set Tuple }
2
3  sig Tuple {  price: Int,  year: Int }
4
5  pred Query(t: Table, result: Table) {
6    result.tuples in t.tuples
7    all p: result.tuples | p.year > 2010
8  }
9
10 pred P(t: Table, result: Int) {
11   lone x0, x1, x2: t.tuples {
12     (result = 0 and no x0 and no x1 and no x2) or
13     (result = x0.price and no x1 and no x2) or
14     (result = x0.price + x1.price and no x2) or
15     (result = x0.price + x1.price + x2.price)
16   }
17 }
18
19 pred Q(t: Table) {
20     some t1:Table | Query[t, t1] && P[t1, 30]
21 }
```

Running pred Q would yield a set of Tuples, t, wherein the sum of the price values for Tuples where year $>$ 2010 would be 30.

3.3 Decision Tree Learning

The input to this component is fully and correctly labeled rows and the output is a decision tree. We created our own implementation of ID3 essentially following the description in Section 2.1. Here are some of the implementation concerns.

Selecting Relevant Attributes The input tables of the buggy query typically have large number of attributes, many of which are irrelevant. We select a subset of the these attributes that satisfies the following conditions: (1) Contain all the attributes projected by the query (2) Attributes that have been frequently used whenever this table has been used earlier in the code(such as key attributes). (3) Attributes having the same data-type and overlapping values with the state variables at that execution point.

Seeding Synthetic Attributes Decision-tree based algorithms are only equipped to learn clauses that compare attribute values with constants. However, WHERE conditions conditions can contain comparison of two attributes. We seed binary-valued equality predicates between attributes as extra attributes into the learning algorithm. These predicates are seeded based on domain knowledge, for instance, if two tables are being used in a query, a comparison of their key attributes is often a part of the filtering condition. Similarly, attributes of the same data-type and having same range of values may be compared to select records.

Note that the condition generated by the decision-tree learning is only a suggested repair. In particular, it cannot express the condition naturally in terms of program variables (e.g. it would discover $netwr < const$ as opposed to $netwr < var$). Obviously, if some condition is not exhibited by training data, the condition would not appear in the decision tree.

3.4 Putting it all together

In this subsection, we describe how the correct-label computation interacts both with label prediction and with decision-tree learning.

First, we briefly describe the data set up. Identification of passing keys, and therefore passing rows, and failing keys/rows is done on the basis of comparing key-wise actual final output with expected final output. To determine the labels of passing rows, since we know that the WHERE condition as written works correctly on the passing rows, we apply the condition per row.

3.4.1 Interaction of SAT with Label Predictions

Assume for the moment that no join is involved. The correct-label computation takes the predictions for the failing rows, and ranks those predictions separately for each failing key. The process described below is carried out *separately* for each failing key.

Step 1. Create three sets: definitely negative (dneg), definitely positive (dpos), and possibly positive (ppos). Initially, dneg includes all rows for which label prediction is negative and dpos includes all rows for which label prediction is positive. Initially, ppos is empty.

If label assignment as per dpos and dneg works for the rest of the program—we validate this using the SAT infrastructure, though native execution would work—we have found a solution.

Step 2. Shrink dneg and/or dpos and move a row from them to ppos. We first move the row for which the absolute value of the prediction confidence is the least.

Step 3. Use SAT-based process for obtaining a feasible solution for ppos. If we get unsat, go back to Step 2.

When the above process terminates, rows in dneg are labeled negatively, rows in dpos positively, and rows in the current ppos, as per the label assignment given by SAT. Note that if no label predictions were available, we could use SAT for all the rows, i.e. assume all the rows are in ppos. However, as we see in our experiments, this may time out if the number of rows in ppos is large.

Correct labelings for each failing key are assembled together with the known correct labeling for the passing keys, and passed on to the decision tree learning.

Multiple Solutions When working with a failing key, for a given ppos, SAT may come up with multiple ways of assigning labels to the set ppos. (This could be an issue regardless of whether we used predications or not.) It is not necessary that each of these solutions (i.e. set of failing rows to which we assign a positive label) would be a good input to decision-tree learning, and we may wish to try them all. Since the labeled rows that handed over to decision-tree component collates solutions for all the failing keys, this can be a problem. If there are f failing keys and (say) k solutions per failing key, the number of distinct label assignments for the entire table would be k^f. Even though k is typically small, as a heuristic, for each failing key we choose the solution based on the best prediction confidence among the rows that constitute that solution.

3.4.2 Dealing with Joined Tables

Consider the query: `SELECT T1.K, T1.V from T1, T2 where T1.K = T2.K`. Operationally, tables `T1` and `T2` are joined, a selection is performed based on the WHERE condition on the joined table, and finally a projection is performed on specific columns of `T1`. In the above query, the WHERE condition is key equality, but it could be more general.

For decision-tree learning to be able to learn the WHERE condition it needs to observe the selection performed on the *joined* table, because after projection (say) to columns of the first table, the selection condition would typically not be observable from the selected and un-selected rows of the first table.

Here we briefly explain label compututation in the case of joined tables. The label assignment for the joined passing rows is determined simply by applying the given WHERE condition. However, additional constraints must be respected when determining labels for a set of (failing) rows in a joined table.

Define a *block* in the joined table as a set of rows obtained from a single row of the first table by cross producting it from all rows of the second table. At most one row from each block can appear in the joined output table, because projection to the columns of the first table can retain only one of them. Therefore, SAT may consider at most one joined row from each block to assign a positive label.

Conceptually, SAT has to carry out two selections for each failing key:

Selection 1. Select one row from each block that pertains to that key; *and*, as before,

Selection 2. Select a subset from the rows selected in *Selection 1* in a way that program correctness is obtained. This is as described in Section 3.4.1.

The SAT problem is more complex—the number of rows in ppos is essentially multipled by the number of rows in second table—and indeed, example *Ex1* in Section 4 was too large for SAT for this reason. Also, there are vastly more solutions to consider when passing them on to decision-tree learning.

Label prediction comes to our rescue again. Instead of modeling *Selection 1* in SAT, we simply choose the row in each block that has the highest numeric prediction, resulting in just one set to be passed to *Selection 2*. This is meaningful, because we are selecting the most likely row from each block to be positively labeled in the complete labeling. It is a heuristic, because we rely on sufficient training data for label predictions to be reasonably accurate.

4 Evaluation

This section first presents an experimental evaluation based on case-studies using a select set of subject programs (Section 4.1). Next, it discusses some limitations of our approach as well as the feasibility of applying a more straighforward, mutation-based approach for repairing the chosen subjects (Section 4.2).

4.1 Case Studies

We selected seven subject programs, which are fragments of ABAP code from industrial applications running on real data sets. The bugs in these programs are actual bugs that occurred in the past. In all cases we know the correct fix to the bug.

The objective of our experiments is to determine whether our technique is efficient and effective in learning an ABAP query, which helps the developer fix the bug (even if the learnt query is not exactly what a human would have manually created). The evaluation was performed on a prototype implementation of our approach using the Alloy 4.2 tool-set (specifically, Forge, Kodkod

Subjects	*# Passing rows*	*# Failing rows*	*Label predictions*	*Correct label computation*				*Decision-tree learning*	
				Basic (w/o predictions)		*Optimized (with predictions)*			
			# correct labels	*time (min)*	*# correct sols*	*time (min)*	*# correct sols*	*time (min)*	*correct cond?*
Ex1	40129	10032	10029	TO	NA	4	1	2	Yes
Ex2	16641	30	11	4	256	2	32	1	Yes
Ex3	316	12	12	1	1	1	1	0.16	Yes
Ex4	274	58	15	1.5	1	3	1	0.03	Yes*
Ex5	993	84	78	5,TO	1,NA	8	1	0.08	Yes
Ex6	90346	1816	1814	TO	NA	5	1	5	Yes
Ex7	13911	2	0	0.5	1	2	1	25	Yes

Table 4: Summary of results. TO- time-out interval set to 15 mins.

and miniSAT), SVM Light in transductive learning mode, and a home-grown implementation of the ID3 decision-tree learning algorithm.

4.1.1 Summary of results

Table 4 summarizes the results. The first column lists the seven subject programs. For every candidate, *# Passing rows* and *# Failing rows* show the number of records corresponding to the passing and failing keys for the respective bug. *Label predictions* column highlights the prediction accuracy by showing the number of failing rows for which the labels were predicted correctly. We present the results of using both our basic (*Basic (w/o predictions)*) and optimized (*Optimized (with predictions)*) approaches to compute the correct labels. We tabulate the SAT solving times and the number of correct solutions generated for both of these approaches (Note that in some cases the basic approach would have sufficed). Finally, we present results on decision-tree learning.

In all the seven cases, our technique was able to find a correct WHERE condition which produces the expected output. Except in one case (*Ex7*), our overall technique takes less than 10 minutes to complete. The predictions were particularly useful in three cases (*Ex1*, *Ex6*, *Ex5*) where the SAT solver timed out. In the first two cases where selections needed to be performed from a joined table, the solver could not find any solution, while for *Ex5*, the solver timed out on 1 failing key. We found that the predictions were not very effective in two cases (*Ex4* and *Ex7*). The condition learned was close to the intended one in all but one of the cases (*Ex4*).

4.1.2 Example application scenarios

Next, we describe details of applying our approach to four subjects to highlight some of its key characteristics and how it handles different types of faults. The other examples are similar to these and are not described here for brevity.

Scenario 1: Repair without predictions. For simplicity, we start with *Ex3*, which illustrates a case in which just the basic approach for repair (without the use of predictions) can successfully produce a valid repair suggestion.

Consider the following buggy SELECT statement in *Ex3*:

```
select * from ekbe into table tab_ekbe
          where ( vgabe eq '2' or vgabe eq '3' )
 //and ebeln in ebeln_range  Needed in the correct query
          order by ebeln ebelp.
```

The WHERE clause is essentially missing two predicates in the form of a missing range-check predicate for the field `ebeln`. This error results in 12 unexpected rows in the output of the program.

Correct Label Computation. The incorrect rows in the program output correspond to 12 failing rows in the input `ekbe` table. SAT quickly finds that none of these records should get selected.

Although it is not required for this scenario, we did run SVM for predicting the labels for the 12 failing rows. The predictions were 100% accurate, and all the 12 records were assigned negative prediction values. We ran SAT marking them as dneg and the labeling was deemed satisfiable within a minute, i.e. not much savings in time over the basic approach.

Decision-tree learning. The condition learned was as given below and was correct in not selecting exactly the 12 failing row.

```
vgabe = '2'and
ebeln <= 4500000229
```

The generation of a comparison on `ebeln` conveys to the programmer that a bound check is missing. Indeed, the source code defines the constant `ebeln_range` as [`ebeln_low`, `ebeln_high`], but the buggy query fails to use it.

The generated repair could not infer the lower bound on `ebeln`, nor it could generate an additional condition on `vgabe`, because such conditions were not warranted by this specific input data. Nonetheless, the repair suggestion is useful in helping the programmer fix the problem. Our technique is intended to generate a useful repair suggestion for the programmer, as opposed to a perfect replacement.

Scenario 2. Use of optimized approach to handle joins. This scenario illustrates the use of predictions to reduce the search space when the buggy statement involves table joins.

The program (*Ex1*) creates a sales order report by calculating order amount and unbilled amount for each sales order. It first creates a table called `p_i_vbrp` using the following query:

```
select vbeln posnr aubel aupos matnr netwr
 from   vbrp, p_i_vbap
 into   table p_i_vbrp
 where  aubel = p_i_vbap—vbeln
 and    aupos = p_i_vbap—posnr
 // and netwr > 0. Needed in the correct query
```

However, the missing $netwr > 0$ predicate from the WHERE condition causes incorrect `p_i_vbrp` formation. For a failing key-value `aupos=102`, here are the computed and expected rows in `p_i_vbrp`:

```
Computed:                     Expected:
aubel aupos  netwr            aubel aupos  netwr
102   20       0.00           102   20   8000.00
102   20    8000.00           102   30  11200.00
102   30       0.00
102   30   11200.00
```

The computation after the SELECT loops over the rows of the `p_i_vbap` table, and for each row, it fetches a row from `p_i_vbrp` based on the `read` statement, which matches `vbeln` in `p_i_vbap` to `aubel` in `p_i_vbrp` and `posnr` in `p_i_vbap` to `aupos` in `p_i_vbrp`. The corresponding `netwr` amounts of the rows read are aggregated for each `vbeln` which contribute to the calculation of the final amount that appears in the report.

The problem arises as the `read` statement reads only the *first* matching row, so for `vbeln` 102 it ends up reading from the incorrect `p_i_vbrp` the rows (102, 20, 0) and (102, 30, 0) corresponding to two `posnr` values 20 and 30, instead of (102, 20, 8000) and (102, 30, 11200), which it would have read from the correct version of the table. This leads to wrong `l_billamt` value and it shows up in the incorrect output for the key 102. There are 20 failing keys with incorrect amounts in the output report.

Correct Label Computation. This illustrates a case where the input space becomes too large for SAT to handle due to the presence of joins, as explained in Section 3.4.2. Each failing key is

considered separately when invoking SAT to look for correct solutions. There are an average of 3 records corresponding to each failing key value of field `aubel` in `vbrp`. Each combination of `<aubel, aupos, netwr>` in `vbrp` maps to a block of 227 records in the joined table. Hence the search space to perform both *Selections 1 and 2*(Section 3.4.2) using SAT would be 227^3 per failing key. SAT times out without finding any solution when applying the basic approach without predictions.

Label Predictions-based reduction of search space. There were 40129 records in the joined table corresponding to the passing keys that were labeled as per their outcome in the existing execution. 10032 failing rows were unlabeled. SVM attached a positive prediction to 19 unlabeled rows and a negative prediction to the remaining.

As explained in the Subsection 3.4.2, *Selection 1* was performed by choosing the record having highest prediction from each block in the joined table.

We used the predictions of the selected subset of records to further reduce the state-space for *Selection 2*. The initial labeling was based on sign of predications. All records whose absolute value of prediction (confidence) is greater than a threshold of 0.0 are considered definitely positive, while the others are marked definitely negative. We verified that this gave perfectly correct labeling for all but three keys. For these three keys, we had to invoke the dpos and dneg shrinking process to arrive at a correct solution. For example, for key 146, initially both rows were marked negative (as shown below) leading to unsatisfiability.

```
aubel aupos netwr pred
146    10      0  -0.99972347
146    10     30  -0.39996994
```

After iteratively moving rows with low confidence to ppos, a correct solution was found when records with confidence below 0.4 were moved out of dneg set. For instance, for key 146, the second row was moved to ppos. SAT assigned positive labels to these rows leading to satisfiable solutions. In all, this process completed in 4 minutes as tabulated.

Decision-tree Learning. Decision-tree learning discovered the correct WHERE condition:

```
aubel = p_i_vbap-vbeln and
aupos = p_i_vbap-posnr
and netwr > 6
```

Note that our approach was able to learn the join condition. The constant discovered is 6 rather than 0, due to the distribution of the data. But it is a good repair suggestion since it points out an important missing clause.

One of the challenges in applying prediction in this example was the large range of values of `netwr` column which required truncation of outlier data points, so that the distribution of values did not get drowned out. Without this data conditioning, the accuracy of the results obtained by SVM was poor.

Scenario 3. Use of predictions to rank candidate solutions. This scenario highlights that our repair algorithm is not restricted only to SELECT statements, and further illustrates a case where predictions aid in reducing the space of candidate solutions on which decision-tree learning has to be performed.

The buggy statement in this example (*Ex2*) is a `DELETE` statement, shown below.

```
DELETE ADJACENT DUPLICATES FROM db_tab
   COMPARING kunnr matnr
   //arktx Needed in the correct query
```

The `DELETE ADJACENT DUPLICATES` statement deletes a row from the table that has same values in its immediately previous row for the fields specified in `COMPARING` clause. This could be modeled as an equivalent SELECT statement as shown next.

```
select * from db_tab_rc as db_tab1, db_tab_rc as db_tab2
where db_tab1.rc = db_tab2.rc+1 and
db_tab1.kunnr = db_tab2.kunnr and
db_tab1.matnr = db_tab2.matnr and
// db_tab1.arktx = db_tab2.arktx Needed in correct query
```

Where `db_tab_rc` has an extra column `rc` in addition to all the columns of `db_tab`. It contains the same records as `db_tab` with the `rc` column populated with the row number.

This statement selects rows that would need to be *deleted* by the original statement. The code after the `DELETE` statement, in a nutshell, aggregates the `netwr` amounts corresponding to every unique values in `monat` field on `db_tab`. The output report had incorrect amounts displayed for two `monat` values – Sep2008 and Oct2008 (2 failing keys).

Correct Label Computation. The `db_tab` had 10 records with `monat` as Sep2008 and 20 records with Oct2008. Note that although the SELECT is over a join and every row of `db_tab_rc` maps to a block of rows in the joined table, we know upfront the exact record that needs to be considered from every block. The only record that can be selected in the block corresponding to every failing row of `db_tab` is the one where `db_tab1.rc = db_tab2.rc+1` is satisfied as this predicate will be present in the correct version of the query. Hence the input state space for SAT remains 10 and 20 respectively for the two failing keys and it becomes feasible to apply the basic approach of label computation without predictions.

SAT is invoked separately on the records for the 2 failing keys to determine the possible subsets of the records that would sum up to the respective expected final amounts. There are 8 possible subsets for Sep2008 and 32 possible subsets for Oct2008, leading to 256 possible correct labelings. It would be inefficient to generate 256 possible WHERE clauses. This is where predictions aid in heuristically selecting the solution that is most likely to yield the ideal WHERE clause. Note that this displays a scenario wherein label predictions aid in reducing the state-space of decision-tree generation even if the number of failing rows may be small enough for SAT to process.

Label Predictions-based solution ranking. There are only 80 positively labeled passing key records compare to 12691 negatively labeled records in the joined table. Hence the prediction accuracy in terms of the classification is low. However, the confidence of the incorrectly predicted labels were lower than the correctly predicted ones. We used predication-value based ranking of the solutions to selecting the desired solution for both for the two failing keys (comprising of 2 records and 17 records respectively).

Decision-tree Learning. The WHERE clause learnt for the SELECT statement was,

```
db_tab1.rc = db_tab2.rc + 1 and
db_tab1.kunnr = db_tab2.kunnr and
db_tab1.arktx = db_tab2.arktx
```

As can be observed, the condition on `arktx` that was missing in the incorrect version is correctly discovered. However, the condition on `matnr` is missing from the learned clause. This is because the `matnr` values are equal for all adjacent records in which the other two conditions are also satisfied. This makes the learned WHERE clause correct for the given input set.

Scenario 4. Impact of incorrect selection on passing keys. *Ex4* displays an interesting scenario which violates our assumption about the correctness of erroneous SELECT statement for the passing keys. The final output corresponding to passing keys is still correct but the SELECT acts incorrectly on some of them.

The erroneous SELECT statement given below leads to the inclusion of 58 extra records for

the failing keys in the actual output of the program, compared to the expected correct output.

```
select  ebeln  ebelp  belnr buzei  bewtp  budat matnr
        werks ernam
 from ekbe
 into table it_ekbe
 where
   budat in s_crdate
  // AND vgabe = 1 Needed in the correct query
```

In this example, for the passing keys too, the erroneous SELECT statement selected some extra rows, but subsequently they got deleted by a delete statement in the program. Consequently, these passing keys yielded the correct final output anyway.

Correct Label Computation. The incorrect labeling for 16 passing key records where `vgabe = 1` impacts the accuracy of predictions as seen from Table 4. Hence the approach of using predictions to determine dpos and dneg record sets performed poorly. It passes through 4 rounds and produces correct solution only when all records are put into the ppos set, which is equivalent to the basic approach, i.e. without prediction.

```
 belnr<=5000000236 OR
(budat = 61111) OR
(budat > 61111 AND
 (ebeln = 4500000022))
```

Decision-tree Learning. The incorrect labeling of the passing key records impacts the WHERE clause condition learned.

The condition is quite different from the one in the correct version of the code. It leads to the expected final output on this data set, but this is not a useful repair suggestion (marked by * in Table 4).

4.2 Discussion

4.2.1 Limitations

1. Our technique assumes that the incorrect selection criteria works correctly for keys that satisfy the final output correctness criteria. Violation of this assumption (*Ex4*) impacts the quality of the predictions and the WHERE condition learned.

2. Sufficient amount of representative passing data is required to make the learning effective. For *Ex7*, the failing rows were all erroneously predicted to be negatively labeled. This is because only 10% of the passing key records were positively labeled. Moreover, only 2 out of 12890 records reflected the condition `vgabe = 3`. SVM's learning algorithm possibly considered these as outliers and made predication based on the majority of records.

3. Attention must be paid to data conditioning, which currently uses heuristics. This was an issue in several examples, and without proper data conditioning we did not get good label predictions.

4. The ID3 algorithm is designed to correctly label all training data. However, if the data in the current execution is not representative enough, then the WHERE condition created may be overfitted to the data (*Ex4*). Techniques to avoid overfitting [20] compromise the accurate labeling of training data. Finding the right balance for our application is the subject of future work.

5. The learning algorithm strives to generate the most compact classifier for the given data. In some cases, this could exclude clauses that would be in general necessary, but do not impact the outcome for the given data. To reiterate, our technique generates useful repair suggestions and not necessarily plug-and-play repairs.

4.2.2 Comparison with Mutation-based repair

In this section we discuss how a standard technique such as a simple repair algorithm based on mutations would perform for the bugs under consideration. The method checks if the WHERE condition could be corrected by either adding one clause, removing one existing clause, or replacing an existing clause with a new one. Clauses of the form `Field Operator Field`, `Field Operator Constant` and `Field Operator Variable` are considered as mutants.

In almost all the cases the search space of the number of mutants is very huge (in the order of 40,000) leading to a blow-up in the worst-case exploration time (in the order of 40 hours assuming an average of 5 seconds to execute 1 mutant). The main reason being the large number of distinct values that could be compared in the clauses of the form `Field Operator Constant`. An algorithm that does not consider clauses that involve constants would work much faster, however it would be unsuccessful in discovering the correct WHERE condition for *Ex1*, *Ex3*, and *Ex7*.

Compared to the worst-case time taken by mutation approach, decision tree based technique is fast. Since the algorithm learns the rule from scratch, the time for discovering the correct condition remains the same irrespective of the number of clauses missing or erroneous in the buggy statement, whereas this has a much worse effect on the mutation technique. However, the condition generated by a technique that looks to make minimal changes to the existing condition, may be closer to the ideally correct version as against our technique which would learnt the most compact clause that works for the given data.

5 Related Work

Recent years have seen much progress in techniques for automated debugging – both for *fault localization* [15], i.e., finding the locations of (likely) faulty lines of code, as well as *program repair* [26], i.e., correcting the faulty lines of code to fix the fault(s), which is the focus of this paper.

Fault localization. The application of machine learning to debugging is largely confined to fault localization. Decision tree generation algorithms, including C4.5, have been used in conjunction with the fault localization tool Tarantula [14] to cluster failing tests in order to help developers manually fix bugs in their code more effectively [14, 3]. Statistical debugging techniques [18, 11] employ statistical analysis on the data collected from passing and failing program runs to determine likely faulty statements.

Program repair. The problem of program repair has been the focus of a number of recent techniques, including those based on evolutionary algorithms [26], specifications [6], program code transformations [5], as well as program state mutations [4]. The key novelty of our technique with respect to previous work is two-fold: (1) previous work has not considered repair of SQL statements, in general, and ABAP programs, in particular; and (2) machine learning and systematic search have not been integrated before for program repair.

Weimer et al. [26] introduced the idea of program repair using genetic programming, where existing parts of code are used to patch faults in other parts of code and patching is restricted to those parts that are relevant to the fault. Ackling et al. [1] repair a program by evolving patches to fix it rather than evolving the fautly program itself, and argue that doing so simplifies the repair problem. Wilkerson et al. [27] present a co-evolutionary approach where code and its tests are

co-evolved to improve the bug finding ability of tests as well as to improve the overall quality of the code in order to provide an automated software correction system.

Chandra et al. [4] use changes to program states in a faulty program to approximate the behavior of a correct program with respect to a given set of passing and failing tests, and use these state mutations to guide syntactic changes to code in order to repair it. Malik et al. [19] use a search-based technique for data structure repair [16] as a basis of program repair. Specifically, they use mutations done on program state to fix corrupt data structures as a basis of synthesizing program statements that abstract those fixes using program variables. Gopinath et al. [6] consider a similar setting of repairing programs that operate on structurally complex data but use a different approach. They introduce nondeterminism in the program's operations and use SAT solvers to generate valid program states (with respect to given specifications), which are then abstracted into program expressions that evaluate to those states and provide the fixes. Jobstmann et al. [13] originally used this technique to replace faulty program expressions with unknowns and formed a model checking problem in order to repair a faulty program with respect to its linear time logic specification. Griesmayer et al. [7] map the problem or repairing boolean programs to finding a memoryless, stackless strategy in a game and explore the game graph to find a repair for the boolean program, and show how it can be used to repair a class of C programs.

Debroy et al. [5] introduced the idea of using *mutations*, i.e., syntactic transformations to the faulty program as a basis of repair. The developed their technique in the context of the Tarantula tool and spectrum-based fault localization using a given set of passing and failing tests to focus mutations. While such code transformations can assist in debugging, the space of variations to explore grows very quickly the feasibility of using such a technique for real applications requires developing novel pruning techniques.

Wei et al. [25] attempt to combine specification-based and test-based repair. Boolean queries are used to build an abstraction of the state, which forms the basis to represent contracts of the class, fault profile of failing tests and a behavioral model based on passing tests. A comparison between failing and passing profiles is performed for fault localization and a subsequent program synthesis effort generates the repaired statements. This technique however only corrects violations of simple assertions, which can be formulated using boolean methods already present in the class.
Program synthesis. A closely related area to program repair is *program synthesis* [9], where a goal is to generate (parts of) a program independently of a given incorrect version. A number of program synthesis techniques are based on specifications. Programming by sketching [23] employs SAT solvers to generate missing parts of a given skeletal program with respect to another reference program that serves as a specification. A SAT solver completes the implementation details by generating expressions to fill the "holes" of the skeletal program by exploring several of its variants. Gulwani et al. [9] use the counterexample guided iterative synthesis paradigm together with SMT solvers to synthesize loop-free programs with respect to given specifications of desired functionality. Kuncak et al. [17] generalize decision procedures into synthesis procedures to synthesize code snippets from specifications.

To alleviate the burden of writing detailed specifications, some recent techniques support synthesis based on given concrete input/output examples. Gulwani [8] presents such a technique for synthesizing string processing code for spreadsheets using examples of how a user processes sample strings. More recently, Singh et al. [22] integrate *scenarios*, which illustrate steps of modifying specific data structure instances, with given code skeletons and inductive definitions to facilitate

program synthesis.

At present, techniques for synthesis have largely been developed independently of techniques for repair.

6 Conclusion

We presented a novel approach to generate repair suggestions for defective database programs, where the faults are in the WHERE condition of a SELECT statement. We use techniques from machine learning to learn a decision tree from the correct behavior shown on the defect-free data as well on correct behavior determined for defect-inducing data. The decision tree guides our repair technique. Our key novelty is to determine the correct behavior of the defect-inducing data using a combination of SAT-based search and prediction generated by support vector machines (SVMs). Experiments using a prototype embodiment of our approach on a suite of real programs show the promise it holds in automated debugging.

References

[1] T. Ackling, B. Alexander, and I. Grunert. Evolving patches for software repair. In *GECCO*, pages 1427–1434, 2011.

[2] K. P. Bennett and C. Campbell. Support vector machines: hype or hallelujah? *SIGKDD Explor. Newsl.*, 2(2):1–13, Dec. 2000.

[3] L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with Tarantula. In *ISSRE*, pages 137–146, 2007.

[4] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *ICSE*, pages 121–130, 2011.

[5] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, pages 65–74, 2010.

[6] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188, Mar. 2011.

[7] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to C. In *CAV*, pages 358–371, 2006.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[9] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[10] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2), Apr. 2002.

[11] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[12] T. Joachims. Making large-scale svm learning practical. *Advances in Kernel Methods - Support Vector Learning*, 1999.

[13] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, 2005.

[14] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA*, pages 16–26, 2007.

[15] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[16] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *SPIN*, pages 123–138, 2005.

[17] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[19] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *ASE*, pages 615–619, Nov. 2009.

[20] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[21] R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1), 1986.

[22] R. Singh and A. Solar-Lezama. SPT: Storyboard programming tool. In *CAV*, pages 738–743, 2012.

[23] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, pages 4–13, 2009.

[24] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, 1995.

[25] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, pages 61–72, 2010.

[26] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[27] J. L. Wilkerson and D. Tauritz. Coevolutionary automated software correction. In *GECCO*, pages 1391–1392, 2010.