Advancements in AI-assisted compilation error repair and program retrieval

A Thesis Submitted

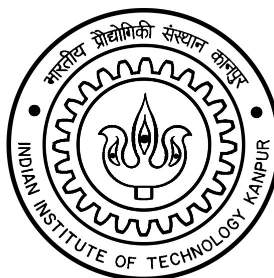
In Partial Fulfillment of the Requirements

For the Degree of M.Tech.

by

Fahad Shaikh

19111034



to the

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

June, 2021

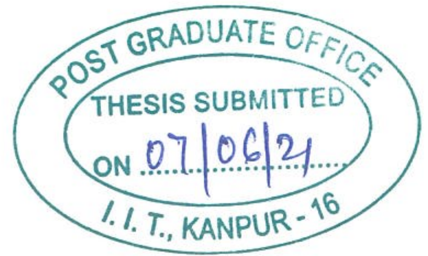
Declaration

This is to certify that at the thesis titled **"ADVANCEMENTS IN AI-ASSISTED COMPILATION ERROR REPAIR AND PROGRAM RETRIEVAL"** has been authored by me. It presents the research conducted by me under the supervision of **PROF. PURUSHOTTAM KAR, PROF. AMEY KARKARE**.

To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgments, in line with established norms and practices.

Name: Fahad Shaikh (19111034)
Program: M.Tech.
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur, Kanpur, 208016.
June, 2021

19111034



Certificate

It is certified that the work contained in the thesis titled "ADVANCEMENTS IN AI-ASSISTED COMPILATION ERROR REPAIR AND PROGRAM RETRIEVAL" by **FAHAD SHAIKH** has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof. Purushottam Kar, Prof. Amey Karkare
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, 208016.
June, 2021

vi

Abstract

Name of student: **Fahad Shaikh** Roll no: **19111034**
Degree for which submitted: **M.Tech.**
Department: **Department of Computer Science and Engineering**
Thesis title: **Advancements in AI-assisted compilation error repair and program retrieval**

Name of thesis supervisor: **Prof. Purushottam Kar, Prof. Amey Karkare**
Month and year of thesis submission: **June, 2021**

The use of AI assisted tools for pedagogical and software engineering applications is an active area for research. For novice programmers, compilation errors pose a major hurdle in learning. Moreover, the compiler provided feedback often targets more seasoned programmers and hence may not make sense to beginners. Automated compilation error repair uses AI based algorithms to generate fixes to erroneous programs and can help a programmer greatly.

We propose MACER++ which breaks down the task of program repair into several modules. Such modular structure was first proposed by MACER [5]. We propose optimizations in almost every module of the pipeline and also propose a synthetic data generation algorithm which enables few shot learning. We evaluate our optimizations on two data-sets, one having 4326 programs with single line errors and other having 6996 programs with multi line errors. Compared to MACER [5] our approach gives an improvement of 4-5% in repair accuracy on both data-sets and 2% improvement in Pred@5 metric which indicates that our fixes more closely resemble the fixes applied by students. Moreover, our approach is significantly faster than another state of the art method DrRepair [16] and provides better performance on single line data-set.

We also present PRIORITY, an AI assisted tool for labelling programming problems. The tool is intended to help tutors of ESC101, a programming course offered at IIT Kanpur. PRIORITY uses semi supervised techniques to label the large corpora of programming problems from previous offerings of ESC101, thus making these problems search-able. This can make the task of tutors much faster and easier.

Acknowledgments

I would like to express gratitude to all the people who helped me during my stay at IIT Kanpur. I especially thank my advisors Purushottam Kar and Amey Karkare for guiding me throughout this thesis work and also to help me improve as a person. They were always there when I got stuck or need help with something.

I would also like to thank Sharath who has been my partner during this thesis and has been a constant source of support. I also extend thanks to my batch mates, friends and family members. A special thanks to Umair Z. Ahmed,post-doc researcher at National University of Singapore, who helped us with some parts in this thesis.

Also I would like to thank the PROSE team at Microsoft Research at Redmond and India. I would like to thank Margaret Price and Sumit Gulwani for initiating the project to build a VS-code extension which utilizes MACER++ to help novice programmers. Also I thank Titus Barik, Danny Simmons, Ivan Radicek and Gustavo Soares among others for providing active feedback and guidance.

This thesis was compiled using a template graciously made available by Olivier Commowick `http://olivier.commowick.org/thesis_template.php`. The template was suitably modified to adapt to the requirements of the Indian Institute of Technology Kanpur.

Fahad Shaikh
June, 2021

Contents

Acknowledgments **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

1.1 Our Contributions 2

2 Related Works **3**

2.1 TEGCER and TRACER: 4

2.2 MACER (Modular Accelerated Compilation Error Repair) 4

2.3 Dr Repair 5

2.4 SampleFix 6

2.5 DECAF 7

2.6 Hierarchical classification 7

3 Proposed Optimizations **9**

3.1 Reworking repair classes: 10

3.2 Using compiler message for Localization: 10

3.3 Learned Hierarchical Classification: 11

3.4 Label features: 12

3.5 Repair application: 14

3.6 Synthetic Data generation: 15

3.7 Atomic Repair: 17

4 PRIORITY **21**

4.1 Data Pre-processing: 22

4.2 Feature Extraction: 23

4.3 ML Model: 25

4.4 Illustration: 26

4.5 What lies ahead: 27

5 Experiments **29**

5.1 Data-sets 29

5.2 Comparison with MACER: 30

5.3 Comparision with DrRepair: 32

5.4 Optimizations that didn’t work: 33

5.5 Priority results 34

6 Conclusion **35**

Bibliography **37**

7 Appendix **39**

7.1 Labels used in PRIORITY: 39

List of Figures

2.1 MACER's pipeline. Image Credit [5] 5

3.1 Prediction Pipeline for atomic repair 19

4.1 Distribution of labelled examples among labels 23
4.2 Label propagation to utilize unlabelled examples 24

5.1 Heavy tail in repair classes 32
5.2 Comparision of repair and pred accuracies for MACER vs MACER++ (MACER++ on right) 32

List of Tables

1.1 Example of repair generated by MACER++ 2

3.1 Examples generated using our Synthetic data generation method. 17

5.1 Better Line Localization using compiler diagnostic 30
5.2 Ablation Studies for MACER++ 31
5.3 Ablation studies for MACER++ using Pred@k measure 31
5.4 Comparison of MACER++ with DrRepair 33
5.5 Comparison among different methods for repair class prediction 33
5.6 Effect of using Label features with MACER++ 34
5.7 Comparison of using different methods for label tagging 34

CHAPTER 1

Introduction

Contents

1.1 Our Contributions **2**

The field of Artificial Intelligence (AI) has seen a remarkable rise in interest in various areas. Of specific interest has been a rise in application of AI tools and algorithms to the field of education [4]. Massive open online courses (MOOCs) are fast becoming a popular way of teaching programming. As the number of students in these courses increases it becomes more and more important to develop tools which can ease the burden on the instructors and teaching assistants (TA's).

For a novice programmer, compilation errors are one of the biggest hurdles to learning. [7] shows that novice programmers spend a lot of time trying to resolve these errors. Moreover, the compiler provided diagnostics are often too cryptic for novice programmers, hence may lead to further confusion instead of helping to solve the issue [15]. Because of this the area of automatic compilation error repair has seen a lot of interest recently. [5] proposes an AI based pipeline that takes as an input an erroneous program and produces a correct target program. Moreover, they break down this process for compilation error repair into various simple modules which makes the pipeline a much better fit for pedagogical applications as compared to other state of the art methods. We improve upon their work by making improvements in multiple modules. Concretely, we show that compiler diagnostic can provide valuable feedback and also show how smarter use of label metadata can augment the pipeline significantly. We also propose a novel synthetic data generation method which aids in training AI models, especially when training data is scarce.

Another issue faced by large MOOC's is that of question/problem re-usability. Consider ESC101, a basic programming course at IIT Kanpur, offered every semester to hundreds of students. In every offering of the course, the instructors and tutors need to put together programming problems which are then solved by students in lab sessions. But since these problems are not indexed, subsequent offerings are not able to utilize this large corpora of problems that has been collected. To remedy this, we develop PRIORITY (PRoblem IndicatOr ReposITorY), an AI based labelling mechanism to label these problems and develop a web app to allow tutors of this course to search for a problem from previous offerings.

```c
#include<stdio.h>
int main() {
    int age;
    scanf( "%d" , &age);
    if (age => 18 ){
        printf( "Can vote" );
    }
}
```

```c
#include<stdio.h>
int main() {
    int age;
    scanf( "%d" , &age);
    if (age >= 18 ){
        printf( "Can vote" );
    }
}
```

Error Message: file.c:6:13: error: expected expression

Repair: Replace => with >= on line 5.

Table 1.1: Example of repair generated by MACER++

1.1 Our Contributions

The key contributions of this thesis are enumerated below:

1. MACER [5] divides the task of erroneous program repair into various modules and introduces the notion of *repair classes* to succinctly describe the repair steps required to correct the erro on a given line of code. We propose efficient yet impactful optimizations to these modules that significantly boost the performance of the overall pipeline.
    - We show how compiler error messages can be used for better error line localization.
    - We show how the repair classes contain valuable information which can be used to improve the performance of the pipeline and how existing XML techniques can be used to learn a hierarchy of repair classes.
    - We also suggest a change in the formation of the repair classes themselves which results in better performance on zero shot cases.
2. We propose novel synthetic data generation techniques that use available training data to generate new synthetic examples. These synthetic examples mimic realistic errors made by novice programmers and augment the amount of training data available to the AI models that boosts performance significantly.
3. We propose an algorithm to label programming problems with pedagogical tags so as to construct finely indexed question banks consisting of questions from previous offerings of programming courses. Such a question bank can then be queried by course admins while preparing questions for the current offering of the course. This greatly reduces time and mental effort required to set questions for large programming courses.

We also developed real world deployments of these tools and discuss how they can benefit the students and course admins. The details of these deployments and user studies are a part of the companion thesis titled "Real World Deployment of AI-assisted compilation error repair and program retrieval" [10].

CHAPTER 2

Related Works

Contents

2.1 TEGCER and TRACER: 4
2.2 MACER (Modular Accelerated Compilation Error Repair) 4
2.3 Dr Repair 5
2.4 SampleFix 6
2.5 DECAF 7
2.6 Hierarchical classification 7

The use of machine learning and deep learning techniques for fixing erroneous programs has seen a lot of interest in recent years. This area has shown a lot of promise especially from a pedagogical perspective. One of the first papers in this area was DeepFix [8], which uses a sequence-to-sequence model to generate a syntactically correct program given an erroneous program. Several other works have been reported in this area since then.

Another approach, TRACER [1], breaks down the process of fixing an incorrect program into more than one step. They use student data, having source and target pairs, collected using the PRUTOR IDE [6], hence are able to capture mistakes made by students rather than relying on synthetically generated data. MACER [5] further breaks down the job of correcting the erroneous programs into several modules and uses light weight machine learning techniques to fix the syntax errors. Thus they not only improve upon TRACER's [1] accuracy but also improve training and testing time. Another approach, SampleFix [9], uses a conditional variational encoder to introduce diversity in the fixes suggested by the model.

One of the state-of-the-art methods for compilation error repair is DrRepair [16]. They use a full fledged sequence to sequence model along with graph attention and a pointer generator network to generate the correct program given an incorrect one in one shot. One disadvantage of this technique is that its resource intensive requiring extensive training on reasonably powerful GPU architectures.

In this thesis we introduce MACER++ , which improves upon MACER [5], by making optimizations to each of its modules. MACER++ also uses the concept of repair classes used by MACER [5] and TRACER [1] but significantly augments its utility to use them to generate synthetic data.

This chapter briefly explains all of the above mentioned approaches along with some extreme multi label classification techniques which were experimented with.

2.1 TEGCER and TRACER:

- We will not be describing the detailed approaches of these papers, but just the notion of a repair class and program abstraction, which was introduced by these papers and is used by MACER++ (and MACER).
- TRACER [1] was one of the first methods that segregated the repair procedure into multiple steps viz. Code Abstraction, Error Localization, Abstract Code Repair and Concretization. Training of MACER++ is inspired by TRACER where training is done on erroneous source line and corrected target line pairs instead of whole source-target program pairs.
- TRACER [1] instead of working on the source programs directly, first abstracts out identifiers and literals which are not much informative for error correction.
- We use TRACERS abstraction module which uses LLVM compiler to replace the identifiers/literals with generic tokens (like V_INT, V_CHAR, L_INT etc.). There are some exceptions to this, and we refer you to the TRACER [1] paper for more details.
- The advantage of working on abstract code instead of concrete code is that it limits the vocabulary size (which could be infinite in case of concrete code). This helps the ML algorithms immensely.
- Moreover, we improve upon TRACER's error line localization module. TRACER uses compiler reported error line and 2 other lines (one above and one below the compiler reported error line) and feeds them (after encoding) to its deep learning model.
- TEGCER [2] is a tool that suggests to students, based on the error they made in their code, similar examples and their fixes made by their peers.
- TEGCER [2] first defines error-repair classes using compiler reported error messages, and the changes made to the erroneous lines by the student to repair them. Then, an encoded feature vector is created using the compiler reported error lines and the unigrams and bigrams of the erroneous line. This feature vector is used to train a dense feedforward neural network for predicting the error-repair classes.

2.2 MACER (Modular Accelerated Compilation Error Repair)

- Since our work is closely related to MACER [5], we explain the architecture of MACER in greater detail. [5] in more detail.
- MACER [5] sets up a modular pipeline that, in addition to locating lines that need repair, 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). They also show that MACER [5] can specifically target certain error types.
- They use the notion of repair classes introduced by TEGCER [2].

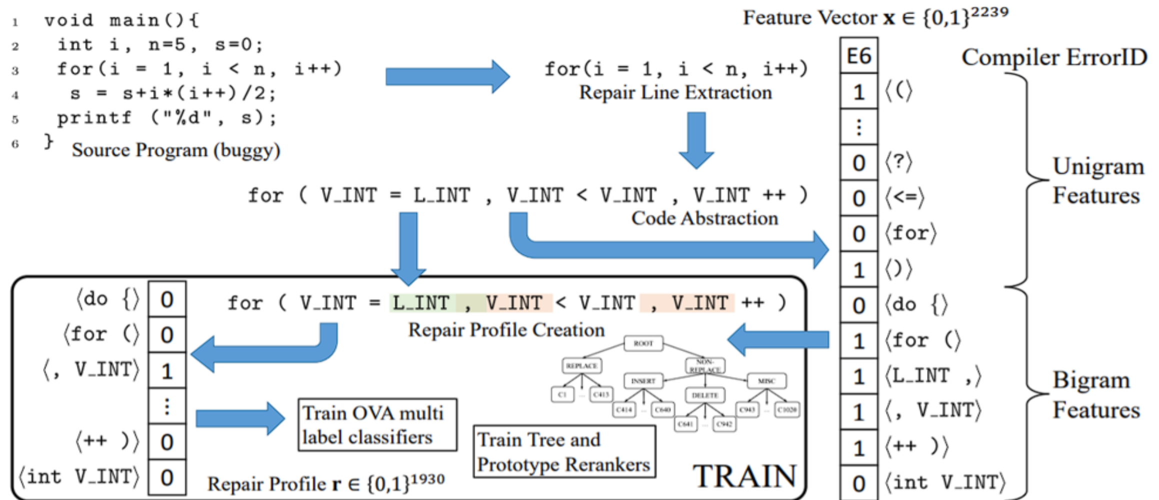


Figure 2.1: MACER's pipeline. Image Credit [5]

- They introduce techniques used in large-scale multi-class and multi-label learning tasks, such as hierarchical classification and reranking techniques, to the problem of program repair.
- As show in Figure 2.1, MACER [5] segregates the repair process into six distinct steps.
  1. Repair Lines: Locate within the source code, which lines are erroneous and require repair. This is done by including the compiler reported error lines, a line above that and a line below that.
  2. Feature Encoding: For each of the identified lines, perform code abstraction and obtain a 2239- dimensional feature vector.
  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. MACER [5] uses a Probabilistic Label Tree (PLT) technique (explained later) for multi class classification of repair classes.
  4. Repair Localization (Repair Profile prediction): Use the feature vector to predict locations within the source lines at which repairs should be applied. MACER [5] takes a DisMEC (explained later) style approach for repair profile prediction which is a multi-label classification problem.
  5. Repair Application: Apply the predicted repairs at the predicted locations.
  6. Repair Concretization: Undo code abstraction and compile.

2.3 Dr Repair

- DrRepair [16] proposes a Graph Attention-based LSTM network for Compiler Error correction using compiler diagnostic feedback. It takes in a program x and diagnostic feedback from a compiler as inputs, encodes them via LSTM and graph attention layers, and decodes the error line index k and repaired code y_k.

- They propose a joint graph representation of a program and diagnostic feedback that captures the underlying semantic structure of symbols (program-feedback graph). Specifically, it takes all identifiers in the source code and any symbols in the diagnostic arguments as nodes and connects instances of the same symbols with edges to encode the semantic correspondence.
- They then design a neural net model with a graph-attention mechanism on the program-feedback graph to model the symbol tracking process.
- They model the probability of a line k being erroneous via a feedforward network, and model its repair y_k, via a pointer-generator decoder.
- Main disadvantage of this approach is that it requires several days to train even on a reasonably powerful GPU infrastructure. Moreover, the one-step approach makes it challenging to fine-tune their method to focus more on certain types of errors than others.
- They also introduced a novel program corruption procedure for generating more synthetic training data.
- For program corruptions they first analysed common compiler errors in three settings: experienced developers, beginner programmers, and predicted code of program synthesis. For each case, they collected statistics from different data-sets, and grouped the errors into four major categories.
  - Syntax: which randomly deletes, inserts, or replaces an operator / punctuation.
  - ID-type: which randomly deletes, inserts, or replaces an identifier (ID) type such as int, float, char.
  - ID-typo: which randomly deletes, inserts, or replaces an identifier.
  - Keyword: which randomly deletes, inserts, or replaces a use of program language keyword or library function, such as if and size ().
- Then, they took syntactically correct programs from codeforces website and, based on the analysis, introduce each of this error based on some probability distribution into this correct programs to generate more source/target pairs. They use this synthetically generated data to pre-train their model.

2.4 SampleFix

- SampleFix [9] proposes an approach focused on tackling ambiguity and diversity in code. Since more than one piece of code can result in the same output, they propose that the problem of fixing a buggy problem be treated as a one-to-many mapping between source and target programs.
- At the core of their approach is a conditional variational autoencoder that is trained to sample accurate and diverse fixes for the given erroneous programs and interacts with a compiler that evaluates the sampled candidate fixes in the context of the given programs.

- They also propose a novel regularizer which encourages diversity by penalizing the distance among candidate samples, which thereby significantly increases the effectiveness by producing more diverse samples.

2.5 DECAF

- Extreme multi-label classification (XML) refers to the task of tagging data points with a relevant subset of labels from an extremely large label set. DECAF [12] paper demonstrates that XML algorithms stand to gain significantly by incorporating label metadata.
- In most of the XML applications and also in repair class prediction for MACER++ labels contain a lot of information.
- Using label metadata means incorporating information contained in labels into the Machine Learning model.
- Label metadata can allow collaborative learning, which especially benefits tail labels. Tail labels are those for which very few training points are available and form most labels in XML applications.
- DECAF [12] learns a separate linear classifier per label based on the 1-vs-All approach. These classifiers critically utilize label metadata and require careful initialization.
- Specific contributions are made in designing a shortlister with a large fan-out and a two-stage training strategy. DECAF [12] also introduces a novel initialization strategy for classifiers that leads to accuracy gains, more prominently on data-scarce tail labels. It scales to XML tasks with millions of labels
- It consists of three components 1) a lightweight text embedding block suitable for short-text applications, 2) 1-vs-All classifiers per label that incorporate label text, and 3) A shortlister that offers a high recall label shortlist for data points.

2.6 Hierarchical classification

- The use of these approaches for MACER++ is explained in section 4. Just a succinct explanation of each technique:
    1. PLT [11] - The Probabilistic Label Tree technique considers a fixed (handcrafted or learnt separately) label tree where every leaf of the tree is a label. The technique offers a way to navigate this tree effectively to retrieve for any data point, the subset of labels (equivalent subset of leaves) which are most applicable to that data point. Each node in the tree gives probability estimates of their children being applicable to a data point.
    2. DiSMEC- It is a highly parallelized implementation of one-vs-all (OvA) classifiers for extreme multi-label classification problems. The authors implement two levels of parallelism: first they divide the labels into batches which can be trained independently

and second, within each batch, classifiers of each label are trained independently and in parallel.

3. Tree Based XML techniques (Parabel & CraftML):

    Tree-based methods (like Parabel, CraftML etc) implement a divide-and-conquer paradigm and scale to large label sets in XMC by partitioning the labels space. As a result, these schemes of methods have the computational advantage of enabling faster training and prediction.

    Tree-based methods transform the initial large-scale problem into a series of small-scale subproblems by hierarchically partitioning the instance set or the label set.

    Parabel [13] learns a hierarchy over labels rather than data points.

    Parabel [13] improves over PLT [11] by learning the label hierarchy rather than using a fixed one, by modelling the joint label distribution rather than the marginals and by developing more efficient optimizers.

    CRAFTML [14] is a forest of decision trees trained with the supervision of the labels where the splitting conditions are based on all the features.

    We experimented by incorporating different Hierarchical classification techniques with the MACER [5] pipeline.

CHAPTER 3

Proposed Optimizations

Contents

3.1 Reworking repair classes: 10

3.2 Using compiler message for Localization: 10

3.3 Learned Hierarchical Classification: 11

3.4 Label features: 12

3.4.1 DECAF: 14

3.5 Repair application: 14

3.6 Synthetic Data generation: 15

3.6.1 Generating Synthetic examples using repair classes. 15

3.7 Atomic Repair: 17

3.7.1 Data Generation: 18

3.7.2 Feature Encoding: 19

3.7.3 Training Process and Model: 19

3.7.4 Prediction Process: 19

3.7.5 Future Work: 20

As discussed earlier, MACER [5] segregates the task of repairing an erroneous program into 6 different modules and uses lightweight machine learning techniques to determine the type of repair to be applied and where to apply them. Thus MACER [5] outperforms previous techniques not only in terms of repair accuracy but also is much less resource intensive. However, MACER [5] suffers from some issues. One major critique of MACER [5] is that it doesn't utilize the information provided by the compiler diagnostic very well. They just use the line number reported by the compiler. DrRepair [16] has shown that using compiler diagnostic can provide some valuable feedback which can significantly boost the performance of the machine learning techniques.

Another issue with MACER [5] is that it suffers from zero shot cases. Zero shot cases are the repairs which do not appear in the training set and hence MACER [5] is unable to repair them. Also MACER [5] doesn't utilize the information contained in the repair classes themselves which can be valuable. We try to tackle all of the above issues as well as try to improve upon the repair class prediction and repair profile prediction modules.

Apart from this we also introduce a novel synthetic data generation technique which uses the concept of repair classes to generate new examples from existing data. Even though other techniques try to generate artificial data, they do this in more or less random manner hence the generated data may not resemble the mistakes made by programmers. Instead our approach can generate examples which much closely resembles the errors made by programmers since the repair classes are created from these errors in the first place.

3.1 Reworking repair classes:

- The notion of repair class is central to MACER++ (and MACER [5]). Given a test program line MACER predicts the repair class which suggests the type of repair to be applied to correct the erroneous line.
- We automatically figure out the set of these repair classes during training using the edit distance between the source and target programs. The repair classes not only contains information about the type of error in the source line but also how to repair the erroneous line.
- These repair classes contain three pieces of information used to repair an erroneous line:
  - The Error ID (s) as reported by the compiler.
  - A enumeration of tokens to be inserted.
  - A enumeration of tokens to be deleted.
- Looking at the formation of the repair class it does not seem intuitive to include the error ids in the repair class. If the repair classes have identical tokens to be inserted / deleted, then the repair required should be the same regardless of the error ids reported by the compiler.
- Hence, we removed the error ids from the formation.
- This has 2 (logical) advantages:
- The first is that the number of repair classes went down from 1000 to 700 (a 30% reduction). Reducing the number of Classes can greatly help the ML algorithms used for multi class classification.
- Second is that since multiple classes are now condensed into one the number of training instance per repair class increases, which can be extremely helpful especially for tail classes (Classes which have very few training examples)
- We did see some improvements in prediction accuracies by applying this simple change.

3.2 Using compiler message for Localization:

- Compilers report line numbers where an error is located but they fail to pinpoint the exact location of error accurately. [15]
- One critique of techniques like MACER [5], TRACER [1] etc is that they do not use information provided by the compiler diagnostic message fully.
- One of the major components of MACER++ pipeline is its line no. localization module. Since, getting an incorrect line number would result in applying corrections on an incorrect line, thus causing MACER++ to fail in correcting the erroneous program.

- As explained earlier, MACER [5] uses the compiler reported error line and lines above and below that to create a set of lines to inspect, and then applies the rest of the pipeline on each of these lines individually.
- However, the compiler message still contains more valuable information that they ignore.
- One such piece of information is the identifier / symbol reported by the compiler in the diagnostic feedback. DrRepair [16] uses this information along with the source code to form an attention graph. It takes all identifiers in the source code and any symbols in the diagnostic arguments as nodes and connects instances of the same symbols with edges.
- Taking inspiration from these we extract the identifiers / symbols present in the compiler diagnostic feedback, then we iterate the source code to find occurrences of these symbols and include those lines where they occur in the set of lines to inspect.
- This improves MACER's localization accuracy, without hurting the running time too much.
- Not only does this make better use of the valuable information provided by the compiler, but also this allows MACER++ to fix some long-range errors which MACER would not have been able to fix.
- Consider this example:

```
1  #include<stdio.h>
2  int main()
3  {
4      int N ;
5      scanf("%d",&N);
6      int a,b,c;
7      int count=0;
8      if (a < b && b < c)
9       count++;
10     }
11     printf("%d",count);
12     return 0;
13 }
14
```

- As you can see the error in the code is unmatched braces. One fix could be to delete the closing brace on line 10. However, the compiler reports this error:

  file.c:13:1: error: extraneous closing brace ('}')

- Thus the line which require edit is not near the compiler reported line. However if we consider the identifier reported by the compiler and search for it in the source text, we can better locate the line which requires edit and thus fix the error.

3.3 Learned Hierarchical Classification:

- If we look at the set of repair classes, there is a natural (fixed) hierarchy Figure 2.1 which can be formed. MACER [5] uses this fixed hierarchy inspired by PLT [11] to predict the

repair classes.

- Other approach for such hierarchical classification could be to learn a hierarchy of labels instead of using a fixed hierarchy. There are existing techniques, generally used for extreme multi label classification (XML for short), which can learn a hierarchy of labels based on different similarity measures and at the same time scale up to a million labels.

- We tried replacing MACER's repair class classification module with some of these XML techniques viz. Parabel [13] and CraftML [14] (from now on refered to as XML algorithms).

- This XML algorithms basically learn an ensemble of Decision trees where leaves of the trees indicate one or more labels. While prediction step labels recommended by a majority of the trees are given a higher score. The output of these algorithms is a ranking of the labels from most relevant to least relevant.

- The feature encoding step is same as that of MACER [5]. We then feed this feature vectors to the XML algorithms which assign a probability score to each label, thus giving us a ranked list of most applicable labels (in our case repair classes). We pick top k of these recommended repair classes (k being a hyper-parameter) and then follow through with repair profile prediction, repair application and concretization steps to generate target programs. Finally, we compile each target program to find the correct solution.

- Even though the results (discussed later) are not that impressive, because of the modular nature of MACER and quick training and testing times, such algorithms can easily be plugged in and tested out as a black box with little effort. Any further improvements in any of these algorithms can directly improve MACER++ 's performance.

- One other advantage of these algorithms is that they are built to run efficiently in extreme cases where the number of labels is in order of millions (or even billions). Hence for applications like MACER++ where the number of labels is comparatively quite small these algorithms are extremely fast (faster than using a fixed hierarchy).

- Also, since MACER++ pipeline takes just a few minutes of time to train and test we can try out a variety of approaches quickly, thus being able to land upon the most suitable method.

3.4 Label features:

- Generally in multi class and multi label classification settings the class or labels are encoded as integers or as one hot vectors. However, recent work shows that use of label metadata, information contained in classes or labels, can help improve the performance of various algorithms.

- One peculiar thing about repair classes is that they themselves pack a lot of information about the repair that must be made. Using this information can help boost the performance of MACER++ pipeline.

- Also, the use of label metadata can enable collaborative learning. For example, consider two repair classes.

  1. [=] [==]: replace an assignment operator with a comparison operator.
  2. [!] [==] [!=]: replace a not and a comparison operator with not equals operator.

- Both classes will not have any common training examples. However, they both share a common token (the comparison operator). Incorporating this information can help the ML model perform better, especially for tail classes (which have very few training examples).

- However, MACER [5] and other techniques ignore this valuable piece of information by merely encoding the repair classes as integers.

- One way of incorporating information from the labels (repair classes) is to use a re-ranker along with the existing method used by MACER [5] for repair class prediction.

- Label text was incorporated into these classifiers as follows: for every training sample *i*, let s_{il} be the relevance score the XML classifier predicted for label *l* for document *i*. As suggested by DECAF [12] we can augment this score to incorporate label text by computing.

$$s_{il}\prime = \alpha \cdot s_{il} + (1 - \alpha)\sigma({x_i}^\intercal z_l)$$

- Here, $\alpha \in$ [0, 1] was fine tuned to offer the best results, x_i is the feature vector for the current training example and z_l is the corresponding label (repair class) representation and σ is the sigmoid function

- To obtain z_l, we simply consider the repair class as a plain text and use uni and bi-grams features. We also append the tokens with either del or add to differentiate between tokens to be deleted or inserted. For e.g., Consider a replace (reworked) repair class representing replacing 2 commas with 2 semicolons:

[, ,][; ;]

- This will be encoded as:

$$[del,][del,][add;][add;][,][,][;][;][,,][,;][;;]$$

- We then experiment with one hot encoders, count vectorizers etc. to convert this representation of repair classes to numerical formats.

- Note that this technique doesn't change the existing pipeline. In fact, there is no change to be made during training. Only during prediction we have another score in the form of $\sigma({x_i}^\intercal z_i)$, which can be thought of as a re-ranker, hence the name naive re-ranker.

3.4.1 DECAF:

- We experimented with DECAF [12], which extracts label features from label metadata and applies deep learning techniques to give state of the art performance on the existing XML benchmarks.
- Much like other XML techniques, given a feature vector DECAF [12] outputs a ranked list of labels (repair classes) and assigns a score to each label. However, they use a shared vocabulary between the feature vectors and label vectors. This enables collaborative learning when two labels share the same tokens which can help improve the performance for rare labels.
- Label vector were generated from the repair classes using the same procedure as discussed above.
- We tried replacing both MACER's [5] repair class prediction module (which is a multi-class classification problem) and the repair profile prediction module (which is a multi-label prediction problem).

3.5 Repair application:

- MACER [5] classifies the repair classes into 4 broad categories:
  - Replace repair class: where the repair can be made using just replace operations.
  - Insert repair class: where tokens are just to be inserted.
  - Delete repair class: where tokens are just to be deleted.
  - Rest (misc.) repair class: some combination of the above classes.
- After predicting which repair class is to be applied, it uses the repair profiles predicted using One vs All classifiers to apply repair classes to source lines in order to obtain the corrected lines. These repair profiles are nothing but an enumeration of bi-grams which appear in the source line and which require an edit to correct the error.
- However, as pointed out earlier, the predicted repair class itself contains information which can be used to improve on the existing repair application step.
- Specifically, for delete and replace classes, the repair class tells us which tokens are to be deleted. These classes can only be applied to the bi-grams which contain the token to be deleted. The predicted repair profile does not account for this fact, hence it is possible that the bi-gram predicted to be edited might not contain this token(s) and hence MACER fails to apply the repair.
- We incorporate this fact to improve on the repair application step of MACER. In more detail, while correcting an erroneous line, if the predicted repair class is a delete or replace class, we adopt a more brute force approach. We scan the line to find bi-grams which contain the tokens to be deleted and then apply the edit (delete or replace) on the discovered bi-grams.

- One problem with this approach is that the same token (to be deleted) might appear multiple times in the erroneous line. To handle this, we generate all combinations of edits possible on the line and then compile all of them to see which one works.
- One issue with this solution is when the number of combinations become too large, it might slow down the pipeline. To avoid this, we define a threshold (hyper-parameter). If the number of combinations become larger than this threshold, we fall back to the repair profile and use it to make the repair. Note that we compute the number of combinations that would be generated before actually generating them hence this doesn't slow down the pipeline even when we have to fall back to the repair profiles.
- This gave us a significant boost in accuracies.

3.6 Synthetic Data generation:

- Techniques like MACER [5], used to solve automatic program repair problem rely on aligned training data i.e., training data must have both source and target programs.
- However, available aligned training data is limited in size (of the order of 10K).
- DrRepair [16] tries to solve this problem using a self-supervised approach. Briefly, they scrape the internet to get source programs and introduce errors in them using a novel corruption module. This technique allows them to get a lot of aligned data points (order of 1 million).
- However, since the corruption module introduces errors randomly (by deleting/ inserting/ replacing tokens) it is difficult to say if the generated synthetic data has a good representation of errors made by novice programmers.
- We introduce our own synthetic data generation module which uses the concept of repair classes to generate more training data by reproducing errors already encountered in the training set. Since these errors were committed by novice programmers themselves, these new training points capture the actual errors in a much more reliable way.
- Some examples of generated synthetic data are included in Table 3.1

3.6.1 Generating Synthetic examples using repair classes.

- Method of generating new training examples vary based on the type of repair class we are trying to introduce:
- **Insert repair classes:** Each insert repair class has tokens to be inserted in an incorrect program line. Thus, to generate an example of this class we need to remove these tokens from the correct line of the program. We do this as follows:

  For each correct program in training data, we iterate over all lines and if any line contains all tokens to be inserted in the insert repair class, delete these tokens from the line, thus creating a new example of corresponding insert repair class.

- **Replace repair class:** Each replace class has a set of tokens to be deleted and another set of tokens to be inserted in place of these deleted tokens. Thus, to generate a new example of replace class we replace the tokens to be inserted with the tokens to be deleted as follows:

  For each correct program iterate over all lines and if any line contains all tokens to be inserted for the repair class, replace these tokens with corresponding tokens to be deleted, thus creating a new example of the corresponding replace repair class.

- **Delete repair class:** Here we want to insert the tokens appearing in the delete repair classes into a syntactically correct line. Inserting these tokens randomly in any line does not make sense since the generated data point will not be related to any actual error made by a programmer and will not create realistic errors. Since we do not have a straightforward way to determine which lines, we should do these for, we have used the following method.

  We have around 3000 examples with delete repair needed in the available training data.

  First, we train an One vs Rest (OVR for short) binary classifier which given a syntactically correct line and a delete repair class tells whether this class should be applied to given line or not.

  Then we train a set of OVR models (taking inspiration from MACER's [5] repair profiles) which tell us where (in which bi-gram) in the given erroneous line should we insert the tokens appearing in given delete class.

  We use the above to models to generate new examples for delete repair classes.

  Let's consider an example from the table (row 3). The original line is an else statement free from errors. We want to generate an example of repair class which deletes a pair of parenthesis (delete ( )). First we use the corresponding binary classifier for this class to determine whether this repair class can be applied to the given source line. If yes, then we use the corresponding OVA classifier for this repair class to determine which bi-grams in the given lines should be edited. The model in this case returns the bi-gram < else { >. So we insert the pair of parenthesis after the else keyword thus generating a new example for this repair class.

- **Rest repair class:** Rest or miscellaneous repair classes require more than one edit (insert/ delete /replace) operation. To create synthetic data of rest repair classes we follow the following steps:

  We first split the rest repair class into three parts having insert, delete and replace operations respectively.

  For insert and replace parts we follow the same steps as a insert and replace repair class respectively.

  For delete part, we randomly insert the tokens in a given line. Note that we don't need a classifier to tell us which lines should be used to insert these tokens since a rest class is guaranteed to have at least one of insert or replace parts along with this delete part.

- Since repair classes are in abstract form, we need to concretize the final buggy target program generated, before adding it to the training set.

- There are a couple of advantages to generating data this way:

  One is that it generates realistic errors as discussed before.

  The second advantage is that if we look at the distribution of repair classes, we observe a heavy tail i.e., a lot of the rare repair classes have very few training examples. By generating training data using these repair classes we give the ML model much more data to work with for these tail classes, thus improving accuracy for these.

  Moreover, to determine how many synthetic training examples to generate for each repair class we use the distribution of errors itself. Briefly, we generate more training examples for classes which are rare (in the training data-set). For this we use the concept of binning. We split the training data-set into same sized bins. Now we place each training example in a bin depending on how rare the repair class for that example is. We try to keep the sizes of each bin approximately the same. At the same time, if two examples belong to the same repair class they must go in the same bin. Once we have obtained the bins, we can use them to determine how many synthetic examples should be generated for repair classes in each bin. For now, we generate 10 extra examples per repair class for top 30% of bins, 25 extra examples for repair classes in next 30 % of bins and 50 examples of the remaining classes.

- **Further Work:** While generating new examples for insert, replace and rest (misc.) classes we don't consider the order of tokens that appear in the repair class. Basically if all the tokens appearing in the repair class also appear in a target line we apply the procedure discussed above to generate a synthetic training example regardless of whether these tokens appear in the same order or not. However this results in introduction of some new repair classes which were not present in the training set. Though this is not a big issue it can be improved upon.

Original Source Line	Generated Source Line	Repair Class
printf (" %d \n " , age) ;	printf (" %d \n " age) ;	[,][]
scanf (" %f %f " , & a , & b) ;	scanf (" %f %f " , * a , * b) ;	[& &][* *]
else {	else ( ) {	[][( )]
scanf(" %d", &a);	scanf(" %d " &a ,);	[,][,]

Table 3.1: Examples generated using our Synthetic data generation method.

3.7 Atomic Repair:

- A major issue with MACER++ is that it is unable to repair errors which require previous unseen repairs. We call this case Zeros Shot cases.

- These zero shot cases arise because MACER uses composite repair classes. That is a single repair class contains all the edits that need to be made to correct an incorrect line. However, since the type of errors a programmer can make is not bounded, theoretically the number

of possible repair classes would be infinite. Practically, though, there is a pattern of errors made by students.

- However, these zero shot cases do appear in the test data and can affect the pipelines performance significantly.
- Consider the following source line:

```
printf("%d" marks)
```

As you can see there are two, more or less unrelated, errors in this line. One, the missing comma between the string literal and the variable and two, the missing semicolon at the end of line. For MACER++ to be able to repair this line, we would need a repair class that inserts a comma and a semicolon. However, since it is rare for a student to make a mistake like this, its possible that we never find this kind of complex error in our training set and hence this becomes a zero shot case for MACER++ .

- One way to solve this issue is to use atomic repair classes i.e., instead of predicting the entire repair needed to correct the erroneous line we first predict one edit (insert/ delete / replace), perform the edit, and get the partially corrected line, then pass this partially corrected line as an input to predict the next edit and so on till the program compiles or some threshold is met.
- For this the repair class used by MACER++ needs to be broken down into atomic repair classes which represent just a single insert, single delete or a single replace operation.
- We came up with an implementation (naive) for this. Details are as follows.

3.7.1 Data Generation:

- For implementing this pipeline, we would need aligned atomic repair data i.e., where the source erroneous line and the correct target line differ only in one atomic edit operation.
- In TRACER's [1] single line data-set, the source and target programs differ just in one line. However, the source line might require multiple edits to be translated to the corresponding target line.
- We generated the atomic repair data from TRACER's [1] single line data by splitting each source and target line pairs into multiple pairs by using the concept of edit distance. Meanwhile we also had to do some data cleaning since, in case of erroneous programs, the abstraction given by clang is not perfect especially when the program has syntax errors.
- The generated data had around 37K source and target pairs which differ only by one atomic edit operation. Note that the target code in this case may not necessarily compile since more edit operations might be required to fix it.
- We also maintain the order of edit operations while generating the data-set so that first all replace edits are done followed by insert and lastly delete operations are applied.

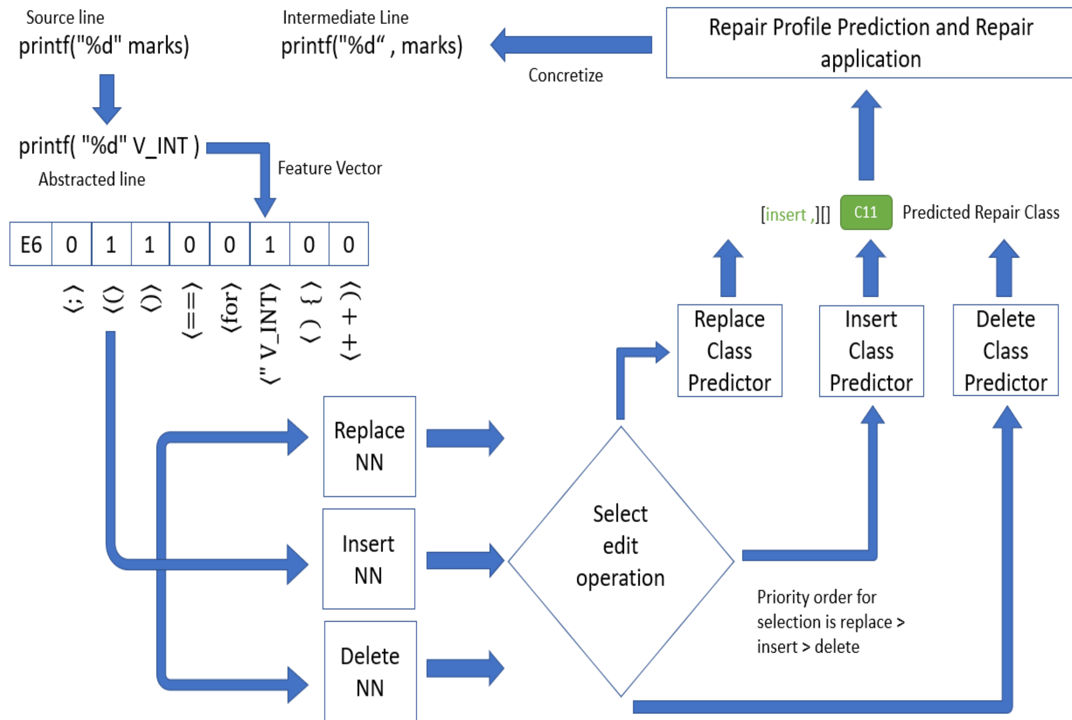


Figure 3.1: Prediction Pipeline for atomic repair

3.7.2 Feature Encoding:

- Used the same feature encoding and program abstraction step as MACER [5].

3.7.3 Training Process and Model:

- To predict whether the given source line requires a (atomic) replace, insert or delete operation we train 3 binary classifiers. Each of these binary classifier is a Fully Connected Neural Network with 2 hidden layers each having 128 neurons followed by a Dropout layer.
- Each of these neural networks are trained on the training examples which require replace, insert or delete operation respectively.
- Then to determine which specific class among the replace, insert, or delete class is required to be applied we train 3 linear models (Logistic Regression) respectively.
- After this we use MACER's [5] repair profile module and repair application module to predict where to apply the repair and apply it.

3.7.4 Prediction Process:

- The prediction process is shown in the Figure 3.1. While predicting we first identify the line number(s) for which the repair must be made using the process described in section 3.2.
- Once we know the set of line(s) is to be edited, we apply the following steps for each line.

- First, we extract the features (uni-grams and bi-grams) and encode it to get the feature vector. Then we ask each Neural network (one for replace, delete and insert) whether the current line needs an edit (replace, delete, or insert resp.). We consider the replace edit to be of the most priority followed by insert and lastly delete (In case if multiple networks say an edit is required).

- Based on which edit is to be made we determine the exact repair class to be applied with the help of the corresponding (linear) model.

- Now we proceed to apply the edit according to the output of the 3 neural networks. For applying the edit we first predict the repair profile just like MACER++ and follow the similar repair application step like MACER++ .

- In the end we get a transformed line(s). This line(s) may not be the correct line and might require further repairs to be applied. Moreover, since the repair application step is not deterministic, we might end up with more than one transformed line in each iteration.

- So, the search space is a general tree. We apply breadth first traversal of this tree till either we find a line that compiles or some threshold number of nodes have been examined.

- Consider the example given in the figure. Here, to repair the erroneous line, 2 edit operations are required. First we need to insert a comma (,) between the variable and the double quotes and second we need to insert a semicolon at the end of line. First we pass the encoded feature vector through the pipeline, which predicts that the repair class is [insert ,]. We use the repair profile prediction and repair application steps as discussed in section 3.5 and section 2.2 to finally get partially corrected line. This partially corrected line then becomes the source line for the next step. This process is repeated till the line compiles (or number of errors in the program are reduced) or some threshold number of attempts have been made.

3.7.5 Future Work:

- Better tree search methods can be applied. Even depth first search might give better results since we would be exploring a path completely before moving on to a new one.

- Using MACER's [5], re-ranking step might help predict the repair class better.

- Moreover we could use a combination of composite and atomic repair classes to get best of both worlds.

- Also, currently we select the edit operation to be applied using this priority order, replace > insert > delete. However, one idea could be to ask the models to provide a confidence score for their prediction the decision to select the edit operation can be made based on this confidence score.

CHAPTER 4

PRIORITY

Contents

4.1 Data Pre-processing: 22

4.1.1 Challenges faced: 22

4.1.2 Observations: 23

4.2 Feature Extraction: 23

4.3 ML Model: 25

4.3.1 Model for label prediction 25

4.3.2 Model for difficulty score 26

4.4 Illustration: 26

4.4.1 Feature extraction and encoding 26

4.4.2 Label Prediction 27

4.5 What lies ahead: 27

4.5.1 Feedback Collection 27

ESC101 is a course offered every semester for novice programmers to teach fundamental principles of programming. In this course labs are conducted every week where students are given programming questions which they have to solve within given time frame. This questions are based on the concepts of programming taught in the ongoing and previous weeks lectures. The tutors (and TA's) of the course are required to formulate these programming questions every week. From all the offerings of this course a large corpora of such questions has been collected using online coding platform PRUTOR [6]. But since these problems are not search able every year the tutors have to start creating these questions from scratch. This takes a lot of effort and time.

The main motive for PRIORITY is to make the large corpus of programs available from previous ESC101 course offering search able. Tutors and TA's should be able to perform a search over these problems based on the skills required to solve these problems, their difficulty etc. This would offer the tutors a good starting point as they can have a look at what kind of problems where used in the previous offerings. This makes the task of setting problems much simpler and faster. To make these problems searchable we came up with 28 programming concepts related tags or labels. In this section we discuss how we tag these problems using some supervised and semi supervised machine learning techniques.

We also developed a web-app so that tutors could easily utilize the problems tagged by our algorithm.

4.1 Data Pre-processing:

- Initially, the (SQL) dump of coding questions (and solutions), collected by PRUTOR [6], was available to us. This data had approximate 2000 questions along with their solution codes and some other data.
- Along with these, with the help of some tutors, we were able to manually label a small set of problems (around 10-15% of the total data samples available).
- These labels represented skills and knowledge of programming concepts that would be required to solve a given problem. The tutors were also instructed to give a difficulty score (1-5) to each problem they labelled. We had a total of 10 major labels, each of which was further divided into 2 to 4 minor labels.
- For example, one of the major label was Conditionals basically indicating that a student requires knowledge of Conditional statements to solve the question. This major label was further divided into 4 sub labels, one or more of which may apply, viz.
  1. Basic : simple if/if-else statements
  2. Switch : use of switch statements
  3. Advanced : use of nested conditionals, ternary statements
  4. Flag : use of flags
- A full list of the labels and there description can be found in the Appendix section 7.1.
- First task, before moving on to applying Machine learning models, was to combine these two sources of data, separate the labelled and unlabelled samples, and convert the data in appropriate form for the Machine learning pipeline.
- Finally the data obtained, after all the pre-processing, was a csv file with information like the problem statement, solution code, template code and a list of labels (if labelled) for each of the coding question in the PRUTOR data-set.

4.1.1 Challenges faced:

- Overlap in the labelled set: Since the data was labelled by multiple tutors, some data points were (deliberately) given to more than one tutor. While cleaning of data we had to decide how to resolve such conflicts where different tutors gave different labels to the same coding question. We resolved this conflict by taking a union over the labels given by the tutors. For determining the difficulty we took the average of the difficulty scores given by the tutors (rounded to closest integer).
- Encoding/ decoding issues: The data dump available from PRUTOR was a SQL database table with entries corresponding to each problem statement along with its solution and some other information. These problem statements and solution codes where stored as a base 64 encoded string. Some these strings were not properly encoded and hence that data sample had to be discarded.

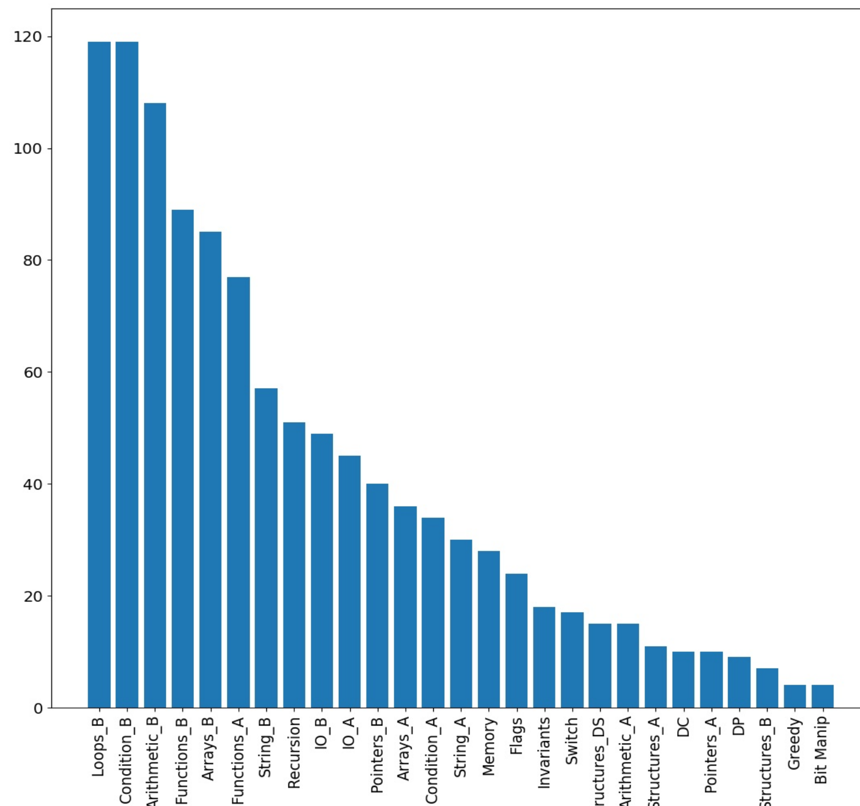


Figure 4.1: Distribution of labelled examples among labels

4.1.2 Observations:

- As you can see the amount of labelled data was very low. Moreover if we split the data points based on the tags (labels) given to them by tutors, we can see in Figure 4.1 that most of the labelled data points belong to only a few popular labels like conditionals, basic loops, terminal i/o etc. while some of the rare labels like DP algorithms, Divide and conquer algorithms etc have less than 10 labelled points. In fact, the 8 most rarest labels have less than 20 data points.

- Hence, if we train a binary classifier, which differentiates the examples which is tagged with the label from the ones that are not, it suffers heavily from a class imbalance problem for these rare labels.

4.2 Feature Extraction:

- After data pre-processing, for each problem we had a problem statement and the solution (gold) code. We decided to use only the gold code to create the feature vector as the problem statement often has a lot of irrelevant information since the tutors are instructed to make a nice story around the problem.

- One (naive) way to encode the solution code is to treat the entire code as a text, and use Bag

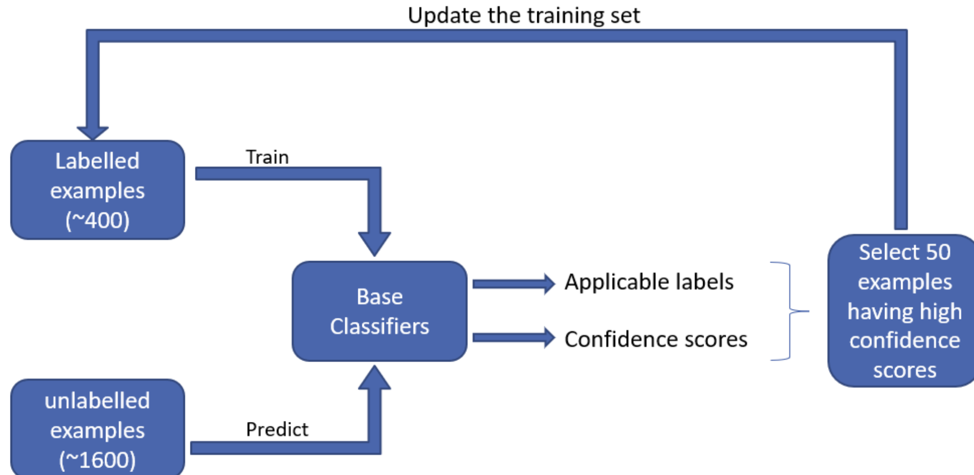


Figure 4.2: Label propagation to utilize unlabelled examples

of words type features (like uni-grams, bi-grams, etc). This method however fails to capture the syntactic details of the source code and also produces very long feature vectors.

- For example, consider the statements:

  z =-1

  z -=1

- The above statements would have very similar uni-gram/bi-gram representations but mean completely different things. One initializes the identifier z with negative 1 and other decrements the value of z by one.

- Hence, we extract the features from the solution code with the help of the abstract syntax tree (AST).

- Abstract Syntax Trees provide a wealth of information about computer programs. It especially helps in analysing the syntactic structure of computer programs.

- We use Pycparser [3] to walk the AST of the solution code and extract relevant features. We use the list of labels (given in appendix chapter 7) to decide what syntactic constructs can help predict the labels. We extract information like number of conditionals used, number of logical & arithmetic operators used, number and dimensions of array's used etc. Apart from these we also check whether the code uses in-built functions like printf, scanf, memory operations etc. which can be a valuable hint towards determining the skills required to solve the problem.

- We encode this information using a CountVectorizer from sklearn to get a 50 dimensional feature vector.

4.3 ML Model:

In this section, we describe the models we used and experimented with to predict both the labels and the difficulty scores of a given programming problem.

4.3.1 Model for label prediction

- The problem at hand is effectively a multi label classification problem where given a feature vector representing the solution code we have to predict which of the 28 labels are applicable and also predict the difficulty score (multi class classification).

- Since we have such a limited labelled data points most of the well known algorithms for supervised multi label classification give us a poor performance. We tried using One vs rest (OVR) approach with different base models. However all these methods gave a poor performance. This could be because, one the amount of labelled data is very little and second the data is heavily skewed in favour of only a few popular labels.

- Another approach we tried is Multi label k nearest neighbours (MLKNN) [17]. This approach is an extension of the k nearest neighbour approach to work with multi label classification settings. MLKNN out-performed the One vs Rest methods with k = 10. However the F1 score was still quite low.

- One disadvantage of these supervised algorithms is that they do not use the unlabelled data points. To utilize these unlabelled data points we used a semi supervised learning technique called Label propagation as illustrated in Figure 4.2.

- For each label we first train a Base classifier on the labelled data points which is 400 examples. Then we predict the labels for the remaining (unlabelled) points which are 1600 examples. We also ask the model to provide confidence scores to all of its predictions. Based on these confidence scores we pick the top 50 examples on which the model is most confident on and add them to our training set. Hence, the training set now consists of 450 samples. We retrain the Base classifier on this updated training set and repeat the process.

- At the end of each iteration, 50 unlabelled points are moved to the training set. We continue this process till all the unlabelled points are labelled.

- We experimented with different Base classifiers but found Balanced Random Forest Classifier to work best. As noted earlier, the rare labels suffer from class imbalance problem. This can hurt a machine learning models performance. A balanced random forest classifier tries to solve this problem by using techniques like over-sampling. Hence, it performs better than a traditional Random Forest classifier.

- A disadvantage of this technique is that it may amplify the base classifiers incorrect predictions (if the model is very confidently wrong).

4.3.2 Model for difficulty score

- Difficulty of a problem is a number between 1 to 5, 1 indicating a very easy problem and 5 indicating a very challenging problem.
- Predicting a difficulty score for a given programming problem is a simple classification problem with 5 classes.
- We experimented with a few multi class classification algorithms and found Random Forest classifier to work best.
- We use the same feature vector for difficulty prediction since the syntactic structure of a program is often a good indicator of how difficult it would be to solve a problem.

4.4 Illustration:

In this section we will go over the entire labelling process by taking an example. Consider the following programming problem

> Write a C Program to compute the area pi*r*r, where the float values are pi=3.14159265, r is a user input and print the resulting area.
> *Input: A floating point number representing radius r of the circle.*
> *Output: Area of circle with radius r. output should contain only 2 digits after the decimal.*

The solution code for above problem would look like this:

```
1  #include<stdio.h>
2  int main(){
3    float pi, r, a;
4    pi=3.1415926;
5    scanf("%f", &r);
6    a = pi*r*r;
7    printf("Area of circle with radius %0.2f is %0.2f\n", r, a);
8
9    return 0;
10 }
```

To solve the above problem a student would require knowledge about the following programming concepts:

1. How to take input from the user and output the answer to the user using the correct format specifiers specified by the question
2. Knowledge about basic arithmetic operators to calculate the area.

4.4.1 Feature extraction and encoding

As discussed in section 4.2, to extract the features from the solution code we generate an Abstract Syntax Tree using Pycparser [3] and walk this AST to extract relevant information. For the given

example, our approach extracts the following information:

- Use of Basic function (main).
- Use of assignment operator.
- Use of console input (scanf)
- Use of console output (printf)
- Use of arithmetic operator (*)
- Use of dereference operator (&)

We encode these features to get our feature vector.

4.4.2 Label Prediction

Once we have our feature vector, we feed it to the Balanced random forest classifier model, trained using label propagation. For each of the 28 labels the model will output a 0 or 1 indicating whether this problem should be tagged with the label or not. For the given example, the model outputs 2 tags viz. Arithmetic_Basic and TerminalIO_Advanced, indicating that the problem requires basic knowledge of arithmetic operators and advanced knowledge of terminal (or console) input/output.

Passing the feature vector to the difficulty score prediction model, the model assigns a difficulty score of 1 to the problem. Difficulty of a problem can be subjective and can vary based on the opinion and proficiency of the student, but the problem at hand is relatively simple and a difficulty score of 1 is justifiable.

4.5 What lies ahead:

As mentioned earlier, we deployed a web-app which exposes the labelled data-set of programming problems, labelled using PRIORITY, to tutors of ESC101 course. Details of this web-app can be found in Sharath's Thesis Real World Deployments of AI assisted compilation error repair and program retrieval [10]. This web-app is currently used by the tutors of ongoing ESC101 offering.

Since most of the problems in this data-set were labelled by the machine learning model, chances are that some of the labels would be incorrect i.e. the model may tag some problems with a label which doesn't apply to the problem or the model may also fail to apply a label to the problem which should have been applied. Hence, one of the important aspect of the web-app is feedback collection.

4.5.1 Feedback Collection

The web-app collects 3 types of feed backs which are as follows:

1. Active Feedback: If the tutor feels that the labels of a problem are incorrect, the web-app allows him/her to specify which labels should/should not be applied. This type of feedback can help the Machine Learning model directly since the tutors are indirectly labelling the

data. Note that it may not be the best idea to take the tutors suggested labels as ground truth and some clever filtering might be needed.

Besides this tutors can also give a star rating to a specific suggestion made by the model. A 5 star rating would mean that they were completely satisfied with the labels while a 1 star rating would mean that there was some mistakes in the labels.

2. Passive Feedback: It's common for any recommendation system to collect passive feedback from its users based on how they interact with the system. PRIORITY collects some passive feedback as well. For example, the web-app records data like how many problems the user had to visit before they were satisfied, how much time a user spent on one problem, did the user copy the problem statement or code, etc. These feed backs can provide valuable hints to whether the users liked the suggestion made by the model or not.

3. Text Feedback: Finally, the web-app allows the tutors to leave comments on a specific search results or on the entire web-app in general.

Please refer to Sharath's Thesis for more details on feedback collection.

All these feed backs can help improve the machine learning model drastically. For example, if the user spends a longer time on a specific question then probably they are going through the details of the question and hence chances are high that they liked the suggestion. The next step of PRIORITY would be incorporate these feedback to further improve the Machine learning models.

CHAPTER 5

Experiments

Contents

5.1 Data-sets **29**

5.2 Comparison with MACER: **30**

5.2.1 Short-hands: 30

5.2.2 Localization Accuracy: 30

5.2.3 Repair Accuracy 30

5.2.4 Pred@k metric 31

5.2.5 Accuracy on tail labels: 31

5.3 Comparision with DrRepair: **32**

5.4 Optimizations that didn't work: **33**

5.4.1 Learned Hierarchy: 33

5.4.2 Label Features: 33

5.4.3 Atomic repair: 33

5.5 Priority results **34**

We applied all the optimizations discussed in chapter 3 for MACER++ and compared the results with base MACER's [5] results as well as with other state of the art methods. In this chapter we describe all the results. We also discuss some ablation studies that we perform. For PRIORITY we discuss why we chose to go with Label Propagation and compare its performance with a few other methods.

5.1 Data-sets

We ran our experiments primarily on TRACER's [1] single line data-set (now on referred to as single line data). This data-set contains 4326 problems each of which has exactly one erroneous line. Apart from this we also compare MACER++ performance on Deepfix [8] data-set. This data-set has 6996 problems which have one or more erroneous lines.

For PRIORITY our data-set has roughly 2000 problems out of which approximately 460 are labelled. We withhold 60 of these as test set and train the model on remaining 400 problems (and also on unlabelled problems in case of semi supervised learning). We report accuracies on these 60 problems. Each of these problems can have one or more of 28 possible labels (section 7.1) as well as a difficulty score.

5.2 Comparison with MACER:

In this section we talk about the optimizations which gave us an improvement (in terms of various metrics) over the reported metrics of the base MACER pipeline.

5.2.1 Short-hands:

Here are some short hands used in later sections.

- Error Line Localization (ELL): Better Line number localization using identifiers/ symbols in compiler error message
- Without error id (weid): Revamped repair classes without error id
- Synthetic Data(md): Used artificially generated data while training
- Repair application(ra): Improved repair application for delete and replace classes by examining tokens in the predicted repair class

5.2.2 Localization Accuracy:

The localization Accuracy refers to how accurately did the pipeline identify the current line which requires repair. Table 5.1 compares the localization accuracy of MACER++ with that of the base MACER [5] pipeline.

As you can see, using compiler message to extract symbols and then using these symbols to identify long range errors does indeed increase the localization accuracy by 3% on the single line data.

Model	Single Line Data
	Localization accuracy
Macer	93
Macer + ell	96

Table 5.1: Better Line Localization using compiler diagnostic

5.2.3 Repair Accuracy

The Repair Accuracy of the model indicates the ratio of programs from the test data which the model was able to repair, i.e. the program compiles after the transformation, to the total number of programs in the data set. Repair@k indicates the repair accuracy while considering the top k repair classes predicted by the model. We accept or reject this predictions by checking if the transformed code compiles or not. The table compares the repair accuracy of MACER++ with that of base MACER [5] pipeline, as well as an ablation study on which optimization gave how much improvement.

As you can see from the table, using compiler feedback as well as using synthetic data improves the repair accuracies on both data sets. Also better repair application by using the repair classes greatly boosts performance of the pipeline.

Model	Single line data		Deepfix data
	Repair @ k		Repair @k
	K = 1	K = 5	K = 5
Macer	69.3	80.2	55.7
Macer + ell + weid	71.3	80.9	57.4
Macer + ell + md + weid	72.1	81.7	58.1
Macer + ell + weid + ra	**73.4**	83.7	59.6
Macer + ell + weid + ra + md	73.3	**84.6**	**59.8**

Table 5.2: Ablation Studies for MACER++

5.2.4 Pred@k metric

The Pred@k metric was introduced by MACER [5]. It indicates the ratio of programs from the test data which the model got exactly correct, i.e. the target code matches exactly with the repair made by the student, to the total number of programs in the data set. Again, the value of k represents how many repair classes predicted by the model were considered. Table 5.3 reports the Pred@k measure of MACER++ along with that of MACER [5] and provides an ablation on which optimization was more beneficial.
Note that MACER [5] reported the Pred@k measure assuming that we know the gold error line i.e. we don't need to localize which line requires repair. However, since MACER++ improves the line localization module, we report Pred@k measure both when we know the gold error line numbers as well as when we don't.

As you can see in the table, these optimizations not only improve repair accuracy but also improve the Pred@k metrics. Thus the improvements not only succeed in repair the erroneous line but also help improve the prediction of what the student actually wanted.

	Gold Lines	Active Localization
	K = 5	K = 5
Macer	69.3	60.9
Macer + Weid + ell	69.3	63
Macer + Weid + ell + md	69.9	63.6
Macer + Weid + ell + ra + md	**71.5**	**65.1**

Table 5.3: Ablation studies for MACER++ using Pred@k measure

5.2.5 Accuracy on tail labels:

All the above metrics are macro measures i.e. they are calculated on the entire dataset, irrespective of the class the repair belongs to. However, as the shown in Figure 5.1, there is a heavy tail in the distribution of examples among the repair classes. Because of this it is easy for the model to have high numbers in macro measures even though the model fails miserably on tail repair classes. Hence, we need some way to compare these models performance on the tail repair classes as well. We propose the following method:
The single line data has approximately 3.4k examples (ignoring the zero shot cases). The goal is to divide these examples into 10 bins of roughly the same size, however we want to keep the

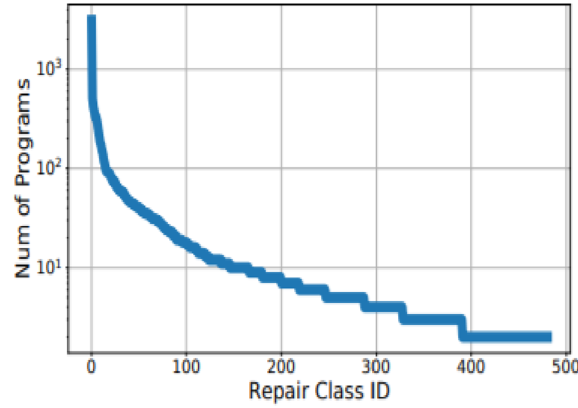


Figure 5.1: Heavy tail in repair classes

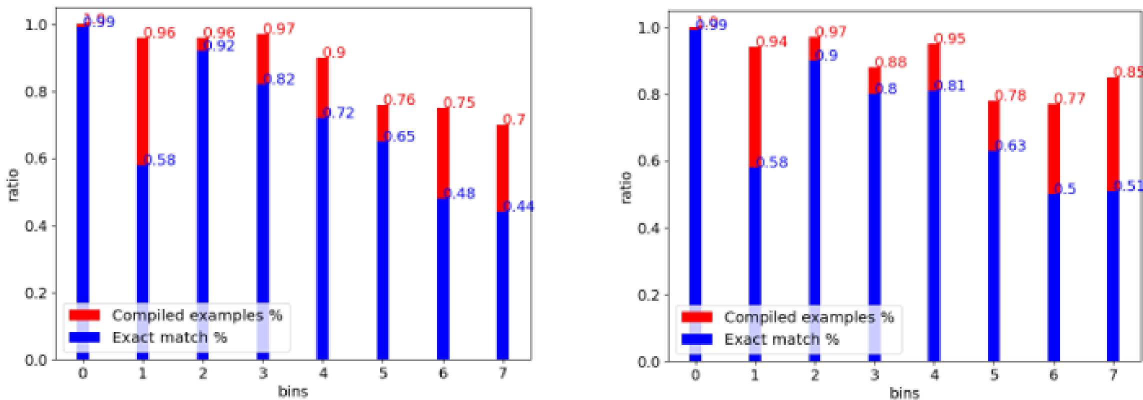


Figure 5.2: Comparision of repair and pred accuracies for MACER vs MACER++ (MACER++ on right)

examples belonging to the same repair class in the same bin. For this we identify the repair class of each of the examples (since we know the target program). We sort the repair classes based on how many examples of that class exist. Then we pick all examples of the most popular repair class and place them in bin 0. Then we pick all examples of next repair class and place it in the same bin. We keep doing this till the bin is full (i.e. has more than total examples / 10 examples) then we move to next bin. In our case the first bin has examples from only 1 repair class while the last bin has examples from around 100 repair classes. Now we calculate the repair accuracy and Pred@k measure for each of this bins separately. Figure 5.2 compares the repair accuracy and exact match accuracy (which is the Pred@k measure without using gold line numbers) of MACER++ with that of MACER for each bin. As you can see, MACER++ improves the accuracies for the tail classes.

5.3 Comparision with DrRepair:

We compared MACER++ with DrRepair [16], one of the state of the art method for compilation error repair. Table 5.4 shows that MACER++ significantly outperforms DrRepair [16] on single line data. Even though it falls quite short of DrRepair on Deepfix [8] data-set, its important to note

that DrRepair is a heavy duty sequence to sequence model which requires a lot of resources (time and compute) to train as well as test. We trained the best model of DrRepair on a Volta class GPU and it took > 5 days to train while MACER++ trains in approximately 10 minutes on a CPU.

	Repair Accuracy		Training Time	Testing time
	Tracer Single Line Dataset	Deepfix Dataset	(For best model)	(one example)
Dr Repair	64.5	68	> 5 days	~5 seconds
MACER++	84.5	59.8	1̃0 minutes	~1 second

Table 5.4: Comparison of MACER++ with DrRepair

5.4 Optimizations that didn't work:

In this section, we discuss the optimizations which seemed promising but didn't provide any improvements over the base MACER [5] pipeline.

5.4.1 Learned Hierarchy:

We tried replacing MACER's [5] repair class prediction module, which uses PLT [11] based fixed hierarchy of labels to predict the repair class, by using algorithms which learn the hierarchy of labels. As you can see Table 5.5, this experiment shows that for this use case using a Fixed hierarchy of labels works much better than letting the algorithm learn the hierarchy between the labels.

	Repair Accuracy
PLT (original MACER)	80.2
Parabel	77.61
CraftML	76.8

Table 5.5: Comparison among different methods for repair class prediction

5.4.2 Label Features:

In this section we compare how using the label text (in this case repair classes) affects the performance of the model. As shown in the Table 5.6, using the label metadata gives use improvements for repair@1 accuracies. However, the performance for repair@5 is poorer as compared to when not using the label metadata. One of the reason for this could be that our label representation isn't capturing enough information and a better label representation might be needed. Nonetheless, this approach can be examined further since the difference in performance is very little.

5.4.3 Atomic repair:

MACER++ , even after applying all these optimizations, still suffers from zero shot cases. As discussed earlier to overcome this problem we tried to break down the repair into elementary

	Repair Accuracy	
	K = 1	K = 5
MACER++	72.1	83.7
DECAF	73.4	82.7
Naive re-ranker using label features	73.2	83

Table 5.6: Effect of using Label features with MACER++

operations. Each elementary operation is characterized by an elementary class. After generating the data (as discussed in section 3.7), we found approximately 600 elementary classes. Applying the method discussed in section 3.7, this new approach gave an accuracy of 50.2% on the single line data. Even though this is considerably less than what MACER++ achieves, its still has a lot of room for improvement. In theory, this method should perform at least as good as MACER++ . This is because all the repair classes formed by MACER++ are some combination of these elementary repair classes. The reason for the sub optimal performance of this method could be:

- Improper search method: As discussed earlier we use BFS to search the search tree, but some other search method might perform better.
- Need for better feature encoding: Using BoW type of features might not be powerful enough to train the 3 neural networks and we might need to experiment with more sophisticated methods.

5.5 Priority results

	F-score	Accuracy
MLKNN	37.75	90.1
Balanced RFC	42.07	87.14
Balanced RFC + label propogation	49.4	88.51

Table 5.7: Comparison of using different methods for label tagging

We tried 3 different algorithms for the label tagging problem. We report the performance in terms of accuracy as well as F-score. Considering a metric like F-score is important because the data suffers from a class imbalance problem especially for the rare labels.

The first algorithm is Multi-label k Nearest Neighbours (MLKNN) [17]. This algorithm is an adaptation of the k nearest neighbour algorithm for multi label classifciation problem. Though this method gives a high accuracy, it suffers from a low recall and hence a low F-score.

The other method is to use a balanced Random Forest Classifier. This method uses a traditional Random Forest Classifier but uses techniques like under-sampling and over-sampling to deal with class imbalance problem. Finally, to better utilize the large number of unlabelled example, we pair the Balanced Random Forest Classifier model with Label Propagation explained earlier and as shown in the Table 5.7, this gives use the best performance.

CHAPTER 6

Conclusion

In this thesis we present MACER++ which proposes various optimizations over MACER [5]. Table 5.2 shows that these optimizations give a significant improvement in performance over the base pipeline. Also, Figure 5.2 shows that our method perform significantly better on rare repair classes. Moreover we explore the use of labels metadata to provide additional information which can be an interesting area to look at for further improvements.

MACER++ also proposes a novel synthetic data generation method which can target specific errors and generate examples for them. This can help with few shot cases where the amount of training data is very low. We also explore the concept of atomic repair which can potentially work on zero shot cases which the current pipeline fails to repair.

We also propose an algorithm that labels the given programming problem with tags indicating the skills required to solve these problems. The algorithm uses label propagation to learn not only from labelled examples, which are scarce, but also from unlabelled examples. As shown in companion thesis, this algorithm can help create tools which can be of great help to tutor's and TA's.

Bibliography

[1] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: for the student programs, from the student programs. *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2018.

[2] Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. Targeted example generation for compilation errors. *In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 327–338, 2019.

[3] Eli Bendersky et al. Pycparser: https://github.com/eliben/pycparser.

[4] Lijia Chen, Pingping Chen, and Zhijian Lin. Artificial intelligence in education: A review. *IEEE Access*, 8:75264–75278, 2020.

[5] Darshak Chhatbar, Umair Z. Ahmed, and Purushottam Kar. MACER: A modular framework for accelerated compilation error repair. *CoRR*, abs/2005.14015, 2020.

[6] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani. Prutor: A system for tutoring CS1 and collecting student programs for analysis. *CoRR*, abs/1608.03828, 2016.

[7] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 07 2012.

[8] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. *In 31st AAAI Conference on Artificial Intelligence (AAAI)*, pages 1345–1351, 2017.

[9] Hossein Hajipour, Apratim Bhattacharyya, and Mario Fritz. Samplefix: Learning to correct programs by sampling diverse fixes. *CoRR*, abs/1906.10502, 2019.

[10] Sharath HP. *Real World Deployments of AI-assisted compilation error repair and program retrieval*. M.tech. thesis, Indian Institute of Technology Kanpur, 2021. unpublished thesis.

[11] Kalina Jasinska, Krzysztof Dembczynski, Róbert Busa-Fekete, Karlson Pfannschmidt, Timo Klerx, and Eyke Hullermeier. Extreme f-measure maximization using sparse probability estimates. *In 33rd International Conference on Machine Learning (ICML)*, pages 1435–1444, 2016.

[12] Anshul Mittal, Kunal Dahiya, Sheshansh Agrawal, Deepak Saini, Sumeet Agarwal, Purushottam Kar, and Manik Varma. Decaf: Deep extreme classification with label features.. *In Proceedings of the Fourteenth ACM International Conference on Web Search and Data Mining (WSDM '21), Virtual Event, Israel. ACM, New York, NY, USA*, 2021.

[13] Yashoteja Prabhu, Anil Kag, Shrutendra Harsola, Rahul Agrawal, and Manik Varma. Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising. *18: Proceedings of the 2018 World Wide Web Conference.*, page 993–1002, April 2018.

[14] Wissam Siblini, Pascale Kuntz, and Frank Meyer. Craftml, an efficient clustering-based random forest for extreme multi-label learning. *Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80.*, 2018.

[15] V. Javier Traver. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 06 2010.

[16] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. *Proceedings of the 37 th International Conference on Machine Learning, Vienna, Austria, PMLR 119.*, 2020.

[17] Min-Ling Zhang and Zhi-Hua Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern recognition*, 40(7):2038–2048, 2007.

CHAPTER 7

Appendix

7.1 Labels used in PRIORITY:

1. Difficulty [1,2,3,4,5]
    - 1. Very easy
    - 2 : Easy
    - 3 : Medium
    - 4 : Difficult
    - 5 : Very difficult
2. Terminal IO [Basic, Advanced]
    - Basic : simple IO with various data-types, use of escape sequences
    - Advanced : pretty patterns/word art/non-trivial formatting, format specifiers e.g. %5.4f or %0.2e, heavily formatted input e.g. (%d-%d-%d) to input (02-12-89)
3. Arithmetic [Basic, Advanced, Bit]
    - Basic : simple arithmetic operations (+,-,*,/,%,++,–), expressions, bracketing
    - Advanced : mixed type operations (e.g. long+int), explicit typecasting, math.h
    - Bit : use of bit-wise operators, left/right shift, bit masks
4. Conditionals [Basic, Switch, Advanced, Flag]
    - Basic : simple if/if-else statements, relational and logical operators
    - Switch : use of switch statements
    - Advanced : use of nested conditionals, ternary statements
    - Flag : use of flags e.g. isSorted, isFirstIteration
5. Loops [Basic, Advanced, In-variants]
    - Basic : simple use of for, while, do-while loops
    - Advanced : nested loops, use of break/continue, use of infinite while loops e.g. while(1)... and loops with empty headers e.g. for(;;)...
    - In-variants : Use of partial sums, running counts, running products and others
6. Arrays [Basic, Advanced, Memory]
    - Basic : 1D numeric arrays, creation, traversal, modification
    - Advanced : 2D or nD arrays
    - Memory : memory management using sizeof, malloc, calloc, realloc, free, stdlib.h
7. Pointers [Basic, Advanced]
    - Basic : referencing, dereferencing, pointer arithmetic
    - Advanced : arrays of pointers, pointers to pointers

8. Char-String [Basic, Advanced]
    - Basic : character IO, character arithmetic, string IO, NULL, EOF
    - Advanced : sub-string manipulation, strings and pointers, string.h
9. Functions [Basic, Advanced]
    - Basic : one or more scalar arguments and scalar return
    - Advanced : pointer/reference/array arguments, pointer/reference/array return
10. Structures [Basic, Advanced, DS]
    - Basic : storing user input in structures, arrays of structures
    - Advanced : pointers to structures, nested structures
    - DS : use/implementation of data structures e.g. linked list, stacks, (circular) queues, trees, graphs possibly using struct, or even using arrays
11. Algorithms [DC, Recursion, Greedy, DP]
    - DC : divide and conquer, bisection search etc
    - Recursion : self/mutual recursion
    - Greedy : greedy algorithms
    - DP : dynamic programming