Learning User Friendly Type-Error Messages

BAIJUN WU, UL Lafayette, USA
JOHN PETER CAMPORA III, UL Lafayette, USA
SHENG CHEN, UL Lafayette, USA

Type inference is convenient by allowing programmers to elide type annotations, but this comes at the cost of often generating very confusing and opaque type error messages that are of little help to fix type errors. Though there have been many successful attempts at making type error messages better in the past thirty years, many classes of errors are still difficult to fix. In particular, current approaches still generate imprecise and uninformative error messages for type errors arising from errors in grouping constructs like parentheses and brackets. Worse, a recent study shows that these errors usually take more than 10 steps to fix and occur quite frequently (around 45% to 60% of all type errors) in programs written by students learning functional programming. We call this class of errors, *nonstructural errors*.

We solve this problem by developing Learnskell, a type error debugger that uses machine learning to help diagnose and deliver high quality error messages, for programs that contain nonstructural errors. While previous approaches usually report type errors on typing constraints or on the type level, Learnskell generates suggestions on the expression level. We have performed an evaluation on more than 1,500 type errors, and the result shows that Learnskell is quite precise. It can correctly capture 86% of all nonstructural errors and locate the error cause with a precision of 63%/87% with the first 1/3 messages, respectively. This is several times more than the precision of state-of-the-art compilers and debuggers. We have also studied the performance of Learnskell and found out that it scales to large programs.

CCS Concepts: • **Software and its engineering → Functional languages**; *Data types and structures*;

Additional Key Words and Phrases: Type error debugging, concrete messages, machine learning, structure-changing errors

ACM Reference Format:
Baijun Wu, John Peter Campora III, and Sheng Chen. 2017. Learning User Friendly Type-Error Messages. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 106 (October 2017), 29 pages. https://doi.org/10.1145/3133930

1 INTRODUCTION

Type systems are a useful tool for catching errors in programs. Type inference has made it easier to program in languages with static type systems, with even mainstream languages like C++, C#, and Java all adopting some form of type inference to eliminate cumbersome type annotations in code. Many functional languages like Haskell and OCaml have near-full type inference. Unfortunately, when type inference fails, understanding error messages and fixing type errors is challenging for both novice programmers [Chambers et al. 2012; Hage and Van Keeken 2006; Heeren 2005;

Authors' addresses: B. Wu, J. Campora III, and S. Chen, The Center for Advanced Computer Studies, School of Computing and Informatics, UL Lafayette. {bj.wu,campora,chen}@louisiana.edu.
Authors' addresses: Baijun Wu, The Center for Advanced Computer Studies, UL Lafayette, 301 E Lewis St, Lafayette, Louisiana, 70503, USA, bj.wu@louisiana.edu; John Peter Campora III, The Center for Advanced Computer Studies, UL Lafayette, 301 E Lewis St, Lafayette, Louisiana, 70503, USA, campora@louisiana.edu; Sheng Chen, The Center for Advanced Computer Studies, UL Lafayette, 301 E Lewis St, Lafayette, Louisiana, 70503, USA, chen@louisiana.com.

2475-1421/2017/10-ART106
https://doi.org/10.1145/3133930

Tirronen et al. 2015] and professional programmers [Lerner et al. 2007; Neubauer and Thiemann 2003]. Type inference can fail for various reasons, for example, using wrong library functions, using constants as functions, applying functions to arguments of wrong types, missing and having extra pairs of parentheses, and so on. We call type errors that are fixed by adding or removing parentheses, adding or removing brackets (or changes having the same effect, such as applying `head` to a variable to remove brackets), or a combination of them *nonstructural errors* and call all other errors *structural errors*.

Numerous approaches have been developed to deliver accurate and informative error messages in the last thirty years [Chen and Erwig 2014a,b; Chen et al. 2017; Chitil 2001; Eo et al. 2004; Haack and Wells 2003; Johnson and Walz 1986; Loncaric et al. 2016; McAdam 2002a; Pavlinovic et al. 2014; Zhang et al. 2015]. Most of them detect type errors by traversing program abstract syntax trees (ASTs), generating constraints relating different parts of ASTs (for example the argument type of the function must match the type of the argument), and locating inconsistencies among the generated constraints. Since parentheses and brackets directly control the structures of ASTs, nonstructural errors cause ASTs to be malformed.[1] Consequently, existing approaches don't work well for nonstructural errors [Wu and Chen 2017].

1.1 Issues with Non-structural Errors

While existing approaches work well for detecting structural errors, they perform poorly for fixing nonstructural errors. This is particularly troublesome because Wu and Chen [2017] show that these errors are common (comprising about 45% to 60% of all type errors observed in the study), and they often take more than 10 steps to fix.

To illustrate, let's consider the following example from the student program database [Hage and van Keeken 2009].

```
groepeer :: Int -> [Int] -> [[Int]]
groepeer n [] = []
groepeer n x  = take n x ++ groepeer n drop n x
```

This function tries to group sublists of length `n` of the given list `x`. The first type error is that, instead of `++`, `:` should be used. The second error is that a pair of parentheses is missing around `drop n x` in the subexpression `groepeer n drop n x`.

For this ill-typed program, GHC version 8.0.2, the de facto standard Haskell compiler, reports the following message. For simplicity, in the rest of this subsection, we only show the message that is related to the second type error, which is nonstructural.

```
* Couldn't match expected type `Int -> [Int] -> [Int]'
    with actual type `[[Int]]'
* The function `groepeer' is applied to four arguments,
    but its type `Int -> [Int] -> [[Int]]' has only two
    In the second argument of `(++)', namely `groepeer n drop n x'
    In the expression: take n x ++ groepeer n drop n
```

GHC infers that the recursive call to `groepeer` has too many arguments, but it fails to make further suggestions about how it could be fixed.

Helium [Heeren et al. 2003], a Haskell implementation that focuses on delivering user-friendly error messages for beginners, reports:

[1] More precisely, the AST for the program containing nonstructural errors is different from the AST that the user is intended.

```
(4,29): Type error in application
 expression       : groepeer n drop n x
 term             : groepeer
   type           : Int -> [Int]               -> [[Int]]
   does not match : Int -> (Int -> [a] -> [a]) -> Int -> [Int] -> [Int]
 because          : too many arguments are given
```

Helium delivers a similar message to GHC, though it's perhaps a bit more readable. Helium states that too many arguments are given to `groepeer`. Although the reason given by GHC and Helium provides the right direction for removing the type error, failing to provide concrete suggestions still poses some challenges to students. For example, it took students 8 steps to fix the type error in this program.

Seminal [Lerner et al. 2007] is one of the few error debuggers that deal with nonstructural errors, and it usually produces change suggestions to remove a type error. It does this by searching around the type error's location in the program AST and trying to remove, commute, or change nodes in the AST. We have translated this program into its equivalent OCaml form, and the error message generated by Seminal is below. The presentation of the message is altered by collapsing rows in order to save space.

```
Your code has several type errors. The following subexpressions are
type-correct on their own, but do not fit with the rest of the program:
    @
If you ignore other surrounding code, try replacing: groepeer
with: groepeer; [[...]]
The actual replacement has type
    int -> (int -> 'a list -> 'a list) -> int -> int list -> 'b
within context
    (fun n y -> (match y with
         | [] -> []
         | x -> ((take n x) @; [[...]] ((groepeer; [[...]]) n drop n x))))
```

For this program, Seminal fails to produce a helpful error message and fix. It suggests to change `groepeer` to a function that can take the four arguments it was erroneously provided, instead of figuring out that a pair of parentheses needed to be added over the larger subexpression `drop n x`.

1.2 Solving the Problem with Nonstructural Errors

From the previous error messages, we observe that it's hard to give good error messages by only considering type conflicts. We solve this problem by establishing a statistical relation between the program information around type errors (like the type conflicts and the tree structure) and how errors are fixed, through machine learning.

Figure 1 depicts the work flow of our approach, which includes a training phase and a prediction phase. In the training phase, we use the ill-typed programs from a student program database [Hage and van Keeken 2009] as our training data. We use feature vectors (Section 4) to store the information extracted from type errors. The feature extraction component can be implemented as a stand-alone algorithm or be implemented on top of some other error debugger or compiler, which we refer to as the underlying compiler. At the end of the training phase, a machine learning model is learned based on the extracted feature vectors.

With the trained model, we can determine whether the type error in a given user program is a nonstructural error or not. In the prediction phase, the classification result of the model and the feature vector are sent to the suggestion generation algorithm, so that a helpful error message for nonstructural errors can be generated.

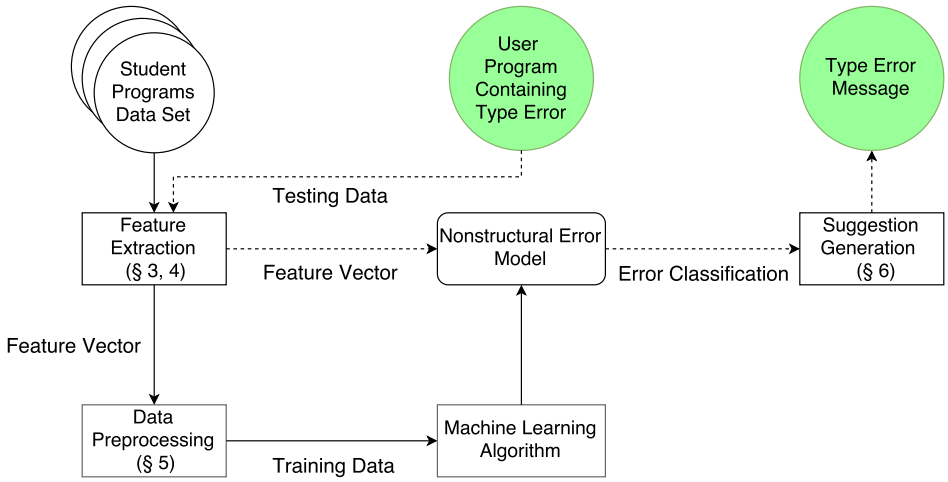


Fig. 1. The overview of Learnskell. The circles denote the input and output of the system. The solid and dashed arrows reflect information flow in the training and prediction stages, respectively.

Learnskell produces the following message for the aforementioned `groepeer` program. We have omitted the information about the expected type and the inferred type.

```
Add an open parenthesis before *drop* and a closing parenthesis after *x*
And needs some further changes
```

This message correctly diagnoses the nonstructural error and provides a concrete change suggestion to guide the user to fix the type error. To get past the barrier of using incorrect constraint information, the key idea in our approach is using machine learning to construct a model that can detect when type errors are likely to be nonstructural. This enables Learnskell to find nonstructural errors and provide concrete change suggestions for them, in a way that is largely orthogonal to other debugging approaches.

Overall, we make the following contributions in this paper.

(1) We have explored the idea of applying machine learning to generate better type error messages. Our experience is very positive, encouraging the adoption of machine learning to improve error reporting in compilers in general.
(2) We have developed an algorithm for computing the differences between two types when they fail to unify and prove its properties. As error debuggers always use, among others, the heuristic of type difference of conflicting types to rank multiple error locations [Chen and Erwig 2014a; Hage and Heeren 2007], our algorithm could be employed to develop more powerful error ranking heuristics.
(3) We have implemented Learnskell and fully evaluated its effectiveness and performance. Our evaluation on 1,537 type errors demonstrates that Learnskell captures 86% of nonstructural errors and locates the exact error cause and generates concrete suggestions in 63%/87% when considering the first 1/3 error messages, respectively, which is significantly higher than other compilers and error debuggers. Learnskell is scalable, generating suggestions within one second for 1,000 LOC Haskell programs. Our implementation can be accessed through the following link: https://bitbucket.org/plcacs/learnskell.

The rest of the paper is organized as follows. Section 2 motivates the use of machine learning in Learnskell. Section 3 shows how to calculate the difference of non-unifiable types provided by

the underlying compiler. Section 4 provides the details of feature vector. Section 5 presents various tricky issues of applying machine learning and our solutions to these problems. Section 6 describes the implementation of LEARNSKELL and Section 7 provides the evaluation results of our approach. Section 8 discusses the relation of our work with others, and Section 9 concludes.

2 MOTIVATION FOR USING MACHINE LEARNING

The goal of LEARNSKELL is to produce user-friendly type error messages for nonstructural errors. We achieve it by adopting the idea of machine learning. In this section, we discuss why machine learning can help to make full use of error information for nonstructural error determination. Along the way, we also gain insights about what error information we need to extract.

We use the following expressions to illustrate the discussion in this section.

```
v1 = map (not True) [False]
v2 = map True [False]
v3 = let func = not True
     in map func [False]
```

For `v1`, the type error is caused by the subexpression `not True`, with the expected type being `a -> b` and the inferred type being `Bool`. While these two types are non-unifiable, we observe that the expected type has two *spine*s, `a` and `b`, and the inferred type has one spine, `Bool`. Intuitively, a spine is the domain of a top-level arrow or the codomain of the last arrow at the top level. For example, the type `(a->b) -> [a] -> [b]` has three spines. A function with n spines can be applied to $n - 1$ arguments. Consequently, the fact that the expected type has one more spine than that of the inferred type in `v1` indicates that the erroneous subexpression has been applied to an extra argument. In other words, we may remove the type error by removing the argument `True`. Besides spine differences, bracket differences in types also provide insights about how type errors may be fixed.

While type differences help to infer possible fixes, they alone are insufficient. Consider, for example, the expressions `v2` and `v3` below `v1`. For both of them, the expected type and the inferred type are exactly the same as those for `v1`. Inferring to remove an argument, as we did in `v1`, is totally misleading. One can't remove an argument from `True` (the error cause in `v2`) or `func` (the error cause in `v3`). We can, however, address this issue by extracting program structure information around the error cause. For example, the location of the error cause in the program AST helps us to decide whether "Remove an argument" is an appropriate suggestion because we can't remove an argument if the error cause is a leaf. Moreover, other kinds of program structure information are also useful. For example, the fact that `func` is a let-bound variable indicates that the type error may be fixed by making `func` into a function if we view it as a point-free function [Peyton Jones 2003].

We may think of extracting all relevant information and constructing changing rules manually. However, this idea suffers from three shortcomings. First, it's likely that if we extract more information we can have better insights into diagnosing a given error. Unfortunately, the dimension of feature vectors could be high and similar errors can be represented by different feature combinations. This hinders us in deciding what kind of information is useful or not when constructing changing rules for errors. Second, it's very hard to come up with appropriate, consistent, and coherent rules that make best use of extracted information. For example, an error in a certain class could have many variants. In practice it's likely the rule we use to handle one error may not be able to handle its variants. Finally, the generated suggestions lack statistical meaning, yielding high ratio of misleading messages. For example, while the suggestion of removing the argument `True` to `not` in `v1` fixes the type error, the suggestion of changing the function `not` (to, for example, `const`) does so, too. The question is, which fix should we suggest?

All these problems could be solved nicely with machine learning. Based on the previous discussion, once a type error is identified by a compiler, we extract three kinds of information: (1) the type conflicts, (2) the program structure around the error location, and (3) the error messages from the underlying compiler. Extracting information from conflicting types is a tricky problem, and we investigate this problem in Section 3. We discuss the extraction of the latter two kinds of information in Section 4.

3 UNIFYING NON-UNIFIABLE TYPES

When a type error is detected, the underlying compiler usually provides only the expected type and inferred type of the identified error cause and the fact that these types fail to unify. Instead, we need to extract more meaningful information for machine learning from them even though they are non-unifiable.

Even though two types fail to unify, investigating the unification process still allows us to capture useful information. We use the following pair of types to illustrate our discussion below.

$$\text{expected type } (\tau_{exp}) : [\alpha] \to \texttt{Int} \qquad \to \texttt{Bool} \to \texttt{Int} \tag{1}$$

$$\text{inferred type } (\tau_{inf}) : \texttt{Int} \to (\texttt{Int} \to \texttt{Bool}) \to \alpha \tag{2}$$

The unification of these two types leads to two subproblems: $[\alpha] =^? \texttt{Int}$ and $\texttt{Int} \to \texttt{Bool} \to \texttt{Int} =^? \texttt{Int} \to \texttt{Bool} \to \alpha$. We use the notation $\tau_1 =^? \tau_2$ to denote the unification problem between τ_1 and τ_2. When solving the first subproblem, the unification fails since we can't unify a list type against `Int`. However, we observe that if we add [] over `Int` (more formally, applying the list type constructor to `Int`), then these types can be unified. One design decision we are facing is, should we record $\alpha \mapsto \texttt{Int}$ in the substitution returned from solving the first subproblem? There are reasons for choosing either to record or not to record. We decide to record because the error in the first unification problem is already fixed by adding [] and ignoring $\alpha \mapsto \texttt{Int}$ will not place correct constraint on later unification problems.

Solving the second unification subproblem by first substituting α with `Int`, we further have to solve the following subproblems, which we name as (3) and (4).

$$\texttt{Int} =^? \texttt{Int} \to \texttt{Bool} \tag{3}$$

$$\texttt{Bool} \to \texttt{Int} =^? \texttt{Int} \tag{4}$$

Both these subproblems fail to solve because we can't unify a function type with a non-function type. However, there is an important difference between them. Namely, the types constituting the problem (3) are the argument types of the original types and those constituting the problem (4) are the return types of the original types. This distinction is important because the former usually indicates that a function has been applied to wrong arguments and the latter indicates that a function is given not enough or too many arguments.

Overall, we compute four components from two conflicting types: (1) *TopFuncDiff*, which represents the top-level function difference (-1 in our example), (2) *TopBracDiff*, which represents the top-level bracket difference (0 in our example), (3) *FuncDiff*, a list of location and function difference pairs in argument types ([(2,1)] in our example), and (4) *BracDiff*, a list of location and bracket difference pairs in argument types ([] in our case).

3.1 Computing Type Differences

We develop an algorithm for computing type differences in Figure 2. This algorithm doesn't directly compute all the four components discussed earlier. Instead, we will show this computation towards the end of this subsection.

$\mathcal{U} : \tau \times \tau \times lev \times idx \rightarrow \theta \times TD$

(a) $\mathcal{U}(\alpha, \tau, lev, idx) \mid \alpha == \tau = (\emptyset, \emptyset)$
$\quad \mid \alpha \in FV(\tau) = \mathcal{D}(\alpha, \tau, lev, idx, 0)$
$\quad \mid \text{otherwise} = (\{\alpha \mapsto \tau\}, \emptyset)$

(b) $\mathcal{U}(\tau, \alpha, lev, idx) = \textbf{let } (\theta, TD) = \mathcal{U}(\alpha, \tau, lev, idx) \textbf{ in } (\theta, -TD)$

(c) $\mathcal{U}(\tau_1 \rightarrow \tau_2, \tau_3 \rightarrow \tau_4, lev, idx) = \textbf{let } (\theta_1, TD_1) = \mathcal{U}(\tau_1, \tau_3, lev + 1, idx)$
$\quad (\theta_2, TD_2) = \mathcal{U}(\theta_1(\tau_2), \theta_1(\tau_4), lev, next(lev, idx))$
$\quad \textbf{in } (\theta_2 \circ \theta_1, TD_1 \cup TD_2)$

(d) $\mathcal{U}([\tau_l], [\tau_r], lev, idx) = \mathcal{U}(\tau_l, \tau_r, lev, idx)$

(e) $\mathcal{U}(\tau_l, \tau_r, lev, idx) \mid \tau_1 == \tau_r \ = (\emptyset, \emptyset)$
$\quad \mid \text{otherwise} = \mathcal{D}(\tau_l, \tau_r, lev, idx, 0)$

$\mathcal{D} : \tau \times \tau \times lev \times idx \times depth \rightarrow \theta \times TD$

(f) $\mathcal{D}(\tau_l, \tau_r, lev, idx, depth)$
$\quad \mid numArity(\tau_l) \neq numArity(\tau_r) = (\emptyset, (\{(lev, idx, numArity(\tau_l) - numArity(\tau_r))\}, \emptyset))$

(g) $\mathcal{D}([\tau_b], \tau, lev, idx, depth) = \textbf{let } (\theta, TD) = \mathcal{D}(\tau, [\tau_b], lev, idx, depth) \textbf{ in } (\theta, -TD)$

(h) $\mathcal{D}(\tau, [\tau_b], lev, idx, depth) = \mathcal{D}(\tau, \tau_b, lev, idx, depth + 1)$

(i) $\mathcal{D}(\tau, \alpha, lev, idx, depth) = (\{\alpha \mapsto \tau\}, (\emptyset, \{(lev, idx, depth)\}))$

(j) $\mathcal{D}(\tau_l, \tau_r, lev, idx, depth) \mid \tau_l == \tau_r \ = (\emptyset, (\emptyset, \{(lev, idx, depth)\}))$
$\quad \mid \text{otherwise} = (\emptyset, \emptyset)$

Fig. 2. A type difference algorithm

The algorithm can be conceptually understood as two parts: the part that handles normal unification ($\mathcal{U}$) and the part that computes type difference once unification fails ($\mathcal{D}$). In the figure, we use the auxiliary function $FV(\tau)$ to compute the set of free type variables in τ and use the function $numArity(\tau)$ to compute the number of spines of τ. The type TD is a pair of two sets, for storing the results of arrow differences and bracket differences, respectively. The members of the set has the form (lev, idx, v), where lev and idx denote the level and the index at where the value v is computed. We will discuss more about lev and idx in the next paragraph. To simplify the presentation, we write $\emptyset$ for TD to denote that both sets of TD are empty. We use the set union operation to union two TDs by taking the union of their respective components. We also write $-TD$ to negate the third component of each set member in TD.

The function $\mathcal{U}$ takes four arguments, the types that are being unified (τ_l and τ_r) and the level lev and index idx that indicate the nesting level and the index of τ_1 and τ_2 in the original types, respectively. The variable lev helps us to distinguish whether the result computed is for top-level difference (if lev is 1) or for arguments (if lev is 2). It's rare for lev to reach more than 2 due to how underlying compilers report type errors. Given an ill-typed expression `f g x`, lev may reach 3 only if the type error is reported at `f` and the argument `g` itself is a higher-order function. However, usually in such cases debuggers report type error at `g` since this is a smaller expression that is ill-typed. When lev is 2, idx specifies argument at which the type difference is computed. This information is important because it tells where type mismatch happens. For example, if we are now solving the problem (3) arising from the problem $\tau_{exp} =^? \tau_{inf}$, then the lev and idx are 2 and 2, respectively. Similar, for the problem (4), they are 1 and 3, respectively. We use the function $next(lev, idx)$ to compute the index, which returns idx if $lev > 1$ and $idx + 1$ otherwise. We do this

to ensure that for $lev \leq 2$ the pair (lev, idx) uniquely determines the part of the type. When $\mathcal{U}$ is first called, both lev and idx are initialized to 1.

In addition to these four arguments, $\mathcal{D}$ takes another argument *depth*, indicating the current depth of the computed bracket difference. Both $\mathcal{U}$ and $\mathcal{D}$ return two values, a substitution mapping type variables to types and a *TD*.

We now briefly go through different cases of $\mathcal{U}$ and $\mathcal{D}$. The structure of $\mathcal{U}$ is similar to traditional unification algorithms, except that it doesn't terminate on unification failure but calls $\mathcal{D}$ for recording type differences. Case (a) unifies the type variable α against the type τ. If α is the same as τ, then we return $\emptyset$ for both the substitution and the *TD*. If the occurs check succeeds, then α and τ fail to unify, and we use $\mathcal{D}$ to record the difference. Otherwise, the unification succeeds and maps α to τ. Case (b) is a dual case of case (a). It delegates the real unification to case (a) and takes the negation of the *TD* returned by (a).

Case (c) unifies two function types. We unify their corresponding argument types and return types, respectively. We also pass in appropriate level and index values when solving subproblems. The composition of two substitutions has a conventional definition. When unifying two list types, we unify the underlying types wrapped in lists. This is handled by case (d). Finally, case (e) takes care of other situations. In particular, if two types are different, we record their differences using $\mathcal{D}$.

We now turn our attention to $\mathcal{D}$. First, in case (f), if two types have different numbers of spines, then we record function differences. The remaining cases of $\mathcal{D}$ handle bracket differences. It always assumes that its second argument has more brackets than its first argument. As a result, in case (g), if the first argument has more brackets, we call $\mathcal{D}$ with the types swapped and then take the negation of the computed *TD*. In case (h), if the second type is a list type, then the bracket difference is increased by one and $\mathcal{D}$ is recursively called. Note that in case (h), the first argument couldn't be a list type. Otherwise, the case would have been captured by case (d) of $\mathcal{U}$. Next, the case (i) handles the situation that the second argument is a type variable α. We solve this case by mapping α to the first argument and recording the bracket differences at the same time. Finally, if two types are the same, we record bracket differences. Otherwise, if τ_l and τ_r are different, we don't record any bracket difference. The reason is that the type error couldn't be fixed by adding or removing brackets.

Given a $TD = (F, B)$, the *TopFuncDiff* is v if $(1, idx, v) \in F$ for some idx and 0 otherwise. The *FuncDiff* collects all the pairs (idx, v) such that $(2, idx, v) \in F$. Similarly, we can compute *TopBracDiff* and *BracDiff* from B.

Usually error debuggers include the difference between two conflicting types as a heuristic to rank all potential error causes [Chen and Erwig 2014a; Hage and Heeren 2007]. The computations in these approaches are quite simple, for example, just computing the difference between the arities of conflicting types. Our algorithms $\mathcal{U}$ and $\mathcal{D}$ could be employed for that purpose and give more precise results.

3.2 Properties

In this subsection, we investigate the properties of $\mathcal{U}$ and $\mathcal{D}$. First, both of them are terminating, as captured in the following theorem.

Theorem 3.1. *For any two types τ_l and τ_r, $\mathcal{U}(\tau_l, \tau_r, 1, 1)$ terminates.*

Proof. We give a proof sketch based on four observations. First, in all cases of $\mathcal{U}$, except case (b), case (a) second guard, and case (e) second guard, the sizes of types to the recursive calls of $\mathcal{U}$ decrease. Moreover, the case (b) can only be called once and other two cases are handled by the termination of $\mathcal{D}$. Second, in all cases of $\mathcal{D}$, except case (g), the sizes of types to all recursive calls decrease. Moreover, case (g) can be called only once. Third, $\mathcal{D}$ doesn't call $\mathcal{U}$, so $\mathcal{U}$ and

$\mathcal{D}$ are not mutually recursive. Finally, all auxiliary functions are terminating. As a result, $\mathcal{U}$ is terminating. □

We want *TD* to capture only differences between types, though we don't record all information about the differences of types. We only try to capture information useful to type error debugging. Consequently, if the types are unifiable, then $\mathcal{U}$ returns an empty *TD*, as captured in the following theorem.

Theorem 3.2. *If τ_1 and τ_2 are unifiable, then $\mathcal{U}(\tau_1, \tau_2, 1, 1) = (\theta, \emptyset)$ for some θ.*

Proof. If two types are unifiable, then the conditions of case (a) second guard and (e) second guard can't be satisfied, and $\mathcal{D}$ will not be called. As a result, the *TD* will be empty. □

Unfortunately, the opposite of Theorem 3.2 doesn't hold. That is, if the $\mathcal{U}$ computes empty *TD*, it doesn't mean that the two types are unifiable. Also, *TD* doesn't capture unification failure caused by other type mismatches. However, $\mathcal{U}$ and $\mathcal{D}$ do capture authentic arrow differences and bracket differences. We express this result in the following theorem.

Theorem 3.3. *Given τ_1 and τ_2, let $\tau_{1p} = \tau_1|_{(lev, idx)}$, $\tau_{2p} = \tau_2|_{(lev, idx)}$, and $(\theta, (F, B)) = \mathcal{U}(\tau_1, \tau_2, 1, 1)$, then for any (lev, idx) with $lev < 3$ then $(lev, idx, f) \in F$ if and only if $numArity(\theta(\tau_{1p})) \neq numArity(\theta(\tau_{2p}))$. Similarly, $(lev, idx, b) \in B$ if $\theta(\tau_{1p})$ and $\theta(\tau_{2p})$ differ by b brackets.*

In the theorem, we write $\tau|_{(lev, idx)}$ to get the part of the type τ at level *lev* and index *idx*. The pair (lev, idx) determines the location when $lev < 3$. For example, $(2, 2)$ refers to the type `Bool` in type (2) and $(2, 2)$ refers to the first `Int` in the type (1). This theorem can be proved by an induction over each case of $\mathcal{U}$ and $\mathcal{D}$. The reason there is no "only if" clause for bracket difference is that recording function differences takes priority in our algorithm. As an example, for `Int → Bool` $=^?$ `[Int → Bool]`, the arity difference makes the function difference rules take priority, even though the types differ by 1 bracket.

4 THE FEATURE VECTOR

This section discusses the features we extracted from compiler error messages and program ASTs. We give an overview of the features and their value ranges in Figure 3. The first two features are extracted from Helium error messages, and thus the values may change when we consider another compiler or error debugger, for example GHC. All the remaining features are from program ASTs, thus independent of Helium. We can reuse the same values and meanings if we build our approach on top of another underlying compiler. In fact, all features 3 through 13 are not only compiler independent, but also language independent. We can thus anticipate a broad application of using our features in machine learning to improve error message reporting.

We discuss compiler dependent features in Section 4.1, compiler and language independent features in Section 4.2, and the language dependent feature in Section 4.3.

4.1 Compiler Dependent Features

Feature 1: Kind of the error message We consider the kind of the error message delivered by the underlying compiler, Helium. Information in the message, such as type errors existing in application or constructor, may correlate with the presence of nonstructural errors. Helium generates 13 different kinds of error messages, and we assign each kind a different value from 1 to 13. We present these message kinds in Appendix A. For example, the information "Type error in application" for the program `groepeer` represents one message kind of Helium, and the feature is assigned the value "1".

Feature ID	Summary	Feature Values
1	Kind of the error message from Helium.	Nominal (1 ~ 13)
2	Kind of change suggestion in the error message.	Nominal (1 ~ 3)
3	Type differences between the conflicting types	Integral (≥ 0)
4	Location of type inconsistency in AST.	Nominal (1 ~ 3)
5	Kind of the offending location.	Nominal (1 ~ 6)
6	Associativity of the operator in Feature 5.	Nominal (-1, 1, 2)
7	Precedence of the operator in Feature 5.	Integral(≥ -1)
8	Involvement of parentheses in the ill-typed expression.	Nominal (0, 1)
9	Involvement of brackets in the ill-typed expression.	Nominal (0, 1)
10	Function parameter style of the ill-typed binding.	Nominal (0, 1)
11	Occurrence frequency of the offending identifier.	Nominal (-1, 1, 2)
12	Index of the offending identifier in the ill-typed expression.	Nominal (-1 ~ 2)
13	Presence of operators with the same associativity.	Nominal (-1 ~ 1)
14	Presence of the `$` operator in the ill-typed expression.	Nominal (0, 1)

Fig. 3. An overview of the feature vector. A feature whose value is "Nominal" can take any value specified in the corresponding range, and a feature whose value is "Integral" can take any value satisfying the corresponding condition. We motivate the use of two different kinds of feature values in Section 5.2.

Feature 2: Kind of change suggestion in the error message We leverage the change suggestions given by the underlying compiler our approach is built on. Based on their concreteness, we classify change suggestions into 3 kinds, namely type-based, reason-based, and expression-based. In type-based messages, only the expected type and inferred type information are given. In reason-based messages, more information about the possible reason of the error cause is included. In expression-based messages, concrete information about how to remove the type error is included, for example, replacing a `concat` with `concatMap` [Hage and Heeren 2007]. For `groepeer`, Helium gives the reason that "Too many arguments are given", and the value for this feature is "2".

4.2 General Features

Hereafter, we use the notion *offending location* to refer to the position in the program AST that causes the type error. The offending location is reported by the underlying compiler.

Feature 3: Differences of the conflicting types We calculate the type differences between the expected type and the inferred type, as described in Section 3.1. The top-level function difference, top-level bracket difference, function difference, and bracket difference are the four sub-features of this feature. For example, for `groepeer`, the values of these sub-features are 0, 2, 0, and 2, respectively.

Feature 4: Location of type inconsistency in AST This feature identifies whether the offending location is a root (for example, when the error cause is an entire binding), an internal node (for example, when the error cause is a subexpression), or a leaf (for example, when the error cause is a variable reference or constant). These cases are given the values 1 to 3, respectively. In the function `groepeer`, the offending location is the identifier, `groepeer`, in the body of the second case of the function. As a result, we assign the value "3" to this feature for `groepeer`.

Feature 5: Kind of the offending location When the offending location is a leaf (that is the value of feature 4 is "3"), this feature identifies whether the error cause is a constant, a library function, a user defined function, an operator, or a reference to a parameter of the enclosing function definition. These cases are assigned the value 1 through 5, respectively. For example, for `groepeer`,

the feature value is "3" since the error cause, `groepeer`, is the function being defined by the user. In the following function, the offending location is `x` in the body. As a result, the value for this feature is "5" since it refers to the parameter `x`.

```
lengths :: [Char] -> [Int]
lengths x = map length x
```

In addition to the previous five cases, this feature has the value "6" if the offending location is an internal node. Take the following student program as an example:

```
type Table = [[String]]
lenTable :: Table -> [Int] -> Table
lenTable tabel lengte = lenTable ((head (transpose tabel)) (head lengte))
```

The program contains a nonstructural error: `(head lengte)` should be pulled out of the parentheses and be given as the second argument of `lenTable`. The offending location is the subexpression `head (transpose tabel)`. As a result, we assign the value "6" to this feature.

Feature 6: Associativity of Feature 5 If the offending location is an operator (that is the value of feature 5 is "4"), the value of this feature is "1" if the operator is left associative, or is "2" if the operator is right associative. Otherwise, the feature value is "-1". This feature provides some insights to determine whether an error is nonstructural or not when the ill-typed expression contains multiple operators with the same associativity. For example, consider the following student program [Hage and van Keeken 2009].

```
uitlijnen :: Int -> [String] -> [String]
uitlijnen _ [] = []
uitlijnen breedte (x:xs) = x ++ replicate (breedte - length x) ' '
                           : uitlijnen breedte xs
```

For this expression, the offending location is `++`, and the type error is fixed by adding an open parenthesis before `x` and a closing parenthesis after `' '`. Note that both `++` and `:` are right associative. Consequently, the body is interpreted by Haskell as

```
x ++ (replicate (breedte - length x) ' ' : uitlijnen breedte xs)
```

rather than the following

```
(x ++ replicate (breedte - length x) ' ') : uitlijnen breedte xs
```

This example shows why the associativity of the erroneous operator may correlate with nonstructural errors.

Feature 7: Precedence of Feature 5 If the offending location is an operator (that is the value of feature 5 is "4"), the value of this feature is the precedence of the operator in the language. Otherwise, the value of this feature is "-1". Some operators may be more related to a certain subclass (for example missing parentheses is a subclass and missing brackets is another subclass) of nonstructural errors, while other operators with the same associativity could be more related to other error subclasses. Knowing the precedence level allows us to differentiate those operators with the same associativity, which can help machine learning models decide whether an error is nonstructural or not and which subclass is the error. The feature values for `groepeer`, `lenTable`, and `uitlijnen` are "-1", "-1", and "5", respectively.

Feature 8: Involvement of parentheses Since by definition nonstructural errors occur due to the misuse of parentheses and brackets, the information about the presence of parentheses in the ill-typed expression should yield helpful information to detect nonstructural errors. For example, when operators like `:` and `++` are involved in the ill-type expression, the presence of parentheses around the error location may suggest nonstructural errors have happened. The value of this feature

is "1" if parentheses are present in the erroneous subexpression and "0" otherwise. For example, the feature values are "0", "1", and "1" for `groepeer`, `uitlijnen`, and `lenTable`, respectively.

Feature 9: Involvement of brackets We also record the presence of brackets, due to the same reason we record the presence of parentheses in feature 8. This feature has the value "1" if the erroneous expression contains brackets and "0" otherwise. The feature value is "0" for all `groepeer`, `uitlijnen`, and `lenTable`.

Feature 10: Function parameter style This feature differentiates the styles of the parameters of the binding (function definition) in the ill-typed expression. It is possible that certain styles are more closely related to nonstructural errors. For example, point-free definitions, which are common in functional programming, often lead to complicated function compositions, which, in turn, may cause nonstructural errors related to parentheses. The value of this feature is "1" if the function is point-free and "0" otherwise. For instance, the feature values are "0" for all `groepeer`, `uitlijnen`, and `lenTable`.

Feature 11: Occurrence frequency of the offending identifier This feature records the number of occurrences of the offending identifier in the ill-typed expression if the identifier is a function or operator. Intuitively, an offending identifier is less likely to be the real error cause if it appears several times in the expression. In such cases, the chance of nonstructural errors increases since changing the offending identifier itself may not fix the type error.

When the offending identifier is a function or operator, the feature value is "1" or "2" if the operator appears in the ill-typed expression exactly once or more than once, respectively. When the offending identifier is neither a function nor an operator, the feature value is "-1". For instance, the feature values are "1", "1", and "2" for `groepeer`, `uitlijnen`, and `lenTable`, respectively.

Feature 12: Index of the offending identifier If the offending identifier is a function or operator, then we record its index, with regard to its occurrence in the ill-typed expression. The value for this feature is "0", "1", or "2" if the offending identifier is a function or operator and it is the first, middle, or last in all the same identifiers, respectively. The feature value is "-1" if the offending identifier is not a function nor an operator. This feature may help to learn if a nonstructural error occurs more often when the offending identifier is at some index. The feature value is "0" for all `groepeer`, `uitlijnen`, and `lenTable`.

Feature 13: Presence of operators with the same associativity This feature records the presence of operators that share the same associativity of the offending operator on the same associative side. Specifically, if the offending operator is right associative, such as `++` or `:`, then this feature considers operators to the right of the offending operator. Otherwise, if the offending operator is left associative, this feature considers operators to the left. To see why this feature may be useful, consider the example `uitlijnen`, where the nonstructural error is caused by the presence of `:` at the right side of the right-associative offending identifier `++`. The value of this feature is "0" if the offending identifier is an operator and operators with the same associativity are not present and is "1" if operators with the same associativity are present. The value is "-1" if the offending identifier is not an operator. The feature values are "-1", "-1", and "1" for for `groepeer`, `lenTable`, and `uitlijnen`, respectively.

4.3 Language Dependent Features

Feature 14: Presence of the `$` operator The value of this feature is "0" if the expression does not contain `$` and "1" otherwise. This feature is relevant to nonstructural errors because the presence of `$` has the same effect of adding parentheses at appropriate locations.

Summary Overall, these 14 features provide a principled way to correlate type errors with program structures. They cover a wide array of useful error information, while avoiding adding complexity and excess dimensions to the feature vector. This enables us to apply machine learning to predict nonstructural errors effectively.

5 LEARNING NONSTRUCTURAL ERRORS

Machine learning can often deliver accurate solutions to problems that seem computationally intractable [Harmeling 2000; Moitra 2014]. In this section, we recast the problem of how to automatically diagnose type errors as a big data problem. Given a set of feature vectors of type errors and the corresponding class of each error, we apply a proper supervised machine learning (ML) algorithm to help us determine if a type error is structural or nonstructural. To evaluate whether an ML model is suitable to our application scenario, it is desirable to have a program set containing both structural and nonstructural errors. Moreover, it would be best to use a large data set with programs written by novices. This information enables ML to learn cues from various errors, which can provide significant value to type error debugging.

There has been work collecting student programs to investigate how beginners learned functional programming. The study by Hage and Van Keeken [2006] built a database with over 60k student programs in Haskell. While students were debugging errors, each program that was compiled with Helium was recorded. This large program data set provides a sufficient amount of type errors for us to construct the training data. By comparing the ill-typed and well-typed versions of a student program, we are able to find out how exactly a type error was fixed in practice, which can be used to guide us to label the class of each error in the training data. We filtered out 1,604 ill-typed programs in year 2003-2004 from the program data set. For every type error, we extracted the feature vector and manually labelled its error class to construct our training data.

In Section 5.1 we review some concepts about measuring the error rate of an ML model and the bias-variance tradeoff. In Section 5.2 we discuss how to reduce the error rate by preprocessing our training data. We resolve the imbalanced classification problem and use a cross-validation method to select the appropriate ML algorithm in Section 5.3

5.1 Preliminaries of Machine Learning

In Learnskell, we automatically extract a set of features for a given type error. We use FE to represent this feature extraction, which converts a type error x into a finite feature vector $v = FE(x)$. The data set is denoted by $D = \{(v_1, l_1), (v_2, l_2), ..., (v_i, l_i)\}$, where l_i is the label value of the error class for the type error x_i. The model $\mathcal{M}$ generated by an ML algorithm can interpret the relationship between any pair of v and l in D. The generalization error ε for the model $\mathcal{M}$ is estimated as follows [Domingos 2000].

$$\begin{aligned}\varepsilon(\mathcal{M}) &= \frac{1}{N}\sum_{i=1}^{N} P((v_i, p_i) \mid D) * L(l_i, p_i) \\ &= E[P((v, p) \mid D) * L(l, p)]\end{aligned}$$

We use $p = \mathcal{M}(v)$ to denote the prediction result produced by this model for any feature vector v in D. The loss function $L(l, p)$ is used to measure the cost of predicting p with respect to the label value l. For example, the zero-one loss function is commonly used in classification, where $L(l, p) = 0$ if $l = p$ and 1 otherwise.

Definition 5.1. Assume the number of all possible values of l is N_l, the optimal prediction for a feature vector v is $p_*(v) = \arg\min_{p'} E_l[L(l, p')]$, where $E_l[L(l, p')] = \frac{1}{N_l} \sum_{i=1}^{N_l} P(l_i \mid v) * L(l_i, p')$. The model $\hat{\mathcal{M}}$ is optimal if $\hat{\mathcal{M}}(v) = p_*$ for every v.

Definition 5.2. Assume the number of all possible values of p is N_p, the main prediction $p_m = \arg\min_{p'} E_p[L(p, p')]$, where $E_p[L(p, p')] = \frac{1}{N_p} \sum_{i=1}^{N_p} P(p_i \mid D) * L(p_i, p')]$. In other words, p_m is the most frequent prediction generated by $\mathcal{M}$.

We apply bias-variance decomposition [Domingos 2000] to $E[P((v, p) \mid D) * L(l, p)]$ and have:

$$\begin{aligned} \varepsilon(\mathcal{M}) &= L(p_*, p_m) + c_1 E[L(p, p_m)] + c_2 E[L(l, p_*)] \\ &= Bias(v) + c_1 Var(v) + c_2 Noise(v) \end{aligned}$$

The smaller the value of $\varepsilon(\mathcal{M})$ is, the better the ML model we obtain. The $Bias(v)$ denotes the systematic error of the model $\mathcal{M}$ by comparing with the optimal model $\hat{\mathcal{M}}$. The $Var(v)$ represents the fluctuations around the central tendency of prediction results with respect to data set D, which is related to the precision of the classes. The $Noise(v)$ is independent of the ML algorithm, and it is usually unavoidable when estimating an ML model.

To minimize the error rate in ML model, intuitively we shall minimize the bias and variance in the model at the same time. However, finding an optimal solution to minimize both bias and variance is very hard. The bias may be increased when reducing the variance, and vice versa. This is called bias-variance tradeoff [Geman et al. 1992]. For example, the variance is 0 in an ML model if this model always produces the same prediction result regardless of the training data. However, the bias in the model would be considerably large. Such a situation in ML is called underfitting. We can gradually increase the number of classes and eventually consider all type errors as one class. By doing so, we minimize the bias in the model, while the variance could be significant due to the large number of type error classes, which causes the ML model to overfit in this situation.

5.2 Data Preprocessing

In this section we first use feature selection techniques to distill the extracted features, and then we describe the strategy used to design the labels for different error classes.

In practice, most of the training data sets contain noise. Moreover, the potential bugs in the implementation of feature extraction and the possible mistakes in manually labelling data could intensify the noise. Reducing the noise is not trivial and most ML applications consider it as irreducible error. To reduce the value of $Noise(v)$, we try to handle the data noise in the training data, to some extent. We consider irrelevant and interdependent features as the noise, since removing these features usually leads to a simpler and more flexible ML model.

In Section 4 we showed that there are more than 10 features extracted to represent the program structure around the error location. Note that there exists dependencies among some features. For example, feature 5 denotes what kind of expression the offending node is. If the offending node is an operator, then we have feature 6 to describe the associativity of the operator. In addition, feature 7 shows the precedence of the offending node if it is an operator. Also, some feature values were redundant. For example, if the offending element is an expression, then its location in the AST is definitely at a node. We want to examine whether these attributes are relevant to the problem of type error debugging and to identify how important they are.

To perform feature selection, we split the training data into two sets. We applied SGC classifier with l_1-norm [Kearns and Vazirani 1994] to help us choose features on one set, and we evaluated the ML model on the other. Feature selection returns a subset of the relevant features and importance

ranking of all the features. Incorporating our domain knowledge about nonstructural errors and the understanding of the extracted features, we can use the result of feature selection to decide whether we should remove a feature. For example, we had a feature to keep track if the ill-typed expression contained more than one identifier. We found out that its feature importance was the smallest, and we further realized that the feature was redundant since its value can be inferred by the location of the error in AST and the type of the offending node. Therefore, it is safe to remove this feature. In fact, the classification accuracy of our ML model was improved by 4% thanks to that removal.

We can also handle the data noise by reducing the complexity of features. It is possible that some features had many values, but only one value was significant. For example, initially we kept track if a function binding had syntactic sugar for lists in its parameters. We checked to see if some parameter was a list with a head and a tail or if the function was point-free, etc. This created a large range of values for this feature, which hurt the ML performance. Eventually, we used a binary value to represent whether a function definition was point-free or not only.

So far we have handled the data noise in the designed feature vector. Next, we try to reduce the variance in the ML model. Learnskell focuses on handling nonstructural errors. Accordingly, we first classify type errors into structural and nonstructural errors, denoted by the error spaces $\mathcal{X}^{s}$ and $\mathcal{X}^{ns}$, respectively. Since fixing nonstructural errors requires restructuring the AST by changing parentheses or brackets, we further shatter $\mathcal{X}^{ns}$ in 4 sub-spaces, namely "add brackets" as $\mathcal{X}^{ns}_{1}$, "remove brackets" as $\mathcal{X}^{ns}_{2}$, "pull identifier from parentheses" as $\mathcal{X}^{ns}_{3}$, and "merge identifiers" as $\mathcal{X}^{ns}_{4}$. We can iteratively shatter each sub-space into another 2 new sub-spaces. For example, the space $\mathcal{X}^{ns}_{3}$ can be shattered in $\mathcal{X}^{ns}_{3_1}$(pull 1 identifier from parentheses), and $\mathcal{X}^{ns}_{3_{>1}}$(pull more than 1 identifiers from parentheses). Moreover, by combining rules from handling parentheses and brackets, we could shatter a space in other sub-spaces like $\mathcal{X}^{ns}_{3\&1}$(pull identifier from parentheses & add brackets), and $\mathcal{X}^{ns}_{3\&2}$(pull identifier from parentheses & remove brackets). For these data spaces that introduce more than one change to represent nonstructural errors, we do not consider them. This is because we can handle such errors by several fixing steps. Nevertheless, some sub-spaces with only one change, which have very few error cases, could also be negligible. For example, $\mathcal{X}^{ns}_{1_4}$ indicates adding 4 brackets to fix the nonstructural error, while in practice this rarely happens.

So when should we stop shattering the space $\mathcal{X}^{ns}$ and which sub-spaces we should assign labels to? Based on the concept of bias-variance tradeoff, the variance in the ML model decreases while the bias increases if we assign labels to the low dimensional spaces, and vice versa if we assign labels to the high dimensional spaces. Our strategy is to reduce the value of $Var(v)$ for the ML model first, and then we reduce the bias by selecting the most appropriate ML algorithm after data preprocessing. Therefore, we assign different labels to the four error spaces $\mathcal{X}^{ns}_{i}$, where $i \in [1, 2, 3, 4]$. Since changing the operator \$ has the same effect of adding or removing parentheses, we created a label to represent such nonstructural error space $\mathcal{X}^{ns}_{s}$. In Section 6, we discuss how our suggestion generation algorithm determines the number of brackets needs to be changed or the number of identifiers needs to be pulled/merged.

5.3 Imbalanced Classification and Model Selection

Most ML algorithms assume, by default, the data distribution of each class is balanced [He and Garcia 2009]. During data labelling, we found out the numbers of errors in class $\mathcal{X}^{ns}_{4}$ and $\mathcal{X}^{ns}_{s}$ are much smaller than other classes. The distribution ratio of these minority classes is only around 6%. Conventional ML models usually fail to provide good prediction accuracy across the classes when learning from imbalanced training data. To deal with the imbalanced classification problem, we can adopt a cost-sensitive ML model or resample the data set.

Cost-sensitive methods [Elkan 2001; Ting 2002] require a proper cost function to describe the penalty for misclassifying any type error. To achieve a good accuracy, we usually need to heavily tune the parameters in the cost function. In addition, for different ML models, we need to use different cost functions and repeating the tuning process. Such approaches aim to improve the efficiency of a certain ML algorithm when having imbalanced data, while our goal is to find an ML algorithm that provides the best classification accuracy. Therefore, we do not use cost-sensitive methods to handle the imbalanced classification problem in this work.

The essential idea of data resampling is to balance the given data set. This can be achieved with undersampling, which removes some existing data in the majority classes [Liu et al. 2009], or oversampling, which creates some new data in the minority classes [Chawla et al. 2002; Jo and Japkowicz 2004; Wang and Japkowicz 2004]. The resampling approach allows us to first balance the training data and then fairly evaluate the performance of different ML algorithms in an efficient way. Since we do not have an extremely large data set, applying undersampling methods to our data set may filter out many useful error cases, and consequently hurt the classification accuracy due to the underfitting problem. A naive way to oversample the imbalanced data is randomly replicating the records in the minority classes. However, this would lead to the overfitting problem.

To properly synthesize new data in the minority classes, the challenge is to make sure the generated data points do not destroy the characteristics of the structures of the original data set. Most of oversampling methods address this challenging by inserting the synthesized data point into an area of the imbalanced data space such that the similarity distances between the new point and its neighbours are small. Thus, the measurement of the similarity distance between two data points directly affects the quality of the oversampled data set.

SMOTE [Chawla et al. 2002] is a popular data oversampling technique. In SMOTE, each feature in the feature vector contributes to the calculation of similarity distance. As a result, SMOTE works well when all features are independent. However, there exists some dependency among the features in our feature vector. For example, the values of both feature 7 and 8 are partially dependent on the value of feature 5, as illustrated in Figure 3. Therefore, directly applying SMOTE in our case may not work well since the contributions of the interdependent features to the similarity distance could be over-considered. Instead of considering all features individually in computing the similarity distance, we use the common vector, which is an intersection of two feature vectors, to measure the similarity distance between two feature vectors. Intuitively, two feature vectors that share a large common vector have a small similarity distance. We formally developed our distance metric based on the idea of information entropy [Shannon 2001] of common vectors. We replaced the distance metric in SMOTE with our own distance metric. We refer to this new approach SMOTE-CV. We defer the detailed discussion of SMOTE-CV to Appendix B. We have compared SMOTE-CV and SMOTE and found out the classification accuracy of the ML algorithm with SMOTE-CV was about 6% higher than that with SMOTE. A more detailed comparison is also available in Appendix B.

For ML model selection, we used cross-validation [Kohavi et al. 1995] to evaluate the performances of different ML algorithms based on the training data. The average accuracy and standard deviation are 0.83 ± 0.08 for logistic regression, 0.64 ± 0.09 for SVM with a degree-2 polynomial kernel, 0.83 ± 0.06 for SVM with a linear kernel, 0.82 ± 0.08 for decision tree, and 0.85 ± 0.07 for random forest with 100 trees. Moreover, we tested random forest with 500 trees and did not observe any difference from random forest with 100 trees. Since random forest yields the best performance over the others, we use it as the model to predict nonstructural errors in this work.

6 IMPLEMENTATION

We have implemented a prototype of Learnskell using both Haskell and Python. We used Haskell for implementing feature extraction and suggestion generation. For each ill-typed program, we first

call Helium to get the relevant information, including the error cause and the expected type and inferred type of the error cause. Note that the location returned by Helium could be very different from the location LEARNSKELL reports. For example, for the `groepeer` example in Section 1, Helium reports the error at `groepeer` after `++`, while our approach reports it at `drop n x`. Based on the information returned by Helium, we apply the algorithms $\mathcal{U}$ and $\mathcal{D}$ from Section 3.1 to compute the type difference and implement other functions to extract the features listed in Section 4.

We implemented SMOTE-CV in Python, and it was used to resample the extracted feature vectors from the training data. We fed the feature vectors to the *scikit-learn* python package [Pedregosa et al. 2011], which generated the ML model for capturing nonstructural errors. For any feature vector of an ill-typed program, we provide it to the model, which will generate the classification result. The result includes two error classes, ordered by their corresponding probabilities.

For each class, we use the following rules to generate suggestions. We collect all of the suggestions and select the first three. If the type error is classified as a structural error, we use the corresponding error message generated by Helium directly. Otherwise, the suggestion generation module first checks the class of the nonstructural error. For the class "has `$`", we simply suggest to remove `$` first. If the class is "Merge" or "Pull" (the meanings of them are defined in Section 5.2), then only one message is generated. In this case, the indices of the first two elements in *FuncDiff* (defined in Section 3) are used to decide where to add the parentheses or pull arguments and the number of arguments to change. For example, for `groepeer`, the class of the second type error is "Merge" and the values of *FuncDiff* are 2 and 2, respectively, indicating that starting from the second argument two arguments should be removed. Therefore, we suggest adding parentheses over three arguments to fix the type error.

If the class is "Delete Brackets" or "Add Brackets", then we generate three suggestions. We generate more messages for dealing with bracket differences because there are numerous possibilities for fixing such errors while the fixing for parentheses errors is quite certain. In this case, the indices of the first two elements of *BracDiff* are used to generate suggestions. We suggest to change the expression specified by the first index, the first identifier of the error cause, and the parent node of the error cause. Of course, we may tune these rules to achieve better precision, but we wanted to keep change suggestions simple, given a classification result.

Sometimes the rule may fail to generate a suggestion with the given nonstructural error class. The reason is either the classification result is wrong, or the *FuncDiff* / *BracDiff* is empty. One example is that the predicted class is "Pull" but we don't have any expressions nested in parentheses. In such cases, we follow the suggestions from Helium.

7 EVALUATION

To evaluate our work, we trained our model on the Year-03 program set [Hage and van Keeken 2009], which contains 1,604 ill-typed programs. We evaluated the effectiveness of LEARNSKELL on the benchmark collected by Chen and Erwig [2014a] and the Year-02 program set. Since the first benchmark is very small, we briefly discuss the result in Section 7.3. The Year-02 contains 2,000 ill-typed programs and we chose 1,000 programs from them randomly for evaluation. Our evaluation focuses on this data set. The programs in Year-03 and Year-02 were written by different students in two different years. In these years, students worked on different programming exercises and assignments [Hage and van Keeken 2009].

Each error message generated by LEARNSKELL consists of three main parts: (1) the information indicates whether the type error is structural or nonstructural (this information is not shown to the user since it is not needed for fixing the type error), (2) the location of the type error in the program, and (3) a suggestion about how to remove the type error. We use the term "accuracy" when discussing the quality of ML classification and part (1) of messages, "precision" when discussing

the quality of part (2) of messages, and "concreteness" when taking about the quality of part (3) of messages.

Our evaluation focuses on four aspects. In Section 7.1, we present the accuracy of both machine learning classification and error messages. In Section 7.2, we compare the error locating precision of Learnskell with GHC and Helium, and in Section 7.3 we compare Learnskell with SHErrLoc. We do not compare Learnskell with Seminal [Lerner et al. 2007] because it works for OCaml only. As Wu and Chen [2017] showed that more concrete error messages tend to be more effective for students to fix type errors, we evaluate the concreteness of our error messages in Section 7.4. Finally, in Section 7.5, we measure the performance of Learnskell by considering large programs containing multiple type errors.

7.1 Learnskell Accuracy

To evaluate the accuracy of our trained ML model and suggestion generation in Learnskell, we first need to determine the ground truth of the error class of every type error in the testing data set. Similar to what we did in Section 5 for the training data set, we manually labeled each type error in the Year-02 program set, and used the label values to test against the evaluation results. Based on our label values, there are no type errors belonging to the "Has $" class in the testing data.

To measure the accuracy, we introduce the metrics precision (*pr*), recall (*re*), and F_1.

$$pr = \frac{t_p}{t_p + f_p} \qquad re = \frac{t_p}{t_p + f_n} \qquad F_1 = 2 * \frac{pr * re}{pr + re}$$

Given a set of errors belonging to a certain class, t_p (true positive) represents the number of errors correctly predicted by our ML model as belonging to the positive class, while f_p (false positive) is the number of errors misclassified by the ML model as belonging to the class. Similar to the definition of t_p and f_p, t_n (true negative) and f_n (false negative) represent the number of errors correctly and incorrectly predicted by our ML model as not belonging to the class, respectively. Therefore, given a type error, the metric *pr* is the probability that our model determines the class of this error correctly. The metric *re* indicates the ratio of type errors in a class our ML model can predict. F_1 score is the harmonic mean of *pr* and *re*, which is usually used to represent the overall classification accuracy of ML models.

Figure 4 shows the number of type errors in each class and the classification accuracy based on the testing data. Our model computes the probability of every class that a given type error belongs to, and we take the class with the highest probability as the classification result of the model. The accuracy of the ML model for capturing nonstructural errors is 80% and that for capturing structural errors is 84%. The overall accuracy for all the 1,537 errors is about 82%, which means that 82% of the time our ML model can correctly determine whether a type error is structural or nonstructural.

With the classification result generated by the ML model, Learnskell uses the first two error classes with the highest probabilities to generate error messages. Take the program `groepeer` as an example. The two most likely error classes and their probabilities are "Merge" (0.68) and "Structural" (0.25). Two error messages are generated based on the results. The first one suggests to add a pair of parentheses in the ill-typed expression, and the second one suggests to use Helium's message. Therefore, by considering the first 3 error messages, Learnskell can achieve a higher classification accuracy than the ML model does.

Figure 5 shows the accuracy of Learnskell for determining the classes of type errors for the Year-02 program set. About 88% of the time Learnskell can successfully identify the class of type errors. Note that Learnskell has higher missing rates in the "Merge" and "Pull" error classes. There are two reasons. First, the recall values of these two classes in Figure 4 are small, which

Error Class		# of Errors	Precision	Recall	F_1
Nonstuctural	Delete Brackets	230	0.76	0.87	0.81
	Merge	104	0.91	0.68	0.78
	Pull	69	0.73	0.67	0.70
	Add Brackets	304	0.81	0.86	0.83
	Overall	707	0.80	0.82	0.80
Structural		830	0.86	0.82	0.84
Overall		1,537	0.83	0.82	0.82

Fig. 4. Accuracy of ML model

Error Class		Precision	Recall	F_1
Nonstructural	Delete Brackets	0.81	0.93	0.87
	Merge	0.93	0.71	0.81
	Pull	0.80	0.70	0.75
	Add Brackets	0.85	0.96	0.90
	Overall	0.84	0.89	0.86
Structural		0.92	0.88	0.89
Overall		0.88	0.88	0.88

Fig. 5. Accuracy of Learnskell

indicates that many of the errors in these classes were not covered by our ML model. Second, we sometimes fail to generate suggestions for errors in these two classes.

7.2 Precision Comparison with GHC and Helium

In this section, we compare the error locating precision of Learnskell with Helium and GHC for the ill-typed programs in the Year-02 data set. We say a message is precise if it locates the type error correctly, matching the place where students changed to fix the type error. For example, based on how students fixed the type error in `groepeer` [Hage and van Keeken 2009], the message from Learnskell is precise while those from GHC and Helium are imprecise.

Figure 6 shows the precision of locating nonstructural errors of Learnskell, with one change suggestion, is 63%, while those of Helium and GHC are 33% and 21%, respectively. The precision of Learnskell rises to 74% if we consider the first two change suggestions and to 87% if we consider the first three. Helium and GHC can only generate one message for each type error. The overall error locating precision of Learnskell showed in Figure 6 is quite good compared to other tools. It achieves almost twice the precision of GHC and performs better than Helium. When considering 3 suggestions, Learnskell is the only tool achieving a precision higher than 50%.

In Figure 7, we compare Learnskell with Helium and GHC to see how frequently nonstructural errors of different classes can be precisely located. The result shows that Learnskell outperforms Helium and GHC in every error class, even when only considering the first change suggestion. Learnskell located the error causes for the errors involving parentheses changes more than 90% of the time. For errors involving bracket changes, about 60% of the time the first suggestion

Class	LEARNSKELL with # of Sugg. 1	2	3	Helium	GHC
Nonstructural	0.63	0.74	0.87	0.33	0.21
Overall	0.41	0.47	0.54	0.44	0.23

Fig. 6. Precision comparison with GHC and Helium

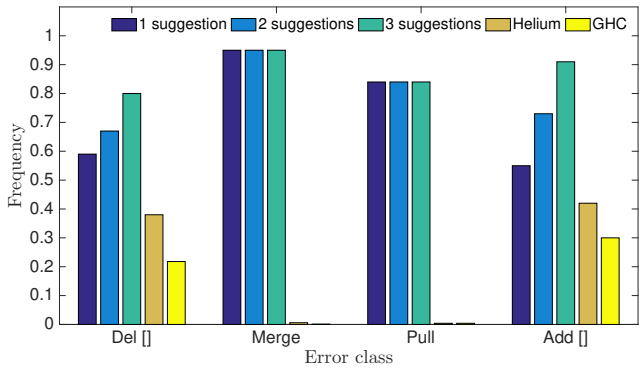


Fig. 7. Localization precision for nonstructural errors

Class	LEARNSKELL with # of Sugg. 1	2	3	SHErrLoc with 3 Sugg.	Helium	GHC
Nonstructural	0.69	0.76	0.82	0.47	0.22	0.13
Structural	-	-	-	0.65	0.51	0.21

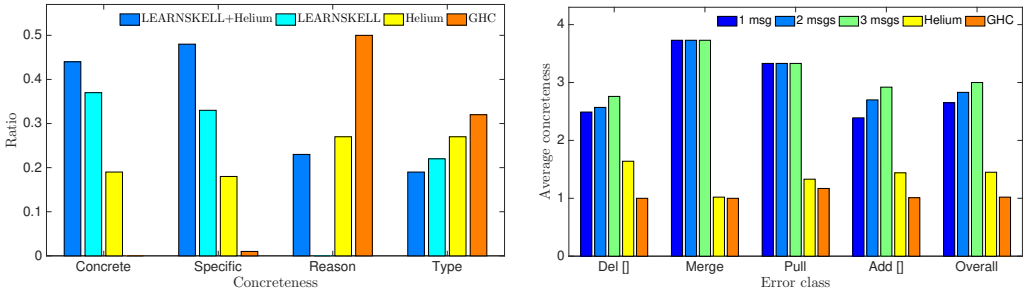
Fig. 8. LEARNSKELL precision with SHErrLoc, Helium and GHC

of LEARNSKELL correctly locates the error causes. When considering 3 suggestions, LEARNSKELL correctly finds the real error location for each class more than 80% of the time.

7.3 Precision Comparison with SHErrLoc

SHErrLoc is a state-of-the-art type error debugger that locates error causes quite well. Consequently, we are interested to evaluate LEARNSKELL against it. SHErrLoc provides an off-line tool for Ocaml, while there is only a web interface for compiling Haskell programs. Therefore, we lacked an easy way to automate the compilation process on a large program set. In this evaluation, we randomly chose 186 programs containing nonstructural errors from Year-02 data set. For each type error, we considered the first three error messages presented by SHErrLoc, leading to a fair comparison of LEARNSKELL and SHErrLoc.

Figure 8 measures the precision of locations reported in messages, similar to Figure 6. Although SHErrLoc performs well in handling nonstructural errors compared to Helium and GHC, LEARNSKELL outperforms it by more than 20%, considering the first suggestion of LEARNSKELL. With the first three suggestions, the precision of LEARNSKELL increases to 82%. This shows that LEARNSKELL addresses the nonstructural problem better than other leading tools.



(a) Concreteness ratio

(b) Concreteness level

Fig. 9. Concreteness of error messages

In Figure 8, we also present the results dealing with 86 programs that contain structural errors only. Since all these type errors are structural, LEARNSKELL did not generate any message. As a result, the entries for the suggestion categories are left empty. We found out that SHErrLoc performed much better than Helium and GHC when locating the real error cause for structural errors. LEARNSKELL and SHErrLoc are strong in different cases, therefore, combining SHErrLoc and our machine learning approach may achieve a better overall precision.

The benchmark from Chen and Erwig [2014a] contains 10 nonstructural errors. For them, LEARNSKELL precisely located 8 errors with the first message and 9 with the first three messages, while SHErrLoc located only 3 of them with the first three messages.

7.4 Suggestion Concreteness

To measure the concreteness of error messages, based on the information error messages include, we classify them into 4 categories, namely *Location*, *Reason*, *Specific*, and *Concrete*. Location messages only report the location and inconsistent types, which are also contained in the error messages of the other 3 categories. Reason messages explain why the type error occurred. Specific messages give a suggestion about how a user could debug the error without a concrete expression. Concrete messages provide the expression-level changes that fix the ill-type program. The study [Wu and Chen 2017] showed that more concrete messages tend to be more effective for users to exploit to fix type errors.

In this work, LEARNSKELL focuses on generating Specific or Concrete messages for nonstructural errors, while only providing Location messages for structural errors. However, Helium can provide Reason, Specific, or Concrete messages for structural errors. Therefore, a simple strategy for our tool to provide more concrete error messages is combining LEARNSKELL and Helium. If LEARNSKELL classifies a type error as structural, then we present the error message from Helium to users. Otherwise, we show the error message generated by LEARNSKELL.

Figure 9(a) shows the evaluation result of the concreteness of error messages. The value of y-axis represents the ratio of messages generated by different tools within a certain kind of suggestion. In Figure 9(a), the standalone LEARNSKELL generates Specific and Concrete suggestions more often than Helium and GHC do. In addition, by combining LEARNSKELL and Helium, we produce more Specific and Concrete error messages, which makes a more helpful tool for error debugging.

We assign a value 1 through 4 to each message if it is Location, Reason, Specific, or Concrete, respectively, that is we assign higher values to more concrete messages. We compute the average concreteness of the messages generated by a tool to give an intuitive idea about which message category it tends to generate. Figure 9(b) presents the concreteness result of different tools for

LOC	10 Errors			20 Errors			40 Errors		
	Extraction	Sugg.	Helium	Extraction	Sugg.	Helium	Extraction	Sugg.	Helium
100	0.006	0.008	1.234	0.012	0.016	2.496	0.026	0.031	4.895
500	0.006	0.008	2.496	0.012	0.016	3.919	0.025	0.031	6.104
1,000	0.006	0.008	4.732	0.013	0.016	5.427	0.026	0.033	8.542

Fig. 10. Running time of Learnskell in seconds. Each time is an average of 10 runs. Times are measured on a laptop with 4 GB of RAM and an AMD A6-3400M quad-core processor running 64 bit Fedora 23 and GHC 7.8.

handling nonstructural errors. For each error class, the average concreteness of GHC and Helium is between 1 and 2, meaning that they largely provide Location messages for all nonstructural errors. For the nonstructural errors involving parentheses changes, the error messages provided by Learnskell are almost entirely Specific and Concrete. For the other error classes, Learnskell reaches an average value of 3 when three messages are considered. The results show that, for different class of nonstructural errors, Learnskell can generate more concrete change suggestions than Helium and GHC do.

7.5 Learnskell Performance

The precision and concreteness of type error messages diminishes in value if error messages arrive too slowly. Users prize quick feedback when debugging type errors, and they will be less likely to continue using an error debugger if it takes a long time to provide error messages in practice. Therefore, scalability is an important factor to consider. We tested the efficiency of Learnskell, and show the results in Figure 10 by evaluating the extraction time and suggestion generation time on programs as large as 1,000 LOC, containing multiple nonstructural and structural errors. Each program contains the same number of nonstructural errors and structural errors. Suggestion generation always takes similar time, regardless of the number of lines of code. The Helium column in Figure 10 is the duration our implementation (that is built on Helium) ran for code that was not part of feature extraction and suggestion generation. The time Helium took increased with the number of lines of code, as well as the number of errors.

The training time for the ML model took around 3 minutes. This is of little impact since it is a one-time cost. The error classification time is usually under 0.01 seconds, which is similarly of little overhead, as with feature extraction and suggestion generation. These results show that the overhead in adding feature extraction and suggestion generation code into a type error debugger is small. As a result, the approach used by Learnskell is scalable, making it a practical design for a type error debugger intended to work on large applications.

8 RELATED WORK

In [Wu and Chen 2017], we studied the current research status of error debugging through an investigation of three Haskell data sets with more than 2,800 type errors, focusing on how type errors were fixed and what students did to fix them. In that work, we observed that while most existing error debuggers work well for type errors caused by single leaves, this class of type errors account for about, depending on the data set, only 30% to 45% of all errors. Nonstructural errors happened very often and usually took students more than 10 steps to fix. Motivated by that situation, this paper develops a solution to address the observed problem.

In the rest of this section, we discuss the relation between Learnskell and three other lines of work: approaches that handle nonstructural errors (Section 8.1), approaches that don't handle

nonstructural errors (Section 8.2), and other applications of machine learning to solve programming language problems (Section 8.3).

8.1 Methods Handling Nonstructural Errors

Helium [Hage and Heeren 2007] is a compiler that operates on a graph representation of the constraints it generates during type inference and uses heuristics to present the most likely error cause. One of these heuristics is the edit distance, which permutes leaves in an AST to reorder function arguments or parts of a tuple. It also tries to remove arguments or insert arguments and then decides whether the resulting program would type check [Heeren et al. 2003]. These heuristics work well in some cases. However, in many situations it reports that functions have too few or too many arguments erroneously, as we saw earlier in the `groepeer` example. The problem is that the constraint set generated is invalid, and these heuristics can't detect the actual error cause and make a suggestion that will fix it. LEARNSKELL doesn't suffer from this problem by learning a relation between error situations (including conflicting types and program structure around the error causes) and the correct fixes.

Seminal works by taking in an AST for the ill-typed program and then proceeding to run a searcher over the AST and making modifications that can remove nodes, change curried functions to tupled functions and vice versa, change a 1-element list containing an n-tuple to an n-element list, or commute leaf nodes [Lerner et al. 2007]. In presence of multiple errors, it can apply triaging [Lerner et al. 2007] to try and fix multiple type errors in different locations It then employs heuristics to rank these fixes. For example, it favors constructive changes over removals. The modifications Seminal applies help to diagnose type errors in many cases. However, Seminal suffers from some problems. First, it doesn't work well when multiple errors happen together because it's hard for Seminal to identify the error cause, despite its triaging method. Its approach to handling multiple related errors is to systematically focus on a particular expression and then remove sibling expressions until the type error is removed. Once the first removal combination is found, it proceeds to consider this case. This prevents exponential explosions in complexity that might come with considering all removal combinations, but can often miss other error fixes. For the `groepeer` example, it misses the error fix over `drop n x`. Second, Seminal doesn't scale well to large programs. For each modification of the program structure, Seminal calls the underlying OCaml compiler to check if the modification yields a well-typed programs. As program size increases, repeatedly calling the compiler becomes time consuming. LEARNSKELL doesn't suffer from these problems because we can always extract a feature vector for each type error and then generate the corresponding error class. For the `groepeer` example, LEARNSKELL generates a concrete suggestion of changing `++` to something else for the type error reported at the whole right hand side of the function.

McAdam [2002b] offers another approach to nonstructural errors, namely unification modulo linear isomorphism. This can be done with the help of AC-unification and rules for currying and uncurrying types, associating and commuting products of types, and some other rules. This approach can detect whether two types like $\tau_1 \to \tau_2 \to \tau_3$ and $\tau_2 \times \tau_1 \to \tau_3$ are isomorphic and then can detect what AST changes can cause a transformation from the actual type to the expected type. Due to the unification methods being used, this approach works the best if the given types already have the same primitive types. If one type has fewer spines, which is often the case when nonstructural error occurs, or if two types have different primitive types, which is the case when multiple type errors happen together, then this approach fails to deduce a nonstructural change. LEARNSKELL doesn't suffer from these problems because the computaion of type difference doesn't require any such rules to hold.

8.2 Approaches That Don't Handle Nonstructural Errors

The majority of existing error debugging approaches don't deal with nonstructural errors. Some approaches first generate constraints and then analyze them. For example, MinErrLoc [Pavlinovic et al. 2014, 2015] mixes user defined criteria with SMT solving to determine the most likely error cause, while Skalpel [Haack and Wells 2003] uses an error slicing methodology to present multiple locations that contribute to the error. All of these debuggers directly analyze type constraints in some manner. Other approaches might work on constraints during AST traversal, but they are still fundamentally constraint-based [Chitil 2001; Duggan and Bent 1995; Lee and Yi 1998, 2000; McAdam 2002b; Wand 1986; Yang 2000]. Chameleon doesn't explicitly work with type constraints. It instead uses Constraint Handling Rules to express type constraints. It simplifies the constraint sets until it finds minimal unsatisfiable constraint sets and reports the locations corresponding to the constraints in that set [Stuckey et al. 2003, 2004; Wazny 2006]. These methodologies work well for debugging structural errors, for example, changing single or a few leaves, but don't work well for nonstructural errors.

SHErrLoc [Zhang et al. 2015] uses a combination of Bayesian prediction and a constraint graph with an SMT solver to reason about type errors, along with a heuristic that favors messages that assume the programmer made less errors. This approach reasons only about the constraints that are generated by other compilers, for example, GHC for Haskell. This means that SHErrLoc is still unlikely to provide an accurate fixes for nonstructural errors. The evaluation in Figure 8 shows that Learnskell performs much better than SHErrLoc for nonstructural errors. Note that SHErrLoc performs better than the underlying compiler Helium, Learnskell and SHErrLoc may be combined to achieve high error locating accuracy.

8.3 Machine Learning on Programming Languages

There have been many approaches that use machine learning to solve programming language problems. Many of them exploit syntactic information of programs. One such work is code modeling and code completion for PHOG [Bielik et al. 2016a]. The work by Allamanis and Sutton [2013] mines a massive amount of code repositories to determine the productivity of programming languages. By considering the uniqueness of identifiers between code bases, this work determines whether code in a Javascript framework is utility or core logic [Allamanis and Sutton 2013]. Our work considers some syntactic information, for example program structures around the error cause, as part of the feature vector for machine learning. There has been some work that involves semantics, such as detecting semantic duplication of code [Sheneamer and Kalita 2016].

The program synthesis community has also adopted machine learning. The work by Ellis et al. [2015] provides a basis for unsupervised learning via program synthesis. DeepCoder [Balog et al. 2016] uses input and output examples and a DSL to learn to write basic programs for programming competitions. The work by Zhu et al. [2016] used machine learning to learn the shape of data structures. Heule et al. [2016] synthesized a semantics for x86-64 instructions by using program synthesis to learn bit-vector formulas, which will be leveraged by SMT solvers. By using program synthesis, Bielik et al. [2016b] constructed a static analyzer from a data set. It generated programs beyond the data set for use as counter-examples, which provide corner cases for the analyzer to verify that the static analysis generalizes well. The main difference between their work and ours is that they learn a static analysis by using runnable programs, while we use language semantics to determine why the input program can't run.

9 CONCLUSION

In this work, we proposed LEARNSKELL, a machine learning-based tool to provide informative error messages for nonstructural errors by learning from a large student program database. We evaluated LEARNSKELL by comparing it with other leading error debuggers and showed that our approach can locate nonstructural errors accurately and generate concrete fixes for them. We also showed that LEARNSKELL is scalable to large programs with multiple type errors. The situations that LEARNSKELL works well for are orthogonal to the underlying compiler, making it a promising method in developing error debuggers. In future, we plan to apply machine learning to improve error reporting in compilers in general.

A KINDS OF HELIUM MESSAGE

String contained in the message	Value
Type error in application.	1
Type error in constructor.	2
Type error in infix application.	3
Type error in literal.	4
Type error in right-hand side.	5
Type error in [variable | enumeration | element of list].	6
Type error in guard.	7
Type error in guarded expression.	8
Type signature is too general.	9
Don't know which instance to choose for [Eq | Num].	10
Type error in [negation | overaloaded function].	11
Type error in [explicitly typed binding | infix pattern application].	12
Other.	13

Fig. 11. Values for feature 1.

B THE DISTANCE MEASUREMENT

There are two general metrics called per-category feature importance (PCF) and cross-category feature importance (CCF) that can be used to measure the similarity distance between two data points [Creecy et al. 1992]. We use N_f to denote the number of extracted features, and N_c denotes the number of different classes. The basic PCF and CCF for two type error feature vectors, u and v, are defined as:

$$PCF(u,v) = \sum_{f}^{N_f} \omega(l_u, f) d(u_f, v_f) \quad where \;\; \omega(l,f) = P(l \mid f)$$

$$CCF(u,v) = \sum_{f}^{N_f} \omega(f) d(u_f, v_f) \qquad where \;\; \omega(f) = (\sum_{i=1}^{N_c} P(l_i \mid f)^2)^k$$

$$d(x,y) = \begin{cases} 1 & if\, x = y \\ 0 & if\, x \neq y \end{cases}$$

We can replace the distance function $d(x, y)$ with any other function, or replace ω with any other weight metric to compute the similarity of u and v. For example, SMOTE uses a different d and sets the parameter $k = 0.5$ for ω to construct its own similarity metric.

Note that for most of the similarity metrics based on PCF/CCF, they compute the contribution of each individual feature to the distance between two data point first, and then combine the contributions of all the features to generate the final result. However, calculating the similarity by following this manner does not work well in our case, since we have lots of interdependent features. For example, our features 6 and 7 are dependent on feature 5. Both values of feature 6 and 7 would be -1 if the offending node in feature 5 is not an operator. Therefore, using PCF/CCF-based methods would over consider the similarity contribution from the interdependent features. As a result, we proposed a new way to measure the similarity distance for a data set with dependent features.

Intuitively, the more similar two feature vectors are, the more number of features they should share. We use cv_{uv} to denote the common vector between two feature vectors u and v, and we want to study how much information of the similarity between u and v that the cv_{uv} can tell us. We use entropy $\delta = log_2(\frac{1}{P(cv_{uv})})$ to represent such information, where $P(cv_{uv})$ is the probability that the common vector cv_{uv} exists among the feature vectors. The entropy δ indicates that infrequent common vectors convey more information about the similarity than frequent common vectors. For example, if cv_{uv} appears more frequently in the feature vectors of the same error class, then it is reasonable to consider the two errors are less similar and vice versa. Therefore, we define the similarity distance between u and v within the class q as:

$$\begin{aligned} \delta'(u, v, q) &= H(cv_{uv} \mid q) \\ &= -\omega' log_2(\omega') - (1 - \omega') log_2(1 - \omega') \\ where\ \ \omega' &= P(cv_{uv} \mid q) \end{aligned}$$

The entropy δ' is similar to the usage of PCF, which only considers the distance between two errors within the same class. The study by Mohri and Tanaka [1994] showed that PCF is sensitive to the distribution of classes. Therefore, directly using δ' as a distance measurement may not be appropriate because of the imbalanced data in our application. To address this issue, we refer to the concept of CCF, which tries to measure the similarity of two data points across all the classes. Accordingly, we define the new distance metric as $\delta(u, v) = \sum_{i=1}^{N_c} P(q_i)\delta'(u, v, q_i)$, which averages δ' over all the error classes.

We implemented a new oversampling method with the distance metric δ, called SMOTE-CV, based on SMOTE. We evaluated the performance of SMOTE-CV by using random forest to classify the Year-03 program set. The result in Figure 12 shows that the performance for predicting errors in imbalanced classes is improved by adopting oversampling techniques. Although SMOTE outperforms than SMOTE-CV by about 1% on the classification of structural errors, SMOTE-CV yields better accuracy than SMOTE by about 6% when dealing with the minority classes like "Pull".

ACKNOWLEDGMENTS

We thank Yichu Li for providing insights on data preprocessing and useful feedback on an initial draft. We thank the anonymous OOPSLA reviewers, whose comments have improved both the presentation and content of this paper.

REFERENCES

Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA,

Error Class	Precision			Recall			F_1		
	Base	SMO	SMO-CV	Base	SMO	SMO-CV	Base	SMO	SMO-CV
Delete Brackets	0.769	0.773	0.807	0.671	0.731	0.819	0.800	0.740	0.804
Merge	0.858	0.824	0.853	0.925	0.944	0.906	0.887	0.878	0.873
Pull	0.875	0.825	0.930	0.510	0.570	0.625	0.623	0.662	0.715
Add Brackets	0.795	0.778	0.797	0.853	0.853	0.853	0.814	0.800	0.816
Has $	0.750	0.723	0.867	0.600	0.850	0.800	0.650	0.741	0.813
Structural	0.847	0.895	0.883	0.868	0.850	0.851	0.856	0.871	0.866

Fig. 12. A comparison of oversampling techniques. We use the short names SMO and SMO-CV for SMOTE and SMOTE-CV, respectively.

207–216. http://dl.acm.org/citation.cfm?id=2487085.2487127

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). http://arxiv.org/abs/1611.01989

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016a. PHOG: Probabilistic Model for Code. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 2933–2942. http://dl.acm.org/citation.cfm?id=3045390.3045699

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016b. Learning a Static Analyzer from Data. *CoRR* abs/1611.01752 (2016). http://arxiv.org/abs/1611.01752

Christopher Chambers, Sheng Chen, Duc Le, and Christopher Scaffidi. 2012. The function, and dysfunction, of information sources in learning functional programming. *Journal of Computing Sciences in Colleges* 28, 1 (2012), 220–226.

Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Int. Res.* 16, 1 (June 2002), 321–357. http://dl.acm.org/citation.cfm?id=1622407.1622416

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. DOI: http://dx.doi.org/10.1145/2535838.2535863

S. Chen and M. Erwig. 2014b. Guided Type Debugging. In *Int. Symp. on Functional and Logic Programming (LNCS 8475)*. 35–51.

Sheng Chen, Martin Erwig, and Karl Smeltzer. 2017. Exploiting diversity in type checkers for better error messages. *Journal of Visual Languages & Computing* 39 (2017), 10 – 21. DOI: http://dx.doi.org/10.1016/j.jvlc.2016.07.001 Special Issue on Programming and Modelling Tools.

Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *ACM Int. Conf. on Functional Programming*. 193–204.

Robert H Creecy, Brij M Masand, Stephen J Smith, and David L Waltz. 1992. Trading MIPS and memory for knowledge engineering. *Commun. ACM* 35, 8 (1992), 48–64.

Pedro Domingos. 2000. A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning. Stanford CA Morgan Kaufmann*. 231–238.

Dominic Duggan and Frederick Bent. 1995. Explaining Type Inference. In *Science of Computer Programming*. 37–83.

Charles Elkan. 2001. The foundations of cost-sensitive learning. In *International joint conference on artificial intelligence*, Vol. 17. Lawrence Erlbaum Associates Ltd, 973–978.

Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2015. Unsupervised Learning by Program Synthesis. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*. MIT Press, Cambridge, MA, USA, 973–981. http://dl.acm.org/citation.cfm?id=2969239.2969348

Hyunjun Eo, Oukseh Lee, and Kwangkeun Yi. 2004. Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Computing* 22, 1 (2004), 1–36. DOI: http://dx.doi.org/10.1007/BF03037279

Stuart Geman, Elie Bienenstock, and René Doursat. 1992. Neural networks and the bias/variance dilemma. *Neural computation* 4, 1 (1992), 1–58.

Christian Haack and J. B. Wells. 2003. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*. 284–301.

Jurriaan Hage and Bastiaan Heeren. 2007. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages*. 199–216.

Jurriaan Hage and Peter Van Keeken. 2006. Mining for Helium. *Technical report UU-CS* 2006-047 (2006).

Jurriaan Hage and Peter van Keeken. 2009. Neon: A Library for Language Usage Analysis. In *Software Language Engineering*. Lecture Notes in Computer Science, Vol. 5452. 35–53.

Stefan Harmeling. 2000. Solving Satisfiability Problems with Genetic Algorithms. (2000).

Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* 21, 9 (2009), 1263–1284.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell '03)*. ACM, New York, NY, USA, 62–71. DOI: http://dx.doi.org/10.1145/871895.871902

Bastiaan J. Heeren. 2005. *Top Quality Type Error Messages*. Ph.D. Dissertation. Universiteit Utrecht, The Netherlands. http://www.cs.uu.nl/people/bastiaan/phdthesis

Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. *SIGPLAN Not.* 51, 6 (June 2016), 237–250. DOI: http://dx.doi.org/10.1145/2980983.2908121

Taeho Jo and Nathalie Japkowicz. 2004. Class imbalances versus small disjuncts. *ACM Sigkdd Explorations Newsletter* 6, 1 (2004), 40–49.

Gregory F. Johnson and Janet A. Walz. 1986. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symp. on Principles of Programming Languages*. 44–57. DOI: http://dx.doi.org/10.1145/512644.512649

Michael J. Kearns and Umesh V. Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA.

Ron Kohavi and others. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Stanford, CA, 1137–1145.

Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. on Programming Languages and Systems* 20, 4 (July 1998), 707–723. DOI: http://dx.doi.org/10.1145/291891.291892

Oukseh Lee and Kwangkeun Yi. 2000. *A Generalized Let-Polymorphic Type Inference Algorithm*. Technical Report. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology.

B. Lerner, M. Flower, Dan Grossman, and Craig Chambers. 2007. Searching for type-error messages. In *ACM Int. Conf. on Programming Language Design and Implementation*. 425–434. DOI: http://dx.doi.org/10.1145/1250734.1250783

Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. 2009. Exploratory undersampling for class-imbalance learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39, 2 (2009), 539–550.

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. In *OOPSLA*. 781–799.

Bruce McAdam. 2002a. Trends in Functional Programming. Intellect Books, Exeter, UK, UK, Chapter How to Repair Type Errors Automatically, 87–98. http://dl.acm.org/citation.cfm?id=644403.644412

Bruce J McAdam. 2002b. *Reporting Type Errors in Functional Programs*. Ph.D. Dissertation. Larboratory for Foundations of Computer Science, The University of Edinburgh.

Takao Mohri and Hidehiko Tanaka. 1994. An Optimal Weighting Criterion of Case Indexing for Both Numeric and Symbolic Attributes. In *In D. W. Aha (Ed.), Case-Based Reasoning: Papers from the 1994 Workshop, Technical Report WS-94-01. Menlo Park, CA: AIII.* AAAI Press, 123–127.

Ankur Moitra. 2014. Algorithmic Aspects of Machine Learning. (2014).

Matthias Neubauer and Peter Thiemann. 2003. Discriminative sum types locate the source of type errors. In *ACM Int. Conf. on Functional Programming*. 15–26. DOI: http://dx.doi.org/10.1145/944705.944708

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *OOPSLA*. 525–542.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2015. Practical SMT-based Type Error Localization. In *ICFP*. 412–423.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Claude E Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review* 5, 1 (2001), 3–55.

A. Sheneamer and J. Kalita. 2016. Semantic Clone Detection Using Machine Learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 1024–1028. DOI: http://dx.doi.org/10.1109/ICMLA.2016.0185

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive type debugging in Haskell. In *ACM SIGPLAN Workshop on Haskell*. 72–83. DOI: http://dx.doi.org/10.1145/871895.871903

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*. 80–91. DOI: http://dx.doi.org/10.1145/1017472.1017486

Kai Ming Ting. 2002. An instance-weighting method to induce cost-sensitive trees. *IEEE Transactions on Knowledge and Data Engineering* 14, 3 (2002), 659–665.

Ville Tirronen, SAMUEL UUSI-MÄKELÄ, and VILLE ISOMÖTTÖNEN. 2015. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming* 25 (2015), e11.

Mitchell Wand. 1986. Finding the source of type errors. In *ACM Symp. on Principles of Programming Languages*. 38–43. DOI: http://dx.doi.org/10.1145/512644.512648

BX Wang and Nathalie Japkowicz. 2004. Imbalanced data set learning with synthetic samples. In *Proc. IRIS Machine Learning Workshop*, Vol. 19.

Jeremy Richard Wazny. 2006. *Type inference and type error diagnosis for Hindley/Milner with extensions.* Ph.D. Dissertation. The University of Melbourne.

Baijun Wu and Sheng Chen. 2017. How Type Errors Were Fixed and What Students Did? *Proc. ACM Program. Lang.* 1, OOPSLA, Article 105 (2017), 1 – 27.

Jun Yang. 2000. Explaining Type Errors by Finding the Source of a Type Conflict. In *Trends in Functional Programming*. Intellect Books, 58–66.

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 12–21.

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically Learning Shape Specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 491–507. DOI: http://dx.doi.org/10.1145/2908080.2908125