Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

代码变更表示学习技术

硕士研究生 赵智洋

2023年10月07日

问题回溯

- 总结反思
  - 语速过快
  - 部分内容（如漏讲片段）过渡生硬
  - 配图过于繁杂，不利于观众快速理解
  - 输入、输出没有量化

- 相关内容
  - 2023.09.24 张浩然：《软件漏洞注入技术》
  - 2023.05.14 孔令迪：《源代码漏洞检测》

内容提要

- 背景简介
- 基本概念
- 算法原理
- 总结

背景简介

- 预期收获
  - 了解代码变更表示学习技术的在软件工程领域中的地位和作用
  - 掌握代码变更表示学习技术的基本方法
  - 明确代码变更表示学习技术的应用领域和发展方向

背景简介

- 代码变更（代码编辑）
  - 对软件源代码的增加、删除和修改
- 代码变更表示
  - 自动分析和理解代码变更的基础
  - 影响众多软件工程任务的重要问题
- 代码变更表示相关的下游任务

Example	Example 1
Commit_id	d924b8d91076346aef4e311cc4a16dbac4c23d5a

diff	@@ −61,7 +61,7 @@
	public class MulticastParallelMiddleTimeoutTest
	extends ContextTestSupport {
	from("direct:a").setBody(constant("A"));
	− from("direct:b").delay(3000)
	.setBody(constant("B"));
	+ from("direct:b").delay(4000)
	.setBody(constant("B"));
	from("direct:c").delay(500)
	.setBody(constant("C"));
	}

Ground-Truth	fix test on ci server
NNGen	camel3 to fix a potential npe on camel jdbc
$NMT_{(Luong)}$	try to fix the camel core test error on a slow box
ATOM	**fix test that may fail on ci server**

个：代码提交日志生成任务示例

代码变更表示下游任务

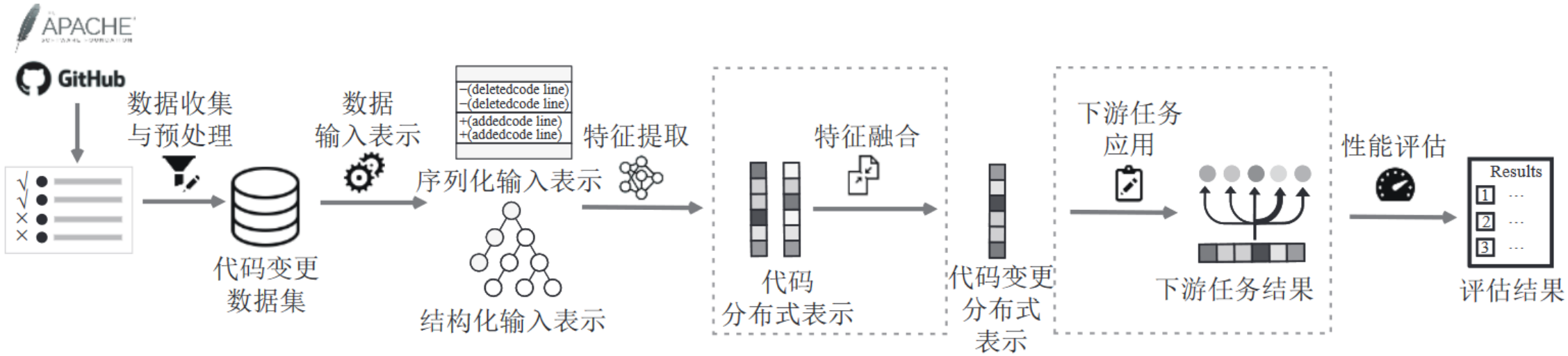
- 生成任务：代码提交日志生成、即时注释更新、代码编辑迁移、代码编辑预测、冲突合并…
- 分类任务：即时缺陷预测、安全漏洞严重性预测、安全漏洞补丁识别、即时代码-注释不一致检测、补丁正确性评估…
- 排序任务：即时缺陷定位、代码评审意见推荐、代码评审优先级排序…

背景简介

- 代码变更表示学习技术发展
  - 传统：特征工程
    - 做法：通过人工设计的特征或特征提取规则将代码变更表示为特征向量
    - 局限
      - 依赖手工分析
      - 只能提取到显式的、浅层的特征
  - 如今：表示学习
    - 做法
      - 将代码变更的语义信息表示为低维稠密的实值向量
      - 即学习代码变更的分布式表示
    - 优势
      - 自动学习、端到端训练
      - 表示准确

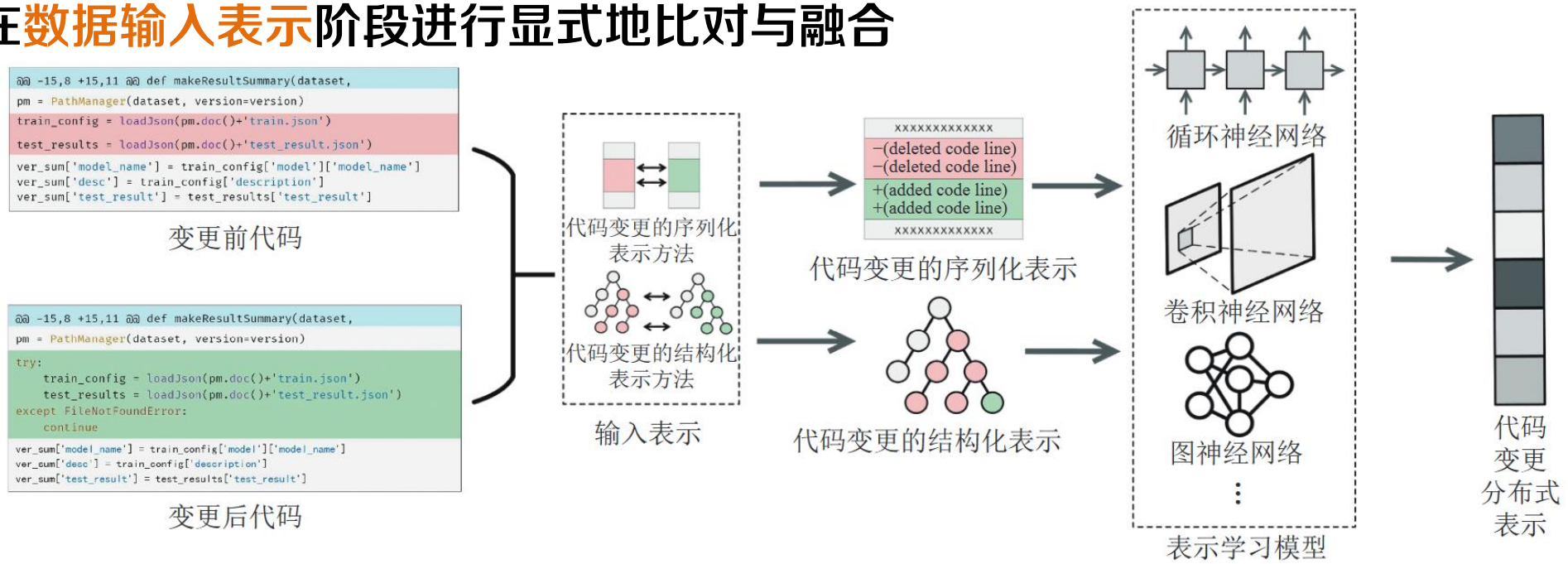
背景简介

- 代码变更表示学习应用的一般框架
  - 数据收集和预处理
    - 目的：过滤噪声数据或不符合要求的数据，提高数据质量
  - 数据输入表示
    - 目的：将经过预处理的代码变更数据转换为表示学习模型能够处理的表示形式
      - 表示形式：序列化输入表示、结构化输入表示



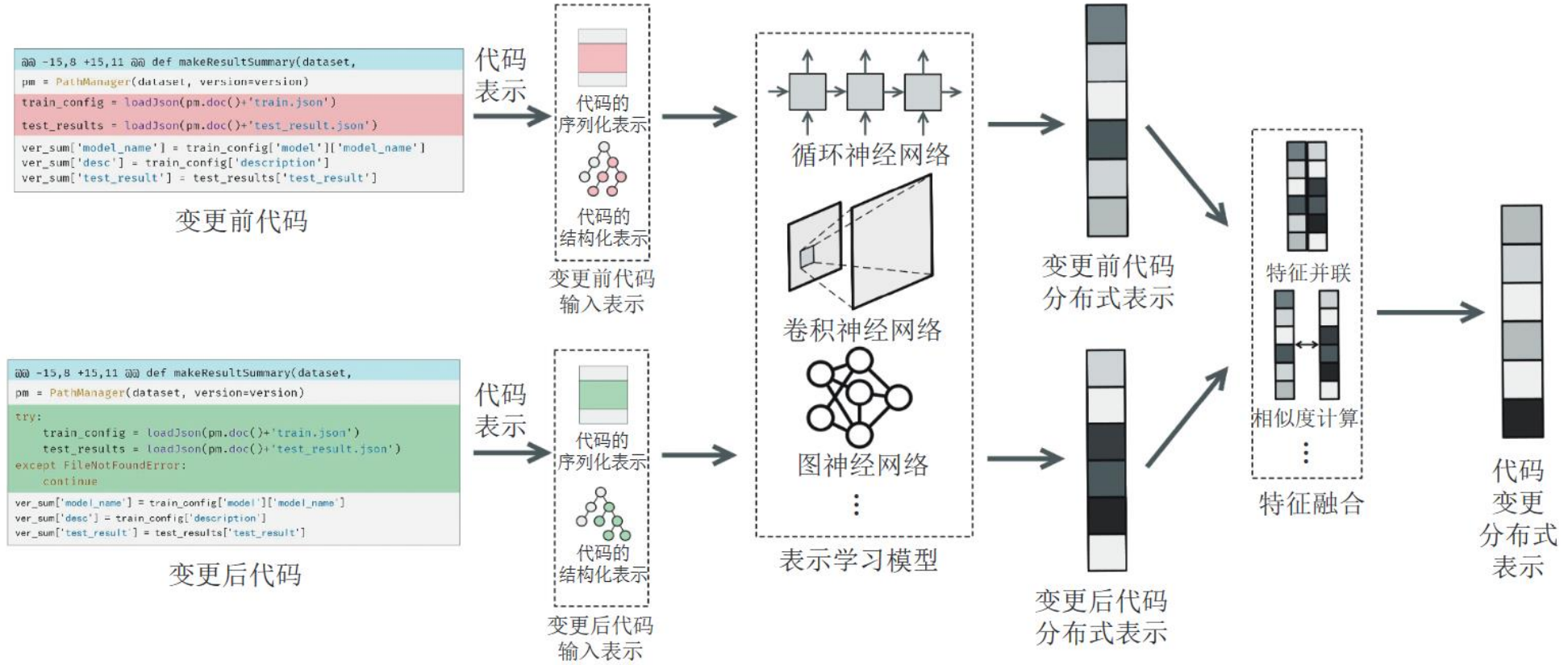
背景简介

- 代码变更表示学习技术分类
  - 分类标准：代码变更信息比对与融合操作的时机
    - 代码变更表示学习的研究对象：变更前代码和变更后代码
  - 分类：基于显式/隐式信息交互的代码变更表示学习
- 基于显式信息交互的代码变更表示学习
  - 在数据输入表示阶段进行显式地比对与融合



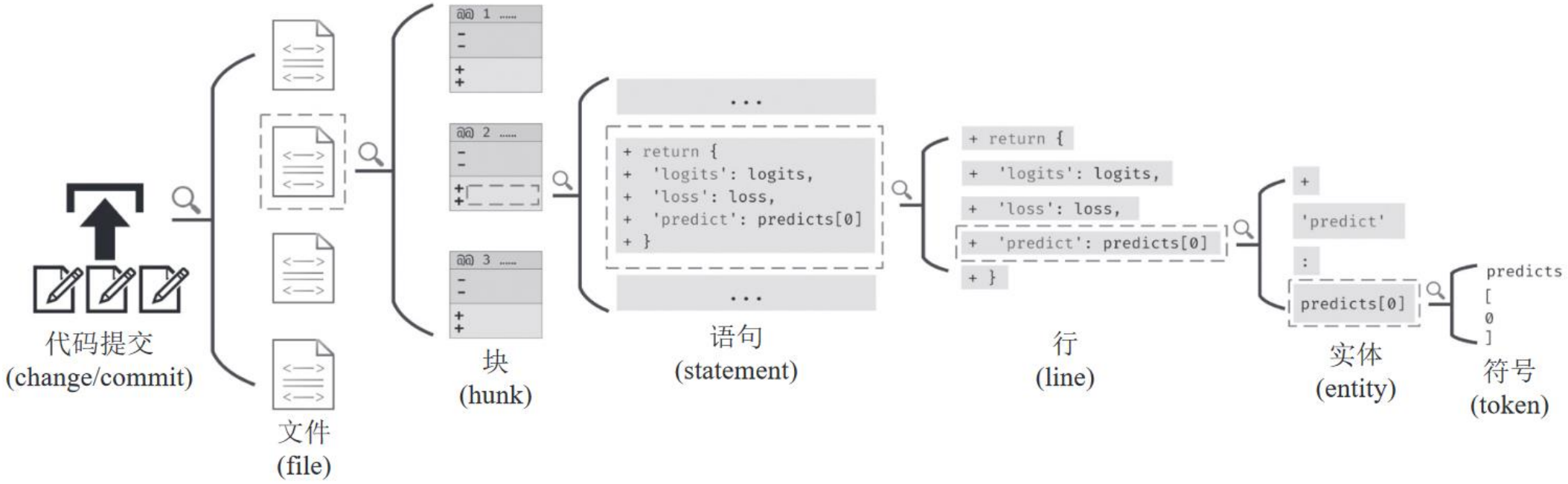
背景简介

- 基于隐式信息交互的代码变更表示学习
  - 在特征融合阶段进行隐式地比对和融合



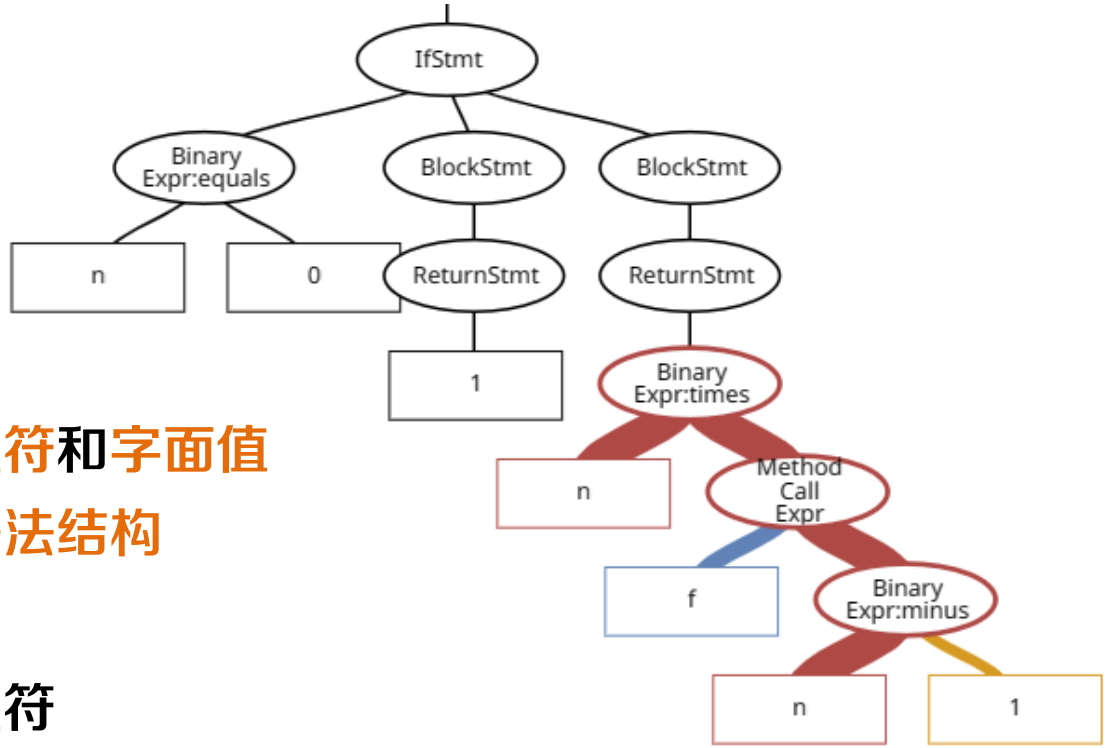
基本概念

- 代码变更表示学习相关概念
  - 代码变更（代码编辑）
    - 对软件源代码的增加、删除和修改
  - 代码变更粒度
    - 代码的结构性使代码变更也具有层次性



基本概念

- 抽象语法树（Abstract Syntax Tree, AST）
  - 源代码语法结构的一种抽象表示
- AST路径（AST path）
  - 定义：AST中2个叶节点之间的最短路径
    - 叶节点：终止节点，用于表示代码中的标识符和字面值
    - 非叶节点：内部节点，用于表示代码中的语法结构
  - 表示：2个标识符在代码语法结构上的关系
    - 两端：终止节点分别对应代码中的一个标识符
    - 中间：内部节点序列
  - 示例：code2vec
    - code2vec类似于自然语言中的word2vec
    - 主要思想是将代码变成嵌入向量，学习代码的分布式表示




```
int f(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * f(n-1);
    }
}
```

基本概念

- 代码提交日志生成（自然语言生成任务）
  - Commit Message Generation
  - 内容：自动为版本管理系统中的代码提交生成描述其内容和意图的自然语言语句
    - 问题：存在开发者倾向于提交时不编写日志、编写的日志内容无意义的情况
    - 输入：代码提交中的代码变更
    - 输出：相应的代码提交日志
  - 研究空白：利用代码结构即AST捕获代码语义
- 自动评估补丁正确性（分类任务）
  - Automated patch correctness assessment，APCA
  - 任务来源：自动程序修复（Automated Program Repair，APR）
    - 生成测试套件，评估补丁正确性
    - 问题：测试套件对预期功能的检查并不充分

算法原理——ATOM

【ATOM】

ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking

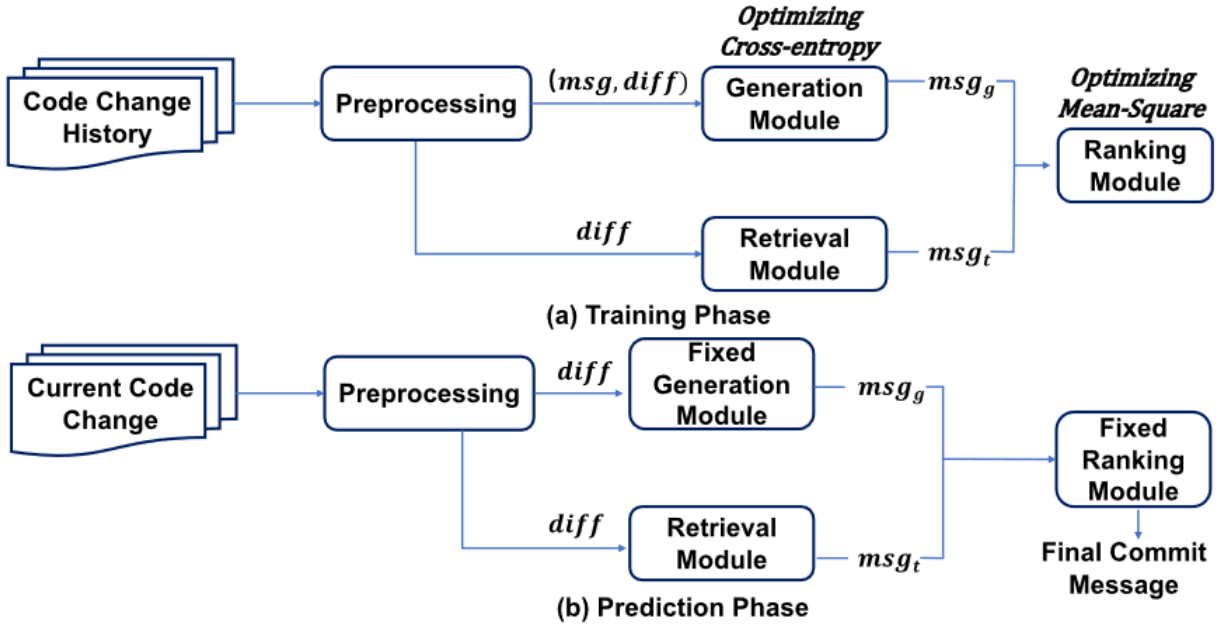
算法原理——ATOM

T	代码提交日志生成
I	1组成对的代码变更集合、1组已知提交日志的代码变更数据库
P	1.预处理：为代码变更构造AST 2.生成：对AST进行编码构造代码变更向量表示，并解码为提交日志 3.检索：根据代码变更相似度检索数据库中的提交日志 4.排序：对生成和检索的提交日志进行排序筛选
O	1组代码变更提交日志集合

P	仅将代码变更视为代码token序列，忽略代码结构信息
C	代码为Java语言、代码变更含完整函数
D	如何提取和利用AST的结构信息
L	TSE 2020 (CCF A)

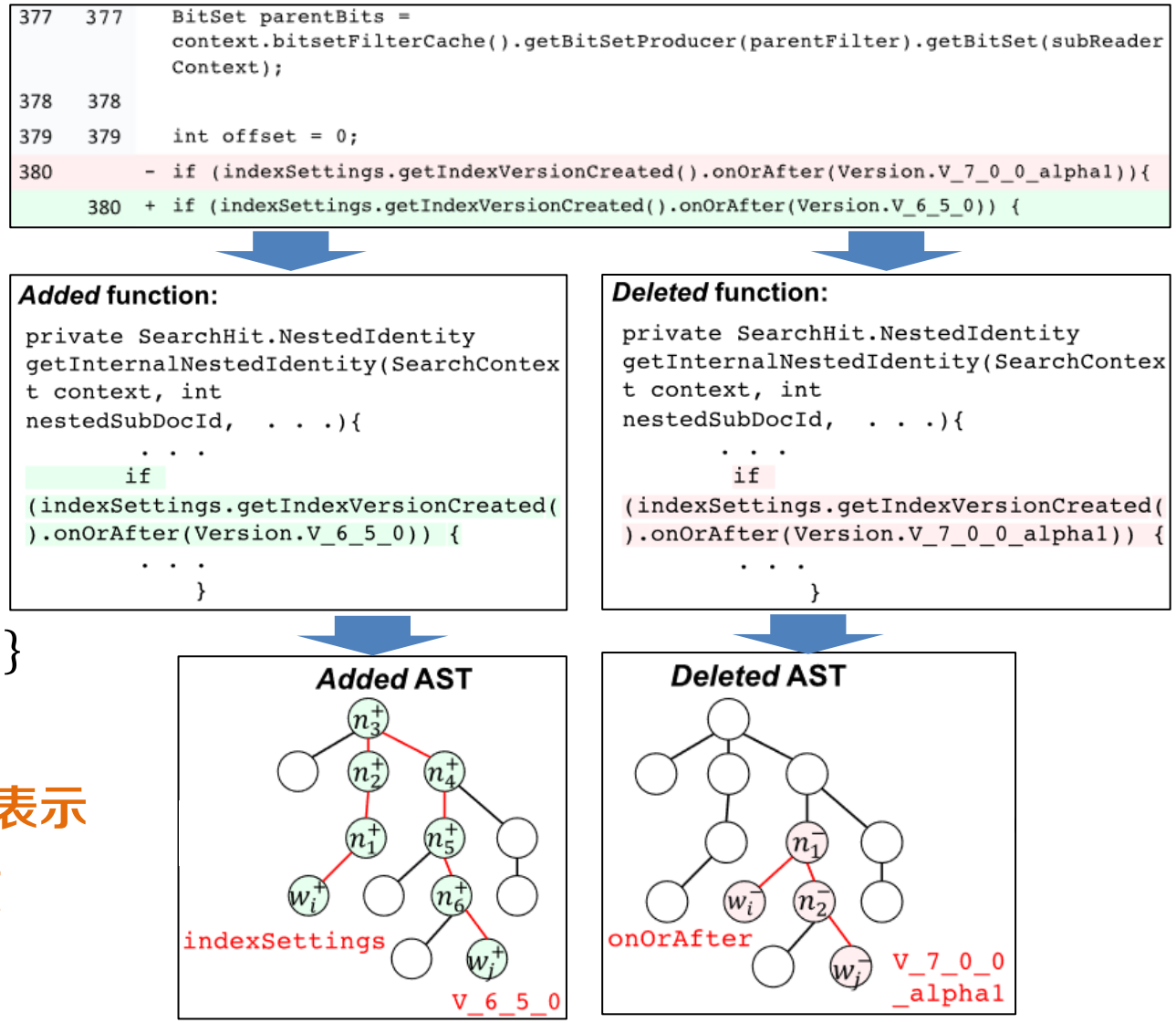
算法原理——ATOM

- 算法原理图
  - 预处理：为代码变更构造AST
  - 生成：对AST进行编码构造代码变更向量表示，并解码为提交日志
    - 模型训练额外输入：代码变更对应的提交日志以训练生成模型
  - 检索：根据代码变更相似度检索数据库中的提交日志
  - 排序：对生成和检索的提交日志进行排序筛选



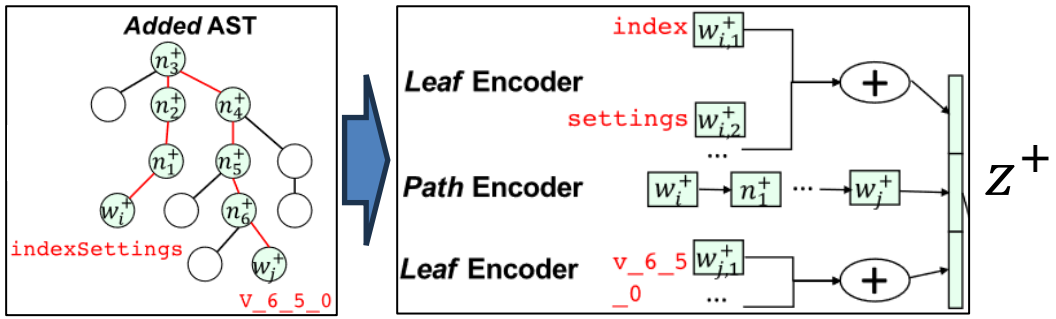
算法原理——ATOM

- **Step 1：预处理——构造AST**
  - 将1份代码变更切分为2个函数片段
    - 添加函数$Added\ function$
    - 删除函数Deleted $function$
  - 为2个函数片段分别构造AST
    - $Added\ AST$、$Deleted\ AST$
  - 抽象AST路径集合表示X^+/X^-
    - $X^+ = \{x_1^+, \ldots, x_p^+\}$、$X^- = \{x_1^-, \ldots, x_k^-\}$
      - p、k表示路径数目
    - 其中，$x = \{w_i, n_1, .., n_l, w_j\}$为路径表示
      - w_i、w_j为该条路径中的叶节点
      - n_l为该条路径中的非叶节点
        - » l表示非叶节点数目

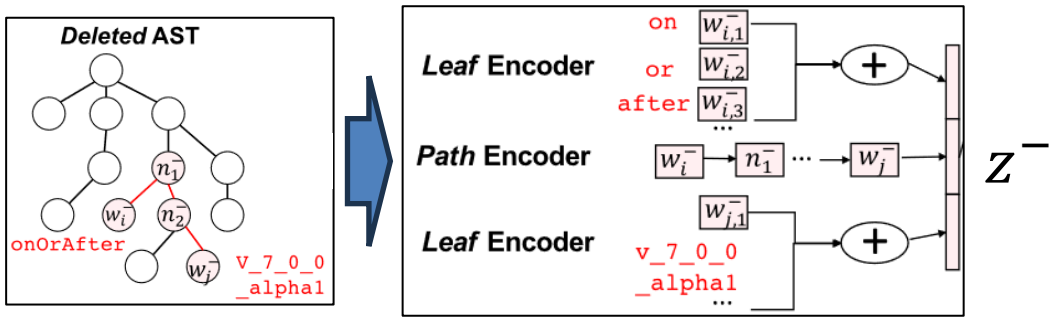


算法原理——ATOM

- **Step 2：生成——编码AST获得代码变更表示**
  - 双向LSTM获取路径嵌入$path_feat^{+/-}$
    - $\boldsymbol{h}_{w_i}^{+/-}, \ldots \boldsymbol{h}_{w_j}^{+/-} = Bi-LSTM(\boldsymbol{E}_{w_i^{+/-}}^{nodes}, \ldots, \boldsymbol{E}_{w_j^{+/-}}^{nodes})$
    - $path\_feat^{+/-} = [\boldsymbol{h}_{w_i^{+/-}}^{\leftarrow}, \boldsymbol{h}_{w_i^{+/-}}^{\rightarrow}]$



  - 子token的矩阵嵌入之和表示叶节点嵌入$leaf_feat^{+/-}$
    - $leaf\_feat^{+/-} = \sum_{s \in split(w^{+/-})} \boldsymbol{E}_{w^{+/-}}^{subtokens}[s]$
  - 全连接层构造AST路径的嵌入表示$z^{+/-}$
    - 输入：1个路径嵌入和2个叶节点嵌入

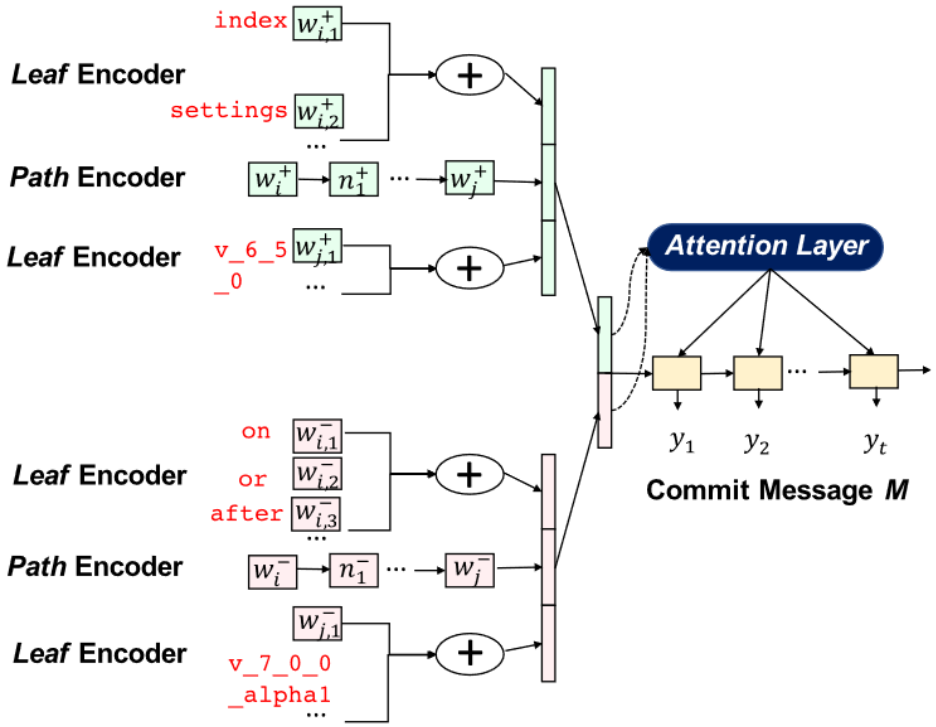


    - $\boldsymbol{z}^{+/-} = layer([\boldsymbol{leaf\_feat}_{w_i}^{+/-}, \boldsymbol{path\_feat}^{+/-}, \boldsymbol{leaf\_feat}_{w_j}^{+/-}])$
  - 获得代码变更表示$\boldsymbol{Z} = [\boldsymbol{z}_p^+; \boldsymbol{z}_k^-]$（任意2个节点之间形成1条AST路径）

    p为添加AST中的路径数目、k为删除AST中的路径数目

算法原理——ATOM

- **Step 2：生成——解码器生成提交日志**
  - 为代码变更表示计算路径注意力
    - $a^t = softmax(\boldsymbol{h}_t W_a \boldsymbol{Z})$
  - 生成提交日志
    - 取路径表示平均值为解码器的初始隐藏状态
      - $\boldsymbol{h}_0 = \frac{1}{p+k}\sum_{i=1}^{p+k} \boldsymbol{z}_i$
    - 预测当前token y_t
      - $p(y_t | y < t, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_{p+k}) = softmax(W_s \tanh(Wc[\boldsymbol{c}_t; \boldsymbol{h}_t]))$
        - » 其中， $\boldsymbol{c}_t = \sum_i^{p+k} a_i^t \boldsymbol{z}_i$为上下文向量
  - 生成模型的损失函数
    - $Loss = \boldsymbol{y} log(e^{\boldsymbol{logits}} / \sum_r e^{\boldsymbol{logits}})$
      - y为真实的提交日志的token，$logits$是生成的提交日志的token



算法原理——ATOM

- Step 3：检索
  - 利用TF-IDF指数计算待检索的代码变更diff与数据库中的代码变更diff的相似度
  - 获取与待检索的代码变更diff最相似的diff对应的1条提交日志
- Step 4：排序筛选
  - 输入：1条生成得到的提交日志、1条检索得到的提交日志
  - 设计相似度匹配矩阵，筛选出1条代码变更提交日志
    - 衡量代码变更diff与提交日志之间的相似度
- 补充：commit和diff
  - Git中使用commit记录不同版本之间的更改
  - commit包含提交日志和代码更改
    - 提交日志：文本格式，辅助理解代码更改
    - 代码更改：即为diff，表征两个代码版本之间的差异

Commit_id	d924b8d91076346aef4e311cc4a16dbac4c23d5a

diff	@@ −61,7 +61,7 @@
	public class MulticastParallelMiddleTimeoutTest
	extends ContextTestSupport {
	from("direct:a").setBody(constant("A"));
	− from("direct:b").delay(3000)
	.setBody(constant("B"));
	+ from("direct:b").delay(4000)
	.setBody(constant("B"));
	from("direct:c").delay(500)
	.setBody(constant("C"));
	}

Ground-Truth	fix test on ci server

实验分析——ATOM

- 实验设置
  - 数据集：自建
    - 原因：公开数据集中，部分代码变更缺乏完整函数，无法直接提取AST
    - 构造：
      - 爬取56个github热门项目
      - 保留160k样本
        - » 过滤：包含过多行的merge/rollback、相同的commit…
    - 使用：10%测试、10%验证、80%训练
  - 评价指标：
    - $BLUE-N$：计算候选序列相对于参考序列的n-gram精度
      - $BLUE-N = BP * \exp(\sum_{n=1}^{N} w_n log p_n)$， $N = 1,2,3,4$

        » $w_n = \frac{1}{N}, BP = \begin{cases} 1, if\ c > r \\ e^{1-r/c}, if c \leq r \end{cases}$（$c$为候选序列的长度，$r$为参考序列的长度）

实验分析——ATOM

- 实验设置
  - 评价指标：
    - ROUGE-L：基于最长公共子序列 (Longest common subsequence，LCS) 的分数，评估2个给定文本之间的相似度F_{lcs}
      - $F_{lcs} = (1+\beta^2)R_{lcs}P_{lcs}/(R_{lcs}+\beta^2 P_{lcs})$
        - » 召回率$R_{lcs} = LCS(X,Y)/m$，m为参考序列的长度
        - » 准确率P_{lcs}=$LCS(X,Y)/\mathrm{n}$，n为生成序列的长度
    - $Meteor$：基于整个语料库上的准确率和召回率，得出最终测度
      - $Meteor = F_{mean}(1-Penalty)$
        - » 调和均值$F_{mean} = 10PR/(R+9P)$
        - » 惩罚因子$Penalty = 0.5(chunks/unigrams_matched)^3$
          - $chunks$：候选序列和参考序列能够对齐的、空间上排列连续的单词形成一个chunks
          - $unigrams\_matched$：能够匹配上的unigram数目

实验分析——ATOM

- 实验-对比、消融实验：代码提交日志生成效果
  - 对比方法：分别基于检索或生成
    - NNGen：基于检索的方法，从训练集中检索最相似的top-k代码变更diff
    - Ptr-Net、CODISUM：将代码视为token序列，忽略结构信息
  - 实验分析与结果
    - 仅采用检索的ATOM优于NNGen，而NNGen比仅采用生成的ATOM有更好的性能
      - 基于检索的方法在代码提交日志生成任务中具有有效性，但面临泛化性差的局限
    - 引入生成的ATOM性能有明显提升
      - ATOM生成模块可以在代码变更中学习更多语义以生成高质量的提交日志

Methods		BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
Baselines	$NMT_{(Luong)}$	13.12	8.01	6.11	5.23	12.73	10.37
	$NMT_{(Bahdanau)}$	12.78	7.66	5.72	4.81	11.95	9.87
	NNGen	16.91	12.01	10.03	8.04	15.20	13.68
	Ptr-Net	5.80	1.72	0.73	0.45	7.61	4.98
	CODISUM	7.82	3.61	2.22	1.75	9.87	8.35
	Commit2Vec	12.72	7.78	6.09	5.38	13.54	10.43
Ours	$ATOM_{Gen}$	15.97	10.70	8.83	7.35	14.80	11.82
	$ATOM_{Ret}$	17.74	12.65	10.55	8.52	15.93	14.35
	ATOM	**23.88**	**15.61**	**12.17**	**10.51**	**22.02**	**18.51**

算法原理——CACHE

【CACHE】

Context-Aware Code Change Embedding for Better Patch Correctness Assessment

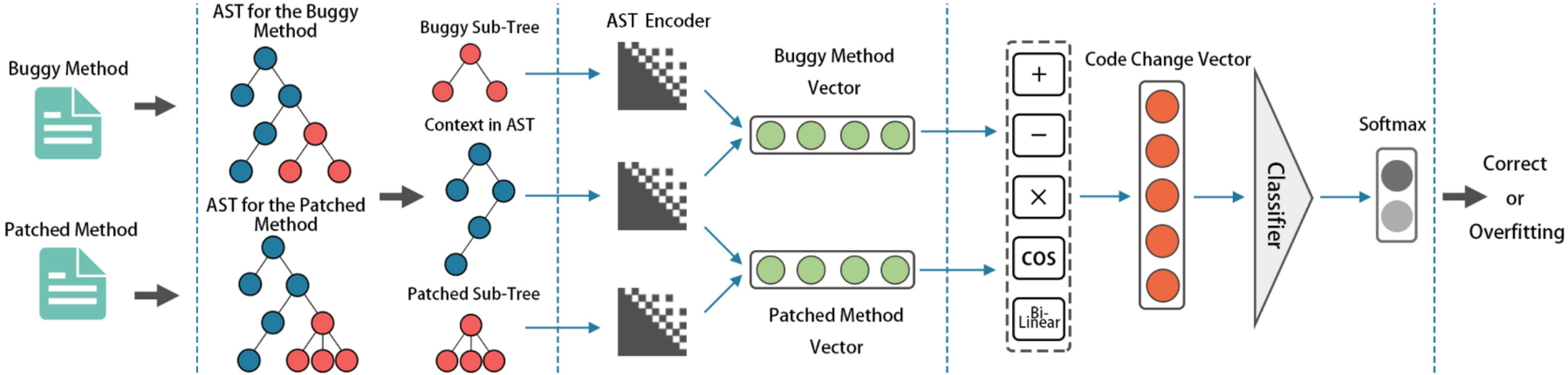
算法原理——CACHE

T	自动评估补丁正确性
I	1组成对的缺陷方法代码和补丁方法代码集合
P	1.上下文抽取：提取表示为AST的缺陷方法代码和补丁方法代码中的子树结构 2.AST嵌入：将子树结构表示为嵌入向量 3.补丁评估：融合嵌入向量，输入分类器评估补丁方法
O	1组补丁方法代码的评估结果（正确或过拟合）集合

P	生成评估补丁正确性的代码嵌入时，忽略上下文信息和程序结构信息
C	代码为Java语言
D	如何提取上下文信息和利用结构信息
L	TOSEM 2022 (CCF A)

算法原理——CACHE

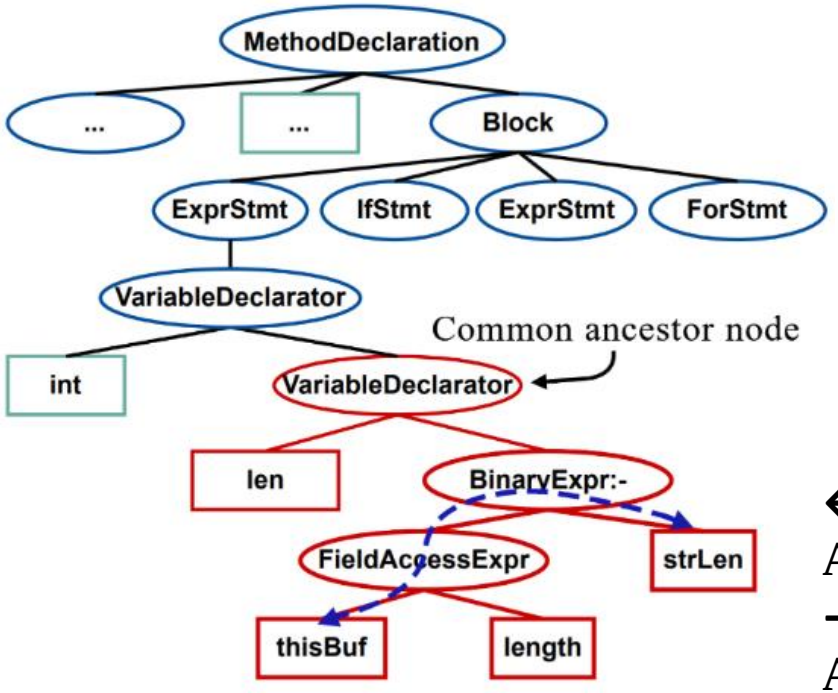
- 算法原理图
  - 上下文抽取：提取表示为AST的缺陷方法代码和补丁方法代码中的子树结构
  - AST嵌入：将子树结构表示为嵌入向量
  - 补丁评估：融合嵌入向量，输入分类器评估补丁方法

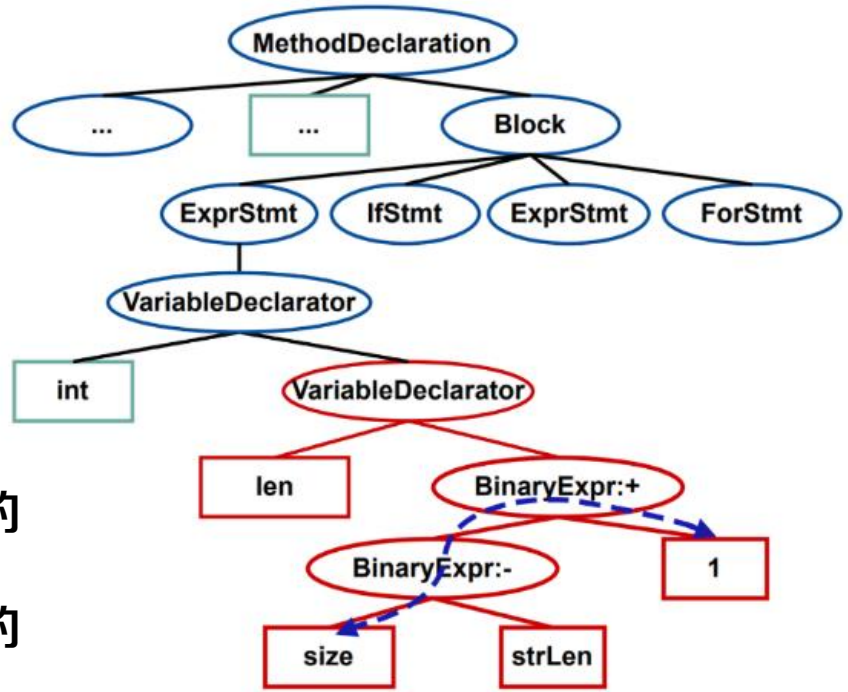


算法原理——CACHE

- **Step 1：上下文抽取**
  - 将缺陷方法代码和补丁方法代码进行AST表示
  - 识别2个AST中的共同祖先节点
  - 抽取缺陷子树和补丁子树，提取上下文子树

```
1          if (strLen > size) {
2              return -1;
3          }
4          char[] thisBuf = buffer;
5  -       int len = thisBuf.length - strLen;
6  +       int len = size - strLen + 1;
7          outer:
8          for (int i = startIndex; i < len; i++) {
9              for (int j = 0; j < strLen; j++) {
```



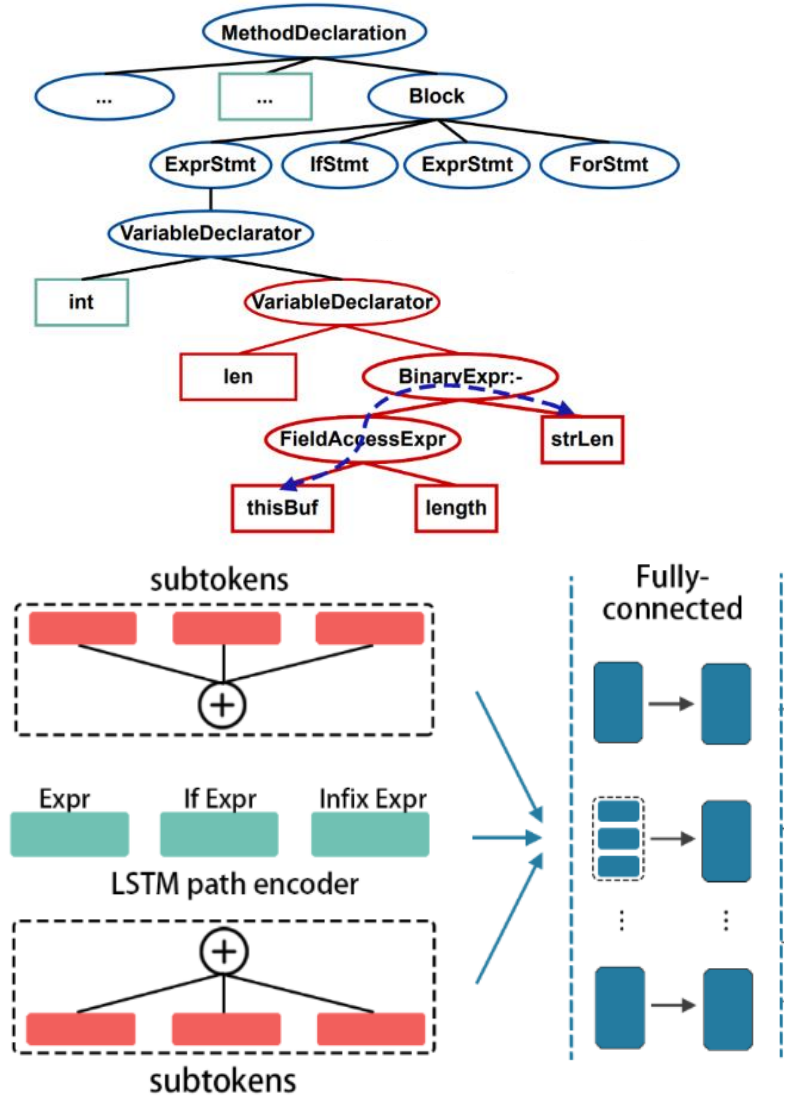


←：缺陷方法代码的AST表示

→：补丁方法代码的AST表示

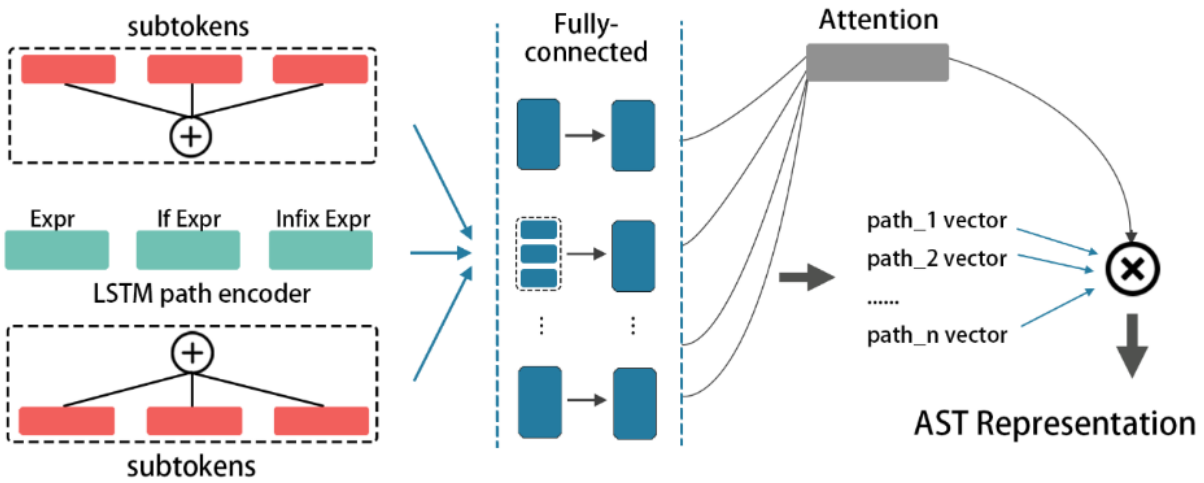
算法原理——CACHE

- Step 2：AST嵌入——获取AST路径表示
  - 提取子树AST路径
    - 分别提取缺陷子树、补丁子树和上下文子树的各2个叶节点之间的路径
  - 进行AST路径表示
    - 2个叶节点分别拆分为子token进行嵌入表示
      - 依据驼峰命名（如strLen）或下划线命名（如str_len），拆分为str和len
    - 学习基于AST路径的节点嵌入表示
    - 构造AST路径表示
      - 连接AST路径节点嵌入表示和2个叶节点嵌入表示

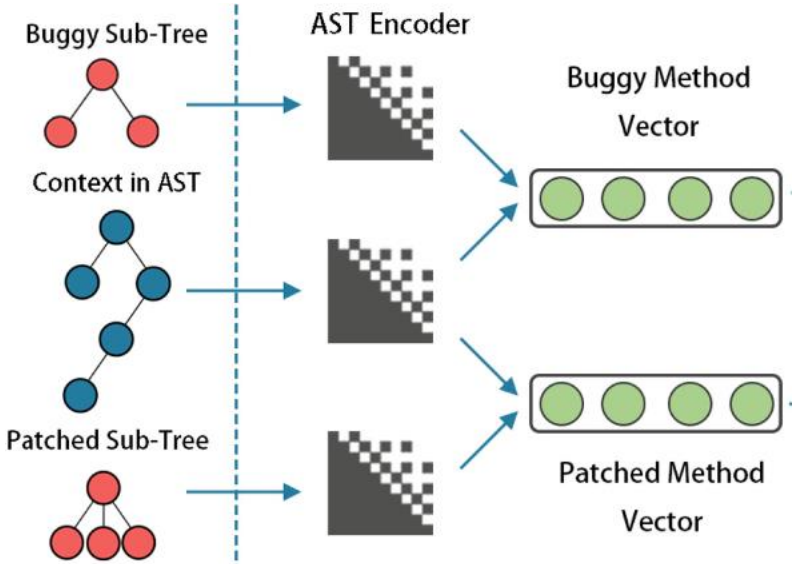


算法原理——CACHE

- **Step 2：AST嵌入——获取缺陷/补丁方法代码表示**
  - 获得子树嵌入表示：加权
    - 将AST路径表示输入全连接层
    - 获取注意力权重
  - 获得方法代码表示：连接向量
    - 缺陷方法代码表示=缺陷子树表示+上下文子树表示
    - 补丁方法代码表示=补丁子树表示+上下文子树表示



↑：获取子树嵌入表示



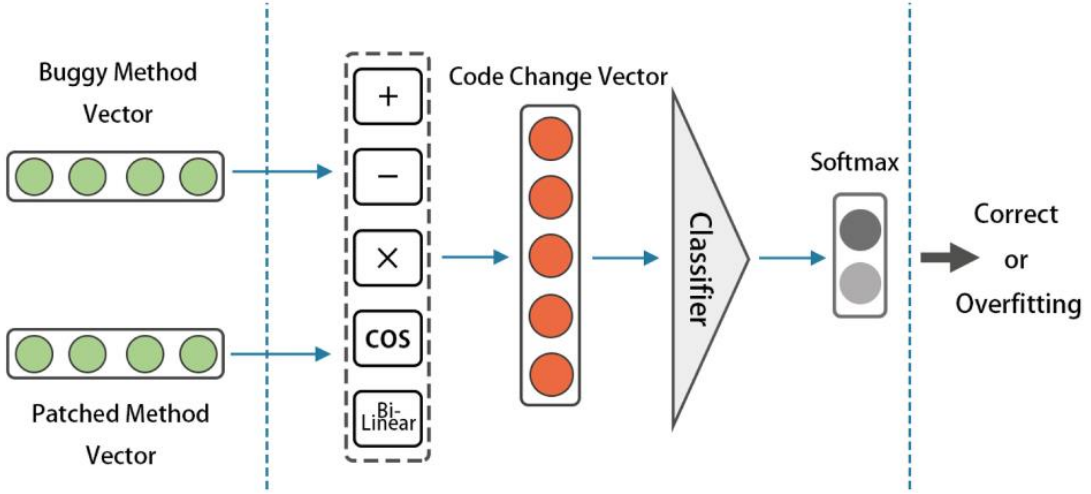
↑：获取方法代码表示

算法原理——CACHE

- **Step 3：补丁评估**
  - 获取代码变更表示
    - 融合表示：5个函数融合缺陷方法代码表示和补丁方法代码表示
      - 加、减、矩阵乘法、余弦相似度和双线性模型
  - 神经网络二元分类器
    - 训练分类器：基于交叉熵损失的有监督训

$$L = \sum_i -[\mathrm{y}_i \cdot \log\left(p\left(E_{\mathrm{out}_c}\right)\right) + (1-\mathrm{y}_i) \cdot \log\left(p\left(E_{\mathrm{out}_c}\right)\right)]$$

    - 获取正确性概率分布
      - 补丁正确：正确的概率大于不正确的概率



实验分析——CACHE

- 实验设置
  - 数据集：含标签（补丁正确/过拟合）、保证数据平衡
    - 删除重复的补丁样本：避免数据泄露
  - 评价指标：准确率、精确率、召回率、F1值和AUC
    - $Accuracy = (TP + TN)/(TP + FP + FN + TN)$
    - $Precision = TP/(TP + FP)$
    - $Recall = TP/(TP + FN)$
    - $F1 = 2 * Precision * Recall/(Precision + Recall)$

Datasets	Subjects	# Correct patches	# Overfitting patches	Total
Small	Tian et al. [76]	468	532	1,000
	Wang et al. [81]	248	654	902
	Filtered	**535**	**648**	**1,183**
Large	ManySStuBs4J [30]	51,433	0	51,433
	RepairThemAll [16]	900	63,393	64,293†
	Filtered	**25,589**	**24,105**	**49,694**

实验分析——CACHE

- 实验1-对比实验：表示学习技术
  - 实验设置
    - 分别在大、小数据集上进行5折交叉验证
    - 大数据集（多为单行补丁）训练、小数据集（多为跨行补丁）验证
  - 实验结果
    - 小数据集：实现78%的F1值，提升至少6%
    - 大数据集：实现98.6%的F1值，提升至少3%
    - 跨数据集：实现75%的F1值，提升至少10.3%
  - 分析
    - CACHE在小数据集中，保证75%召回率的同时，保持79.5%的精确率
    - CACHE泛化性能好，在跨数据集中指标领先

Classifier	Embedding	Acc.	Pre.	Recall.	F1	AUC
Decision Tree [8]	BERT [15]	63.5	65.3	70.9	67.9	63.7
	CC2vec [26]	66.1	69.4	68.0	68.7	66.5
	code2vec [5]	65.1	68.1	68.3	68.1	64.4
	code2seq [2]	60.1	63.5	64.0	63.7	60.0
	Doc2Vec [35]	61.2	64.5	65.3	64.8	60.8
Logistic Regression [32]	BERT [15]	64.8	66.5	72.4	69.2	68.7
	CC2vec [26]	64.9	62.4	90.1	73.7	68.6
	code2vec [5]	66.8	68.6	72.9	70.6	70.2
	code2seq [2]	60.7	63.3	67.6	65.3	63.1
	Doc2Vec [35]	63.7	65.7	70.8	68.0	68.9
Naïve Bayes [66]	BERT [15]	61.6	64.8	65.7	65.0	64.7
	CC2vec [26]	60.0	58.3	**94.6**	72.2	58.1
	code2vec [5]	57.7	58.1	81.5	67.8	55.6
	code2seq [2]	57.0	59.0	70.5	64.2	60.6
	Doc2Vec [35]	64.1	65.8	72.4	68.7	67.0
CACHE		**75.4**	**79.5**	76.5	**78.0**	**80.3**

↑：小数据集实验结果 ↓：跨数据集实验结果

Classifier	Embedding	Acc.	Pre.	Recall.	F1	AUC
Decision Tree [8]	BERT [15]	59.7	65.9	54.6	59.7	59.9
	CC2vec [26]	46.2	52.1	22.7	31.6	47.7
	code2vec [5]	59.5	64.0	60.6	62.3	62.9
	code2seq [2]	59.9	63.3	63.9	63.6	59.6
	Doc2Vec [35]	49.3	54.4	46.5	50.1	47.8
Logistic Regression [32]	BERT [15]	53.6	56.4	67.5	61.5	55.9
	CC2vec [26]	44.9	9.0	0.3	0.6	41.9
	code2vec [5]	57.8	61.5	61.3	61.4	59.1
	code2seq [2]	57.9	59.8	70.5	64.7	57.1
	Doc2Vec [35]	47.0	52.0	41.4	46.1	47.3
Naïve Bayes [66]	BERT [15]	46.7	52.1	34.8	41.8	49.4
	CC2vec [26]	45.2	NaN	0.0	NaN	50.0
	code2vec [5]	42.4	43.9	18.5	26.1	46.4
	code2seq [2]	48.1	54.1	34.6	42.2	50.7
	Doc2Vec [35]	41.8	44.8	27.5	34.1	43.6
CACHE		**70.1**	**69.1**	**81.9**	**75.0**	**73.2**

考虑上下文信息和程序结构对于评估补丁正确性非常重要

实验分析——CACHE

- 实验2-消融实验：模块贡献
  - 实验设置
    - 嵌入模块的6个方面
      - 不考虑上下文节点、不拆分代码token、不使用LSTM学习AST路径特征、不使用注意力模型、不考虑AST路径特征、不分离3个（缺陷、补丁、上下文）子树
    - 融合模块方面
      - 分别消融1个函数
  - 实验结果和分析
    - 上下文信息对CACHE贡献最大
    - LSTM编码AST路径最多能够提升10.1%的F1值
      - 隐藏在AST中的信息应被仔细提取和学习

Dataset / Model	Small		Large		Cross-dataset	
	F1	↓ (%)	F1	↓ (%)	F1	↓ (%)
No Context	68.3	**12.4**	96.5	**2.1**	67.9	**9.5**
No token splitting	70.8	9.2	97.4	1.2	69.4	7.5
No LSTM encoder	70.1	10.1	98.1	0.5	69.2	7.7
No attention	70.5	9.6	97.2	1.4	68.5	8.7
No program structure	71.4	8.5	98.0	0.6	68.8	8.3
No split-embedding	72.1	8.2	98.2	0.4	70.1	6.5
No addition	71.9	7.8	98.2	0.4	70.2	6.4
No subtraction	76.4	2.1	98.5	0.1	71.8	4.3
No Hadamard product	72.3	7.3	98.3	0.3	70.1	6.5
No cosine similarity	70.8	9.2	97.9	0.7	69.2	7.7
No Bilinear model	75.2	3.6	98.4	0.2	71.5	4.7
CACHE (original model)	78.0		98.6		75.0	

上下文信息最有价值!

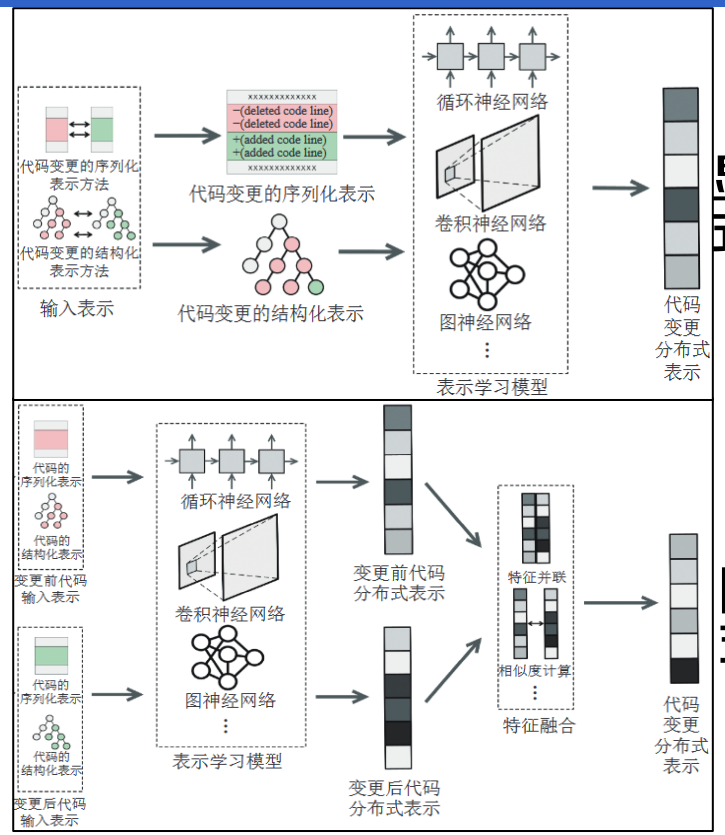
总结

代码变更表示学习技术

- 代码特点
  - 强结构性、长依赖、可执行性
    - 建模代码的结构信息
      - 如嵌套结构、条件结构等
    - 捕获标识符之间的长依赖
      - 如标识符的声明和使用
- 输入表示形式：序列化和结构化
  - 序列化
    - 基于不同的对齐粒度表示代码变更
  - 结构化：更好地捕捉代码变更中的语义信息和依赖关系
    - 直接或间接地基于AST构造代码变更表示

代码变更表示学习技术

- 代码变更表示学习技术分类：显式/隐式信息交互
  - 基于显式信息交互的代码变更学习技术
    - 利用领域知识和人工设计的规则
      - 能有效抽取代码中的显式特征（如被变更的标识符）
    - 局限：过于聚焦显示特征、泛化性差
  - 基于隐式信息交互的代码变更学习技术
    - 具备学习隐式特征（如语义特征）的能力
    - 局限：对数据质量、数据规模和模型设计更加敏感



方法	处理过程			优点	缺点
	输入表示形式	信息交互阶段	信息交互方式		
基于显式信息交互的代码变更表示学习	代码变更的单一表示(例如diff)	数据输入表示阶段	基于人工设计的规则和算法	能有效抽取显式特征(例如变更的标识符)	在特征难以显式提取的任务和大规模数据集上泛化性受限
基于隐式信息交互的代码变更表示学习	变更前代码和变更后代码	特征融合阶段	基于学习	能自动学习隐式特征(例如语义信息)	对数据质量、数据规模和模型设计更敏感

代码变更表示学习技术

- 关键挑战
  - 结构信息利用困难
    - 结构化信息提取工具的鲁棒性有限
    - 基于结构化信息的表示学习模型计算复杂度高
  - 内在评估方法缺乏
    - 依靠下游任务的性能指标间接评估代码变更表示的质量
  - 基准数据集缺失
    - 自建数据集：规模小、质量参差、缺乏维护
  - 跨语言学习
- 研究趋势
  - 预训练、多任务代码变更表示学习...

参考文献

[1] Liu S, Gao C, Chen S, et al. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking[J]. IEEE Transactions on Software Engineering, 2020, 48(5): 1800-1817.

[2] Lin B, Wang S, Wen M, et al. Context-aware code change embedding for better patch correctness assessment[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2022, 31(3): 1-29.

[3] Hoang T, Kang H J, Lo D, et al. Cc2vec: Distributed representations of code changes[C]. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 518-529.

[4] Alon U, Zilberstein M, Levy O, et al. code2vec: Learning distributed representations of code[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 1-29.

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！