自动化测试 2022 工具复现实验报告（有效警告识别方向）

南京大学软件学院
蔡之恒 林浩然 沈霁昀 熊丘桓
{201250127, 201250184, 201250048, 201250172}@smail.nju.edu.cn

0. 小组构成和工作总览

0.1 小组构成

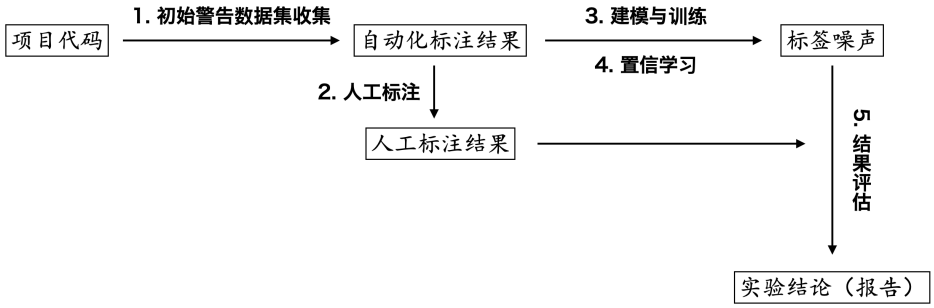
- 组名：紫东华策史
- 编号：6
- 成员：
  - 蔡之恒（201250127）
  - 林浩然（201250184）
  - 沈霁昀（201250048）
  - 熊丘桓（201250172）

0.2 工作总览

我们的选题是有效警告识别方向（基于置信学习的警告数据集去噪技术）。

根据说明文档的指导，小组依次完成了初始警告数据集收集、人工标注、初始警告数据集收集、建模和训练、置信学习、结果分析五个步骤（对应下文中各主要部分）。项目 `git` 仓库中涉及到的主要代码和数据文件将在下文中详细说明。

小组选择的项目为 `Commons Imaging: a Pure-Java Image Library` ，我们从 `apache/commons-imaging` 中获取项目代码和 `git` 记录。



总体工作流程如上图所示，该项目的最终实验结论将在5.5 实验结论中详细说明，而关于实验过程和结果的一些重点讨论将在 6. 中详细进行。

1. 初始警告数据集收集

按照配置文档 `Readme.pdf`的指导，在本地（MacOS M1）上搭建了 JAVA8 运行环境，Neo4j 数据库配置在华为云（Ubuntu 20.04）上。

1.1 相关文件

按照配置文档，将这一部分涉及到的相关文件 / 文件夹列出。

```
tool-build
├── automated-annotation-results      # 这一部分工作结束后小组成员整理的实验结果
│   ├── fixed-alarms.list             # 自动标记的已修复警告列表（该阶段主要实验结果 #1）
│   ├── summary-project-vtype.csv
│   ├── unfixed-commons-imaging.csv   # 自动标记的未修复警告列表（该阶段主要实验结果 #2）
│   └── vtype-stat.csv
├── findbugs-violations               # 需配置的项目代码主体
├── repos                             # 配置文档中涉及的代码仓库文件夹
└── workings                          # 配置文档中涉及的 shell 脚本、部分配置、代码仓库文件夹
```

1.2 项目选择

在项目选择上，小组成员一共有过三次尝试：

- 第一次尝试中，选用的项目是 apache/commons-email（见 `repos/repos-a/`），收集这个项目的警告信息在我们的机器上大约总共需要 4 小时的运行时间，可能是由于项目规模不够，仅收集到个位数条自动标注的有效警告信息，这导致了后续的训练学习无法有效展开。
- 第二次尝试中，选用的项目是 apache/openmeetings（见 `repos/repos-b/`），这个项目的前四个版本使用配置好的 maven 均无法编译成功，猜测为 `maven` 的版本原因。
- 第三次尝试中，选用的项目是 apache/commons-imaging（见 `repos/repos-c/`），即最终的项目。这个项目的规模（以 github 上的提交次数来衡量）大约为我们尝试的第一个项目的 1.7 倍，但实际总运行时间大约为第一个项目的 6 ~ 7 倍。使用 spotbugs 扫描每个版本的警告（第一个脚本）大约用去 3 小时时间，而在 git trace 中分析不同版本之间的警告（第二个脚本）总共用去约 24 小时的时间。我们猜测一方面是因为这份代码中扫描出的警告的绝对数量较多，而另一方面则是因为这个项目的 git trace 的路径相比第一次尝试中复杂了很多。这个项目总共收集到 206 条有效警告，33474 条无效警告（分别见 `automated-annotation-results/fixed-alarms.list` 和 `automated-annotation-results/unfixed-commons-imaging.csv`）。在这里我们保留了需要复现的代码的原始输出，这部分原始输出将在建模与训练中被整理成统一和机器友好的形式。

1.3 经验与教训

小组成员认为，这一部分的难点主要是相关环境的配置，而导致这一问题的罪魁祸首就是这份收集警告的项目代码的实现年份较早，很多依赖都已经迭代了好几个大版本。一个最明显也非常极端的例子就是 Neo4j 数据库，代码文件的依赖库 `lib` 中打包支持的数据库驱动版本最高仅能支持到 `Neo4j 3.3.*` ，而现在 Neo4j 的稳定版本号已经来到了 `Neo4j 5.2.0` 。我们没能在官网上找到相应的操作手册和发行版，最后只在 Neo4j 发行版仓库 中找到了相应的源码，我们还曾尝试使用对应版本的 maven 编译却未能成功，最后我们在网上偶然间发现了一个 binary release 的压缩包，然后到华为云的机器上安装好对应的 JAVA 8 环境才将这个版本的数据库运行起来。

负责这一部分的小组成员在运行了这一部分的 scala 脚本之后，私心以为这一份代码写的实在是有点不好，对于一份在主流机器上单线程运行、且运行时间需要以小时来衡量的代码来说，这份脚本的交互方式实在是不够友善，既没有存档点，也没有进度条的处理让人仿佛回到了上个世纪，而且由于版本不兼容的原因，需要复现的代码不停地向外抛出报错信息，实在是非常糟心。

小组成员总结经验教训如下：

- 严格按照相应版本配置运行环境，应该说环境是一个通病，年限较早的代码或多或少都会遇到这个问题。但有一些通用的跨平台的方式（比如 JVM 虚拟机）可以有效减少环境不兼容带来的麻烦，哪怕是在 MacOS/Aarch64 上，只要能找到相应的 JDK 支持，就可以将 java 代码和运行环境解耦，这也为我们自己写代码带来了一些启发。
- 好的工具很重要！使用现代的 IDE 工具（比如 Jetbrains IDEA）可以方便地管理配置相应的运行环境并配置参数，能够极大程度地减少麻烦，提高效率。

2. 人工标注

按照有效警告识别方向说明文档的指导，完成了 `open / close / unknown` 的标注。

2.1 相关文件

```
tool-build
└── hand-annotation-results           # 这一部分结束之后的主要结果
    ├── fixed-alarms.list             # 人工标注的有效警告列表，原列表的一份副本（主要结果 #1）
    └── unfixed-alarms-samples.csv    # 人工标注的无效警告列表，原列表的一份副本（主要结果 #2）
```

2.2 标注方式

- 对于有效警告，截取一条为例。

  ```
  EI_EXPOSE_REP2:commons-imaging:9ddad4b1ed76b8f405098f4220ff21bed05acecc:src/main/java/org/apache/commons/imaging/formats/psd/PsdHeaderInfo.java:34:34=commons-imaging:3f6c6c269cbd3d2e3a09e287a9080335c99d8cb1:src/main/java/org/apache/commons/imaging/formats/psd/PsdHeaderInfo.java#close
  ```

  其中分隔符 `#` 是小组成员在标注过程中人为插入的。 `#` 前的部分是原有效警告列表中的一条，而 `#` 后面的部分则是 `open / close / unknown` 人工标注的结果。

  需要说明的是，在完成这一部分时，由于小组成员的疏忽，开头部分警告的条目中 `⇒` 变成了 `=` ，导致格式有一些不统一，希望助教和老师在检查时谅解。

  我们一共完成了全部 206 条有效警告的标记。

- 对于无效警告，类似的，也截取一条为例。

  ```
  THROWS_METHOD_THROWS_CLAUSE_BASIC_EXCEPTION,commons-imaging,0ba49a1ce3d5944ea7b8979d3970584ec5d1a691,src/test/java/org/apache/commons/imaging/color/ColorCieLabTest.java,50,51#close
  ```

  我们一共完成了抽样出的 300 条无效警告的标记。

有关人工标注的准确性和误差等问题，将在下文中讨论。

3. 建模与训练

这一部分使用 1. 中得到的自动化标注结果训练一个简单的二分类器以拟合自动化标注结果。

3.1 相关文件

```
tool-build
├── datasets                  # 数据处理相关部分（主要实验结果 #1）
│   ├── pyscripts             # 一些 python 脚本，将在下文的详细说明
│   ├── raw                   # 1. 2. 中得到的部分原数据
│   └── results               # 数据处理结果
├── model1                    # 一个训练模型的尝试（主要实验结果 #2）
│   ├── README.md
│   ├── __init__.py
│   ├── __pycache__
│   ├── model.py              # 模型定义
│   └── train.py              # 数据预处理、训练、评估全过程
└── models                    # 另一个训练模型的尝试，已失败
```

```
    ├── MyModel.py
    ├── __init__.py
    ├── __pycache__
    ├── data_preprocess.py
    └── pretrain.py
```

3.2 特征选择和数据集生成

3.2.1 特征分析

我们观察了现有的警告的信息，发现主要有两部分，其一是 1. 中自动标记出的有效警告和无效警告的列表，其二是 1. 中扫描每个具体版本时留下的警告信息（见 `workings/working-c/reports/`），这两部分相互补充，再加上项目文件源代码中的一些信息，构成了现有的警告的特征。

小组成员认为，在建模时我们需要对于警告的特征进行一些挖掘，对标记条目进行简单的文本分类似乎不会是好的选择，我们使用了以下若干警告特征。

3.2.1.1 优先级 priority

`H` 或 `M` 或 `L`，由 spotbugs 自动生成的严重程度。

3.2.1.2 分类 type

spotbugs 将警告的类别分成若干个大类，每个大类下还有一些详细的小类（详细的分类信息可以参考 spotbugs 的手册-bugDescriptions），这些信息保存在 `reports/` 文件夹下的具体报告中，而“大类”和“小类”是分开的，例如：

```
"V","EI_EXPOSE_REP2"
```

我们直接将所有的小类信息作为警告的类别，为了避免出现不同大类中出现相似小类的情况出现，我们将大类信息与小类信息通过 `#` 连接起来，例如：

```
V#EI_EXPOSE_REP2
```

3.2.1.3 包名 package_name

包名（例如 `org.apache.commons.imaging.formats.psd.psdheaderinfo`）和类名（例如 `src/main/java/org/apache/commons/imaging/formats/jpeg/JpegPhotoshopMetadata.java`）比较相似，我们选用包名来作为表示警告位置的一个特征。

3.2.1.4 包（类）深度 package_depth

我们使用包深度这一特征来衡量某一个具体警告在项目中的位置到底有多“深”，比如包 `org.apache.commons.imaging.formats.psd.psdheaderinfo` 的包深度为 8（`len(package_name.split('.')) + 1`）。

3.2.1.5 代码行数 line_id_range

在警告的标记信息中给出了代码起始和结束行的前提下，我们使用这个简单的指标对于警告所在位置的代码“复杂度”进行一定程度的衡量。考虑到绝对的代码行位置似乎不能对于警告的建模起到良好的效果，而相对的位置应当具有一定意义。这个特征最大的优势在于计算方便，在我们选取的项目代码工整、格式化程度高的前提下，我们认为这个简单的特征可能会起到比预想中更好的效果。

小组成员也曾经尝试使用一些其他特征（最简单的，词法信息，比如警告所在涉及代码的 token 数、最长 token 长度和平均 token 长度等）来衡量警告所在代码的复杂度，但是考虑到 Java 变量名通常很长、调用通常很深等现状，我们认为上述特征的提取可能是复杂却不讨好。

3.2.1.6 提交时间 commit_time

在警告的信息中，还有一条不容易忽略的信息就是警告对应的版本的版本号（16 进制哈希值），直接使用这个字符串当做特征进行训练似乎丢失了太多有价值的信息，而且这个字符串本来就是一个随机生成的哈希值，很难说其本身对于警告的特征有太大的标识度，只有当其放入真正的 git graph 之中时，才有比较大的意义。小组成员认为，对于 git graph 的建模意义不凡，但是如何对图结构进行建模，似乎并不是一个简单的问题。

小组成员观察 git graph 之后发现，这个图基本上就是一条链，除去少量分支的 checkout 和 merge ，我们可以采用简单的方法表示两个版本的先后顺序，因此我们根据警告特征中的 commit-id 找到了相应的提交时间（机器时间戳中的绝对时间），尝试以此来衡量提交版本号的前后顺序。

3.2.1.7 有效性 is_true

这并不是一个真正的特征，这是警告的标记，已经在 1. 中自动化获取。

我们将上述特征整理为一个 csv 文件，其结构如下。

priority	type	package_name	package_depth	line_id_range	commit_time	is_true
m	v#ei_expose_rep2	org.apache.commons.imaging.formats.psd.psdheaderinfo	8	1	1476008766	1

需要说明的是，我们发现 1. 中脚本的输出中似有 bug ，部分大小写未统一，这导致了部分脚本的失配，因此我们直接将所有的英文转化为小写表示。

3.2.2 数据集生成

我们首先介绍一下各个脚本的功能。

```
datasets/pyscripts
├── gen_categories.py                          # 对于类别特征（priority, type, package_name）生成全集，unused now
├── gen_fixed_alarms_samples.py                # 将 1. 中的有效警告标记转化成无效警告标记的 csv 的格式
├── gen_handmarked_fixed_alarms_samples.py     # 处理人工标注的有效警告数据
├── gen_handmarked_unfixed_alarms_samples.py   # 处理人工标注的无效警告数据
├── gen_testing_dataset.py                     # 生成测试数据集（人工标注数据集）
├── gen_training_dataset.py                    # 生成训练数据集（自动标注数据集）
├── gen_unfixed_alarms_samples.py              # 从全部无效警告列表中随机筛选出若干条样本
├── single_violation_formatter.py              # 将单个警告的信息封装成所需的 csv 格式
└── utils                                      # 一些工具代码
    ├── git_repo_helper.py                     # 从 git 仓库中查找 commit_time
    └── raw_violation_helper.py                # 从 reports/ 文件夹中找到对应的原警告信息
```

由于时间和人力资源都有限，我们不可能标注所有无效警告，因此我们首先使用 `gen_unfixed_alarms_samples.py` ，从约 33000 条无效警告中随机筛选了 300 条无效警告，这也是 2. 中标注的部分。然后我们使用 `gen_fixed_alarms_samples.py` 格式化有效警告标注，使用 `gen_training_dataset.py` 生成了符合要求的数据集 csv 文件。

我们对于单条警告的格式化处理进行了封装（ `single_violation_formatter` ），它可以返回一个以 `,` 隔开的 csv 模式串，其中按顺序包含了 7 个特征。这个函数依赖于 `utils/` 中的两个子函数。这是生成数据集的核心函数， `gen_testing_data` 和 `gen_training_data` 都依赖于这个函数，它对于自动标注数据和人工标注数据的统一处理的能力为实验提供了很大的便利。

在获取到人工标注之后，我们使用 `gen_handmarked_{fixed, unfixed}_alarms_samples.py` 格式化人工标记的部分，然后再使用 `gen_testing_dataset.py` 生成测试数据集（人工标注数据集）。为了适应不统一且可能存在错误的数据格式，这些脚本的思想非常简单，实现却非常繁琐。

这一阶段我们生成的数据集如下。

```
datasets/results
├── fixed-commons-imaging-training-samples.csv      # 格式化之后的自动标注的警告样本集
├── fixed-testing-samples.csv                       # 人工标注的有效警告样本集
├── testing-dataset-ignoring-unknown.csv            # 不包含 unknown 的人工标注数据集，unused now
├── testing-dataset.csv                             # *测试数据集（人工标注数据集）
├── training-dataset.csv                            # *训练数据集（自动标注数据集）
├── unfixed-commons-imaging-samples.csv             # 等待人工标注的无效警告样本集
├── unfixed-commons-imaging-training-samples.csv    # 自动标注的无效警告训练样本集
└── unfixed-testing-samples.csv                     # 人工标注的无效警告测试样本集
```

除去两个标 `*` 的文件，其余都是一些中间结果，这两个文件将作为下阶段的输入。

`traning-dataset.csv` 包含一共 956 条训练数据，其中包含 206 条有效警告、300 条等待人工标注的无效警告以及 450 条随机的无效警告，而 `testing-dataset.csv` 包含一共 506 条测试数据，包含 206 条已经经过人工标记的（原）有效警告和 300 条已经经过人工标记的（原）无效警告。

有关测试数据的筛选问题，将在下一小节中讨论。

3.3 数据预处理、模型搭建和训练

3.3.1 数据预处理

3.3.1.1 测试数据选择

为了增强模型拟合自动化标注结果的能力，一个看上去理所应当的选择是将所有自动标注的结果都“喂”给模型，但当我们观察我们收集到的警告数据集，有效警告有 206 条，但无效警告约有 33000 条，我们认为这样极端失衡的数据会使得分类器选择“摆烂” —— 不管输入什么，都输出 0 的预测结果，这样甚至可以达到很高的准确率。

在文献综述的过程中我们了解了真实世界中的一些统计数据，有效警告大约占全部警告的 $20\% \sim 30\%$ 。我们想要模拟有效警告在真实世界中的分布情况，因此，除去测试集中的 506 条警告之外，我们额外引入了 450 条随机的无效警告。这样有效警告在全部警告中的比例达到了 21.548% 。

3.3.1.2 数据预处理

在我们建模的特征中，priority、type、package_name 都是类别特征（categorized_data），而剩下的 package_depth、line_id_range、commit_time 都是数字特征（numrical_data）。类别特征的向量化表示（embedding）一直都是有难度的一个问题。

我们尝试直接使用 `sklearn.preprocessing.OneHotEncoder` 对其进行最简单的建模，并尝试使用 `skorch` 中的轮子直接拟合这些特征（见 `models/` 中的代码）。这一次尝试得到了匪夷所思的结果，那就是分类器真的开始“摆烂”了。小组成员研究后发现，这样子直接建模后警告可用一个 277 维的向量表示，而绝大多数（除去最后三个数字特征）的维度上都是一个 0 ，这样不明显的特征学习起来可能确实非常有难度，我们认为这是导致这次尝试失败的原因。

我们在查阅资料的过程中发现了一篇教程 Introduction to PyTorch for Classification，觉得这和我们的任务比较类似，因此仿照教程中的文章搭建了几乎完全相同的神经网络。这个分类器将类别特征和数字特征分开，在 embedding 的时候限制了维度以防这些类别特征过于膨胀。以下是部分 embedding 的代码。

```
categorical_column_sizes = [len(dataset[column].cat.categories) for column in categorical_columns]
categorical_embedding_sizes = [(col_size, min(50, (col_size+1)//2)) for col_size in categorical_column_sizes]
```

事实证明，这个分类器的拟合效果还不错，这也是我们在下一阶段中使用的模型（见 `model1/` ）。

3.3.2 模型搭建与训练

3.3.2.1 模型

我们最终使用的模型结构如下。

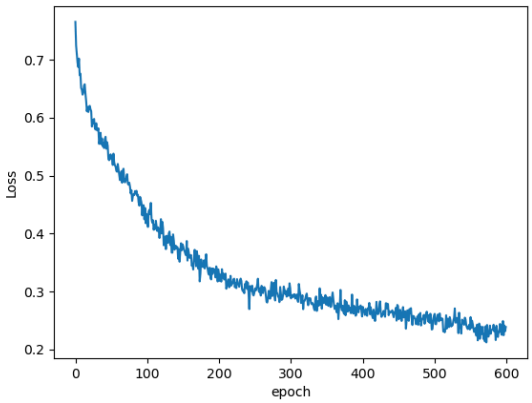
```
Model(
  (all_embeddings): ModuleList(
    (0): Embedding(3, 2)
    (1): Embedding(20, 10)
    (2): Embedding(179, 50)
  )
  (embedding_dropout): Dropout(p=0.4, inplace=False)
  (batch_norm_num): BatchNorm1d(3, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (layers): Sequential(
    (0): Linear(in_features=65, out_features=200, bias=True)
    (1): ReLU(inplace=True)
    (2): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Dropout(p=0.4, inplace=False)
    (4): Linear(in_features=200, out_features=100, bias=True)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): Dropout(p=0.4, inplace=False)
    (8): Linear(in_features=100, out_features=50, bias=True)
    (9): ReLU(inplace=True)
    (10): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): Dropout(p=0.4, inplace=False)
    (12): Linear(in_features=50, out_features=2, bias=True)
  )
)
```

由于时间有限，我们来不及对于这个模型的结构和部分参数进行细致的调整，但是对于一个只有少数几层的分类器来说，意义似乎也不大。

3.3.2.2 训练

训练的过程其实高度规范化，我们使用 `nn.CrossEntropyLoss()` 作为损失函数进行训练。

我们在尝试的过程中发现当训练轮数为 600 的时候，模型基本收敛（甚至有一些过拟合），我们将损失函数和训练轮数的关系简单绘制如下。



训练完成后，我们使用同一份数据集（自动标注数据）对模型的拟合情况进行简单的评估，以下为 `sklearn.metrics.classification_report` 得到的评估结果。

```
              precision    recall  f1-score   support

           0       0.93      0.96      0.94       750
           1       0.83      0.72      0.77       206

    accuracy                           0.91       956
   macro avg       0.88      0.84      0.86       956
weighted avg       0.90      0.91      0.90       956
```

尽管这样的测试可能意义并不大，但是应该足以说明该模型对于自动化标注数据的拟合效果不错，对该模型进行置信学习的尝试能够在一定程度上代表置信学习在该自动化警告标注技术中的效果。

这一部分的代码见 `model1/model.py` 和 `model1/train.py` 。

4. 置信学习

这一部分对于 3. 中的训练得到的模型应用置信学习技术，尝试进行标签的去噪。

4.1 相关文件

```
tool-build
└── confident_learning
    ├── clean_samples.py            # 清洗数据集中的噪声
    ├── cross_validation.py         # k 折交叉验证，生成概率预测矩阵
    ├── data_preprocess.py          # 数据预处理，将输入数据打包成模型的输入
    ├── evaluate.py                 # 简单评估置信学习的效果
    ├── evaluate_model.py           # 评估模型的效果
    ├── model.py                    # 模型的定义
    ├── train_model.py              # 训练模型
    └── work.py                     # 程序的入口，总 workflow
```

在这一部分中，我们对于 `model1/` 中的部分进行解耦，将模型的定义、训练的过程、评估模型的部分分别放入 `model.py` 、 `train_model.py` 、 `evaluate.py` 中，以方便后续的使用。

4.2 生成概率预测矩阵 `pred_prob`

根据置信学习开源包 CleanLab 和其 官方文档 的指导。由于我们的模型不满足 sklearn-compatible 的特点，因此我们选择使用 `cleanlab.filter .find_label_issues` 找出训练数据中的可能存在的噪声，需要为其提供模型预测的概率矩阵。

由于我们采用的模型最后一层的输出为一个二维向量，在 3. 中我们曾使用 `np.argmax()` 来表示模型的预测结果，在这一步我们对模型的最后一层输出应用 softmax 函数来表示相应的概率。

一种最简单的方法是直接将训练数据集放入模型中得到概率预测矩阵，我们首先尝试了这种方法， `find_label_issues` 为训练数据找出了约 80 个可疑噪声；而根据文档描述，更好的方法是生成“样本外”(out-of-sample) 的概率预测矩阵，查阅资料后得知这种方法被称为 k 折交叉验证 (kth fold cross validation) 我们根据Computing Out-of-Sample Predicted Probabilities with Cross-Validation的指导**手动复现**了这部分代码（见 `cross-validation.py` ），当我们采用 k = 20 时， `find_label_issues` 为我们找出了大约 280 个可疑噪声。

在一次具体实验中，我们找出了 285 个可以噪声，其中有 197 个存在于前 206 + 300 个人工标注的数据中，根据人工标注的结果，有 97 个真正的噪声。关于具体的实验结果分析，请参考 5. 结果分析。

5. 结果分析

这一部分我们将对置信学习对自动化标注的去噪效果进行分析，所有的结论和图表都将呈现在这份实验报告中，小组提交的代码仓库中也没有“相关文件”。

5.1 参数设置

将这一部分需要的一部分关键参数列举如下：

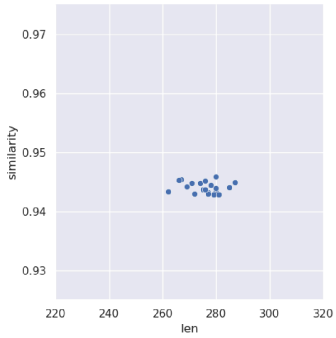
- `k` 折交叉验证的 `k` 取 `20`。
- 使用 `'self_confidence'` 作为 `find_label_issues` 的 `return_indices_rank_by` 的参数。
- 使用默认的 `prune_by_noise_rate` 作为 `find_label_issues` 的 `filter_bt` 的参数。

本节所有结论都在以上条件下控制并产生，我们认为本实验主要验证置信学习对于自动化标注的去噪效果，而其去噪效果的有无理应和参数的选择没有特别大的关系（只要不选择太过离谱的参数），而我们尝试过的部分有关参数的调整和选择问题将在 `6.` 中讨论。

5.2 稳定性讨论

小组成员进行连续 `20` 次重复实验，每一次实验中，我们利用同样的数据集重复训练模型并重复调用 `find_label_issues` 观察置信学习输出的结果是否相似。

为了衡量每次找出的可疑噪声集的相似程度，我们使用雅卡尔相似度（Jaccard's Similarity）度量两个可以噪声集的距离。对于每一个特定噪声集，使用它到其他所有噪声集的平均距离作为其“距离”的 `y` 轴参数，而噪声集长度作为 `x` 轴参数，绘制简单的散点图。



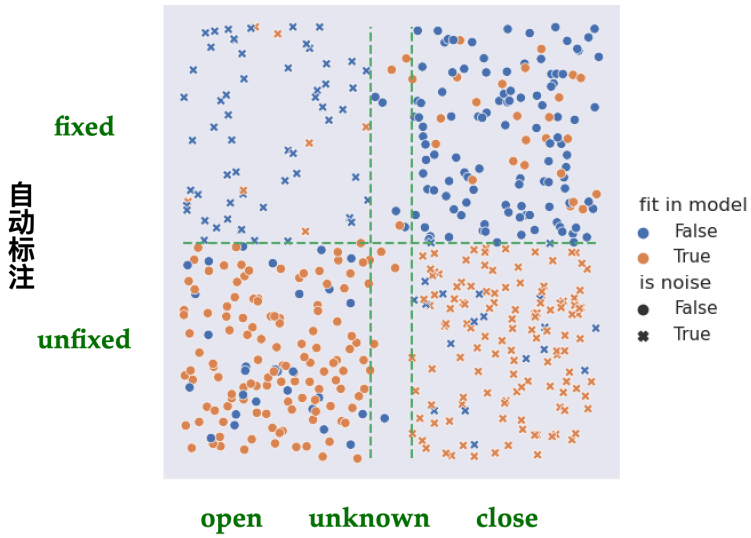
观察散点图可以发现，这些可疑噪声集的平均相似度很高（接近 `1`），尽管其长度分布不太均匀，但总体误差不大，可以认为 `find_label_issues` 具有一定的稳定性。

为了确保数据的统一，我们对这 `20` 次实验结果中出现的可疑噪声加权排序，并筛选出“最像噪声”的前 `275` 个（`20` 次噪声集大小的平均数）警告作为可疑噪声集，下面的讨论都基于这个可疑噪声集展开。

5.3 去噪效果

为了衡量去噪效果，需要先对自动化标记的部分中原来的噪声情况进行衡量。以下为原数据集（训练数据集）中警告有效性的分布情况（不考虑最后添加的 `450` 条无效警告，仅使用前 `506` 条）。

在这部分的图表中，为了更好的展示效果，我们为每一条警告根据其所在的象限随机设置了横纵坐标值，使其能够比较均匀地分布在平面上，因此散点图中点的横纵坐标的数值是没有意义的，有意义的是点在象限中的分布情况。

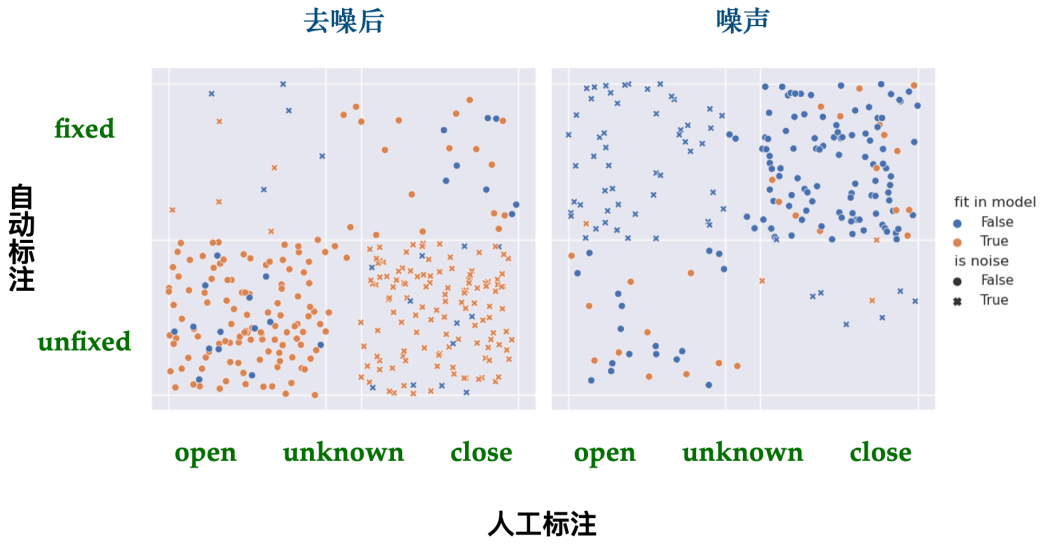


假如我们认为人工标记的具有绝对准确性，那么可以观察到原数据集中存在大量的噪声，我们的模型可能就是由于拟合了过多右下角的噪声导致其对右上角的警告拟合程度不佳，具体数据统计如下。

自动标注 | 人工标注	open	unknown	close
unfixed	154	4	142
fixed	64	7	135

至少有噪声 206 条，比例至少为 $206/506 \approx 40.711\%$，这样的噪声率对于训练数据集来说应当是不可接受的。

我们同时标出“可疑噪声集”在上图中的分布和去噪后的警告有效性的分布情况。



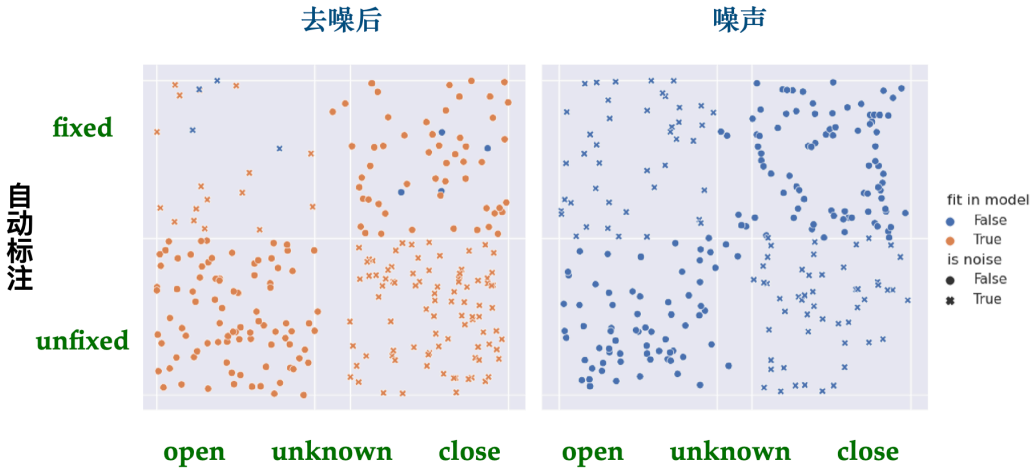
观察上图可以发现，在置信学习找出的可疑噪声集中，绝大部分点都是原来模型无法拟合的，这应该也符合置信学习的“本职工作”。上一张图片中我们发现，我们设计的模型过度拟合了右下角（`unfixed - close`）的警告，导致模型对于真正右上角（`fixed - close`）的拟合程度太低，这一部分数据集在置信学习的过程中被理所当然地视为“噪声”。除了此处存在较大的误差之外，置信学习基本上找出了左上角（`fixed - open`）的噪声。我们将去噪之后的数据统计如下。

自动标注	人工标注	open	unknown	close
unfixed		127	3	132
fixed		10	3	23

至少有 $127 + 23 = 150$ 条噪声，比例至少为 $150/298 = 50.333\%$（甚至变高了），置信学习不仅筛出了大量的真噪声，也一起“干掉”了一大部分的假噪声。

5.4 有关测试数据选择的讨论

在我们小组的实验设置中，为了适应有效警告在真实世界中的分布情况，除了 506 条需要人工标记的警告之外，我们还选取了 450 条额外无效警告，而在置信学习找出的可疑噪声集中，仅有 $142/275 \approx 51.636\%$ 条警告落在前 506 条警告中，还有接近一半的可疑噪声落在多余的这 450 条警告中。我们不禁思考，如果不使用这 450 条额外的警告训练，那么找出的“可疑噪声集”会有怎样的效果。以下是仅使用人工标记的 506 条警告训练的结果。



不得不说，这张图表是我们本次实验中最匪夷所思的实验结果，在仅使用前 506 条训练数据训练模型的情况下，置信学习的效果出奇地“好”，它几乎找出了所有模型不能拟合的训练数据，但是这些数据中噪声和非噪声（在每个象限都）非常均匀地分布，也就是说，几乎没能对真实世界中的噪声产生“去噪”的效果。

5.5 实验结论

综合以上事实与讨论，我们认为，**置信学习在一定程度上能够对自动化警告标记技术进行去噪，但效果应当比较有限**。

在 3.2.2.2 中，我们尝试利用一些 open metrics 证明我们搭建的模型能够较好地拟合自动化标记技术的结果，而在 5.3 中，我们也发现置信学习对于 fixed-open 部分的去噪效果相当不错（虽然它同时也错误地去除了大量 fixed - close 部分的有效数据），从总体上来说，还是应该认为置信学习在寻找真实世界的噪声中能够起到一定的效果。

尽管我们得出了以上的结论，但是我们认为**这个结论的前提并不稳固**，这个结论甚至是主要基于经验而不是基于实验事实，有以下几点原因：

- 数据量太小。我们仅仅选取了一个项目中的 206 条有效警告和 300 + 450 = 750 条无效警告，这个数据量对于分类器来说应该是远远不够的，因此可能产生比较大的偏差，影响置信学习的效果。
- 原始数据太“脏”，模型能力不佳。在上文的讨论中我们发现置信学习可以很好地找出模型中无法拟合的数据，但是就训练模型来说，5.3 节已经呈现，训练数据集中噪声的比例达到了 40% ，这对于模型能力的影响应当是非常致命的；另一方面来说，好的（强鲁棒性的）模型本身就应该具备一定的抗干扰能力，我们搭建的这个小模型在这样狭小的数据集下只能当做一个玩具。为了佐证以上的观点，需要更强大的模型和更海量的数据。

为什么说这个结论是基于经验的，我们想要在下一节中详细讨论。

6. 讨论、反思与总结

6.1 背景知识

我们小组在文献综述的过程中选择的也是这个方向，因而对于这些知识有过一些粗浅的认识。Is there a "golden" feature set for static warning identification?: an experimental evaluation对于警告特征的建模方法做出了一些总结，而 Learning to recognize actionable static code warnings (is intrinsically easy) 对于常见的机器学习模型使用 Golden Features 进行学习的效果进行了比较成熟的实验和比较，而 Detecting False Alarms from Automatic Static Analysis Tools: How Far are We? 不仅对于上面两篇文章中的结论进行了深入的总结和讨论，还设计评估了本次课程作业工具复现的第一步—自动化警告标记技术的有效性。而事实上，我们在文献综述中也花去较大的篇幅对这一部分的方法进行了思考与讨论。因此这一部分的知识和上一阶段的思考也一定程度上影响了我们实验的设计。

6.2 参数选择

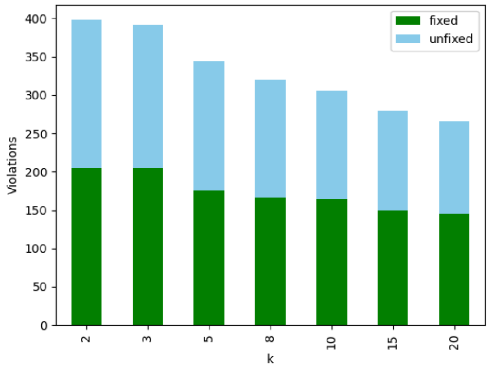
在实验的过程中，有很多可以调整的参数，我们主要将其分为三类：模型参数、交叉验证参数 k 、置信学习参数。

6.2.1 模型参数

说实话我们认为这部分的参数比较复杂，似超出了我们的能力范围，再加上时间有限，由于 Learning to recognize actionable static code warnings (is intrinsically easy) 中得出了的结论是警告的内在特征是很低维的，因此我们认为尝试不同的模型意义不大，再加上本项目的重点也不是设计合适的数据特征和建模训练，在得到了一个当时认为效果尚可的模型之后，我们就没有对这一部分调整展开实验。

6.2.2 交叉验证参数 k

在 5. 中我们选取 k = 20 ，这个数字应该是比较大的，小组成员查阅资料后了解到，在一般的 k 折交叉验证中通常取 k = 3, 5 较多，最多应该不超过 10 。小组成员对于 k 的选择展开了实验，以下是收集到的可疑噪声集与 k 的变化关系图。



同时我们将统计结果摘录如下。

k	fixed count	unfixed count
2	205	193
3	205	186
5	175	169
8	167	153
10	164	141
15	150	130
20	145	121

观察这张表格可以发现另一个比较奇怪的事实，那就是当 `k` 比较小的时候，置信学习将几乎所有的有效警告都放入了可疑噪声集（一共只有 206 条有效警告），而有效警告的数目基本上都要显著多于无效警告，这里面可能与我们建模的方式有比较大的关系。为了不损失太多的有效警告，我们选择 `k = 20` 以控制可疑噪声集的大小。

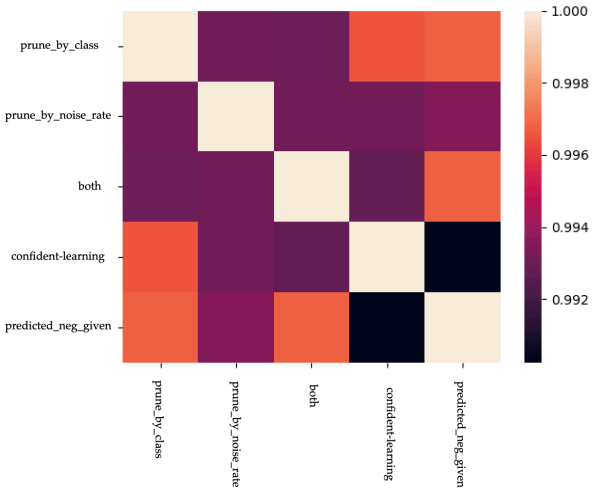
6.2.3 置信学习参数

我们认为，我们使用的 `find_label_issues` 有主要以下两个可以调整的参数（详见官方文档）：

- `filter_by`：是置信学习筛选噪声的方式，较为重要。
- `return_indices_ranked_by`：是结果返回的顺序，似乎不太重要。

我们控制 `5.` 中的变量，对于 `{'prune_by_class', 'prune_by_noise_rate', 'both', 'confident_learning', 'predicted_neq_given'}` 的输出结果进行简单的比较。

对于这几种方式输出的结果，我们依然使用雅各布相似度衡量集合之间的距离，绘制热力图如下。



可以发现，所有结果的输出都很近似，因此我们 `5.` 中的分析具有一定代表性。

6.3 误差分析和可能的改进方向

6.3.1 特征建模

说实话，我们认为这个问题实在是不简单，因为警告的特征本就难以衡量，而警告的有效性似乎又带有很多的“人工”成分，并不太自然，除了从结果来看警告有效性的分类结果天然是一个二分类器的任务，我们都不认为简单的机器学习方法可以取得比较优秀的效果。如何选取警告的特征、到底哪些特征是有效的，在只能凭借着一点点的经验的前提下，只能通过猜测而来不及通过大量实验佐证就显得有一些煎熬。我们也尝试对于这一部分特征根据我们自己的理解进行建模（`3.`），但是从最终实验效果来看我们的建模可能并不好，在这里我们提出几点可能的改进方向：

- 对于 `package_name` 我们简单地将其处理为了一个类别特征，而忽略了包名的语义以及代码文件的层次结构，但表示这些并不容易，似乎也是当前比较开放的一些问题。
- 我们的建模没有深入到代码层面，仅仅使用了静态分析器的输出作为分类的标准，而在警告的生成过程中，静态分析中的信息流可能可以反应更多有价值的信息，如何深入挖掘这一部分信息，可能也是一个可以入手的方向。
- 如何处理类别特征？在上文中提到，我们尝试使用简单粗暴的 `OneHotEncoder()` 对于类别进行建模，但是由于特征向量中的 `0` 太多导致分类器开始“摆烂”，因此类别特征的处理在特征建模上应该也具有非常重要的意义，而警告的特征中，动辄上百个类别，对这些特征进行 `embedding` 似乎也是一个比较复杂的问题。

~~在这一节中我们提出的改进方向似乎全是非常非常复杂的 open problem。~~

6.3.2 模型选择和训练

这个问题和上一个问题其实关联紧密。我们从 Learning to recognize actionable static code warnings (is intrinsically easy) 中了解到一个结论，那就是警告的内在特征是很低维的，简单的机器学习模型反而能够起到更好的效果，这也是我们在本项目中选择自己动手搭建一个简单模型的原因之一。在模型选择上，理应经过大量实验测试，还有前人珠玉在前，如果时间富余的话我们也想要进行一些其他的尝试（比如，声称效果最好的 CNN）。

而另外一个重点就是训练数据的选择，大量的、干净的训练数据（尤其是有效警告）对于分类器来说非常重要。我们猜测，正是因为我们引入了一部分多余的无效警告，使得我们的模型过多地拟合了 unfixed 部分的内容，才使得置信学习对于 fixed - open 的结果有比较精准的判断，而模型能力的不足同时也导致了 fixed-close 部分的拟合程度较低，即没能很好的学习到有效警告的特征（当然，也有可能是因为我们的特征建模存在更大的问题），我们推测这也是导致了 5.4 中匪夷所思的结论的原因。

所以这个方向可能的改进方向应当与上一节类似，还应加入对于模型能力的探索和尝试。

6.3.3 人工标注

小组成员在人工标注的过程中有一些感想，总结起来就是其实人工标注也并不是一件非常容易的事情，大概有以下困难：

- 机械化劳动，容易犯错。
- 难以理解代码，非项目作者在阅读 apache 这种大型项目的源码时，其实在短时间内难以形成深入的理解，增加犯错的概率。

由于没有其他更好的方法，我们只能认为人工标注是绝对准确的。但人工标注的数据中噪声分布的情况到底如何？我们对此持保留意见。

7. *感想与后记

> 总算写完了，qaq
>
> 今天是 2022 年 12 月 2 日，有人在写操作系统，有人在写需求，有人在写软件分析，还有大学牲在写一门三个星期前就考完了试的课程作业，我不说是谁。
>
> # 查询我的精神状态 🤣

写这份作业，最大的感受就是“难”，从构思到代码实现到配环境没有一个步骤是简单的，最后煞有介事的洋洋洒洒写了一个十几页的实验报告，好像干了很多事情，其实做的工作大多琐碎，大多数时间都是在和 python 的类型搏斗。

我们在作业的进行过程中和沈于杰同学的小组就作业进行了多次（遵守学术诚信的）交流，他们的一些思考对我们启发很大，让我们受益匪浅，在这里我想要感谢他们。

依然是感谢其他三位小组成员，帮助我分担了许多工作，但是由于我的个人能力和统筹能力不足，这份作业应该有更好的方法去完成，不至于这么痛苦。

当然还是要感谢老师和助教的辛勤工作，虽然难，但是这可能是我大三在南京大学软件学院做过的最有价值的一份作业🤣。在作业中，我们依然提出了许多激进的观点，我们没有冒犯任何人的意思，如有不敬，非常抱歉。我也知道我们的许多工作其实非常幼稚，不恰当和错误之处，恳请各位老师同学批评指正。

蔡之恒

2022 年 12 月 2 日。