MACER: A Modular Framework for Accelerated Compilation Error Repair

Darshak Chhatbar[1], Umair Z. Ahmed[2], and Purushottam Kar[1(✉)]

[1] Indian Institute of Technology Kanpur, Kanpur, India
{darshak,purushot}@cse.iitk.ac.in
[2] National University of Singapore, Singapore, Singapore
umair@comp.nus.edu.sg

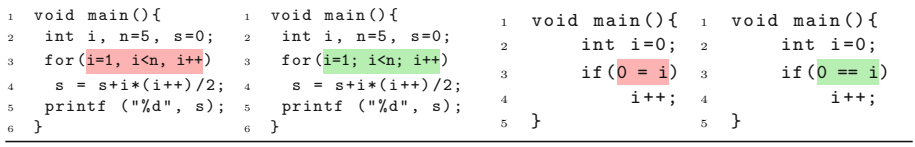
Abstract. Automated compilation error repair, the problem of suggesting fixes to buggy programs that fail to compile, has pedagogical applications for novice programmers who find compiler error messages cryptic and unhelpful. Existing works frequently involve black-box application of generative models, e.g. sequence-to-sequence prediction (TRACER) or reinforcement learning (RLAssist). Although convenient, this approach is inefficient at targeting specific error types as well as increases training costs. We present MACER, a novel technique for accelerated error repair based on a modular segregation of the repair process into repair identification and repair application. MACER uses powerful yet inexpensive learning techniques such as multi-label classifiers and rankers to first identify the type of repair required and then apply the suggested repair. Experiments indicate that this fine-grained approach offers not only superior error correction, but also much faster training and prediction. On a benchmark dataset of 4K buggy programs collected from actual student submissions, MACER outperforms existing methods by 20% at suggesting fixes for popular errors while being competitive or better at other errors. MACER offers a training time speedup of 2× over TRACER and 800× over RLAssist, and a test time speedup of $2-4\times$ over both.

Keywords: Introductory programming · Compilation error · Program repair · Multi-label learning · Structured prediction

1 Introduction

Programming environment feedback such as compiler error messages, although formally correct, can be unhelpful in guiding novice programmers on correcting their errors [14]. This can be due to 1) use of technical terms in error messages which may be unfamiliar to beginners, or 2) the compiler being unable to comprehend the intent of the user. For example, for an integer variable `i` in the C programming language, the statement `0 = i;` results in an error that the "expression is not assignable". Although the issue was merely the direction of assignment, the error message introduces concepts of expressions and assignability which may confuse a beginner (see Fig. 1 for examples). For beginners, navigating

I. I. Bittencourt et al. (Eds.): AIED 2020, LNAI 12163, pp. 106–117, 2020.
https://doi.org/10.1007/978-3-030-52237-7_9



Error Message E6: expected ';' in 'for' statement specifier
Repair Class [E6 [,,] [;;]] (see Sec 2 for details)

Error Message E10: expression is not assignable
Repair Class [E10 [=] [==]] (see Sec 2 for details)

Fig. 1. Two examples of actual repairs by MACER.

such feedback often means seeking guidance from a human mentor which is not scalable [5]. In this work we report MACER, a tool that automatically suggests repairs to programs with compilation errors to reduce loads on human mentors.

Related Works. DeepFix [9] was one of the first methods to use deep learning (sequence-to-sequence models) to jointly locate and repair errors. TRACER [1] reported better performance by segregating the repair pipeline into repair line localization and repair prediction, and introduced the stringent Pred@k metric that compares the predicted repair against the actual repair desired by student, as opposed to the existing *repair accuracy* metric that simply counts reduction in compilation errors. RLAssist [8] introduced the use of reinforcement learning to eliminate the need for labeled training data but suffers from slow training times. Sect. 4 offers explicit experimental comparisons of MACER to DeepFix, TRACER and RLAssist. Apart from this, [10] used variational auto-encoders to introduce diversity in the suggested repairs. [19] considered *variable-misuse* errors that occur due to similar-looking identifier names. TEGCER [2] focused on *repair demonstration* by showing examples of fixes made by other students rather than repairing the error, which can be argued to have greater pedagogical utility.

Our Contributions. In addition to locating lines that need repair, MACER further segregates the repair pipeline by identifying *what* is the type of repair needed on each line (the *repair-class* of that line), and *where* in that line to apply that repair (the *repair-profile* of that line). Methods like TRACER and DeepFix perform the last two operations in a single step using some heavy-duty generative mechanism. MACER's repair pipeline is end-to-end and entirely automated[1] i.e. steps such as creation of repair classes can be replicated for any programming language for which static type inference is possible. In addition to this,

1. MACER is able to pay individual attention to each repair class to offer superior error repair. MACER also introduces the use of highly scalable multi-label learning techniques, such as hierarchical classification and re-ranking. To the best of our knowledge, the use of these techniques is novel in this domain.
2. MACER accurately predicts the repair class (see Table 1). Thus, instructors can manually rewrite helpful feedback (to accompany MACER's suggested repair) for popular repair classes which may offer greater pedagogical value.

[1] The MACER tool-chain is available at https://github.com/purushottamkar/macer/.

ErrorID	Error Message	Freq.
E1	Expected ☐ after expression	4999
E2	Use of undeclared identifier ☐	4709
E3	Expected expression	3818
E6	Expected ☐ in ☐ statement specifier	720
E10	Expression is not assignable	538
E23	Expected ID after return statement	128
E57	Unknown type name ☐	23
E76	Non-object type ☐ is not assignable	11
E98	variable has incomplete type ID	3
E148	Parameter named ☐ is missing	1

ClassID	[ErrorID [Del] [Ins]]	Type	Freq.
C1	[E1 [∅] [;]]	Insert	3364
C2	[E2 [INVALID] [INT]]	Replace	585
C12	[E6 [,] [;]]	Replace	173
C22	[E23 [;] [∅]]	Delete	89
C31	[E6 [,,] [;;]]	Replace	62
C64	[E3 [)] [∅]]	Delete	33
C99	[E45 [==] [=]]	Replace	19
C115	[E3 [∅] [‘]]	Insert	16
C145	[E24 [.] [->]]	Replace	11
C190	[E6 [for] [while]]	Replace	9

Fig. 2. (**Left**) Some of the 148 compiler errorIDs listed in decreasing order of frequency in the train set. Some errorIDs are frequent whereas others are very rare. The symbol ☐ is a placeholder for program specific tokens such as identifiers, reserved keywords, punctuation marks etc. E.g., an instance of E6 is shown in Fig. 1. An instance of E1 could be “Expected ; after expression”. (**Right**) Some of the 1016 repair classes used by MACER listed in decreasing order of frequency in the train set. E.g., ClassID C145 concerns inappropriate use of the dot operator to access member fields of a (pointer to a) structure and requires replacement with the arrow operator. ∅ indicates that no token need be inserted/deleted for that class, e.g., no token need be inserted to perform repair for C22 whereas no token need be deleted to perform repair for C115. Please see the text in Sect. 2 for a description of the notation used in the second column.

2 MACER: Data Pre-processing

The training data for MACER is in the form of (source-target) program pairs where the source program failed to compile and the target is the student-repaired program. Similar to [1], we train only on pairs where the two programs differ in a single line (although MACER is tested on programs where multiple lines require repairs as well). The differing line in the source (resp. target) program is called the *source line* (resp. *target line*) (e.g. line 3 in Fig. 1). With every such program pair, we also receive the errorID and message generated by the Clang compiler [13] when compiling the source program. Figure 2 lists a few errorIDs and error messages. Some error types are extremely rare whereas others are very common.

Notation. We use angular brackets to represent n-grams e.g. the statement $a = b + c$; contains unigrams ⟨a⟩, ⟨=⟩, ⟨b⟩, ⟨+⟩, ⟨c⟩, ⟨;⟩, and bigrams ⟨a =⟩, ⟨= b⟩, ⟨b +⟩, ⟨+ c⟩, ⟨c ;⟩, ⟨; EOL⟩. Including an end-of-line character EOL helps MACER distinguish this location since several repairs (such as insertion of expression termination symbols) require edits at the end of the line.

Feature Encoding. Source lines contain user-defined literals and identifiers that are diverse yet uninformative for error repair. Thus, we perform *abstraction* by replacing literals and identifiers with an abstract LLVM token type [13], while retaining keywords and symbols, e.g. the raw/*concrete* statement `int abc = 0;` is converted to the *abstract* statement `int VARIABLE_INT = LITERAL_INT ;`. An exception is string literals where format-specifiers (e.g. `%d` and `%s`) are retained since these are often a source of error themselves. Such abstraction is

common in literature [1,2]. The token `INVALID` is used for unrecognized identifiers. This gave us a vocabulary of 161 uni and 1930 bigrams (trigrams did not offer significant improvements). A source line is represented as a 2239 (148 + 161 + 1930) dimensional vector storing one-hot encodings of the compiler errorID (see Fig. 2), and uni and bigram feature encodings of the abstracted source line. Note that the feature encoding step does not use the target line in any way.

Repair Class Creation. The *repair class* of a source line encodes *what* repair to apply to that line. The Clang compiler offers 148 distinct errorIDs in our training dataset. However, diverse repair strategies may be required to handle all instances of a single errorID. E.g., errorID E6 can either signal missing semicolons `';'` within the `for` loop statement specifier (as in Fig. 1), or missing semicolon at the end of a `do-while` block, or missing colons `':'` in a `switch` case block. To address this, similar to TEGCER [2], we expand the 148 compiler errorIDs into 1016 *repair classes*. These repair classes are generated automatically from training data and do not require any manual supervision. For each training example, the diff of the abstracted source and target lines reveals the set of tokens that must be inserted/deleted to/from the abstracted source line to obtain the abstracted target line. The *repair class* of this example is then simply a tuple enumerating the compiler error ID followed by the tokens to be inserted/deleted (in order of their occurrence in the source line from left to right).

$$[\text{ErrID } [\texttt{TOK}_1^- \ \texttt{TOK}_2^- \ \ldots][\texttt{TOK}_1^+ \ \texttt{TOK}_2^+ \ \ldots]]$$

We identified 1016 such classes (see Fig. 2). Repair classes requiring no insertions (resp. no deletions) are called *Delete* (resp. *Insert*) classes and others are called *Replace* classes. Repair classes, like error IDs, exhibit a heavy tail distribution with a few popular repair classes having hundreds of training examples whereas most repair classes having single digit training examples (see Fig. 2).

Repair Profile Creation. The *repair profile* of a source line encodes *where* in that line to apply the repair encoded in its repair class. For every erroneous program, the diff between its abstracted source and target lines tells us which bigrams in the abstracted source line require edits (insert/delete/replace). The repair profile for a training example is given as a one-hot representation of the set of bigrams i.e. $\mathbf{r} \in \{0,1\}^{1930}$ which require modification. We note that the repair profile is a sparse fixed-dimensional binary vector (that does not depend on the number of tokens in the source line) and ignores repetition information. Thus, even if a bigram requires multiple edit operations, or appears several times in the source line and only one of those occurrences requires an edit, we record a 1 in the repair profile corresponding to that bigram. This was done in order to simplify prediction of the repair profile for erroneous programs at testing time.

Working Dataset. After the above pre-processing steps, we have with us, corresponding to every training source-target example pair, a class-label $y^i \in [1016]$ telling us the repair class for that source line, a feature representation $\mathbf{x}^i \in \{0,1\}^{2239}$ that tells us the errorID along with the uni/bigram representation of the source line, and a sparse Boolean vector $\mathbf{r}^i \in \{0,1\}^{1930}$ that tells us the repair profile. Altogether, this constitutes a dataset of the form $\left\{(\mathbf{x}^i, y^i, \mathbf{r}^i)\right\}_{i=1}^n$.

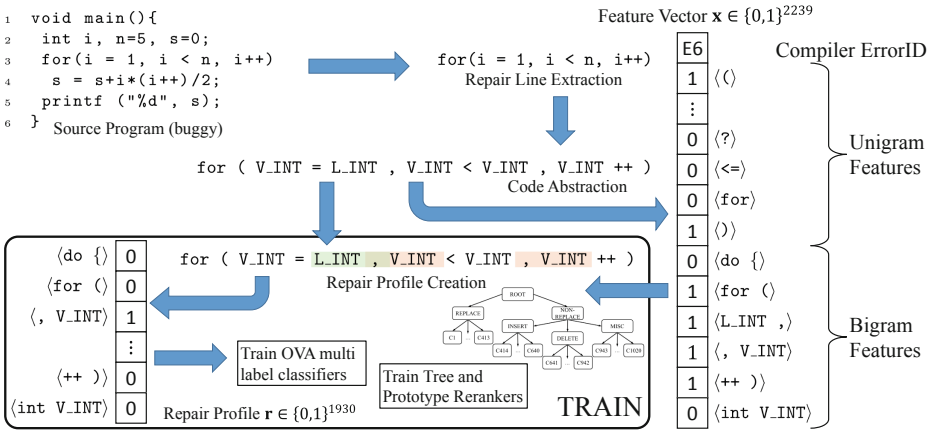


Fig. 3. The training pipeline for MACER, illustrated using the example used in Fig. 1. `L_INT` and `V_INT` are shorthand for `LITERAL_INT` and `VARIABLE_INT`.

3 MACER: Training and Prediction

MACER segregates the error repair process (at test time) into six distinct steps

1. **Repair Lines**: Locate which line(s) are erroneous and require repair.
2. **Feature Encoding**: A 2239-dimensional feature vector for each such line.
3. **Repair Class Prediction**: Use the feature vector to predict which of the 1016 repair classes is applicable i.e. which type of repair is required.
4. **Repair Localization**: Use the feature vector to predict locations within the source line where repairs should be applied.
5. **Repair Application**: Apply repairs at the predicted locations.
6. **Repair Concretization**: Undo code abstraction and compile.

MACER departs notably from previous works in segregating the repair process into these steps. Apart from faster training and prediction, this allows MACER to learn a customized repair location and repair application strategy for every repair class, e.g. if it is known that the repair requires the insertion of a semicolon, then the possible repair locations are narrowed down significantly.

Repair Lines. In addition to the compiler reported error line numbers, MACER samples 2 additional lines, one above and one below, as candidate repair lines. The same technique was used by TRACER [1], and achieves a *repair line localization* recall of around 90% on our training dataset.

Repair Class Prediction. Given the large number of repair classes, MACER uses a probabilistic hierarchical classification trees [12,17] for fast and accurate prediction. Given a source line feature vector $\mathbf{x} \in \{0,1\}^{2239}$, they assign a likelihood score $s_c^{\mathsf{tree}}(\mathbf{x})$ for each repair class $c \in [1016]$ that is used to rank the classes. The tree used by MACER (see Fig. 4) uses a feed-forward network with 2 hidden layers with 128 nodes each at the root node and linear one-vs-rest classifier at other nodes, all trained on cross entropy loss. However, given the large number

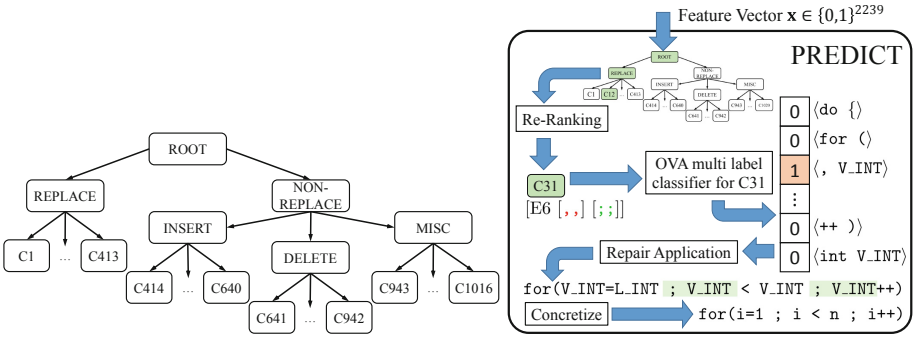


Fig. 4. (Left) The prediction hierarchy used by MACER to predict the repair class. **(Right)** The repair pipeline for MACER, illustrated using the example used in Fig. 1. A situation is depicted where a wrong repair class gets highest score from the classification tree, but reranking corrects the error. Table 1 shows that this is indeed common.

Table 1. Performance benefits of reranking. Here, Top@k reports the fraction of test examples on which the correct errorID/repair tokens were predicted within top k locations of the ranking. MAP refers to mean-averaged precision. Reranking significantly boosts MACER's performance. MAP error indicates that reranking ensures that the correct errorID/repair tokens were almost always predicted within the first two ranks.

	Top@1	Top@3	Top@5	MAP
Reranking Off (use $s_c^{\mathsf{tree}}(\mathbf{x})$ to rank repair classes)	0.66	0.83	0.87	0.40
Reranking On (use $0.8 \cdot s_c^{\mathsf{tree}}(\mathbf{x}) + 0.2 \cdot s_c^{\mathsf{prot}}(\mathbf{x})$ instead)	0.67	**0.88**	**0.90**	**0.50**

of extremely rare repair classes (Fig. 2 shows that only ≈150 of the 1016 repair classes have more than 10 training examples), there is room for improvement.

Repair Class Reranking. To improve classification performance on rare repair classes, MACER uses *prototype classifiers* [11,16]. Prototypes vectors are obtained for each repair class $c \in [1016]$ (with say n_c training examples) by clustering training examples associated with that class into $k_c = \lceil \frac{n_c}{25} \rceil$ clusters with centroids $\tilde{\mathbf{x}}_c^1, \ldots, \tilde{\mathbf{x}}_c^{k_c}$. For a source line $\mathbf{x} \in \{0,1\}^{2239}$, the prototypes are used to assign a new score to each repair class $s_c^{\mathsf{prot}}(\mathbf{x}) := \max_{k \in [k_c]} \exp\left(-\frac{1}{2}\left\|\mathbf{x} - \tilde{\mathbf{x}}_c^k\right\|_2^2\right)$. This score is combined with the earlier (hierarchical classification tree) score as $s_c(\mathbf{x}) = 0.8 \cdot s_c^{\mathsf{tree}}(\mathbf{x}) + 0.2 \cdot s_c^{\mathsf{prot}}(\mathbf{x})$ and $s_c(\mathbf{x})$ is used to rank the repair classes. Table 1 outlines how the reranking step significantly boosts MACER's ability to accurately predict the relevant compiler errorID and the repair class.

Repair Localization. MACER predicts the repair profile vector by solving a multi-label learning problem with 1930 "labels" corresponding to the bigrams in our vocabulary. MACER trains a separate "one-vs-rest" (OVR) classifier [4] per repair class that allows it to adapt to needs of different repair classes. Only those OVR classifiers that correspond to bigrams actually present in the source line are invoked. This offers good localization with a Hamming loss of just 1.43.

Repair Application. Having obtained the nature and location of the repairs from the above steps, MACER uses frugal but effective techniques to apply the repairs. Due to lack of space, we postpone details to the full version. Let $\mathcal{B}$ denote the ordered set of all bigrams (and their locations, ordered from left to right) in the source line which the predicted repair profile considers edit-worthy.

1. **Insertion Repairs**: In most cases of insertion repair, all tokens need to be inserted at the same location, e.g., `for(i=0;i<5)` → `for(i=0;i<5;i++)` with repair class [E6 [∅] [`; VARIABLE_INT ++`]]. MACER concatenates all tokens marked for insertion and tries inserting this ensemble into all bigrams in $\mathcal{B}$.
2. **Deletion Repairs**: Tokens marked for deletion in the predicted repair class are deleted at the first bigram in the set $\mathcal{B}$ that has that token.
3. **Replace Repairs**: The repair class specifies a list of pairs of tokens (`TOK`$^-$, `TOK`$^+$) where `TOK`$^-$ needs to be replaced with `TOK`$^+$. Similar to deletion repairs, MACER attempts this edit at the first bigram that contains `TOK`$^-$.
4. **Miscellaneous Repairs**: for unstructured repair classes where insertions and deletions are both required but an unequal number of tokens are inserted and deleted, MACER first ignores insertion tokens and performs edits as if this were a deletion repair class instance and then performs all insertions. This approach leaves room for improvement but nevertheless performs relatively well.

Repair Concretization. To make the repaired program compilable, abstract LLVM tokens such as `LITERAL_INT` are replaced with concrete program tokens such as literals and identifiers. Each abstract token is replaced with the most recently used concrete variable/literal of the same type, that already exists in the current scope. The process, although approximate, nevertheless recovers the correct replacement in 90+% of the instances in our datasets. Each candidate repair line reported by the *repair line localizer* is replaced with MACER's repair prediction, if it reduces the number of compilation errors in the program.

4 Experiments

We compared MACER's performance against previous works, as well as performed ablation studies to study the relative contribution of its components. All MACER implementations[2] were done using standard machine learning libraries such as `sklearn` [15] and `keras` [6]. Experiments were performed on a system with Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz × 8 CPU having 32 GB RAM.

Datasets. We report results on 3 different datasets, all curated from CS-1 course offerings at IIT-Kanpur (a large public university) with 400+ students attempting 40+ programming assignments. The datasets were recorded using Prutor [7], an online IDE. The DeepFix dataset[3] contains 6,971 programs that fail to compile, each with max 400 tokens [9]. The *single-line* (17,669 train + 4,578 test) and *multi-line* (17,451 test programs) datasets[4] contain program pairs where error-repair is required, respectively, on a single line or multiple lines [1].

[2] The MACER tool-chain is available at https://github.com/purushottamkar/macer/.
[3] https://www.cse.iitk.ac.in/users/karkare/prutor/prutor-deepfix-09-12-2017.zip.
[4] https://github.com/umairzahmed/tracer.

Metrics. We report i) repair accuracy, the fraction of test programs that were successfully repaired by a tool, and ii) Pred@k, the fraction of programs where at least one of the top k abstract repair suggestions exactly matched the student's own abstract repair. This is a metric introduced in [1] motivated by the fact that the goal of program repair, especially in pedagogical settings, is not to merely generate any program that compiles (see below).

The Importance of Pred@k. We consider a naive method *Kali'* that simply deletes all lines where the compiler reported an error. This is inspired by Kali [18], an erstwhile state-of-art semantic-repair tool that repaired programs by functionality deletion alone. *Kali'* gets 48% repair accuracy on the DeepFix dataset whereas DeepFix [9], TRACER [1] and MACER get respectively 27%, 44% and 56% (Table 3). Although *Kali'* seems to offer better repair accuracy than TRACER, its Pred@1 accuracy on the single-line dataset is just 4%, compared to 59.6% and 59.7% by TRACER and MACER respectively (Table 2). This shows the weakness of the repair accuracy metric and the need for the Pred@k metric.

Results. Of the total 7 min train time (see Table 3), MACER took less than 5 s to create repair classes and repair profiles from the raw dataset. The rest of the training time was taken up more or less evenly by repair class prediction training (tree ranking + reranking) and repair profile prediction training.

Comparisons with Other Methods. The values for Pred@k (resp. Rep@k) were obtained by considering the top k repairs suggested by a method and declaring success if any one of them matched the student repair (resp. removed compilation errors). For Pred@k computations, all methods were given the true repair line and did not have to perform repair line localization. For Rep@k computations, all methods had to localize then repair. Tables 2 and 3 compare MACER with competitor methods. MACER offers superior repair performance at much lesser training and prediction costs. Figure 6 shows that MACER outperforms TRACER by $\approx$ 20% on popular classes while being competitive or better on others.

Ablation studies with MACER. To better understand the strengths and limitations of MACER, we report on further experiments. Figure 6 shows that MACER is effective at utilizing even small amounts of training data and that its prediction accuracy drops below 50% only on repair classes which have less than 30 examples in the training set. Figure 5 offers examples of actual repairs by MACER. Although it performs favorably on repair classes seen during training, it often fails on *zero-shot* repair classes which were never seen during training. Table 4 presents an explicit ablation study analyzing the differential contributions of MACER's individual components on the single-line dataset. Re-ranking gives 10–12% boost to both Pred@k and repair accuracy. Predicting the repair class (resp. profile) correctly accounts for 5–12% (resp. 6%) of the performance. MACER loses a mere 6% accuracy on account of improper repair application. For all figures and tables, details are provided in the captions due to lack of space.

Table 2. TRACER vs MACER on single, multi-line datasets. Although comparable on single line, MACER outperforms TRACER by 14% on multi-line dataset. P@k, R@k are shorthand for Pred@k, Rep@k resp.

Dataset	Single			Multi
Metric	P@1	P@5	R@5	R@5
TRACER	0.596	0.683	0.792	0.437
MACER	0.597	0.691	0.805	**0.577**

Table 3. All methods on the DeepFix dataset. Values take from *[9] and †[8]. MACER offers the highest repair accuracy with a margin of 12.5% over the next best method, a prediction time that is at least 2× faster, and a train time 2× faster than TRACER and 800× faster than RLAssist.

	DeepFix	RLAssist	TRACER	MACER
Repair Acc	0.27*	0.267†	0.439	**0.566**
Test Time	<1s†	<1s†	1.66 s	**0.45 s**
Train Time	-	4 Days	14 min	**7 min**

#	Source-line	Target-line	MACER's Top Prediction	Pred?	Repair?	Zero-shot?
1	`scanf("%c",&a[i] ;`	`scanf("%c",&a[i]);`	`scanf("%c",&a[i] );`	Yes	Yes	No
2	`for (i =0;i<n;i++)`	`for(int i=0;i<n;i++)`	`for(int i=0;i<n;i++)`	Yes	Yes	No
3	`if(x==y)printf("Y"); break;`	`if(x==y)printf("Y");`	`if(x==y)printf("Y") ;`	Yes	Yes	No
4	`for(i=0; i=<N ;i++)`	`for(i=0;i<=N;i++)`	`for(i=0; i<N ;i++)`	No	Yes	No
5	`if (a[j]==' ')`	`if(a[j]==' ')`	`if((a[j]==' '))`	No	Yes	No
6	`int n; n=q;`	`int n;`	`int n; n=0;`	No	Yes	Yes
7	`c=sqrt( a^2+b^2 );`	`c=sqrt(a*a+b*b);`	`c=sqrt( a^2+b^2 );`	No	No	Yes

Fig. 5. Some examples of repairs by MACER on test examples. Pred? = Yes if MACER's top suggestion exactly matched the student's abstracted fix. Rep? = Yes if MACER's top suggestion removed all compilation errors. ZS? = Yes for "zero-shot" test examples i.e. the corresponding repair class was absent in training data. MACER offers exactly the student's repair for the first three examples. Note that the second example involves an undeclared identifier. For the next three examples, although MACER does not offer exactly the student repair, it nevertheless offers sane fixes that eliminate all compilation errors. The last two are zero-shot examples – MACER handles one of them.

5 Conclusion

We presented MACER, a novel technique that offers superior repair accuracy and increased training and prediction speed by finely segregating error repair into efficiently solvable ranking and labeling problems. Targeting rare error classes and "zero-shot" cases (Fig. 5) is an important area of future improvement. A recent large scale user-study [3] demonstrated that students who received automated repair feedback from TRACER [1] resolved their compilation errors faster on average, as opposed to human tutored students; with the performance gain increasing with error complexity. We plan to conduct a similar systematic user study in the future, to better understand the correlation between our improved Pred@k metric scores and error-resolution efficiency of students.

Acknowledgments. The authors thank the reviewers for helpful comments and are grateful to Pawan Kumar for support with benchmarking experiments. P. K. thanks Microsoft Research India and Tower Research for research grants.

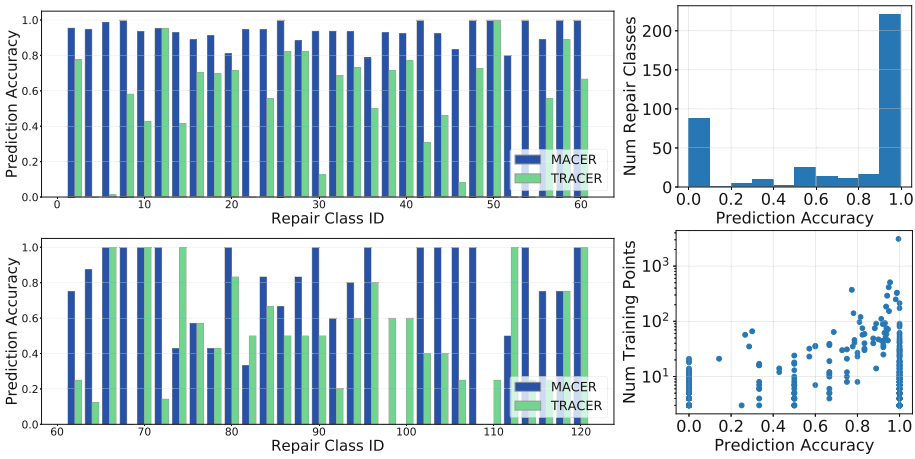


Fig. 6. (Left-Top and Left-Bottom) MACER vs TRACER on the 60 most frequent (head) and top 60–120 (torso) repair classes. To avoid clutter, only 30 classes from each category are shown. MACER outperforms TRACER by around 20% in terms of Pred@k on head classes and is competitive or better on torso classes. **(Right-Top and Right-Bottom)** Prediction (exact match) accuracy for MACER on the 391 repair classes that had at least 3 training points. On a majority of these classes 221/391 = 56%, MACER offers greater than 90% Pred@k. On a much bigger majority 287/391 = 73%, MACER offers more than 50% prediction accuracy. MACER's prediction accuracy drops below 50% only on classes which have less than around 30 points. This indicates that MACER is effective at utilizing even small amounts of training data.

Table 4. An ablation study on the differential contributions of MACER's components. ZS stands for "zero-shot". For the "ZS included" column, all test points were considered while the "ZS excluded" column took only those test points whose repair class was seen at least once in the training data. RR stands for reranking. RCP stands for Repair Class Prediction, RLP stands for Repair Location Prediction. RCP = P (resp. RLP = P) implies that we used the repair class (resp. repair location) predicted by MACER. RCP = G (resp. RLP = G) implies that we used the true (G for gold) repair class (resp. true repair profile vector). The difference in the first two rows shows that reranking gives 10–12% boost to both Pred@k and repair accuracy. Predicting the repair class (resp. profile) correctly accounts for 5–12% (resp. 6%) of the performance. The final row shows that MACER loses 6–8% performance owing to improper repair application/concretization. In the last two rows, Pred@1 is higher than Rep@1 (1–2% cases) owing to concretization failures – even though the predicted repair matched the student's repair in abstracted form, the program failed to compile after abstraction was removed.

			ZS included		ZS excluded	
RR	RCP	RLP	Pred@1	Rep@1	Pred@1	Rep@1
OFF	P	P	0.492	0.599	0.631	0.706
ON	P	P	0.597	0.703	0.757	0.825
ON	G	P	–	–	0.885	0.877
ON	G	G	–	–	0.943	0.926

References

1. Ahmed, U.Z., Kumar, P., Karkare, A., Kar, P., Gulwani, S.: Compilation error repair: for the student programs, from the student programs. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), pp. 78–87 (2018). https://doi.org/10.1145/3183377.3183383
2. Ahmed, U.Z., Sindhgatta, R., Srivastava, N., Karkare, A.: Targeted example generation for compilation errors. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 327–338. IEEE (2019). https://doi.org/10.1109/ASE.2019.00039
3. Ahmed, U.Z., Srivastava, N., Sindhgatta, R., Karkare, A.: Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In: 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET) (2020, to appear)
4. Babbar, R., Schölkopf, B.: DiSMEC - distributed sparse machines for extreme multi-label classification. In: 10th ACM International Conference on Web Search and Data Mining (WSDM), pp. 721–729 (2017). https://doi.org/10.1145/3018661.3018741
5. Camp, T., Zweben, S.H., Walker, E.L., Barker, L.J.: Booming enrollments: good times? In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE), pp. 80–81. ACM (2015). https://doi.org/10.1145/2676723.2677333
6. Chollet, F., et al.: Keras: the python deep learning library (2015). https://keras.io
7. Das, R., Ahmed, U.Z., Karkare, A., Gulwani, S.: Prutor: a system for tutoring CS1 and collecting student programs for analysis (2016). arXiv:1608.03828 [cs.CY]
8. Gupta, R., Kanade, A., Shevade, S.: Deep reinforcement learning for syntactic error repair in student programs. In: 33rd AAAI Conference on Artificial Intelligence (AAAI), pp. 930–937 (2019). https://doi.org/10.1609/aaai.v33i01.3301930
9. Gupta, R., Pal, S., Kanade, A., Shevade, S.: DeepFix: fixing common C language errors by deep learning. In: 31st AAAI Conference on Artificial Intelligence (AAAI), pp. 1345–1351 (2017)
10. Hajipour, H., Bhattacharyya, A., Fritz, M.: SampleFix: learning to correct programs by sampling diverse fixes (2019). arXiv:1906.10502v1 [cs.SE]
11. Jain, H., Prabhu, Y., Varma, M.: Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In: 22nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), pp. 935–944 (2016). https://doi.org/10.1145/2939672.2939756
12. Jasinska, K., Dembczyński, K., Busa-Fekete, R., Pfannschmidt, K., Klerx, T., Hüllermeier, E.: Extreme F-measure maximization using sparse probability estimates. In: 33rd International Conference on Machine Learning (ICML), pp. 1435–1444 (2016)
13. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, p. 75. IEEE Computer Society (2004)
14. McCauley, R., et al.: Debugging: a review of the literature from an educational perspective. Comput. Sci. Educ. **18**(2), 67–92 (2008). https://doi.org/10.1080/08993400802114581

15. Pedregosa, F., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**(85), 2825–2830 (2011)
16. Prabhu, Y., et al.: Extreme multi-label learning with label features for warm-start tagging, ranking & recommendation. In: 11th ACM International Conference on Web Search and Data Mining (WSDM), pp. 441–449 (2018). https://doi.org/10.1145/3159652.3159660
17. Prabhu, Y., Kag, A., Harsola, S., Agrawal, R., Varma, M.: Parabel: partitioned label trees for extreme classification with application to dynamic search advertising. In: 27th International World Wide Web Conference (WWW), pp. 993–1002 (2018). https://doi.org/10.1145/3178876.3185998
18. Qi, Z., Long, F., Achour, S., Rinard, M.C.: An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: International Symposium on Software Testing and Analysis, pp. 24–36. ACM (2015). https://doi.org/10.1145/2771783.2771791
19. Vasic, M., Kanade, A., Maniatis, P., Bieber, D., Singh, R.: Neural program repair by jointly learning to localize and repair. In: 7th International Conference on Learning Representations (ICLR) (2019)