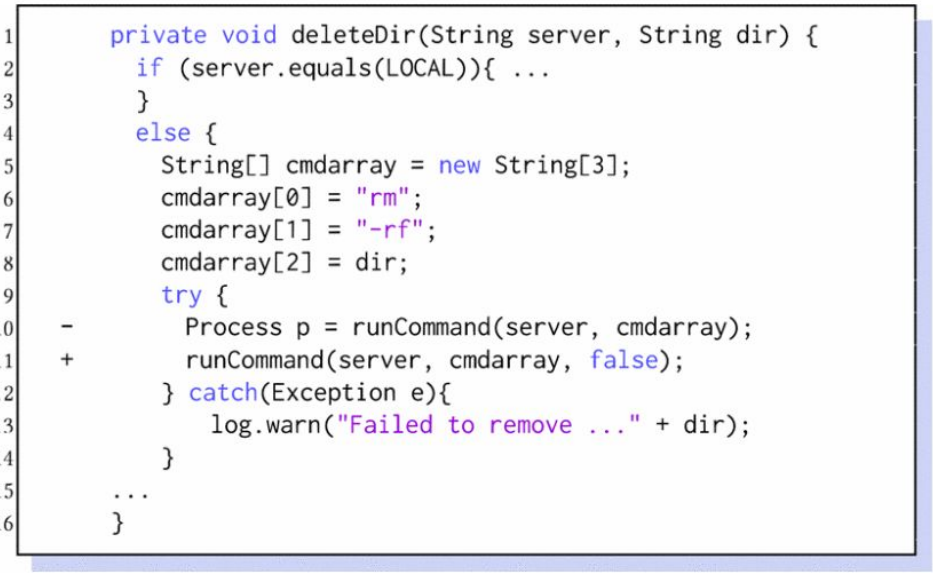
DLFix: Context-based Code Transformation Learning for Automated Program Repair

Yi Li; Shaohua Wang; Tien N. Nguyen

Background : Automatic Program Repair

Automatic program repair approach takes as input a **program** and some **evidence** that the program has a bug (commonly, a failing test) and produces a **patch** for that program's source to fix that bug, ideally without negatively influencing other correct functionality.



```
1       private void deleteDir(String server, String dir) {
2         if (server.equals(LOCAL)){ ...
3         }
4         else {
5           String[] cmdarray = new String[3];
6           cmdarray[0] = "rm";
7           cmdarray[1] = "-rf";
8           cmdarray[2] = dir;
9           try {
10  -         Process p = runCommand(server, cmdarray);
11  +         runCommand(server, cmdarray, false);
12          } catch(Exception e){
13              log.warn("Failed to remove ..." + dir);
14          }
15      ...
16      }
```

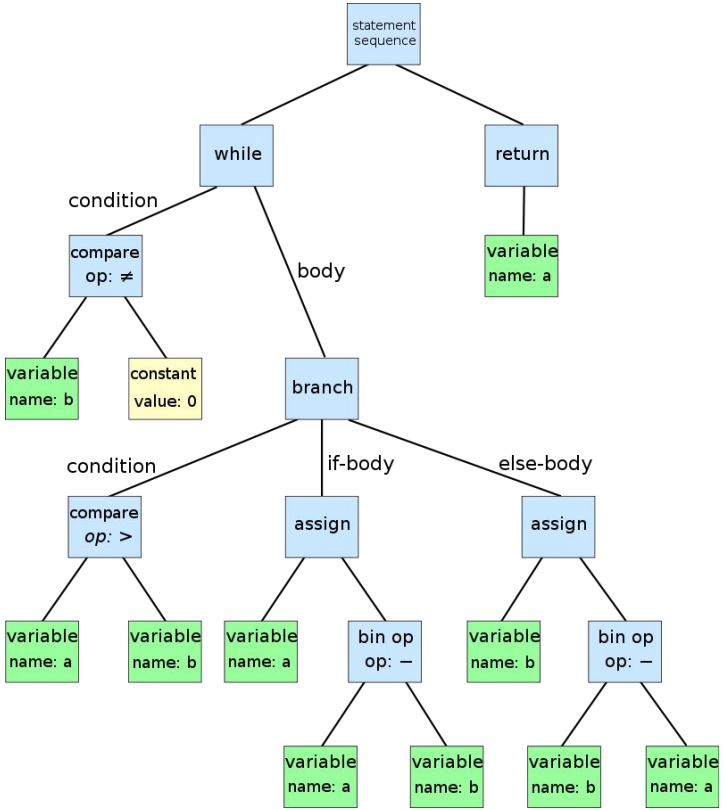
A bug-fixing example

Background: Abstract Syntax Tree(AST)

- An AST is a **tree** model of an entire program or a certain “program structure” (e.g., a statement or an expression in a program)
- Parser generate(tools: esprima, acorn,espree, etc.)
  - Include structural information, variable and function declarations, control flow, data flow, types, etc.
  - Do not include **inessential punctuation and delimiters** (braces, semicolons, parentheses, etc.).
- Representation of the semantics of source code

An example of abstract syntax tree

```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```



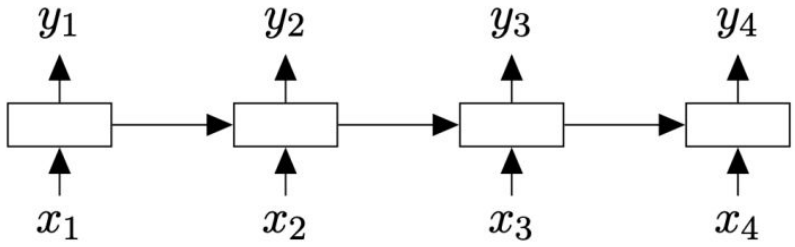
Background: Recurrent Neural Networks(RNN)

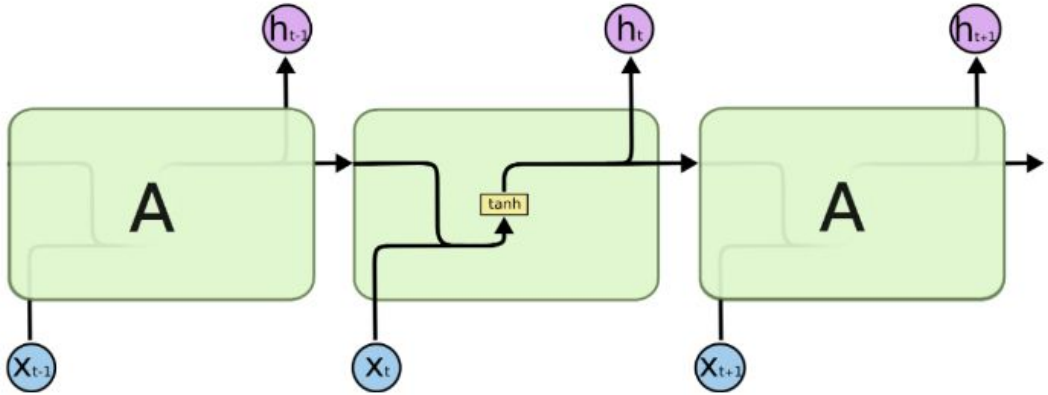
Recurrent Neural Networks(RNN)

$h_t \in \mathbb{R}^n \rightarrow$ hidden state vector

$h_t = \tanh\left(Wx_t + Uh_{t-1} + b\right)$

Exploding or vanishing gradients!





Simple RNN

Background: Long Short-Term Memory(LSTM) Network

Long Short-Term Memory (LSTM) networks

$$i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}\right), \quad \rightarrow \text{input gate}$$

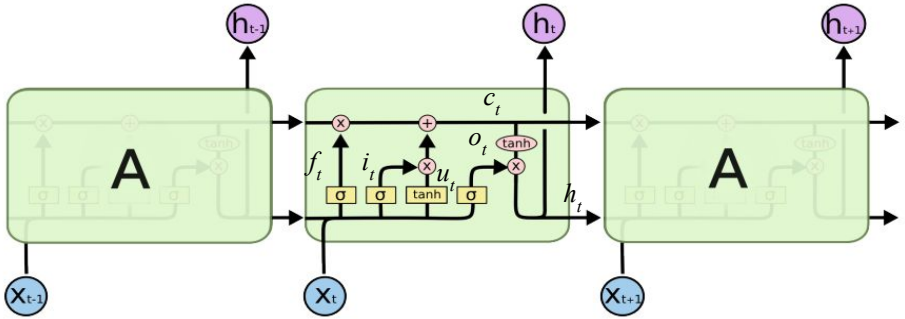
$$f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}\right), \quad \rightarrow \text{forget gate}$$

$$o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}\right), \quad \rightarrow \text{output gate}$$

$$u_t = \tanh\left(W^{(u)}x_t + U^{(u)}h_{t-1} + b^{(u)}\right),$$

$$c_t = i_t \odot u_t + f_t \odot c_{t-1}, \quad \rightarrow \text{memory cell}$$

$$h_t = o_t \odot \tanh(c_t), \quad \rightarrow \text{hidden state}$$



Bidirectional & Multi-layer LSTMs

Tree-Structured LSTMS

– N-ary Tree-LSTM: Tree structures where the branching factor is at most N and where children are ordered (indexed from 1 to N)

– Child-Sum Tree-LSTM: High branching factor or unordered children

Child-Sum Tree-LSTM

$$h^j_{sum} = \sum_{k \in Child(j)} h_k \quad (1)$$

$$i_j = \sigma(W_i x_j + U_i h^j_{sum} + b_i) \quad (2)$$

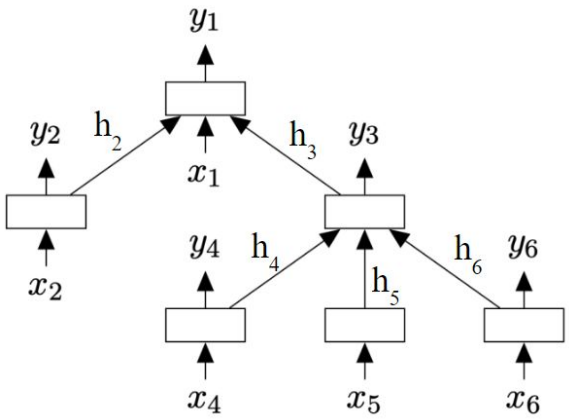
$$f_{jk} = \sigma(W_f x_j + U_f h_k + b_f) \quad (3)$$

$$o_j = \sigma(W_o x_j + U_o h^j_{sum} + b_o) \quad (4)$$

$$u_j = tanh(W_u x_j + U_u h^j_{sum} + b_u) \quad (5)$$

$$c_j = i_j \circ u_j + \sum_{k \in Child(j)} f_{jk} \circ c_k \quad (6)$$

$$h_j = o_j \circ tanh(c_j) \quad (7)$$



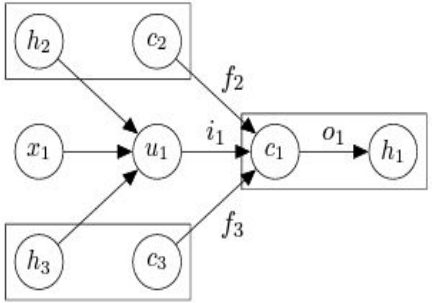


Figure 2: Composing the memory cell c_1 and hidden state h_1 of a Tree-LSTM unit with two children (subscripts 2 and 3). Labeled edges correspond to gating by the indicated gating vector, with dependencies omitted for compactness.

Contribution

- **Deep learning (DL) for Automatic Program Repair (APR)**

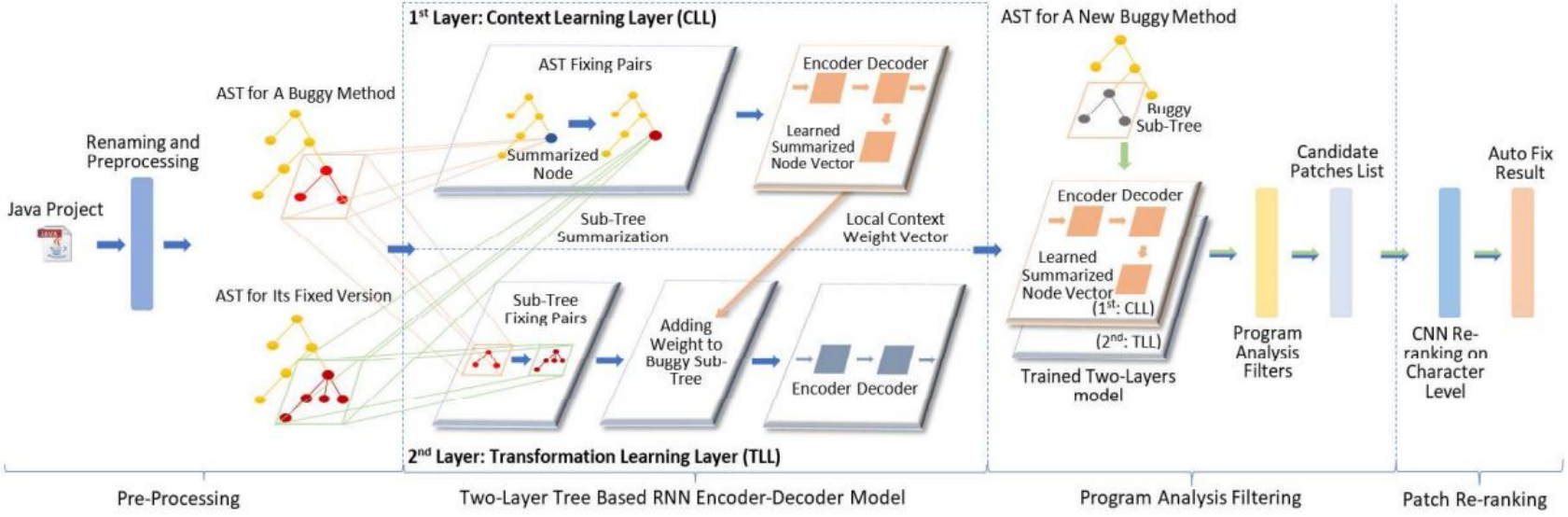
  DLFix is the first DL APR that generates comparable and complementary results with powerful pattern-based tools, as recently published DL-based APR can only fix very few bugs on Defects4J. DLFix helps confirm that further research on building advanced DL to improve APR is promising and valuable.

- **Model**

  A novel DL-based APR approach with a novel two-layer tree-based model, effective program analysis techniques, and Convolutional Neural Network (CNN)-based re-ranking approach to better identify correct patches.

- **Empirical Results**
  - **Comparable and Complementary to Pattern-Based APR**
  - **Improving Over all the DL-Based APR**

DLFix: Context-Based Transformation Learning



Pre-Processing

Step1. Renaming

- increase the chance for the model to learn the fix in one place to apply to another in the similar/same scenarios

Step2. AST Generation

- generate four ASTs: T_b, T_f, T_b^{sub} and T_f^{sub}

Step3. Token Vectors Learning

- use Word2Vec on all the sentences in the corpus to obtain the vectors for all the code tokens

Step4. Sub-tree Summarization

- represent the changed/buggy sub-tree as a single vector representation

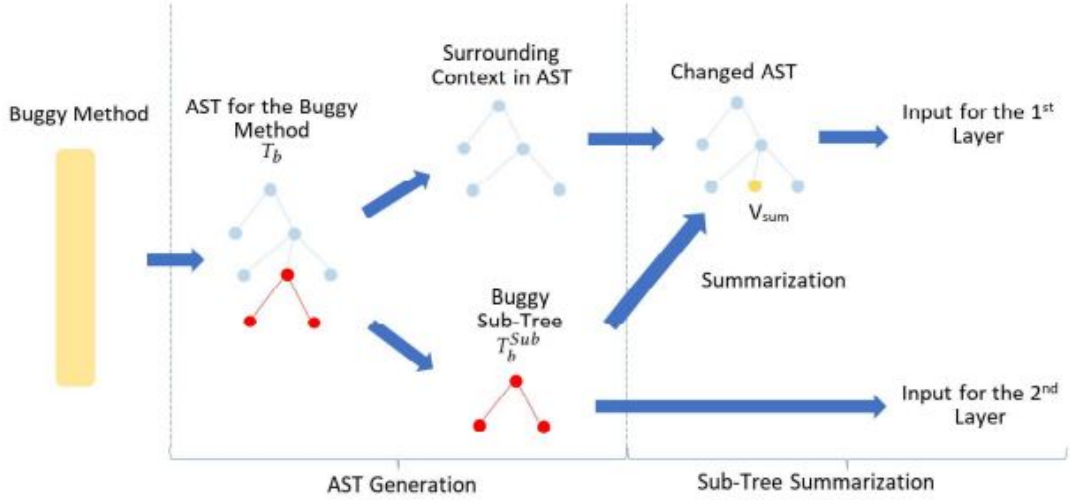


Figure 3: Key Steps of Pre-Processing

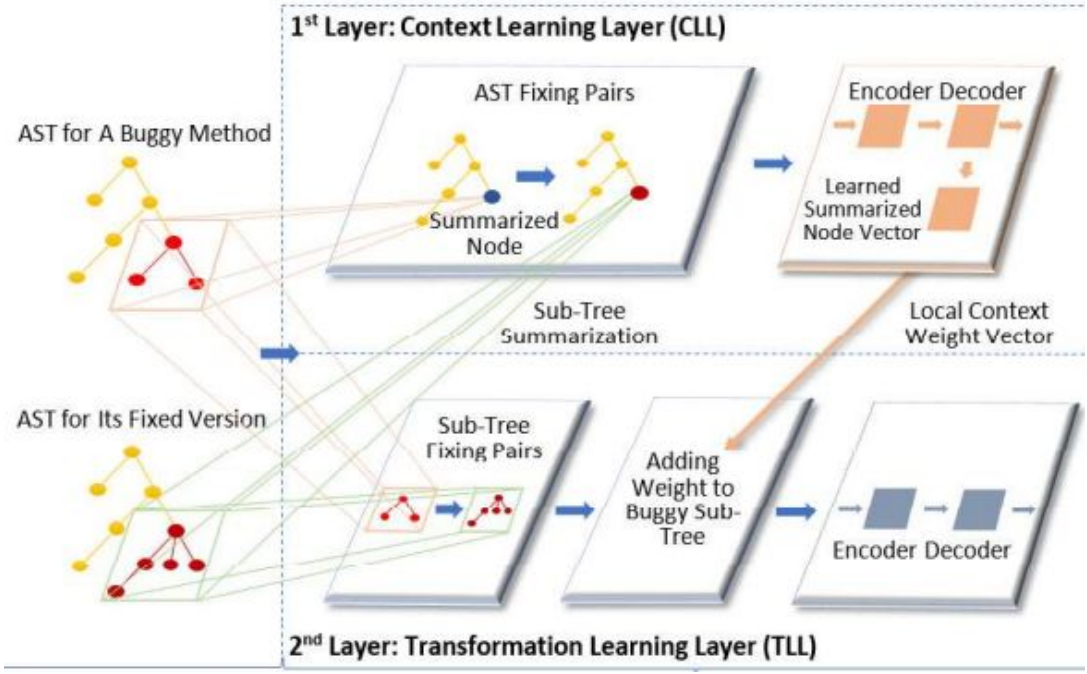
Two-Layer Tree Based LSTM Encoder-Decoder Model

1st Layer: Context Learning Layer

- learn **local context** of the code surrounding the bug-fixing changes
- T_b^{Sub} and T_f^{Sub} are replaced by **summarized nodes**
- encoder-decoder model with a **tree-based LSTM**

2nd Layer: Transformation Learning Layer

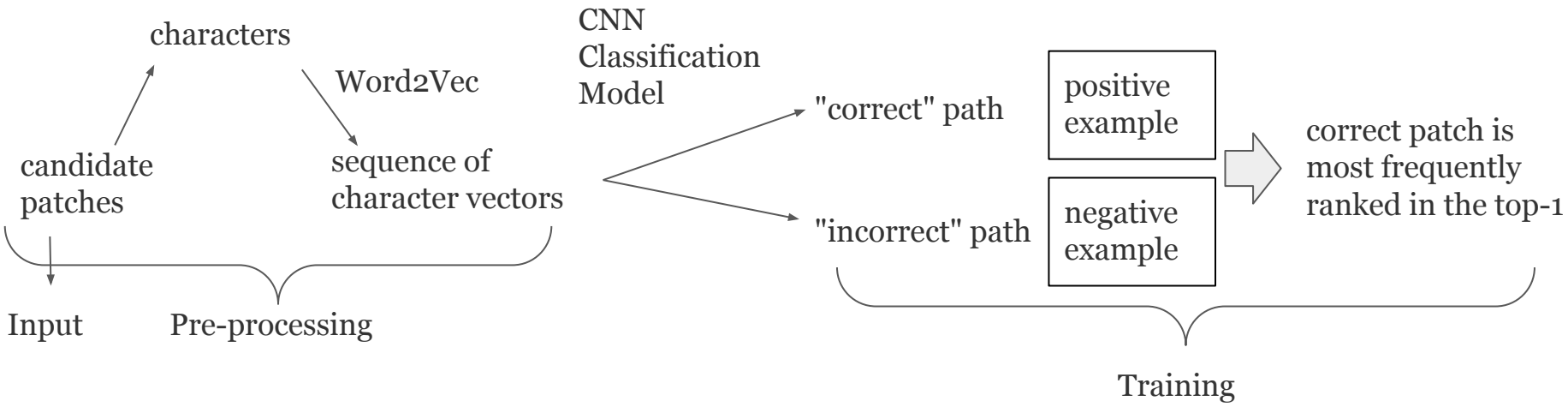
- learn the **code transformations of the bug fix**
- use **cross-product multiplication** to combine the weight vector with the vector of each AST node in a sub-tree



Program Analysis Filtering

- Goal:
  - recover the candidate fixes in the source code form
  - filter the candidate fixes that violate certain pre-defined program analysis rules

Patch Re-Ranking



RQ1: Pattern-Based APR Comparative Study

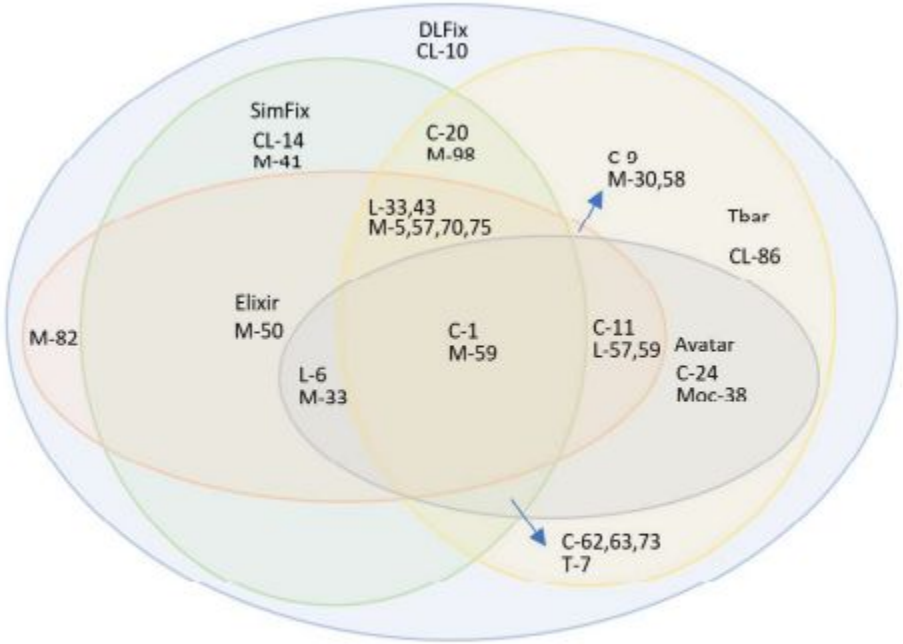


Figure 5: Overlapping Analysis for RQ1. Project names: C:Chart, CL:Closure, L:Lang, M:Math, Moc:Mockito, T:time.

RQ1: Pattern-Based APR Comparative Study

Table 2: RQ1. Comparison with the Pattern-based APR Baselines on Defect4J.

Project	jGenProg	HDRepair	Nopol	ACS	ELIXIR	ssFix	CapGen	SketchFix	FixMiner	LSRepair	AVATAR	SimFix	TBar	DLFix
Chart	0/7	0/2	1/6	2/2	4/7	3/7	4/4	6/8	5/8	3/8	5/12	4/8	**9/14**	5/12
Closure	0/0	0/7	0/0	0/0	0/0	2/11	0/0	3/5	5/5	0/0	**8/12**	6/8	**8/12**	6/10
Lang	0/0	2/6	3/7	3/4	8/12	5/12	5/5	3/4	2/3	8/14	5/11	**9**/13	5/**14**	5/12
Math	5/18	4/7	1/21	12/16	12/19	10/26	12/16	7/8	12/14	7/14	6/13	14/26	**19/36**	12/28
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	**2/2**	0/0	1/**2**	1/1
Time	0/2	0/1	0/1	1/1	**2**/3	0/**4**	0/0	0/1	1/1	0/0	1/3	1/1	1/3	1/2
Total	5/27	6/23	5/35	18/23	26/41	20/60	21/25	19/26	25/31	19/37	27/53	34/56	**43/81**	30/65
P(%)	18.5	26.1	14.3	78.3	63.4	33.3	**84.0**	73.1	80.6	51.4	50.9	60.7	53.1	46.2

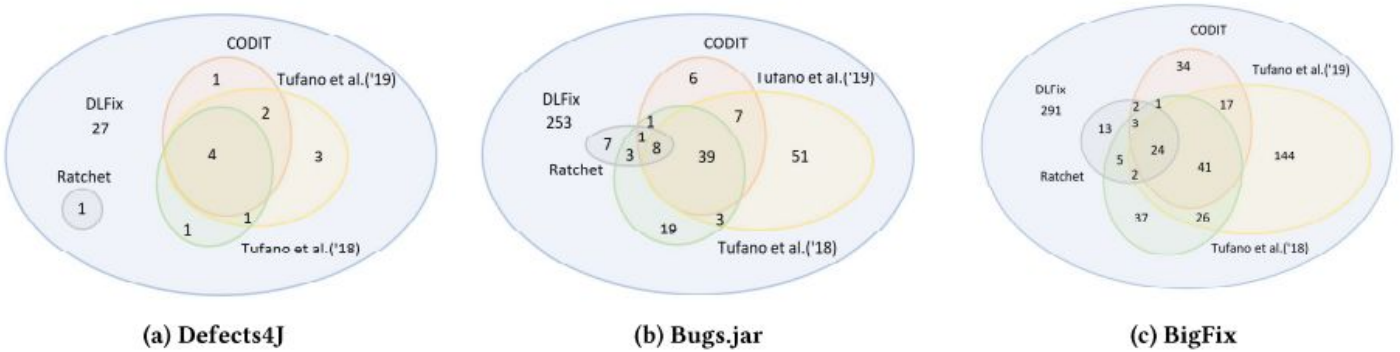
Note: P is the probability of the generated plausible patches to be correct.
In the cells, x/y: x means the number of correct fixes and y means the number of candidate patches that can pass all test cases. For example, for DLFix, 65 candidate patches can pass all test cases. However, 30 out of 65 are the correct fixes compared with the fixes in the ground truth.

RQ2: Deep Learning-Based APR Comparative Study

Table 3: RQ2. Accuracy Comparison with DL-based APR approaches on three Datasets.

Approach	Defect4J (101 Bugs in Testing)			Bugs.jar (1,158 Bugs in Testing)			BigFix (2,176 Bugs in Testing)		
	Top1	Top5	Top10	Top1	Top5	Top10	Top1	Top5	Top10
Ratchet	2.0%	4.0%	6.9%	2.4%	4.4%	6.8%	3.0%	4.1%	6.9%
Tufano et al.('18)	6.9%	9.9%	11.9%	8.4%	11.1%	12.9%	7.9%	10.6%	12.1%
CODIT	8.9%	13.9%	15.8%	7.0%	11.8%	14.8%	6.9%	13.7%	18.3%
Tufano et al.('19)	15.8%	20.8%	23.8%	13.5%	18.7%	23.1%	15.4%	17.3%	21.4%
DLFix	**39.6%**	**43.6%**	**48.5%**	**34.2%**	**36.4%**	**37.9%**	**29.4%**	**31.1%**	**33.4%**

Top K is the number of times that a correct patch is in the ranked list of top K candidate patches.



(a) Defects4J (b) Bugs.jar (c) BigFix

Figure 6: RQ2. Qualitative Analysis Results from DL-based APR Approaches on three Datasets.

RQ3: Sensitivity Analysis

Table 4: Sensitive Analysis – Impact of Different Factors on DLFix's Accuracy in terms of Top1 on BigFix Dataset.

Models	Top1	Improvement
Seq2Seq	1.8%	
Seq2Seq + PAT	6.4%	256%
Two-Layer-EDM	11.7%	550%
Two-Layer-EDM + PAT	24.4%	109%
Two-Layer-EDM + PAT + Re-ranking	29.4%	20.5%

Seq2seq: a simple sequence-to-sequence model; Two-Layer-EDM: Two-layer tree-based LSTM encoder-decoder model; PAT: program analysis (PA) Techniques including Renaming and PA Filters.

Limitations

1. DLFix does not work well on very **unique bugs**. DLFix is the deep learning-based approach that needs a large amount of data for training. But even we have a very large dataset, very unique bugs can still exist. Therefore, if there is not sufficiently similar data in the training, DLFix cannot fix the unique bug well.
2. DLFix does not work well on **multiple bugs** in one method. Our approach can only deal with one bug at a time.
3. DLFix only works on **one statement bugs** and **the bug and fix locations have to be the same**.

Thanks