Learning to Blame

Localizing Type Errors with Data-Driven Diagnosis

ERIC L. SEIDEL, UC San Diego
HUMA SIBGHAT, UC San Diego
KAMALIKA CHAUDHURI, UC San Diego
WESTLEY WEIMER, University of Virginia
RANJIT JHALA, UC San Diego

Localizing type errors is challenging in languages with global type inference, as the type checker must make assumptions about what the programmer intended to do. We introduce Nate, a *data-driven* approach to error localization based on supervised learning. Nate analyzes a large corpus of training data — pairs of ill-typed programs and their "fixed" versions — to automatically *learn a model* of where the error is most likely to be found. Given a new ill-typed program, Nate executes the model to generate a list of potential blame assignments ranked by likelihood. We evaluate Nate by comparing its precision to the state of the art on a set of over 4,500 ill-typed OCaml programs drawn from introductory programming classes. We show that when the top-ranked blame assignment is considered, Nate's data-driven model is able to correctly predict the exact sub-expression that should be changed 73% of the time, 28 points higher than OCaml and 17 points higher than the state-of-the-art SHErrLoc tool. Furthermore, Nate's accuracy surpasses 86% when we consider the top *two* locations and reaches 91% if we consider the top *three*.

ACM Reference format:
Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame. 1, 1, Article 1 (May 2017), 23 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When viewed fully, types are awe-inspiring. Languages like OCaml and Haskell make the value-proposition for types even more appealing by using constraints to automatically synthesize the types for all program terms without troubling the programmer for any annotations. Unfortunately, this automation has come at a price. Type annotations signify the programmer's intent and help to correctly blame the erroneous sub-term when the code is ill-typed. In the absence of such signifiers, automatic type inference algorithms are prone to report type errors far from their source [Wand 1986]. While this can seem like a minor annoyance to veteran programmers, Joosten et al. [1993] have found that novices often focus their attention on the *location* reported and disregard the *message*.

Localizing Errors. Several recent papers have proposed ways to improve feedback via error *localization*. At a high-level these techniques analyze the set of typing constraints to find the minimum (weighted) subset that, if removed, would make the constraints satisfiable and hence, assertion-safe [Jose and Majumdar 2011] or well-typed [Chen and Erwig 2014a; Loncaric et al. 2016; Pavlinovic et al. 2014; Zhang and Myers 2014]. The finger of blame is then pointed at the sub-terms that yielded those constraints. This minimum-weight approach suffers from two drawbacks. First, they are not *extensible*: the constraint languages and algorithms for computing the minimum weighted subset must be designed afresh for different kinds of type systems and constraints [Loncaric et al. 2016]. Second, and perhaps most importantly, they are not *adaptable*: the weights are fixed in an ad-hoc

2017. XXXX-XXXX/2017/5-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fashion, based on the *analysis designer's* notion of what kinds of errors are more or less likely, rather than adapting to the kinds of mistakes programmers actually make in practice.

A Data-Driven Approach. In this paper, we introduce NATE[1] a *data-driven* approach to error localization based on supervised learning (see Kotsiantis 2007 for a survey). NATE analyzes a large corpus of training data — pairs of ill-typed programs and their subsequent fixes — to automatically *learn a model* of where errors are most likely to be found. Given a new ill-typed program, NATE simply executes the model to generate a list of potential blame assignments ranked by likelihood. We evaluate NATE by comparing its precision against the state-of-the-art on a set of over 4,500 ill-typed OCAML programs drawn from introductory programming classes. We show that, when restricted to a *single* prediction, NATE's data-driven model is able to correctly predict the exact sub-expression that should be changed 73% of the time, 28 points higher than OCAML and 17 points higher than the state-of-the-art SHERRLOC tool. Furthermore, NATE's accuracy surpasses 86% when we consider the top *two* locations and reaches 91% if we consider the top *three*. We achieve these advances by identifying and then solving three key challenges.

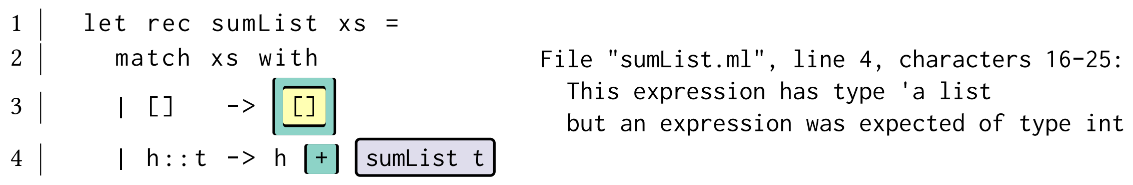
Challenge 1: Acquiring Labeled Programs. The first challenge for supervised learning is to acquire a corpus of training data, in our setting a set of ill-typed programs *labeled* with the exact sub-terms that are the actual cause of the type error. Prior work has often enlisted expert users to manually judge ill-typed programs and determine the "correct" fix [*e.g.* Lerner et al. 2007; Loncaric et al. 2016], but this approach does not scale well to a dataset large enough to support machine learning. Worse, while expert users have intimate knowledge of the type system, they may have a blind spot with regards to the kinds of mistakes novices make, and cannot know in general what novice users intended.

Our *first contribution* (§ 2) is a set of more than 4,500 labeled programs, giving us an accurate ground truth of the kinds of errors and the (locations of the) fixes that novices make in practice. We obtain this set by observing that software development is an iterative process; programmers eventually fix their own ill-typed programs, perhaps after multiple incorrect exploratory attempts. To exploit this observation we instrumented the OCAML compiler to collect fine-grained traces of student interactions over two instances of an undergraduate Programming Languages course.[2] We then post-process the resulting time-series of programs submitted to the OCAML compiler into a set of pairs of ill-typed programs and their subsequent *fixes*, the first (type-) correct program in the trace suffix. Finally, we compute the blame labels using a *tree-diff* between the two terms to find the exact sub-terms that were changed in the fix.

Challenge 2: Modeling Programs as Vectors. Modern supervised learning algorithms work on *feature vectors*: real-valued points in an *n*-dimensional space. While there are standard techniques for computing such vectors for documents, images, and sound (respectively word-counts, pixel-values, and frequencies), there are no similarly standard representations for programs.

Our *second contribution* (§ 3) solves this problem with a simple, yet expressive, representation called a *Bag-of-Abstracted-Terms (BOAT)* wherein each program is represented by the *bag* or multiset of (sub-) terms that appears inside it; and further, each (sub-) term is *abstracted* as a feature vector comprising the numeric values returned by *feature abstraction* functions applied to the term. We can even recover *contextual* information from the parent and child terms by *concatenating* the feature vectors of each term with those of its parent and children (within a fixed window). We have found this representation to be particularly convenient as it gives us flexibility in modeling the syntactic and semantic structure of programs while retaining compatibility with off-the-shelf classifiers, in contrast to, *e.g.*, Raychev et al. [2015], who had to develop their own variants of classifiers to obtain their results.

[1]"Numeric Analysis of Type Errors"; any resemblance to persons living or dead is purely coincidental.
[2] at AUTHOR's INSTITUTION (IRB HIDDEN).



```
1 | let rec sumList xs =
2 |   match xs with
3 |   | []   -> []
4 |   | h::t -> h + sumList t
```

```
File "sumList.ml", line 4, characters 16-25:
  This expression has type 'a list
  but an expression was expected of type int
```

Fig. 1. (left) An ill-typed OCaml program that should sum the elements of a list, with highlights indicating three possible blame assignments based on: (1) the OCaml compiler; (2) the fix made by the programmer; and (3) minimizing the number of edits required. (right) The error reported by OCaml.

Challenge 3: Training Precise Classifiers. Finally, the last and most important challenge is to use our BOAT representation to train classifiers that are capable of *precisely* pinpointing the errors in real programs. The key here is to find the right set of feature abstractions to model type errors, and classification algorithms that lead to precise blame assignments. Fortunately, our BOAT model allows us a great deal of latitude in our choice of features. We can use abstraction functions to capture different aspects of a term ranging from syntactic features (*e.g.* is-a-data-constructor, is-a-literal, is-an-arithmetic-operation, is-a-function-application, *etc.*), to semantic features captured by the type system (*e.g.* is-a-list, is-an-integer, is-a-function, *etc.*). We can similarly model the blame labels with a simple feature abstraction (*e.g.* is-changed-in-fix).

Our *third contribution* (§ 4) is a systematic evaluation of our data-driven approach using different classes of features like the above, and with three different classification algorithms: logistic regression, decision trees, and neural networks. Our evaluation lets us identify the best features and classifier and empirically characterize the importance of different features for error localization. In particular, we find that while machine learning over syntactic features of each term in isolation performs worse than existing purely constraint-based approaches (*e.g.* OCaml, SHErrLoc), augmenting the data with a single feature corresponding to the *type error slice* [Tip and Dinesh 2001] brings our classifiers up to par with the state of the art, and further augmenting the data with *contextual* features allows our classifiers to outperform the state of the art by 17 percentage points.

Thus, by combining modern statistical methods with domain-specific feature engineering, Nate opens the door to a new data-driven path to precise error localization. In the future, we could *extend* Nate to new languages or forms of correctness checks by swapping in a different set of feature abstraction functions. Furthermore, our data-driven approach allows Nate to *adapt* to the kinds of errors that programmers (specifically, novices who are in greatest need of precise feedback) actually make rather than hardwiring the biases of compiler authors who, by dint of their training and experience, may suffer from blind spots with regards to such problems. In contrast, our results show that Nate's data-driven diagnosis can be an effective technique for localizing errors by collectively learning from past mistakes.

2 OVERVIEW

Let us start with an overview of Nate's approach to localizing type errors by collectively learning from the mistakes programmers actually make.

The Problem. Consider the `sumList` program in Figure 1, written by a student in an undergraduate Programming Languages course. The program is meant to add up the integers in a list, but the student has accidentally given the empty list as the base case, rather than `0`. The OCaml compiler collects typing constraints as it traverses the program, and reports an error the moment it finds an inconsistent constraint. In this case it blames the recursive call to `sumList`, complaining that `sumList` returns a `list` where an `int` was expected by the `+` operator.

This *blame* assignment is inconsistent with the programmer's intention and may not be helpful debugging information to a novice.

It may appear obvious to the reader that `[]` is the correct expression to blame, but how is a type checker to know that? Indeed, recent techniques like SHErrLoc and Mycroft [Loncaric et al. 2016; Pavlinovic et al. 2014; Zhang and Myers 2014] fail to distinguish between the `[]` and `+` expressions in Figure 1; it would be equally valid to blame *either* of them alone. The `[]` on line 3 could be changed to `0`, or the `+` on line 4 could be changed to either `@` (list append) or `::`, all of which would give type-correct programs. Thus, these state-of-the-art techniques are forced to either blame *both* locations, or choose one *arbitrarily.*

Solution: Localization via Supervised Classification. Our approach is to view error localization as a *supervised classification* problem [Kotsiantis 2007]. A *classification* problem entails learning a function that maps *inputs* to a discrete set of output *labels* (in contrast to *regression*, where the output is typically a real number). A *supervised* learning problem is one where we are given a *training set* where the inputs and labels are known, and the task is to learn a function that accurately maps the inputs to output labels and *generalizes* to new, yet-unseen inputs. To realize the above approach for error localization as a practical tool, we have to solve four sub-problems.

(1) How can we acquire a *training set* of blame-labeled ill-typed programs?
(2) How can we *represent* blame-labeled programs as numeric vectors amenable to machine learning?
(3) How can we find *features* that yield predictive models?
(4) How can we use the models to give localized *feedback* to the programmer?

2.1 Step 1: Acquiring a Blame-Labeled Training Set

The first step is to gather a training set of ill-typed programs, where each erroneous sub-term is explicitly labeled. Prior work has often enlisted expert users to curate a set of ill-typed programs and then *manually* determine the correct fix [*e.g.* Lerner et al. 2007; Loncaric et al. 2016]. This method is suitable for *evaluating* the quality of a localization (or repair) algorithm on a small number (*e.g.* 10s–100s) of programs. However, in general it requires a great deal of effort for the expert to divine the original programmer's intentions. Consequently, is difficult to scale the expert-labeling to yield a dataset large enough (*e.g.* 1000s of programs) needed to facilitate machine learning. More importantly, this approach fails to capture the *frequency* with which errors occur in practice.

Solution: Interaction Traces. We solve both the scale and frequency problems by instead extracting blame-labeled data sets from *interaction traces.* Software development is an iterative process. Programmers, perhaps after a lengthy (and sometimes frustrating) back-and-forth with the type checker, eventually end up fixing their own programs. Thus, we instrumented the OCaml compiler to record this conversation, *i.e.* record the sequence of programs submitted by each programmer and whether or not it was deemed type-correct. For each ill-typed program in a particular programmer's trace, we find the *first subsequent* program in the trace that type checks and declare it to be the fixed version. From this pair of an ill-typed program and its fix, we can extract a *diff* of the abstract syntax trees, and then assign the blame labels to the *smallest* sub-tree in the diff.

Example. Suppose our student fixed the `sumList` program in Figure 1 by replacing `[]` with `0`, the diff would include only the `[]` expression. Thus we would determine that the `[]` expression (and *not* the `+` or the recursive call `sumList t`) is to blame.

2.2 Step 2: Representing Programs as Vectors

Next, we must find a way to translate highly structured and variable sized *programs* into fixed size n-dimensional numeric *vectors* that are needed for supervised classification. While the PL literature is full of different program representations, from raw sequences of tokens, to richly-structured abstract syntax trees (AST) or control-flow graphs, it is unclear how to embed the above into a vector space. Furthermore, it is unclear whether recent program representations that are amenable to one learning task, *e.g.* code completion [Hindle et al. 2012a; Raychev et al. 2014] or decompilation [Bielik et al. 2016; Raychev et al. 2015] are suitable for our problem of assigning blame for type errors.

Solution: Bags-of-Abstracted-Terms. We present a new representation of programs that draws inspiration from the theory of abstract interpretation [Cousot and Cousot 1977]. Our representation is parameterized by a set of *feature abstraction* functions, (abbreviated to feature abstractions) $f_1, \ldots, f_n$, that map terms to a numeric value (or just $\{0, 1\}$ to encode a boolean property). Given a set of feature abstractions, we can represent a single program's AST as a *bag-of-abstracted-terms* (BOAT) by: (1) decomposing the AST (term) t into a *bag* of its constituent sub-trees (terms) $\{t_1, \ldots, t_m\}$; and then (2) representing each sub-term t_i with the n-dimensional vector $[f_1(t_i), \ldots, f_n(t_i)]$. Our choice of working with ASTs is natural as that is the representation over which the type-checker operates.

Modeling Contexts. Each expression occurs in some surrounding *context*, and we would like the classifier to be able make decisions based on the context as well. The context is particularly important for our task as each expression imposes typing constraints on its neighbors. For example, a `+` operator tells the type checker that both *children* must have type `int` and that the *parent* must accept an `int`. Similarly, if the student wrote `h sumList t` *i.e.* forgot the `+`, we might wish to blame the application rather than `h` because `h` *does not* have a function type. The BOAT representation makes it easy to incorporate contexts: we simply *concatenate* each term's feature vector with the *contextual features* of its parent and children.

2.3 Step 3: Feature Discovery

Next, we must find a *good* set of features, that is, a set of features that yields predictive models. Our BOAT representation lets us iteratively solve this problem by just starting with a simple set of features, and then repeatedly adding more and more to capture important aspects needed to improve precision. Our set of feature abstractions captures the *syntax*, *types*, and *context* of each expression.

Syntax and Type Features. We start by observing that at the very least, the classifier should be able to distinguish between the `[]` and `+` expressions in Figure 1 because they represent different *syntactic* expression forms. We model this by introducing feature abstractions of the form is-`[]`, is-`+`, *etc.*, for each of a fixed number of expression forms. Modeling the syntactic class of an expression gives the classifier a basic notion of the relative frequency of blame assignment for the various program elements, *i.e.* perhaps `[]` is *empirically* more likely to be blamed than `+`. Similarly, we can model the *type* of each sub-expression with features of the form is-`int`, is-`bool`, *etc.*. We will discuss handling arbitrary, user-defined types in § 5.

Contextual Features: Error Slices. Our contextual features include the syntactic class of the neighboring expressions and their inferred types (when available). However, we have found that the most important contextual signal is whether or not the expression occurs in a minimal type error slice [Haack and Wells 2003; Tip and Dinesh 2001] which includes *a minimal subset* of all expressions that are necessary for the error to manifest. (That is, replacing any subterm with `undefined` or `assert false` would yield a well-typed program.) We propose to use type error slices to communicate to the classifier which expressions could *potentially* be blamed — a change to an expression outside of the minimal slice cannot possibly fix the type error. We empirically demonstrate that the type error slice is so important (§ 4.3) that it is actually beneficial to automatically discard expressions that are not part of the slice, rather than letting the classifier learn to do so. Indeed, this domain-specific insight is crucial for learning classifiers that significantly outperform the state-of-the-art.

Example. When Nate is tasked with localizing the error in the example program of Figure 1, the `[]` and `+` sub-terms will each be given their own feature vector, and we will ask the classifier to predict for each *independently* whether it should be blamed. Table 1 lists some of the sub-expressions of the example from Figure 1, and their corresponding feature vectors.

Table 1. Example Feature Vectors

Expression	Is-[]	Is-Match-List-P	Expr-Size	Type-Int-C1	Type-List	In-Slice
`[]`	1	1	1	0	1	1
`hd + sumList tl`	0	1	5	1	0	1
`sumList tl`	0	0	3	0	1	1
`tl`	0	0	1	0	1	0

A selection of the features we would extract from the `sumList` program in Figure 1. A feature is considered *enabled* if it has a non-zero value, and *disabled* otherwise. A "-P" suffix indicates that the feature describes the parent of the current expression, a "-C*n*" suffix indicates that the feature describes the *n*-th (left-to-right) child of the current expression. Note that, since we rely on a partial typing derivation, we are subject to the well-known traversal bias and label the expression `sumList tl` as having type $[\cdot]$. The model will have to learn to correct for this bias.

2.4 Step 4: Generating Feedback

Finally, having trained a classifier using the labeled data set, we need to use it to help users localize type errors in their programs. The classifier tells us whether or not a sub-term *should be* blamed (*i.e.* has the blame label) but this is not yet particularly suitable as *user feedback*. A recent survey of developers by Kochhar et al. [2016] found that developers are unlikely to examine more than around five potentially erroneous locations before falling back to manual debugging. Thus, we should limit our predictions to a select few to be presented to the user.

Solution: Rank Locations by Confidence. Fortunately, many machine learning classifiers produce not only a predicted label, but also a metric that can be interpreted as the classifier's *confidence* in its prediction. Thus, we *rank* each expression by the classifier's confidence that it should be blamed, and present only the top-k predictions to the user (in practice $k = 3$). This use of ranking to report the results of a program analysis is popular in other problem domains [see, *e.g.* Kremenek and Engler 2003]; we focus explicitly on the use of data-driven machine learning confidence as a ranking source. In § 4 we show that Nate's ranking approach yields a high-precision localizer: when the top three locations are considered, at least one matches an actual student fix 91% of the time.

3 LEARNING TO BLAME

In this section, we describe our approach to localizing type errors, in the context of λ^{ML} (Figure 2), a simple lambda calculus with integers, booleans, pairs, and lists. Our goal is to instantiate the blame function of Figure 3, which takes as input a Model of type errors and an ill-typed program e, and returns an ordered list of subexpressions from e paired with the confidence score C that they should be blamed.

A Model is produced by train, which performs supervised learning on a training set of feature vectors $\mathcal{V}$ and (boolean) labels $\mathcal{B}$. Once trained, we can evaluate a Model on a new input, producing the confidence C that the blame label should be applied. We describe multiple Models and their instantiations of train and eval (§ 3.3).

Of course, the Model expects feature vectors $\mathcal{V}$ and blame labels $\mathcal{B}$, but we are given program pairs. So our first step must be to define a suitable translation from program pairs to feature vectors and labels, *i.e.* we must define the extract function in Figure 3. We model features as real-valued functions of terms, and extract a feature vector for each *subterm* of the ill-typed program (§ 3.1). Then we define the blame labels for the training set to be

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & \mid & n \mid e + e \\
 & \mid & b \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \langle e, e \rangle \mid \texttt{match}\ e\ \texttt{with}\ \langle x, x \rangle \rightarrow e \\
 & \mid & [] \mid e :: e \mid \texttt{match}\ e\ \texttt{with} \begin{cases} [] \rightarrow e \\ x :: x \rightarrow e \end{cases} \\
n & ::= & 0, 1, -1, \ldots \\
b & ::= & \texttt{true} \mid \texttt{false} \\
t & ::= & \alpha \mid \texttt{bool} \mid \texttt{int} \mid t \rightarrow t \mid t \times t \mid [t]
\end{array}
$$

Fig. 2. Syntax of λ^{ML}

$$
\begin{array}{lcl}
\mathcal{V} & \doteq & [\mathcal{R}] \\
C & \doteq & \{r \in \mathcal{R} \mid 0 \leq r \leq 1\} \\
\text{features} & : & [e \rightarrow \mathcal{R}] \\
\text{label} & : & e \times e \rightarrow [e] \\
\text{extract} & : & [e \rightarrow \mathcal{R}] \rightarrow e \times e \rightarrow [\mathcal{V} \times \mathcal{B}] \\
\text{train} & : & [\mathcal{V} \times \mathcal{B}] \rightarrow \text{Model} \\
\text{eval} & : & \text{Model} \rightarrow \mathcal{V} \rightarrow C \\
\hline
\text{blame} & : & \text{Model} \rightarrow e \rightarrow [e \times C]
\end{array}
$$

Fig. 3. A high-level API for converting program pairs to feature vectors and labels.

the subexpressions that changed between the ill-typed program and its subsequent fix, and model blame as a function from a program pair to the set of expressions that changed (§ 3.2). The extract function, then, extracts features from each subexpression and computes the blamed expressions according to label.

3.1 Features

The first issue we must tackle is formulating our learning task in machine learning terms. We are given programs over λ^{ML}, but learning algorithms expect to work with *feature vectors* $\mathcal{V}$ — vectors of real numbers, where each column describes a particular aspect of the input. Thus, our first task is to convert programs to feature vectors.

We choose to model a program as a *set* of feature vectors, where each element corresponds an expression in the program. Thus, given the `sumList` program in Figure 1 we would first split it into its constituent sub-expressions and then transform each sub-expression into a single feature vector. We group the features into five categories, using Table 1 as a running example of the feature extraction process.

Local syntactic features. These features describe the syntactic category of each expression e. In other words, for each production of e in Figure 2 we introduce a feature that is enabled (set to 1) if the expression was built with that production, and disabled (set to 0) otherwise. For example, the Is-[] feature in Table 1 describes whether an expression is the empty list [].

We distinguish between matching on a list vs. on a pair, as this affects the typing derivation. We also assume that all pattern matches are well-formed — *i.e.* all patterns must match on the same type. Ill-formed match expressions would lead to a type error; however, they are already effectively localized to the match expression itself. We note that this is not a *fundamental* limitation, and one could easily add features that specify whether a match *contains* a particular pattern, and thus have a match expression that enables multiple features.

Contextual syntactic features. These are similar to local syntactic features, but lifted to describe the parent and children of an expression. For example, the Is-Match-List-P feature in Table 1 describes whether an expression's *parent* matches on a list. If a particular e does not have children (*e.g.* a variable x) or a parent (*i.e.* the root expression), we leave the corresponding features disabled. This gives us a notion of the *context* in which an expression occurs, similar to the *n-grams* used in linguistic models [Gabel and Su 2010; Hindle et al. 2012b].

Expression size. We also propose a feature representing the *size* of each expression, *i.e.* how many sub-expressions does it contain? For example, the Expr-Size feature in Table 1 is set to three for the expression `sumList tl` as it contains three expressions: the two variables and the application itself. This allows the model to learn that, *e.g.*, expressions closer to the leaves are more likely to be blamed than expressions closer to the root.

Typing features. A natural way of summarizing the context in which an expression occurs is with *types*. Of course, the programs we are given are *untypeable*, but we can still extract a *partial* typing derivation from the type checker and use it to provide more information to the model.

A difficulty that arises here is that, due to the parametric type constructors $\cdot \rightarrow \cdot$, $\cdot \times \cdot$, and $[\cdot]$, there is an *infinite* set of possible types — but we must have a *finite* set of features. Thus, we abstract the type of an expression to the set of type constructors it *mentions*, and add features for each type constructor that describe whether a given type mentions the type constructor. For example, the type `int` would only enable the `int` feature, while the type `int` $\rightarrow$ `bool` would enable the $\cdot \rightarrow \cdot$, `int`, and `bool` features.

We add these features for parent and child expressions to summarize the context, but also for the current expression, as the type of an expression is not always clear *syntactically*. For example, the expressions `tl` and `sumList tl` in Table 1 both enable Type-List, as they are both inferred to have a type that mentions $[\cdot]$.

Note that our use of typing features in an ill-typed program subjects us to *traversal bias* [McAdam 1998]. For example, the `sumList tl` expression might alternatively be assigned the type `int`. Our models will have to learn good localizations in spite this bias (see § 4).

Type error slice. Finally, we wish to distinguish between changes that could fix the error, and changes that *cannot possibly* fix the error. Thus, we compute a minimal type error *slice* for the program (*i.e.* the set of expressions that contribute to the error), and add a feature that is enabled for expressions that are part of the slice. The In-Slice feature in Table 1 indicates whether an expression is part of such a minimal slice, and is enabled for all of the sampled expressions except for `tl`, which does not affect the type error. If the program contains multiple type errors, we compute a minimal slice for each error.

In practice, we have found that In-Slice is a particularly important feature, and thus include a post-processing step that discards all expressions where it is disabled. As a result, the `tl` expression would never actually be shown to the classifier. We will demonstrate the importance of In-Slice empirically in § 4.3.

3.2 Labels

Recall that we make predictions in two stages. First, we use eval to predict for each subexpression whether it should be blamed, and extract a confidence score C from the Model. Thus, we define the output of the Model to be a boolean label, where “false” means the expression *should not* change and “true” means the expression *should* change. This allows us to predict whether any individual expression should change, but we would actually like to predict the *most likely* expressions to change. Second, we *rank* each subexpression by the confidence C that it should be blamed, and return to the user the top k most likely blame assignments (in practice $k = 3$).

We identify the fixes for each ill-typed program with an expression-level diff [Lempsink 2009]. We consider two sources of changes. First, if an expression has been removed wholesale, *e.g.* if $f\ x$ is rewritten to $g\ x$, we will mark the expression f as changed, as it has been replaced by g. Second, if a new expression has been inserted *around* an existing expression, *e.g.* if $f\ x$ is rewritten to $f\ x + 1$, we will mark the application expression $f\ x$ (but not f or x) as changed, as the + operator now occupies the original location of the application.

3.3 Learning Algorithms

Recall that we formulate type error detection at a single expression as a supervised classification problem. This means that we are given a training data set $S : [\mathcal{V} \times \mathcal{B}]$ of labeled examples, and our goal is to use it to build a *classifier*, *i.e.* a rule that can predict a label b for an input v. Since we apply the classifier on each expression in the program to determine those that are the most likely to be type errors, we also require the classifier to output a *confidence score* that measures how sure the classifier is about its prediction.

There are many learning algorithms to choose from, existing on a spectrum that balances expressiveness with ease of training (and of interpreting the learned model). In this section we consider four standard learning algorithms: (1) logistic regression, (2) decision trees, (3) random forests, and (4) neural networks. A thorough introduction to these techniques can be found in introductory machine learning textbooks [*e.g.* Hastie et al. 2009].

Below we briefly introduce each technique by describing the rules it learns, and summarize its advantages and disadvantages. For our application, we are particularly interested in three properties – expressiveness, interpretability and ease of generalization. Expressiveness measures how complex prediction rules are allowed to be, and interpretability measures how easy it is to explain the cause of prediction to a human. Finally ease of generalization measures how easily the rule generalizes to examples that are not in the training set; a rule that is not easily generalizable might perform poorly on an unseen test set even when its training performance is high.

Logistic Regression. The simplest classifier we investigate is logistic regression: a linear model where the goal is to learn a set of weights W that describe the following model for predicting a label b from a feature vector v:

$$\Pr(b = 1|v) = \frac{1}{1 + e^{-W^\top v}}$$

The weights W are learnt from training data, and the value of $\Pr(b|v)$ naturally leads to a confidence score C. Logistic regression is a widely used classification algorithm, preferred for its simplicity, ease of generalization, and interpretability. Its main limitation is that the prediction rule is constrained to be a linear combination of the features, and hence relatively simple. While this can be somewhat mitigated by adding higher order (quadratic or cubic) features, this often requires substantial domain knowledge.

Decision Trees. Decision tree algorithms learn a tree of binary predicates over the features, recursively partitioning the input space until a final classification can be made. Each node in the tree contains a single predicate of the form $v_j \leq t$ for some feature v_j and threshold t, which determines whether a given input should proceed down the left or right subtree. Each leaf is labeled with a prediction and the fraction of training samples that would reach it; the latter quantity can be interpreted as the decision tree's confidence in its prediction. This leads to a prediction rule that can be quite expressive depending on the data used to build it.

Training a decision tree entails finding both a set of good partitioning predicates and a good ordering of the predicates based on data. This is usually done in a top-down greedy manner, and there are several standard training algorithms such as C4.5 [Quinlan 1993] and CART [Breiman et al. 1984].

Another advantage of decision trees is their ease of interpretation — the decision rule is a white-box model that can be readily described to a human, especially when the tree is small. However, the main limitation is that these trees often do not generalize well, though this can be somewhat mitigated by *pruning* the tree.

Random Forests. Random forests improve generalization by training an *ensemble* of distinct decision trees and using a majority vote to make a prediction. The agreement among the trees forms a natural confidence score. Since each classifier in the ensemble is a decision tree, this still allows for complex and expressive classifiers.

The training process involves taking N random subsets of the training data and training a separate decision tree on each subset — the training process for the decision trees is often modified slightly to reduce correlation between trees, by forcing each tree to pick features from a random subset of all features at each node.

The diversity of the underlying models tends to make random forests less susceptible to the overfitting, but it also makes the learned model more difficult to interpret.

Neural Networks. The last (and most complex) model we use is a type of neural network called a *multi-layer perceptron* (see Nielsen 2015 for an introduction to neural networks). A multi-layer perceptron can be represented as a directed acyclic graph whose nodes are arranged in layers that are fully connected by weighted edges. The first layer corresponds to the input features, and the final to the output. The output of an internal node v is

$$h_v = g(\sum_{j \in N(v)} W_{jv} h_j)$$

where $N(v)$ is the set of nodes in the previous layer that are adjacent to v, W_{jv} is the weight of the (j, v) edge and h_j is the output of node j in the previous layer. Finally g is a non-linear function, called the activation function, which in recent work is commonly chosen to be the *rectified linear unit* (ReLU), defined as $g(x) = \max(0, x)$ [Nair and Hinton 2010]. The number of layers, the number of neurons per layer, and the connections between layers constitute the *architecture* of a neural network. In this work, we use relatively simple neural networks which have an input layer, a single hidden layer and an output layer.

A major advantage of neural networks is their ability to discover interesting combinations of features through non-linearity, which significantly reduces the need for manual feature engineering, and allows high expressivity. On the other hand, this makes the networks particularly difficult to interpret and also difficult to generalize unless vast amounts of training data are available.

4 EVALUATION

We have implemented our technique for localizing type errors for a purely functional subset of OCaml with polymorphic types and functions. We seek to answer three questions in our evaluation:

- **Blame Accuracy** How often does Nate blame a *correct* location for the type error? (§ 4.2)
- **Feature Utility** Which program *features are required* to localize errors? (§ 4.3)
- **Interpretability** Are the models *interpretable* using our intuition about the causes of type errors? (§ 4.4)

In the sequel we present our experimental methodology § 4.1 and then drill into how we evaluated each of the questions above. However, for the impatient reader, we begin with a quick summary of our main results:

1. Data Beats Algorithms. Our main result is that for type error localization, data is indeed unreasonably effective [Halevy et al. 2009]. Nate's most sophisticated *neural network*-based classifier's top-ranked prediction blames the correct sub-term 73% of the time which is a good 17 points higher than the state of the art SHErrLoc's 56%. However, even Nate's simple *logistic regression* based classifier is correct 60% of the time, *i.e.* 4 points better than SHErrLoc. When the top three predictions are considered, Nate is correct 91% of the time.

2. Slicing Is Critical. However, data is effective *only* when irrelevant sub-terms have been sliced out of consideration. In fact, perhaps our most surprising result is that type error slicing and local syntax alone yields a classifier that is 10 points better than OCaml and on par with SHErrLoc. That is, once we focus our classifiers on slices, purely syntactic features perform as well as the state-of-the-art.

3. Size Doesn't Matter, Types Do. We find that (after slices) typing features provide the biggest improvement in accuracy. Furthermore, we find contextual syntax features to be mostly (but not entirely) redundant with typing features, which supports the hypothesis that the context's *type* nicely summarizes the properties of the surrounding expressions. Finally, we found that the *size* of the sub-expression was not very useful. This was unexpected, as we thought smaller expressions would be simpler, and hence, more likely causes.

4. Models Learn Typing Rules. Finally, by investigating a few of the predictions made by the *decision tree*-based models, we found that the models do indeed capture some simple and intuitive rules for predicting well-typedness. For example, if the left child of an application is a function, then the application is likely correct.

4.1 Methodology

We answer our questions on two sets of data gathered from the undergraduate Programming Languages course at UC San Diego (IRB #140608). We recorded each interaction with the OCaml top-level system over the course of the first three assignments, capturing ill-typed programs and, crucially, their subsequent fixes. The first dataset comes from the Spring 2014 class (SP14), with a cohort of 46 students. The second comes from the Fall 2015 class (FA15), with a cohort of 56 students. The extracted programs are relatively small, but they demonstrate a range of functional programming idioms, *e.g.* higher-order functions and (polymorphic) algebraic data types.

Feature Selection. We extract a set of 282 features from each sub-expression in a student program, including:

(1) 45 local syntactic features. In addition to the syntax of λ^{ML}, we support the full range of arithmetic operators (integer and floating point), equality and comparison operators, character and string literals, and a user-defined arithmetic expressions. We discuss the challenge of supporting other types in § 5.
(2) 180 contextual syntactic features. For each sub-expression we additionally extract the local syntactic features of its parent and first, second, and third (left-to-right) children. If an expression does not have a parent or children, these features will simply be disabled. If an expression has more than three children, the classifiers will receive no information about the additional children.
(3) 55 typing features. In addition to the types of λ^{ML}, we support `int`s, `float`s, `char`s, `string`s, and the user-defined `expr` mentioned above. These features are extracted for each sub-expression and its context.
(4) One feature denoting the size of each sub-expression.
(5) One feature denoting whether each sub-expression is part of the minimal type error slice. We use this feature as a "hard" constraint, sub-expressions that are not part of the minimal slice will be preemptively discarded. We justify this decision in § 4.3.

Blame Oracle. Recall from § 3.2 that we automatically extract a blame oracle for each ill-typed program from the (AST) diff between it and the student's eventual fix. A disadvantage of using diffs in this manner is that students may have made many, potentially unrelated, changes between compilations; at some point the "fix" becomes a "rewrite". We do not wish to consider the "rewrites" in our evaluation, so we discard outliers where the fraction of expressions that have changed is more than one standard deviation above the mean, establishing a diff threshold of 45%. This accounts for roughly 14% of each dataset, leaving us with 2,425 program pairs for SP14 and 2,325 pairs for FA15.

Accuracy Metric. All of the tools we compare (with the exception of the standard OCaml compiler) can produce a list of potential error locations. However, in a study of fault localization techniques, Kochhar et al. [2016] show that most developers will not consider more than around five potential error locations before falling back to manual debugging. Type errors are relatively simple in comparison to general fault localization, thus we limit our evaluation to the top three predictions of each tool. We evaluate each tool on whether a changed expression occurred in its top one, top two, or top three predictions.

4.2 Blame Accuracy

In this experiment we compare the accuracy of our predictions to the state of the art in type error localization.

Baseline. We provide two baselines for the comparison: a random choice of location from the minimized type error slice, and the standard OCaml compiler.

State of the Art. Mycroft [Loncaric et al. 2016] localizes type errors by searching for a minimal subset of typing constraints that can be removed, such that the resulting system is satisfiable. When multiple such subsets exist it can enumerate them, though it has no notion of which subsets are *more likely* to be correct, and thus the order is arbitrary. SHErrLoc [Zhang and Myers 2014] localizes errors by searching the typing constraint graph for constraints that participate in many unsatisfiable paths and few satisfiable paths. It can also enumerate multiple predictions, in descending order of likelihood.

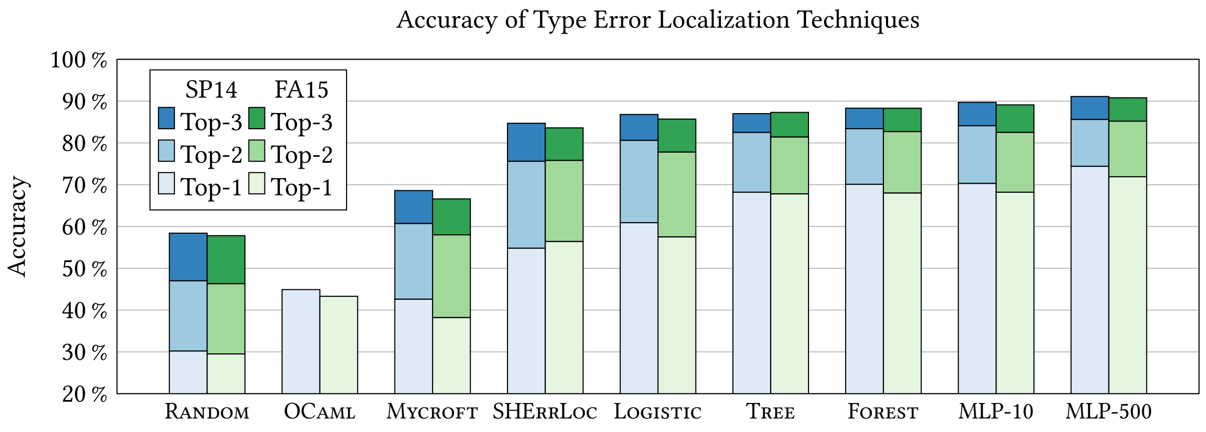


Fig. 4. Results of our comparison of type error localization techniques. We evaluate all techniques separately on two cohorts of students from different instances of an undergraduate Programming Languages course. Our classifiers were trained on one cohort and evaluated on the other. All of our classifiers outperform the state-of-the-art techniques Mycroft and SHErrLoc.

Comparing source locations from multiple tools with their own parsers is not trivial. Our experimental design gives the state of the art tools the "benefit of the doubt" in two ways. First, when evaluating Mycroft and SHErrLoc, we did not consider programs where they predicted locations that our oracle could not match with a program expression: 6–8% of programs for Mycroft and 3–4% for SHErrLoc. Second, we similarly ignored programs where Mycroft or SHErrLoc timed out (after one minute) or where they encountered an unsupported language feature: another 5% for Mycroft and 12–13% for SHErrLoc.

Our Classifiers. We evaluate five classifiers, each trained on the full set of features. Our classifiers are:

Logistic A logistic regression trained with a learning rate $\eta = 0.001$, an L_2 regularization rate $\lambda = 0.001$, and a mini-batch size of 200.

Tree A decision tree trained with the CART algorithm [Breiman et al. 1984] and an impurity threshold of 10^{-7} (used to avoid overfitting via early stopping).

Forest A random forest [Breiman 2001] of 30 estimators, trained with an impurity threshold of 10^{-7}.

MLP-10 and MLP-500 Two multi-layer perceptron neural networks, both trained with $\eta = 0.001$, $\lambda = 0.001$, and a mini-batch size of 200. The first MLP contains a single hidden layer of 10 neurons, and the second contains a hidden layer of 500 neurons. This gives us a measure of the complexity of the MLP's model, *i.e.* if the model requires many compound features, one would expect MLP-500 to outperform MLP-10. The neurons use rectified linear units (ReLU) as their activation function, a common practice in modern neural networks.

All classifiers were trained for 20 epochs on one dataset — *i.e.* they were shown each program 20 times — before being evaluated on the other. The logistic regression and MLPs were trained with the Adam optimizer [Kingma and Ba 2014], a variant of stochastic gradient descent that has been found to converge faster.

Results. Figure 4 shows the results of our experiment. Localizing the type errors in our benchmarks amounted, on average, to selecting one of 3 correct locations out of a slice of 10. Our classifiers consistently outperform the competition, ranging from 60% Top-1 accuracy (87% Top-3) for the Logistic classifier to 73% Top-1 accuracy (91% Top-3) for the MLP-500. Our baseline of selecting at random achieves 30% Top-1 accuracy (59% Top-3), while OCaml achieves a Top-1 accuracy of 45%. Interestingly, one only needs two *random* guesses to outperform

OCaml, with 47% accuracy. SHErrLoc outperforms both baselines, and comes close to our Logistic classifier, with 56% Top-1 accuracy (85% Top 3), while Mycroft underperforms OCaml at 40% Top-1 accuracy.

Surprisingly, there is little variation in accuracy between our classifiers. With the exception of the Logistic model, they all achieve around 70% Top-1 accuracy and around 90% Top-3 accuracy. This suggests that the model they learn is relatively simple. In particular, notice that although the MLP-10 has 50× *fewer* hidden neurons than the MLP-500, it only loses around 4% accuracy. We also note that our classifiers consistently perform better when trained on the FA15 programs and tested on the SP14 programs than vice versa.

4.3 Feature Utility

We have shown that we can train a classifier to effectively localize type errors, but which of the feature classes from § 3.1 are contributing the most to our accuracy? We focus specifically on feature *classes* rather than individual features as our 282 features are conceptually grouped into a much smaller number of *categorical* features. For example, the syntactic class of an expression is conceptually a feature but there are 45 possible values it could take; to encode this feature for learning we split it into 45 distinct binary features. Analyses that focus on individual features, *e.g.* ANOVA, are difficult to interpret in our setting, as they will tell us the importance of the binary features but not the higher-level categorical features. Thus, to answer our question we investigate the performance of classifiers trained on various subsets of the feature classes.

4.3.1 ***Type Error Slice.*** First we must justify our decision to automatically exclude expressions outside the minimal type error slice from consideration. Thus, we compare our classifiers on three sets of features:

(1) A baseline with only local syntactic features and no preemptive filtering by In-Slice.
(2) The features of (1) extended with In-Slice.
(3) The same features as (1), but we preemptively discard samples where In-Slice is disabled.

The key difference between (2) and (3) is that a classifier for (2) must *learn* that In-Slice is a strong predictor. In contrast, a classifier for (3) must only learn about the syntactic features, the decision to discard samples where In-Slice is disabled has already been made by a human. This has a few additional advantages: it reduces the set of candidate locations by a factor of 7 on average, and it guarantees that any prediction made by the classifier can fix the type error. We expect that (2) will perform better than (1) as it contains more information, and that (3) will perform better than (2) as the classifier does not have to learn the importance of In-Slice.

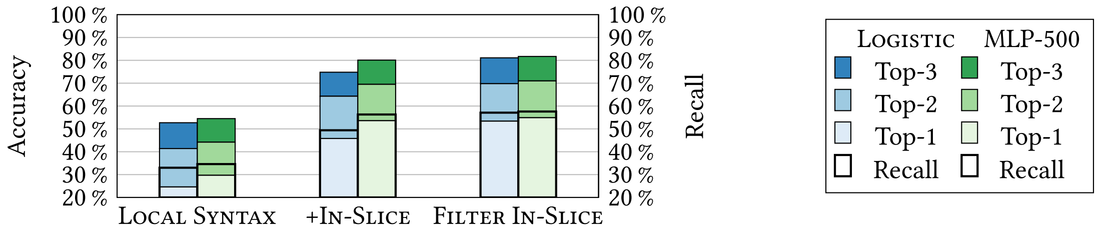
We tested our hypothesis with the Logistic and MLP-500[3] classifiers, cross-validated ($k = 10$) over the combined SP14/FA15 dataset. We trained for a single epoch on feature sets (1) and (2), and for 8 epochs on (3), so that the total number of training samples would be roughly equal for each feature set. In addition to accuracy, we report each classifier's *recall* — *i.e.* "How many true changes can we remember?" — defined as

$$\frac{|\text{predicted} \cap \text{oracle}|}{|\text{oracle}|}$$

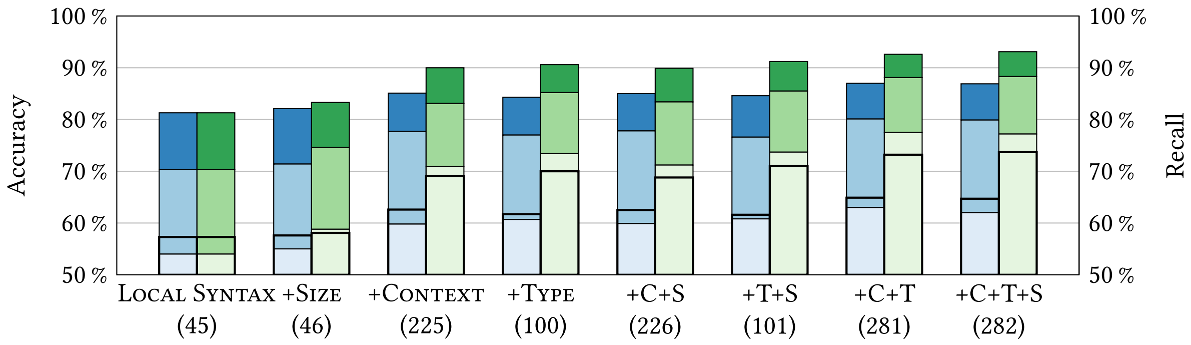
where predicted is limited to the top 3 predictions, and oracle is the student's fix, limited to changes that are in the type error slice. We make the latter distinction as: (1) changes that are not part of the type error slice are noise in the data set; and (2) it makes the comparison easier to interpret since oracle never changes.

Results. Figure 5a shows the results of our experiment. As expected, the baseline performs the worst, with a mere 28% Logistic Top-1 accuracy. Adding In-Slice improves the results substantially with a 46% Logistic Top-1 accuracy, demonstrating the importance of a minimal error slice. However, filtering out expressions that are not part of the slice *further* improves the results to 56% Logistic Top-1 accuracy. Interestingly, while the MLP-500 performs similarly poor with no error slice features, it recovers nearly all of its accuracy after being

[3]A layer of 500 neurons is excessive when we have so few input features — we use MLP-500 for continuity with the surrounding sections.



(a) Impact of type error slice on blame accuracy.



(b) Impact of contextual features on blame accuracy. The total number of features for each set is given in parentheses.

Fig. 5. Results of our experiments on feature utility.

given the error slice features. Top-1 accuracy jumps from 33% to 55% when we add In-Slice, and only improves by 3% when we filter out expressions that are not part of the error slice. Still, the accuracy gain comes at zero cost, and given the other benefits of filtering by In-Slice— shrinking the search space and guaranteeing our predictions are actionable — we choose to filter all programs by In-Slice.

4.3.2 ***Contextual Features.*** We investigate the relative impact of the other three classes of features discussed in § 3.1, assuming we have discarded expressions not in the type error slice. For this experiment we consider again a baseline of only local syntactic features, extended by each combination of (1) expression size; (2) contextual syntactic features; and (3) typing features. As before, we perform a 10-fold cross-validation, but we train for a full 20 epochs to make the differences more apparent.

Results. Figure 5b summarizes the results of this experiment. The Logistic classifier and the MLP-500 start off competitive when given only local syntactic features, but the MLP-500 quickly outperforms as we add features.

Expr-Size is the weakest feature, improving Logistic Top-1 accuracy by less than 1% and MLP-500 by only 3%. In contrast, the contextual syntactic features improve Logistic Top-1 accuracy by 4% (*resp.* 14%), and the typing features improve Top-1 accuracy by 6% (*resp.* 16%). Furthermore, while Expr-Size does provide some benefit when it is the only additional feature, it does not appear to provide any real increase in accuracy when added alongside the contextual or typing features. This is likely explained by *feature overlap*, *i.e.* the contextual features of "child" expressions additionally provide some information about the size of the subtree.

As one might expect, the typing features are more beneficial than the contextual syntactic features. They improve Top-1 accuracy by an additional 2%, and are much more compact — requiring only 55 typing features compared to 180 contextual syntactic features. This aligns with our intuition that types should be a good summary of the context of an expression. However, typing features do not appear to *subsume* contextual syntactic features, the MLP-500 gains an additional 4% Top-1 accuracy when both are added.

4.4 Interpreting Specific Predictions

Next, we present a *qualitative* evaluation that compares the predictions made by our classifiers with those of SHErrLoc. In particular, we demonstrate, with a series of example programs from our student dataset, how our classifiers are able to use past student mistakes to make more accurate predictions of future fixes. We also take this opportunity to examine some of the specific features our classifiers use to assign blame. For each example, we provide (1) the code; (2) SHErrLoc's prediction; (3) our Tree's prediction; and (4) an *explanation* of why our classifier made its prediction, in terms of the features used and their values. We choose the Tree classifier for this section as its model is more easily interpreted than the MLP. We also exclude the Expr-Size feature from the model used in this section, as it makes the predictions harder to motivate, and as we saw in § 4.3 it does not appear to contribute significantly to the model's accuracy.

We explain the predictions by analyzing the paths induced in the decision tree by the features of the input expressions. Recall that each node in a decision tree contains a simple predicate of the features, *e.g.* "is feature v_j enabled?", which determines whether a sample will continue down the left or right subtree. Thus, we can examine the predicates used and the values of the corresponding features to explain *why* our Tree made its prediction. We will focus particularly on the enabled features, as they generally provide more information than the disabled features. Furthermore, each node is additionally labeled with the ratio of "blamed" vs "not-blamed" training expressions that passed through it. We can use this information to identify particularly important decisions, *i.e.* we consider a decision that changes the ratio to be more interesting than a decision that does not.

4.4.1 ***Failed Predictions.*** We begin with a few programs where our classifier fails to make the correct prediction. For these programs we will additionally highlight the correct blame location.

Constructing a List of Duplicates. Our first program is a simple recursive function `clone` that takes an item `x` and a count `n`, and produces a list containing `n` copies of `x`.

```
let rec clone x n =
  let loop acc n =
    if n <= 0 then
      acc
    else
      clone ([x] @ acc) (n - 1) in
  loop [] n
```

The student has defined a helper function `loop` with an accumulator `acc`, likely meant to call itself tail-recursively. Unfortunately, she has called the top-level function `clone` rather than `loop` in the `else` branch, this induces a cyclic constraint `'a = 'a list` for the `x` argument to `clone`.

ES:♣ FIX: our first/second predictions are now swapped♣ Our classifier incorrectly predicts that the use of `x` in the recursive call is the most likely source of the error. Our second prediction coincides with SHErrLoc (and OCaml), blaming the the first argument to `clone`. This is also incorrect, but may be more helpful than our first prediction — if our student decides that she has certainly provided the correct *argument*, an alternative explanation is that perhaps she has called the wrong *function*.

We confess that both of these predictions are difficult to explain by examining the induced paths. In particular, both predictions only reference the expression's context, which is surprising. Much clearer is why we fail to blame the occurrence of `clone`, the two enabled features on the path are: (1) the parent is an application; and (2) `clone` has a function type. The model seems to have learned that programmers typically call the correct function.

Currying Considered Harmful? Our next example is another ill-fated attempt at `clone`.

```
let rec clone x n =
  let rec loop x n acc =
    if n < 0 then
      acc
    else
      loop (x, (n - 1), (x :: acc)) in
  loop (x, n, [])
```

The issue here is that OCaml functions are *curried* by default — *i.e.* they take their arguments one at a time — but our student has called the inner `loop` with all three arguments in a tuple. Many experienced functional programmers would choose to keep `loop` curried and rewrite the calls, however our student decides instead to *uncurry* `loop`, making it take a tuple of arguments. SHErrLoc blames the recursive call to `loop` while our classifier blames the tuple of arguments — a reasonable suggestion, but not the answer the student expected.

We fail to blame the definition of `loop` because it is defining a function. First, note that we represent `let f x y = e` as `let f = fun x -> fun y -> e`, thus a change to the pattern `x` would be treated as a change to the outer `fun` expression. With this in mind, we can explain our failure to blame the definition of `loop` (the outer `fun`) as follows: (1) it has a function type; (2) its child is a `fun`; and (3) its parent is a `let`. Thus it appears to the model that the outer `fun` is simply part of a function definition, a common and innocuous phenomenon.

4.4.2 *Correct Predictions.*
Next, we present a few indicative programs where our first prediction is correct, and all of the other tools' top three predictions are incorrect.

Extracting the Digits of an Integer. Consider first a simple recursive function `digitsOfInt` that extracts the digits of an `int`.

```
let rec digitsOfInt n =
  if n <= 0 then
    []
  else
    [n mod 10] @ [ digitsOfInt (n / 10) ]
```

Unfortunately, the student has decided to wrap the recursive call to `digitsOfInt` with a list literal, even though `digitsOfInt` already returns an `int list`. Thus, the list literal is inferred to have type `int list list`, which is incompatible with the `int list` on the left of the `@` (list append) operator. Both SHErrLoc and the OCaml compiler blame the recursive call for returning a `int list` rather than `int`, but the recursive call is correct!

As our Tree correctly points out (with high confidence), the fault lies with the list literal *surrounding* the recursive call, remove it and the type error disappears. An examination of the path induced by the list literal reveals that our Tree is basing its decision on the fact that (1) the expression is a list literal; (2) the child expression is an application, whose return type mentions `int`; and (3) the parent expression's type mentions `list`. Interestingly, Tree incorrectly predicts that the child application should change as well, but it is less confident of this prediction and ranks it below the correct blame assignment.

Padding a list. Our next program, padZero, is given two `int lists` as input, and must left-pad the shorter one with enough zeros that the two output lists have equal length. The student first defines a helper `clone`.

```
let rec clone x n =
  if n <= 0 then
    []
  else
    x :: clone x (n - 1)
```

Then she defines padZero with a branch to determine which list is shorter, followed by a `clone` to zero-pad it.

```
let padZero l1 l2 =
  let n = List.length l1 - List.length l2 in
  if n < 0 then
    (clone 0 ((-1) * n) @ l2, l2)
  else
    (l1, clone 0 n :: l2)
```

Alas, our student has accidentally used the `::` operator rather than the `@` operator in the `else` branch. SHErrLoc and OCaml correctly determine that she cannot cons the `int list` returned by `clone` onto `l2`, which is another `int list`, but they decide to blame the call to `clone`, while our Tree correctly blames the `::` constructor.

Examining the path induced by the `::`, we can see that our Tree is influenced by the fact that: (1) `::` is a constructor; (2) the parent is a tuple; and (3) the leftmost child is an application. We note that first fact appears to be particularly significant; an examination of the training samples that reach that decision reveals that, before observing the Is-Constructor feature the classifier is slightly in favor of predicting "blame", but afterwards it is heavily in favor of predicting "blame". Many of the following decisions change the balance back towards "no blame" if the "true" path is taken, but the `::` constructor always takes the "false" path. It would appear that our Tree has learned that constructors are particularly suspicious, and is looking for exceptions to this general rule.

Our Tree correctly predicts that the recursive call blamed by SHErrLoc should not be blamed; a similar examination suggests that the crucial observation is that the recursive call's parent is a data constructor application.

4.5 Threats to Validity

Although our experiments demonstrate that our technique can pinpoint type errors more accurately than the state of the art and that our features are relevant to blame assignment, our results may not generalize.

One threat to validity associated with supervised machine learning is overfitting (*i.e.* learning a model that is too complex with respect to the data). A similar issue that arises in machine learning is model stability (*i.e.* can small changes to the training set produce large changes in the model?). We mitigate these threats by: (1) using separate training and testing datasets drawn from distinct student populations (§ 4.2), demonstrating the generality of our models; and (2) via cross-validation on the joint dataset (§ 4.3), which demonstrates the stability of our models by averaging the accuracy of 10 models trained on distinct subsets of the data.

Our benchmarks were drawn from students in an undergraduate course and may not be representative of other student populations. We mitigate this threat by including the largest empirical evaluation of type error localization that we are aware of: over 4,500 pairs of ill-typed programs and fixes from two instances of the course, with programs from 102 different students. We acknowledge, of course, that students are not industrial programmers and our results may not translate to large-scale software development; however, we are particularly interested in aiding novice programmers as they learn to work inside the type system.

A related threat to construct validity is our definition of the immedate next well-typed program as the intended ground truth answer (see § 2, Challenge 2). Students may, in theory, submit intermediate well-typed program "rewrites" between the original ill-typed program and the final intended answer. Our approach to discarding outliers (see § 4) is designed to mitigate this threat.

Our removal of program pairs that changed too much, where our oracle could not identify the blame of the other tools, or where the other tools timed out or encountered unsupported language features is another threat to validity. It is possible that including the programs that changed excessively would hurt our models, or that the other tools would perform better on the programs with unsupported language features. We note however that (1) outlier removal is a standard technique in machine learning; and (2) our Top-1 accuracy margin is large enough that even if we assumed that SHErrLoc were perfect on all excluded programs, we would still lead by 9 points.

Examining programs written in OCaml as opposed to Haskell or any other typed functional language poses yet another threat, common type errors may differ in different languages. OCaml is, however, a standard target for research in type error localization and thus our choice admits a direct comparison with prior work. Furthermore, the functional core of OCaml that we support does not differ significantly from the functional core of Haskell or SML, all of which are effectively lambda calculi with a Hindley-Milner-style type system.

Finally, our use of student fixes as oracles assumes that students are able to correctly identify the source of an error. As the students are in the process of learning the language and type system, this assumption may be faulty. It may be that *expert* users would disagree with many of the student fixes, and that it is harder to learn a model of expert fixes, or that the state of the art would be better at predicting expert fixes. As we have noted before, we believe it is reasonable to use student fixes as oracles because the student is the best judge of what she *intended*.

5 LIMITATIONS

We have shown that we can outperform the state of the art in type error localization by learning a model of the errors that programmers make, using a set of features that closely resemble the information the type checker sees. In this section we highlight some limitations of our approach and potential avenues for future work.

User-Defined Types. Probably the single biggest limitation of our technique is that we have (a finite set of) features for specific data and type constructors. Anything our models learn about errors made with the `::` constructor or the `list` type cannot easily be translated to new, user-defined datatypes the model has never encountered. We can mitigate this, to some extent, by adding generic syntactic features for data constructors and match expressions, but it remains to be seen how much these help. Furthermore, there is no obvious analog for transferring knowledge to new type constructors, which we have seen are both more compact and helpful.

As an alternative to encoding information about *specific* constructors, we might use a more abstract representation. For example, instead of modeling `x :: 2` as a `::` constructor with a right child of type `int`, we might model it as some (unknown) constructor whose right child has an incompatible type. We might symmetrically model the `2` as an integer literal whose type is incompatible with its parent. Anything we learn about `::` and `2` can now be transferred directly to yet unseen types, but we run the risk generalizing *too much* — *i.e.* perhaps programmers make different mistakes with `lists` than they do with other types, and are thus likely to choose different fixes. Balancing the trade-off between specificity and generalizability appears to be a challenging task.

Additional Features. There are a number of other features that could improve the model's ability to localize errors, that would be easier to add than user-defined types. For example, each occurrence of a variable knows only its type and its immediate neighbors, but it may be helpful to know about *other* occurrences of the same variable. If a variable is generally used as a `float` but has a single use as an `int`, it seems likely that the latter occurrence (or context) is to blame. Similarly, arguments to a function application are not aware of the constraints imposed on them by the function (and vice versa), they only know that they are occurring in the context of an application. Finally, *n-grams* on the token stream have proven effective for probabilistic modeling of programming languages

[Gabel and Su 2010; Hindle et al. 2012b], we may find that they aid in our task as well. For example, if the observed tokens in an expression diverge from the n-gram model's predictions, that indicates that there is something unusual about the program at that point, and it may signal an error.

Independent vs Joint Predictions. We treat each sub-expression as if it exists in a vacuum, but in reality the program has a rich *graphical* structure, particularly if one adds edges connecting different occurrences of the same variable. Raychev et al. [2015] have used these richer models to great effect to make *interdependent* predictions about programs, *e.g.* de-obfuscating variable names or even inferring types. One could even view our task of locating the source of an error as simply another property to be predicted over a graphical model of the program. One of the key advantages of a graphical model is that the predictions made for one node can influence the predictions made for another node, this is known as *structured learning*. For example, if, given the expression `1 + true`, we predict `true` to be erroneous, we may be much less likely to predict + as erroneous. We compensate somewhat for our lack of structure by adding contextual features and by ranking our predictions by "confidence", but it would be interesting to see how structured learning over graphical models would perform.

6 RELATED WORK

In this section we describe two relevant aspects of related work: programming languages approaches to diagnosing type errors, and software engineering approaches to fault localization.

Localizing Type Errors. It is well-known that the original Damas-Milner algorithm $\mathcal{W}$ produces errors far from their source, that novices percieve as difficult to interpret [Wand 1986]. The type checker reports an error the moment it finds a constraint that contradicts one of the assumptions, blaming the new inconsistent constraint, and thus it is extremely sensitive to the order in which it traverses the source program (the infamous "left-to-right" bias [McAdam 1998]). Several alternative traversal have been proposed, *e.g.* top-down rather than bottom-up [Lee and Yi 1998], or a *symmetric* traversal that checks sub-expressions independently and only reports an error when two inconsistent sets of constraints are merged [McAdam 1998; Yang 1999]. Type error *slicing* [Haack and Wells 2003; Rahli et al. 2010; Tip and Dinesh 2001] overcomes the constraint-order bias by extracting a complete and minimal subset of terms that contribute to the error, *i.e.* all of the terms that are required for it to manifest and no more. Slicing typically requires rewriting the type checker with a specialized constraint language and solver, though Schilling [2011] shows how to turn any type checker into a slicer by treating it as a black-box. While slicing techniques guarantee enough information to diagnose the error, they can fall into the trap of providing *too much* information, producing a slice that is not much smaller than the input.

Finding Likely Errors. Thus, recent work has focused on finding the *most likely* source of a type error. Zhang and Myers [2014] use Bayesian reasoning to search the constraint graph for constraints that participate in many unsatisfiable paths and relatively few satisfiable paths, based on the intuition that the program should be mostly correct. Pavlinovic et al. [2014] translate the localization problem into a MaxSMT problem, asking an off-the-shelf solver to find the smallest set of constraints that can be removed such that the resulting system is satisfiable. Loncaric et al. [2016] improve the scalability of Pavlinovic et al. by reusing the existing type checker as a theory solver in the Nelson-Oppen [1979] style, and thus require only a MaxSAT solver. All three of these techniques support *weighted* constraints to incorporate knowledge about the frequency of different errors, but only Pavlinovic et al. use the weights, setting them to the size of the term that induced the constraint. In contrast, our classifiers learn a set of heuristics for predicting the source of type errors by observing a set of ill-typed programs and their subsequent fixes, in a sense using *only* the weights and no constraint solver. It may be profitable to combine both approaches, *i.e.* learn a set of good weights for one of the above techniques from our training data.

Explaining Type Errors. In this paper we have focused solely on the task of *localizing* a type error, but a good error report should also *explain* the error. Wand [1986], Beaven and Stansifer [1993], and Duggan and Bent [1996] attempt to explain type errors by collecting the chain of inferences made by the type checker and presenting them to the user. Gast [2004] produces a slice enhanced by arrows showing the dataflow from sources with different types to a shared sink, borrowing the insight of dataflows-as-explanations from MrSpidey [Flanagan et al. 1996]. Hage and Heeren [2006] catalog a set of heuristics for improving the quality of error messages by examining errors made by novices. Heeren et al. [2003], Christiansen [2014], and Serrano and Hage [2016] extend the ability to customize error messages to library authors, enabling *domain-specific* errors. Such *static* explanations of type errors run the risk of overwhelming the user with too much information, it may be preferable to treat type error diagnosis as an *interactive* debugging session. Bernstein and Stark [1995] extend the type inference procedure to handle *open* expressions (*i.e.* with unbound variables), allowing users to interactively query the type checker for the types of sub-expressions. Chitil [2001] proposes *algorithmic debugging* of type errors, presenting the user with a sequence of yes-or-no questions about the inferred types of sub-expressions that guide the user to a specific explanation. Seidel et al. [2016] explain type errors by searching for inputs that expose the *run-time* error that the type system prevented, and present users with an interactive visualization of the erroneous computation.

Fixing Type Errors. Some techniques go beyond explaining or locating a type error, and actually attempt to *fix* the error automatically. Lerner et al. [2007] searches for fixes by enumerating a set of local mutations to the program and querying the type checker to see if the error remains. Chen and Erwig [2014a] use a notion of *variation-based* typing to track choices made by the type checker and enumerate potential changes that would fix the error. They also extend the algorithmic debugging technique of Chitil by allowing the user to enter the expected type of specific sub-expressions and suggesting fixes based on these desired types [2014b]. Our classifiers do not attempt to suggest fixes to type errors, but it may be possible to do so by training a classifier to predict the syntactic class of each expression in the *fixed* program — we believe this is an exciting direction for future work.

Fault Localization. Given a defect, *fault localization* is the task of identifying "suspicious" program elements (*e.g.* lines, statements) that are likely implicated in the defect — thus, type error localization can be viewed as an instance of fault localization. The best-known fault localization technique is likely Tarantula, which uses a simple mathematical formula based on measured information from dynamic normal and buggy runs [Jones et al. 2002]. Other similar approaches, including those of Chen et al. [2002] and Abreu et al. [2006, 2007] consider alternate features of information or refined formulae and generally obtain more precise results; see Wong and Debroy [2009] for a survey. While some researchers have approached such fault localization with an eye toward optimality (*e.g.* Yoo et al. [2013] determine optimal coefficients), in general such fault localization approaches are limited by their reliance on either running tests or including relevant features. For example, Tarantula-based techniques require a normal and a buggy run of the program. By contrast, we consider incomplete programs with type errors that may not be executed in any standard sense. Similarly, the features available influence the classes of defects that can be localized. For example, a fault localization scheme based purely on control flow features will have difficulty with cross-site scripting or SQL code injection attacks, which follow the same control flow path on normal and buggy runs (differing only in the user-supplied data). Our feature set is comprised entirely of syntactic and typing features, a natural choice for type errors, but it would likely not generalize to other defects.

7 CONCLUSION

We have presented Nate, which combines modern statistical methods with domain-specific feature engineering to open the door to a new data-driven path towards precise error localization, significantly outperforming the state of the art on a new benchmark suite comprising 4,500 student programs. We found that while machine learning over syntactic features of each term in isolation performs worse than existing purely constraint-based approaches, augmenting the data with a single feature corresponding to the type error slice brings our classifiers up to par with the state of the art, and further augmenting the data with features of an expression's parent and children allows our classifiers to outperform the state of the art by 17 percentage points.

As with other forms of machine learning, a key concern is that of *data-set bias*: are Nate's models specific to our data set, would they fail on *other* programs? We address this concern in two ways. First, we partition the data by year, and show that models learned from one year generalize to, *i.e.* perform nearly as well on, the programs from the other year. Second, we argue that in our setting this bias is a *feature* (and not a bug): it allows Nate to *adapt* to the kinds of errors that programmers (specifically novices, who are in greatest need of precise feedback) actually make, rather than hardwiring the biases of experts who may suffer from blind spots. In this regard, we are particularly pleased that our classifiers can be trained on a modest amount of data, *i.e.* a single course's worth, and envision a future where each course comes equipped with a model of its students' errors.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Rui Abreu, Peter Zoeteweij, and Arjan J C van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC '06)*. 39–46. https://doi.org/10.1109/PRDC.2006.18

Rui Abreu, Peter Zoeteweij, and Arjan J C van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

Mike Beaven and Ryan Stansifer. 1993. Explaining Type Errors in Polymorphic Languages. *ACM Lett. Program. Lang. Syst.* 2, 1-4 (March 1993), 17–30. https://doi.org/10.1145/176454.176460

Karen L Bernstein and Eugene W Stark. 1995. *Debugging Type Errors.* Technical Report. State University of New York at Stony Brook.

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33rd International Conference on Machine Learning (ICML '16)*.

Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (1 Oct. 2001), 5–32. https://doi.org/10.1023/A:1010933404324

Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees.* CRC press.

M Y Chen, E Kiciman, E Fratkin, A Fox, and E Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings International Conference on Dependable Systems and Networks*. 595–604. https://doi.org/10.1109/DSN.2002.1029005

Sheng Chen and Martin Erwig. 2014a. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2535838.2535863

Sheng Chen and Martin Erwig. 2014b. Guided Type Debugging. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, 35–51. https://doi.org/10.1007/978-3-319-07151-0_3

Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/507635.507659

David Raymond Christiansen. 2014. Reflect on your mistakes! Lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*.

P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for the static analysis of programs. In *POPL 77*. ACM, 238–252.

Dominic Duggan and Frederick Bent. 1996. Explaining type inference. *Science of Computer Programming* 27, 1 (July 1996), 37–83. https://doi.org/10.1016/0167-6423(95)00007-0

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. 1996. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation (PLDI '96)*, Vol. 31. ACM, 23–32. https://doi.org/10.1145/249069.231387

Mark Gabel and Zhendong Su. 2010. A Study of the Uniqueness of Source Code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/1882291.1882315

Holger Gast. 2004. Explaining ML Type Errors by Data Flows. In *Implementation and Application of Functional Languages*. Springer Berlin Heidelberg, 72–89. https://doi.org/10.1007/11431664_5

Christian Haack and J B Wells. 2003. Type Error Slicing in Implicitly Typed Higher-Order Languages. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 284–301. https://doi.org/10.1007/3-540-36575-3_20

Jurriaan Hage and Bastiaan Heeren. 2006. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages*. Springer Berlin Heidelberg, 199–216. https://doi.org/10.1007/978-3-540-74130-5_12

Alon Halevy, Peter Norvig, and Fernando Pereira. 2009. The unreasonable effectiveness of data. *IEEE Intelligent Systems* 24, 2 (2009), 8–12.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer New York. https://doi.org/10.1007/978-0-387-84858-7

Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. 2003. Scripting the type inference process. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, Vol. 38. ACM, 3–13. https://doi.org/10.1145/944705.944707

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012a. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12).* IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012b. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12).* IEEE Press, Piscataway, NJ, USA, 837–847.

James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02).* ACM, New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397

Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. 1993. Teaching functional programming to first-year students. *J. Funct. Programming* 3, 01 (Jan. 1993), 49–65. https://doi.org/10.1017/S0956796800000599

Manu Jose and Rupak Majumdar. 2011. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. *SIGPLAN Not.* 46, 6 (June 2011), 437–446. https://doi.org/10.1145/1993316.1993550

Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. (22 Dec. 2014). arXiv:cs.LG/1412.6980

Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016).* ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/2931037.2931051

S B Kotsiantis. 2007. Supervised Machine Learning: A Review of Classification Techniques. *Informatica* 31, 3 (2007), 249–268.

Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Static Analysis*, Radhia Cousot (Ed.). Lecture Notes in Computer Science, Vol. 2694. Springer Berlin Heidelberg, Berlin, Heidelberg, 295–315. https://doi.org/10.1007/3-540-44898-5_16

Oukseh Lee and Kwangkeun Yi. 1998. Proofs About a Folklore Let-polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 707–723. https://doi.org/10.1145/291891.291892

Eelco Lempsink. 2009. *Generic type-safe diff and patch for families of datatypes.* Master's thesis. Universiteit Utrecht.

Benjamin S Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-error Messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07).* ACM, New York, NY, USA, 425–434. https://doi.org/10.1145/1250734.1250783

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A practical framework for type inference error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, 781–799. https://doi.org/10.1145/2983990.2983994

Bruce J McAdam. 1998. On the Unification of Substitutions in Type Inference. In *Implementation of Functional Languages (Lecture Notes in Computer Science)*, Kevin Hammond, Tony Davie, and Chris Clack (Eds.). Springer Berlin Heidelberg, 137–152. https://doi.org/10.1007/3-540-48515-5_9

Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10).* 807–814.

Greg Nelson and Derek C Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct. 1979), 245–257. https://doi.org/10.1145/357073.357079

Michael A Nielsen. 2015. *Neural Networks and Deep Learning.* Determination Press.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14).* ACM, New York, NY, USA, 525–542. https://doi.org/10.1145/2660193.2660230

John Ross Quinlan. 1993. *C4.5: Programs for Machine Learning.* Morgan Kaufmann.

Vincent Rahli, J B Wells, and Fairouz Kamareddine. 2010. *A constraint system for a SML type error slicer.* Technical Report HW-MACS-TR-0079. Herriot Watt University.

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15).* ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* ACM, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

Thomas Schilling. 2011. Constraint-Free Type Error Slicing. In *Trends in Functional Programming.* Springer Berlin Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

Eric L Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually Go Wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 228–242. https://doi.org/10.1145/2951913.2951915

Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 672–698. https://doi.org/10.1007/978-3-662-49498-1_26

F Tip and T B Dinesh. 2001. A Slicing-based Approach for Locating Type Errors. *ACM Trans. Softw. Eng. Methodol.* 10, 1 (Jan. 2001), 5–55. https://doi.org/10.1145/366378.366379

Mitchell Wand. 1986. Finding the Source of Type Errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '86)*. ACM, New York, NY, USA, 38–43. https://doi.org/10.1145/512644.512648

W Eric Wong and Vidroha Debroy. 2009. *A survey of software fault localization.* Technical Report UTDCS-45-09. University of Texas at Dallas.

Jun Yang. 1999. Explaining Type Errors by Finding the Source of a Type Conflict. In *Selected Papers from the 1st Scottish Functional Programming Workshop (SFP '99)*. Intellect Books, Exeter, UK, 59–67.

Shin Yoo, Mark Harman, and David Clark. 2013. Fault Localization Prioritization: Comparing Information-theoretic and Coverage-based Approaches. *ACM Trans. Softw. Eng. Methodol.* 22, 3 (July 2013), 19:1–19:29. https://doi.org/10.1145/2491509.2491513

Danfeng Zhang and Andrew C Myers. 2014. Toward General Diagnosis of Static Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 569–581. https://doi.org/10.1145/2535838.2535870