Аннотация

Применение методов машинного обучения в статистическом анализе

Назаров Константин Олегович

Статические анализаторы широко используются в промышленности для нахождения и исправления уязвимостей в коде в процессе разработки, тем самым улучшая качество кода и упрощая дальнейшую разработку. Однако их применение ограничивается тем, что на больших и сложных кодовых базах статические анализаторы выдают большое количество false positives наряду с предупреждениями о действительных уязвимостях. Поэтому значительная часть данной работы посвящена тому, чтобы применить методы машинного обучения для классификации и приоритизации сообщений статического анализатора, что позволит сократить время, затрачиваемое разработчиками на ручную обработку всех сообщений анализатора. Основной проблемой при обучении подобного классификатора является составление тренировочного датасета, т.к. неочевиден алгоритм разметки данных и признаки для обучающей выборки. Также большим фактором является скорость генерации: признаки не должны вычисляться долго, т.к. сам статический анализ занимает очень большое время. В данной работе рассматривается способ генерации датасета при помощи тестовых сюит для статических анализаторов, а также показывается, что включение в обучающий датасет token-based представления кода помогает увеличить способность к распознаванию разных шаблонов ошибок, не увеличивая при этом сложности генерации датасета. Полученный датасет был использован для обучения двух моделей: decision tree и gradient boosting. Полученные классификаторы имеют высокую точность на тестовой выборке.

Содержание

1 Введение

1.1 Обозначения и сокращения

1. False Positive, FP - ошибка бинарной классификации, при которой классификатор ошибочно предсказывает положительный результат (например диагноз болен, когда болезни на самом деле нет).

2. True Positive, TP - случай, когда бинарный классификатор верно предсказал положительный результат (диагноз болен, когда болезнь есть)

3. Датасет - это таблица, строки которой называются объектами, а столбцы – признаками этих объектов. Иногда датасет включает в себя еще и метку для объектов.

4. Token-Based представление - представление кода в виде последовательности токенов, получаемых в результате лексического анализа.

5. Тестовая сюита - специализированный набор тестов.

6. Abstract Syntax Tree, AST - представлнение исходного кода в виде дерева абстрактного синтаксиса.

7. Application Programming Interface, API - описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой.

8. Common Weakness Enumeration, CWE [21] - Список уязвимостей программного и аппаратного обеспечения. Является общепринятым языком для описания природы уязвимости.

9. NIST - The National Institute of Standards and Technology.

10. SATE - Static Analysis Tool Expositions.

1.2 Введение

Статические анализаторы широко используются в промышленности для нахождения и исправления уязвимостей в коде в процессе разработки, тем

самым улучшая качество кода и упрощая дальнейшую разработку. Однако их применение ограничивается Множеством факторов. Статья "A Few Billion Lines of Code Later. Using Static Analysis to Find Bugs in the Real World"[4] раскрывает множество проблем с применением статических анализаторов на практике, среди которых: большое количество false positives ($35\% - 91\%$ [12]) на сложных или больших кодовых базах; пользователи, помечающие непонятные им сообщения как ложные; а также сложность обработки большого количества ошибок пользователем, которая приводит к тому, что анализаторы выдают меньше сообщений, чем могли бы. В связи с этим, Moritz Marc Beller с коллегами [3] обнаружили, что на практике очень малое число проектов с открытыми исходным кодом интегрирует статические анализаторы в процесс разработки. Также большая часть проектов не поддерживается правила, что предупреждения компиляции должны рассматриваться как ошибки. Поэтому значительная часть данной работы посвящена тому, чтобы применить методы машинного обучения для классификации и приоритизации сообщений статического анализатора, что позволит сократить время, затрачиваемое разработчиками на ручную обработку всех сообщений анализатора а также облегчит эффективное использование анализаторов на практике. Основной проблемой при обучении подобного классификатора является составление тренировочного датасета, т.к. неочевиден алгоритм разметки данных и признаки для обучающей выборки. Также большим фактором является скорость генерации: признаки не должны вычисляться долго, т.к. сам статический анализ занимает очень большое время (для относительно простых анализаторов время работы примерно в 2 раза превышает время компиляции, но может быть и много больше [14]). В промышленности эта проблема решается тем, что датасет зачастую имеется[17] [11] или может быть легко получен из статистики использования или из истории проекта[19]. В данной работе рассматривается способ генерации датасета при помощи тестовых сюит для статических анализаторов[10] (т.е. репозиториев с "образцовыми"программами, специально написанными для тестирования статических анализаторов), а также показывается, что включение в обучающий датасет token-based представления кода помогает увеличить способность к распо-

знаванию разных шаблонов ошибок, не увеличивая при этом сложности генерации датасета. Данные для обучения были получены путем применения нескольких статических анализаторов к кодовой базе Juliet C/C++ Test Suite [1]. Разметка была произведена автоматически из метаданных тестовой сюиты. Полученный датасет был использован для обучения двух моделей, которые впоследствии были протестированы на тестовой выборке.

2 Постановка задачи

Основной целью данной работы является сокращение времени, затрачиваемого разработчиком на обработку результатов статического анализа. Также важной целью является применение полученных результатов для анализа работы статического анализатора и его дальнейшей настройки. Для достижения обозначенной цели требуется решить следующие задачи:

1. Определить признаки для будущего датасета.
2. Определить метод генерации датасета и его разметки. Собрать данные.
3. Обучить классификатор сообщений статического анализатора.
4. Проанализировать полученные результаты.

Для того, чтобы результат работы считался удовлетворительным и применимым на больших масштабах, решения для поставленных задачи должны удовлетворять следующим требованиям:

1. Сложность извлечения признаков не должна превосходить сложности статического анализа.
2. Разметка датасета должна быть автоматизированной, т.к. ручная разметка занимает слишком долгое время и требует наличия проанализированной большой кодовой базы [2].
3. Классификатор не должен быть ограничен конкретными анализаторами, т.е. должен иметь возможность обработать и классифицировать предупреждение от любого анализатора после необходимой обработки.

3 Обзор существующих решений и литературы

Необходимость поддержки множества анализаторов обуславливается еще и тем, что каждый анализатор покрывает лишь конечное число уязвимостей, упуская остальные[8, 9, 4]. Поэтому для получения наиболее полного по типам ошибок датасета необходимо уметь обрабатывать сообщения от разных анализаторов. Решением данной проблемы является приведение всех кодов ошибок к кодам CWE [21] на этапе обработки сообщения и использование соответствующего кода CWE в датасете при обучении и классификации. Таким образом, добавление поддержки нового анализатора заключается в трансляции кода ошибок анализатора в CWE. Более подробно данный метод будет описан в дальнейшем в соответствующем разделе 4.1.4.

Sarah Heckman и Laurie Williams в своем исследовании[12] выделили следующие признаки, используемые в наиболее релевантных работах о классификаторах, подобных тому, который является целью данного диплома:

1. Характеристики ошибки - атрибуты предупреждения, сгенерированного статистиким анализатором, например: тип ошибки (double free, etc.), местоположение в коде (файл, класс, функция, строка, etc.), а также приоритет, который анализатор присвоил ошибке.

2. Характеристики кода - метрики кода, в котором содержится предупреждение. Эти метрики могут быть извлечены при помощи дополнительного анализа или из самого кода (цикломатическая сложность, количество строк в файле, etc.).

3. Метрики репозитория с исходным кодом - атрибуты репозитория с исходным кодом (история коммитов, частота изменения кода, история ревью, etc.)

4. Метрики базы данных с уязвимостями - информация об известных уязвимостях может быть связана с изменениями в исходном коде для того, чтобы идентифицировать уязвимость и необходимые исправления.

5. Метрики динамического анализа - гибридное использование статиче-

ского и динамического анализа может помочь устранить затраты, связанные с исполнением каждой техники анализа.

В данной работе внимание было сосредоточено целиком на первых двух пунктах: характеристики ошибки и характеристики кода. Изучив соответствующие статьи, было выяснено, что основными характеристиками кода для последующего анализа являются метрики, такие как число вложенности функции, число условных переходов, цикломатическая сложность, etc [10]. Однако все эти метрики являются метаданными, которые лишь характеризуют код в целом, не давая понятия о его структуре. Для того, чтобы каким либо образом анализировать структуру кода обычно используется представление AST[18]. Такой подход хорош для таких языков как Java, Python, где AST может быть легко получено. Для получения же AST в C/C++ необходима полная компиляция проекта. Т.к. каждый проект имеет свои систему для сборки, то невозможно описать данную процедуру единым простым путем. Поэтому в данной работе был опробован подход на основе token-based представления кода, предложенного в [18] для задач определения стиля проекта и вывода правил использования API. Идея состоит в том, что хоть представление в виде токенов и является неточным, но отфильтровав его, и анализируя наряду с метаданными кода, может быть получена более полная картина, нежели чем при использовании только метаданных. Включение токенов в обучающую выборку производилось в предположении, что все false positives следуют определенным шаблонам, которые могут быть определены при помощи предварительно обработанного token-based представления кода. Данное предположение было подробно изучено в работе Zachary P. Reynolds[16], в которой была произведена классификация этих шаблонов для нескольких статических анализаторов. Более подробно о token-based представлении в качестве признаков будет рассказано в соответствующем разделе 4.1.6.

В большинстве рассмотренных работ для классификации используются либо решающие деревья, либо LSTM нейронные сети[10, 19]. В данной работе будет рассмотрен только классификатор на основе решающих деревьев, и решения на основе рекуррентных нейронных сетей являются предметом дальнейших исследований, т.к. могут помочь находить более сложные шаб-

лоны ошибок.

Для разметки датасета предлагаются разные способы. Anshul Tanwar с коллегами в своей работе [19] предлагают способ разметки false positive на основе истории коммитов репозитория. В своей статье авторы брали исправления разработчиков в течение жизни проекта и сопоставляли их с отчетами статического анализатора. После чего производилась разметка: если исправление касалось куска кода, в котором анализатор находил уязвимость, то ставилась метка true positive, тоесть верное предсказание. Иначе, если разраобтчик помечал данный кусок кода как безопасный, то сообщение анализатора помечалось как false positive. Этот подход обладаем многими преимуществами, такими как больший охват различных сценариев срабатывания анализатора, тоесть обученный на подобных данных классификатор будет более гибким, т.к. запомнит более реалистичные шаблоны, нежели классификатор, обученный на рукописных тестах. Также преимуществом данного подхода является относительная простота реализации. Минусами же данного подхода являются три вещи. Первая - это необходимость наличия сторого формализованной истории коммитов в репозитории, чтобы суметь эффективно извлечь информацию об исправлениях. Вторая - необходимость наличия сообщений, помеченных разработчиками как false positive, что возможно только при должном уровне интеграции статического анализа в процесс разработки, что, как показал предыдущий анализ литературы, является большой редкостью. Третья же вещь - необходимость запускать анализ проекта много раз на разных этапах жизни для корректной разметки, что является очень ресурсоемким и долгим процессом. Другой подход, предложенный в работе Lori Flynn с коллегами [10], избегает многих минусов первого подхода за счет уменьшения охвата кодовой базы. В этой статье авторы использовали Juliet Test Suite - набор рукописных тестов, предназначенных для валидации эффективности различных статистиких анализаторов[1] для составления и разметки своего датасета. Подход к генерации датасета, используемый в данной работе был позаимствован из этой работы.

4 Построение решений задач

После анализа литературы и уже существующих решений стало ясно, как реализовывать каждую из поставленных ранее задач. Далее будет рассказано о том, какие методы были выбраны для выполнения этих задач. Порядок, в котором они были объявлены в разделе 2 сохранен.

4.1 Признаки для датасета

Обсудим признаки, которые были выбраны для обучения классификатора. В предыдущем разделе 3 было сказано, что из всех признаков, указанных в [12], данная работа сфокусируется лишь на первых двух - характеристики ошибки и характеристики кода, опуская последние три - метрики репозитория с исходным кодом, метрики базы данных с багами, метрики динамического анализа. Далее будет описано, почему при составлении датасета был опущен каждый из перечисленных классов признаков, а также будет описано то, какие признаки все же были представлены в датасете в настоящей работе.

4.1.1 Метрики репозитория с исходным кодом

Как было обозначено в обзоре литературы 3, для того, чтобы хоть сколько нибудь эффективно анализировать историю коммитов проекта, она должна быть строго организована, иначе поиск нужных мест в коде сводится к полному анализу всего проекта на каждом из этапов его разработки. В том же разделе было объяснено, что в открытом доступе очень малое число проектов соблюдает подобные правила. Плюс, даже при их соблюдении, множественный анализ проекта очень дорог вычислительно и по времени. Плюс, для того, чтобы размечать false positives, необходимо иметь доступ к уже собранной статистике использования анализатора, которой не существует в открытом доступе. Таким образом, анализ истории коммитов и других метрик репозитория отпадает.

4.1.2 Метрики базы данных с уязвимостями

Как и в предыдущем пункте, применение данных метрик ограничивается их отсутствием в открытом доступе.

4.1.3 Метрики динамического анализа

Для проведения динамического анализа требуется полная сборка всего проекта, что добавляет сложности к и без того затратному процессу анализа. Также, основным предметом изучения данной работы является именно статический анализ, поэтому рассмотрение метрик динамического анализа было решено оставить как предмет для будущих исследований.

4.1.4 Характеристики ошибки

Для того, чтобы удовлетворить требованию языковой независимости, все сообщения анализаторов требуется приводить к общему виду. Таким общим видом было выбрано представление Common Weakness Enumeration (CWE)[21]. Таким образом, добавление нового статического анализатора к списку поддерживаемых сводится к добавлению интерфейса, транслирующего код ошибки анализатора в код CWE. К счастью, для большинства распространенных статических анализаторов существуют таблицы соответствия или настройки, позволяющие выводить код ошибки сразу в формате CWE.

4.1.5 Характеристики кода

В проанализированной литературе основными признаками, по которым производилось обучение, являлись метрики кода. Чаще всего речь шла о метриках таких как вложенность, цикломатическая сложность, количество путей через выделенный фрагмент кода, etc. В данной работе метрики кода считались только для функции, в которой анализатор обнаружил уязвимость. Замеры производились при помощи утилиты ccsm[5], в которую были дополнительно внесены изменения. Реализована данная утилита как инструмент в инфраструктуре Clang[6]. Этот инструмент вызывает парсер

Clang для указанного файла, и работает с полученным частичным промежуточным представлением. Т.к. полная компиляция проекта не требуется (парсинг производится лишь на выбранном файле), то извлечение этих метрик не сказывается на общем времени сбора датасета. Метрики кода, использованные для обучения моделей в данной работе можно разделить на следующие группы:

1. Количество ключевых слов, контролирующих поток исполнения программы (for, if, else, etc.)
2. Различные способы подсчета цикломатической сложности
3. Количество различных обращений к памяти (разименование указателя, обращение к полю, обращение к полю по указателю, etc.)
4. Характеристики самой функции (количество путей через нее, вложенность)

4.1.6 Token-based представление кода

Описанные выше метрики характеризуют код лишь косвенно, не давая представления о его структуре. Таким образом теряется способность классификатора распознавать шаблоны, которые приводят к false positive. Чтобы бороться с этой проблемой был предложен подход на основе токенов. Токены были выбраны, т.к. анализ AST является затруднительным и более затратным. Токены, с другой стороны, можно получить еще на этапе лексического анализа. Вывод токенов был также реализован в качестве инструмента для Clang. Чтобы увеличить шансы на распознавание шаблонов false positive, представление кода в виде токенов было предварительно обработано: были оставлены только ключевые слова, влияющие на поток исполнения, операторы и идентификаторы.

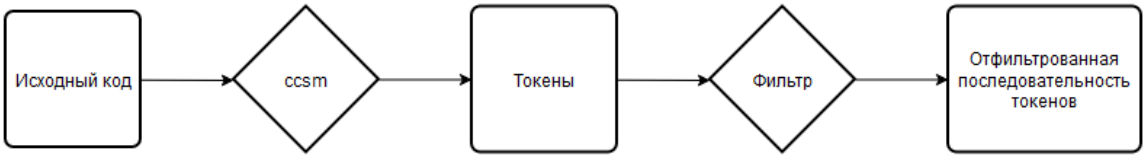

Рис. 1: Шаги получения отфильтрованной последовательности токенов

После предварительной обработки, из получившейся последовательности токенов выбирается подпоследовательность фиксированной длины, такая, что токен, на который указывает сообщение об ошибке находится ровно посередине подпоследовательности. Получившаяся подпоследовательность фиксированной длины называется окном и контролируется гиперпараметром WINDOW_SIZE, отвечающим за ширину этого окна. Соответственно, в датасете появляются признаки token[0], token[1], token[2], ... token[WINDOW_SIZE - 1]. Предположение данной работы состоит в том, что последовательности отфильтрованных токенов наряду с метаданными о коде должно хватить для распознавания большинства шаблонов false positive, указанных в [16].

Резюмируя, датасет, используемый для обучения моделей в данной работе, имеет следующий вид:

код CWE	token[0]	token[1]	...	token[WINDOW_SIZE - 1]	метрики кода	метка класса

Таблица 1: Общий вид элемента датасета

4.2 Метод генерации и разметки датасета

Для генерации и разметки датасета была выбрана кодовая база Juliet Test Suite[1]. Имеющиеся в коде метаданные позволяют без проблем классифицировать большинство сообщений анализатора как false positive или true positive. Рассмотрим пример кода Juliet Test Suite:

```
// CWE415_Double_Free__malloc_free_char_08.c
...
void CWE415_Double_Free__malloc_free_char_08_bad()
{
    ...
}
...

// manifest.xml
```

```
...
<testcase>
  <file path="CWE415_Double_Free__malloc_free_char_08.c">
    <flaw line="47" name="CWE-415: Double Free"/>
  </file>
</testcase>
...
```

Juliet Test Suite содержит два типа метаданных, которые полезны при оценке верности предупреждений анализатора:

1. manifest.xml - этот файл содержит точную информацию о каждой ошибке, включая номер строки, файл и тип ошибки.

2. Названия функций - в документации к тестовой сюите говорится, что если название функции содержит строку 'GOOD', то указанный тип CWE в данной функции не встречается, а если название функции содержит строку 'BAD', то данный CWE в этой функции присутствует.

Для разметки датасета были собраны следующие данные: для каждого файла, в котором анализаторы находили уязвимости были собраны метрики кода, а также token-based представление для каждой из функций. Далее стоит учитывать, что Juliet Test Suite указывает только на наличие или отсутствие конкретного CWE, указанного в названии файла, тоесть мы не можем выносить суждение о других типах уязвимостей, о которых сообщил анализатор в данном файле. Иными словами, если уязвимость, найденная анализатором не совпадает с CWE, указанным в manifest.xml или названии файла, то метаданные не могут быть использованы для того, чтобы пометить данное сообщение анализатора как true positive / false positive. Таким образом, после того, как вся тестовая сюита была проанализирована несколькими анализаторами, их сообщения об уязвимостях были слиты в единую базу данных, где каждый код ошибки был предварительно заменен на эквивалентный CWE либо при помощи встроенных в анализатор средств, либо при помощи уже заранее составленных таблиц соответствия. Далее, для каждого сообщения об ошибке была получена метка класса. Разметка происходила следующим образом:

1. Если информация об уязвимости в manifest.xml (путь, строка, тип) совпадает с той, которую сообщил анализатор, то ставилась метка true positive

2. Если уязвимость обнаружена в строках, принадлежащих функции с именем 'GOOD', и код ошибки совпадает с CWE, указанным в имени файла, то ставилась метка false positive.

Для остальных случаев невозможно точно определить метку. Однако существуют спекулятивные методы, которые являются предметом для дальнейших исследований и не были рассмотрены в данной работе. Их применение позволит расширить кодовую базу, на которой обучается классификатор, что увеличит качество и количество распознаваемых шаблонов false positive.

Тестовая выборка была получена путем разбиения всего датасета в соотношении 30 : 70, с сохранением этих пропорций для каждого из классов true positive и false positive.

4.3 Обучение классификатора

Выше было введено понятие обучения классификатора на размеченном датасете. Далее будут пояснены все использованные выше термины, такие как датасет, а также будут объяснены детали каждой из используемых моделей.

4.4 Обучение с учителем

Задачей машинного обучения является поиск функции, максимально приближающей заданную зависимость между элементами множества X (объектами) и элементами множества Y (таргетами), т.е. поиск $f(X)$:

$$f(X) = Y + \varepsilon$$

где ε - случайная величина, ошибка предсказания

Функция $f(X)$, отображающая объекты в таргеты, именуется моделью, а имеющийся у нас набор объектов иногда ещё называют обучающей выборкой или датасетом. Датасет состоит из:

1. Объекты (признаки) - элементы множества X

2. Метки (таргеты) - правильные ответы для $f(X)$.

В обучении с учителем мы хотим при помощи обучающей выборки построить модель, предсказания которой достаточно хороши. Обычно, качество предсказаний измеряют с помощью метрик качества, то есть функций, которые показывают, насколько сильно полученные предсказания, выдаваемые моделью, похожи на правильные ответы. Цель обучения обычно состоит в том, чтобы получить как можно более лучшее (наибольшее или наименьшее возможное, в зависимости от ситуации) значение метрики.

Выделяют два основных типа задач обучения с учителем в зависимости от того, каким может быть множество Y всех возможных ответов (таргетов):

1. Регрессия - $Y \in \mathbb{R}$ или $Y \in \mathbb{R}^M$. Примерами задач регрессии является предсказание погоды на завтра (температуры, влажности, давления).

2. Классификация - $Y = \{1,...,K\}$. Предсказание принадлежности объекта к одному из K классов. В случае $K = 2$ данная задача называется бинарная классификация. Именно эта задача и решается в данной работе. Необходимо определить, является ли объект (уязвимость, предсказанная анализатором и ее описание, как описано выше в 4.1) элементом класса false positive или true positive.

4.4.1 Решающее дерево. Decision tree

Одной из самых простых, но при этом эффективных для хорошо отделимых классов, моделей для решения задачи бинарной классификации является решающее дерево. Это такое бинарное дерево, в котором:

1. Каждой внутренней вершине v приписан предикат $B_v : X \to 0, 1$.

2. каждой листовой вершине v приписан прогноз $c_v \in Y$, где Y — область значений целевой переменной (в случае классификации листу может быть также приписан вектор вероятностей классов).

В ходе предсказания осуществляется проход по этому дереву к некоторому листу. Для каждого объекта выборки x движение начинается из корня. В очередной внутренней вершине v проход продолжится вправо, если $B_v(x) = 1$, и влево, если $B_v(x) = 0$. Проход продолжается до момента, пока не будет достигнут некоторый лист, и ответом алгоритма на объекте x считается прогноз c_v, приписанный этому листу[22].

Предикат B_v может иметь, вообще говоря, произвольную структуру, но, как правило, на практике используют просто сравнение с порогом $t \in \mathbb{R}$ по какому-то j-му признаку:

$$B_v(x,j,t) = [x_j \leq t]$$

При проходе через узел дерева с данным предикатом объекты будут отправлены в правое поддерево, если значение j-го признака у них меньше либо равно t, и в левое, если больше. В дальнейшем рассказе мы будем по умолчанию использовать именно такие предикаты.

Из структуры дерева решений следует несколько интересных свойств:[22]

1. Выученная функция является кусочно-постоянной, из-за чего производная равна нулю везде, где задана. Следовательно, о градиентных методах при поиске оптимального решения можно забыть.

2. Решающее дерево (в отличие от, например, линейной модели) не сможет экстраполировать зависимости за границы области значений обучающей выборки.

3. Решающее дерево способно идеально приблизить обучающую выборку и ничего не выучить (тоесть такой классификатор будет обладать низкой обобщающей способностью): для этого достаточно построить такое дерево, в каждый лист которого будет попадать только один объект. Следовательно, при обучении нам надо не просто приближать обучающую выборку как можно лучше, но и стремиться оставлять дерево как можно более простым, чтобы результат обладал хорошей обобщающей способностью.

Теперь разберемся с тем, как построить дерево. Другими словами, обучить модель. Построение дерева будем рассматривать итеративно. Пусть X - исходное множество объектов обучающей выборки, а X_m - множество объектов, попавших в текущий лист (в самом начале $X_m = X$). Тогда алгоритм можно описать следующим образом:

1. Рассмотрим вершину v. Если выполнен критерий остановки (например глубина дерева или степень ветвления), то останавливаемся, объявляем эту вершину листом и ставим ей в соответствие ответ.

2. Иначе, находим предикат $B_{j,t}$, который даст наилучшее разбиение текущего множества X_m на два подмножества X_l и X_r, максимизируя критерий ветвления $Branch(X_m,j,t)$.

3. Рекурсивно повторяем эту процедуру для каждого образованного подмножества. При этом в каждой вершине проверяем условие остановки. Если условие выполняется, то прекращаем рекурсию и объявляем текущую вершину листом. Иначе, продолжаем рекурсивно спускаться.

Остановимся более подробно на функции $Branch(X_m, j, t)$, т.к. ее выбор определит то, какую информацию можно будет извлечь из конечного предсказания. Для начала определим общую идею критерия ветвления. Пусть $c \in \mathbb{R}$ - ответ классификатора, тоесть метка класса, и пусть задана функция потерь $L(y_i, c)$, характеризующая то, насколько предсказание y_i отличается от ответа c. В момент, когда мы ищем оптимальное разделение X_m на X_l и X_r, мы можем вычислить для объектов из X_m то предсказание c, которое дало бы дерево, если бы текущая вершина была терминальной. Это предсказание c должно минимизировать среднее значение функции потерь:

$$\frac{1}{|X_m|} \sum_{(x_i,y_i)\in X_m} L(y_i, c)$$

Оптимальное значение этой величины

$$H(X_m) = \min_{c\in Y} \frac{1}{|X_m|} \sum_{(x_i,y_i)\in X_m} L(y_i, c)$$

Называется информативностью. Чем она ниже, тем лучше объекты в листе приближаются константным значением. Определим функцию потерь решающего пня (одного узла решающего дерева):

$$\frac{1}{|X_m|}(\sum_{(x_i,y_i)\in X_l} L(y_i, c_l) + \sum_{(x_i,y_i)\in X_r} L(y_i, c_r))$$

В качестве информативности удобно взять энтропию распределения классов. Пусть мы предсказываем вероятностное распределение классов $(c_1, c_2, ..., c_K)$ и пусть p_i - вероятность того, что объект принадлежит классу c_i. Тогда для информативности имеем[22]:

$$H(X_m) = -\sum_{(x_i,y_i)\in X_m} p_k \log p_k$$

В случае бинарного распределения:

$$H(X_m) = -p \log p - (1-p) \log (1-p)$$

Так как $p_k \in [0,1]$, то энтропия неотрицательна. Если случайная величина принимает только одно значение, то она абсолютно предсказуема и её энтропия равна 0. Наибольшего значения энтропия достигает для равномерно распределённой случайной величины - и это отражает тот факт, что среди всех величин с данной областью значений она наиболее "непредсказуема".

Однако у решающего дерева есть минусы. Самый главный - невозможность построения оптимального решения. Ниже приведен пример для решения задачи XOR. Какой бы критерий ни оптимизировали, решение никогда не приблизится к оптимальному. Картинка ниже иллюстрирует данную проблему:

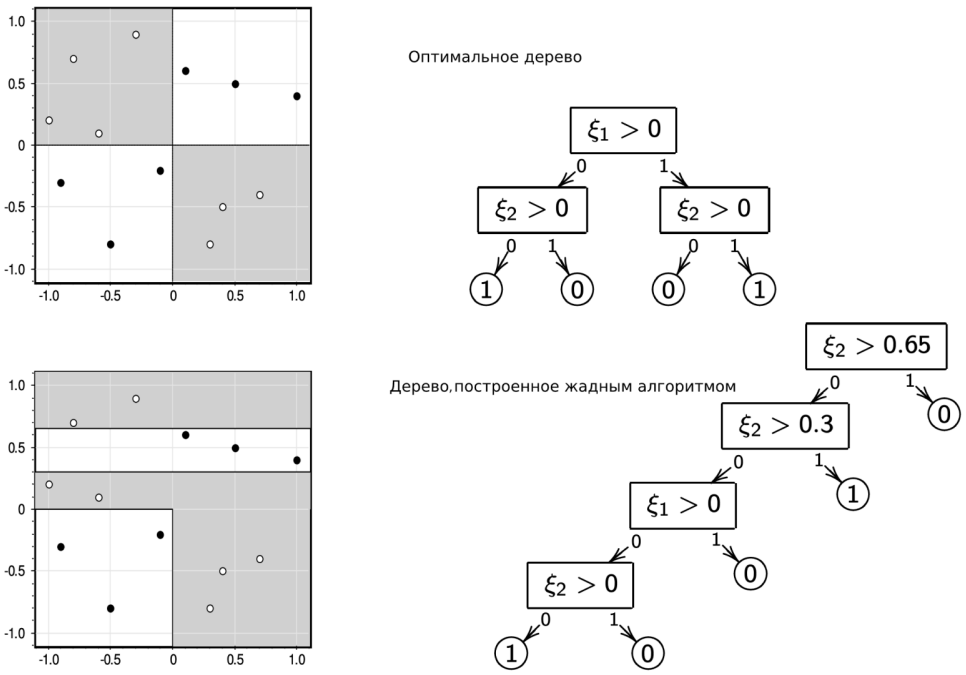

Рис. 2: Иллюстрация неоптимальности разбиения решающего дерева [22]

Для того, чтобы избежать переобучения дерева (ситуации, когда обобщающая способность классификатора начинает падать из-за слишком хорошей точности на обучающей выборке), в данной работе ограничивалась глубина дерева.

4.4.2 Градиентный бустниг. Gradient boosting

Кратко опишем идею, стоящую за градиентным бустингом (В данной работе говорится именно о GBDT, т.е. градиентном бустинге, построенном на решающих деревьях). Эта модель хорошо показывает себя на неоднородных данных, т.е. данных, представимых в виде набора признаков, таблицы и способен хорошо находить и приближать нелинейные зависимости. Идея градиентного бустинга проста: будем использовать ансамбль простых моделей, каждая из которых плохо приближает истинную зависимость, чтобы давать предсказания. Причем каждая последующая модель в ансамбле добавляется так, чтобы уменьшить ошибку уже имеющегося ансамбля. Теперь более формально. Для решения будем строить композицию из K базовых алгоритмов (в нашем случае - деревьев):

$$a(x) = a_K(x) = b_1(x) + b_2(x) + ... + b_K(x)$$

При дальнейшем построении используется следующая идея: при обучении одного дерева $b_1(x)$ нам известны объекты, на которых его предсказание было неверным. Тогда мы можем обучить следующее дерево $b_2(x)$ предсказывать разницу $s_i^1 = y_i - b_1(x_i) = y_i - a_1(x_i)$ между результатами первого дерева и истинной зависимостью для каждого объекта обучающей выборки. Данная разница выражается через градиент функции потерь:

$$s_i^k = y_i - a_1(x_i) = -\frac{\partial L(y_i, z)}{\partial z}|_{z=a_1(x_i)}$$

Или, в общем случае:

$$s_i^k = y_i - a_k(x_i) = -\frac{\partial L(y_i, z)}{\partial z}|_{z=a_k(x_i)}$$

Поэтому бустинг и называется градиентным. Аналогично будет обучено третье дерево и т.д. На каждом k-м шаге вычисляется разница между целевой переменной и предсказанием композиции $k-1$ алгоритмов, после чего k-е дерево учится предсказывать эту разницу, после чего композиция $a(x)$ обновляется:

$$a_k(x) = a_{k-1}(x) + b_k(x)$$

Обучение K базовых алгоритмов завершает построение композиции и обучение модели.

5 Результаты проделанной работы

Если в рамках работы писался какой-то код, здесь должно быть его описание: выбранный язык и библиотеки и мотивы выбора, архитектура, схема функционирования, теоретическая сложность алгоритма, характеристики функционирования (скорость/память).

В этом разделе будут приведены детали реализации классификатора, схема его работы и будут обсуждены возникшие в процессе реализации проблемы.

5.1 Структура проекта

Весь проект был реализован на языке python3 с использованием библиотек sklearn и XGBoost, из которых были взяты модели решающего дерева и градиентного бустинга соответственно. Структурно, проект можно разбить на три модуля:

1. Модуль, генерирующий датасет из Juliet Test Suite. Принцип его работы был описан в разделе 4.1.

2. Модуль, извлекающий признаки из указанного файла, чтобы можно было приоритизировать уязвимости, найденные в нем анализатором. Этот модуль работает аналогично предыдущему, с единственным отличием в том, что он не размечает полученные данные.

3. Классификатор, определяющий принадлежность сообщения анализатора к true positive или false positive. Этот модуль принимает на вход датасет составленный первым модулем и список объектов от второго, после чего обучает две модели на тренировочной выборке и дает предсказания для полученных объектов.

5.2 Пример работы

Пусть есть исходный файл input.c, который нам необходимо проанализировать. Первым делом необходимо передать этот файл на вход второму модулю, в результате чего получим примерно следующий вывод:

```
// data.csv
401,0.0,0.0,0.0,0.0,0.0,5.0,21.0,22.0,5.0,21.0,93.0, ...
775,22.0,5.0,5.0,21.0,13.0,13.0,22.0,91.0,21.0,5.0, ...
415,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0, ...
```

Как было обусловлено в разделе 4.1, первый столбец этой таблицы - это CWE код найденной анализатором ошибки. Далее идет окно токенов, после которого идут метрики кода. Нули в окне токенов означают выход за границы функции, в которой обнаружена ошибка.

Следующим шагом будет подать полученный data.csv на вход третьему модулю. После того, как обе модели обучатся на тренировочной выборке, будет выведен результат предсказания: метки и распределения классов в каждом из листьев. Ниже приведено предсказание для полученных выше объектов:

```
predict tree
[1 0 0] // решающее дерево предсказало, что CWE401 - true
        // positive, а CWE775 и CWE415 - false positive
[[0.00204499 0.99795501]  // Здесь приведены распределения по
 [1.         0.        ]  // классам для каждого из узлов
 [1.         0.        ]]
predict xgboost
[1 0 1] // предсказание модели градиентный бустинг отличается от
        // предыдущего тем, что CWE415 - тоже true positive
[[0.0170452  0.9829548 ]  // Уверенность ниже, но зато больше
 [0.9532441  0.0467559 ]  // обобщающая способность
 [0.03121328 0.9687867 ]]
```

Как видно из листинга выше, предсказания двух моделей расходятся, и только модель на основе градиентного бустинга корректно классифицировала все ошибки. Как видно из распределения классов, решающее дерево на 100% уверено в своем неверном вердикте, что говорит о том, что модель скорее всего была переобучена. Модель на основе градиентного бустинга же наоборот, хоть и менее уверена в своем ответе, но верно классифицировала все три случая, что говорит о ее хорошей обобщающей способности.

Также этот успех говорит о том что распознавание шаблонов false positive работает с достаточно хорошей общностью, т.к. структура у типичного теста Juliet Test Suite и input.c немного, но различалась.

5.3 Полученные результаты, их анализ

Главной проблемой полученного классификатора является ограниченность обучающей выборки. В ней на примерно 3000 объектов приходится примерно 400 объектов класса false positive, тоесть очень велика склонность к переобучению (как мы видели на примере решающего дерева выше) или неспособности вывести какой либо шаблон из тренировочных данных. Такое малое количество false positive объясняется качеством анализаторов и малым разнообразием кодовой базы тестовой сюиты. Другими словами, на таких простых примерах современные статические анализаторы выдают малое количество false positive, а однообразие тестов обеспечивает то, что если false positive и обнаружится, то он будет только одного шаблона. Возможным решением данной проблемы является внедрение кодовых баз из других тестовых сюит, а также внедрение спекулятивных методов разметки датасета. Однако это уже является предметом дальнейших исследований.

Для каждой из моделей были замерены две величины: precision и recall. Определим их:

1. precision - отношение true positive решений классификатора к сумме true positive и false positive предсказаний классификатора. В нашем случае false positive классификатора - это пометка верного сообщения статического анализатора как false positive. Этот сценарий крайне нежелателен, т.к. сводит к нулю смысл всей работы (если все равно придется перепроверять все сообщения, помеченные классификатором как false positive, то цель по сокращению затрачиваемого времени не может считаться достигнутой). Таким образом, нам крайне важно, чтобы precision стремился к 1.

2. recall - отношение true positive решений классификатора к сумме true positive и false negative предсказаний классификатора. В нашем случае это доля тех false positives, которые мы не упустили. Хоть в идеале и

эта величина должна стремиться к 1, однако ее отклонение от 1 не так критично.

Ниже приведены средние значения precision и recall для обеих моделей:

модель	precision	recall
решающее дерево	0.95	0.88
градиентный бустинг	0.98	0.94

Таблица 2: Результаты классификатора для обеих моделей на тестовой выборке

Как видно из результатов, градиентный бустинг лучше справляется с задачей классификации и обладает большей обобщающей способностью на тестовой выборке.

Для того, чтобы проверить предположение о целесообразности добавления token-based представления в качестве признаков, было произведено повторное обучение на частичном датасете, не включающем окно токенов. Были получены следующие результаты:

модель	precision	recall
решающее дерево	0.89	0.43
градиентный бустинг	0.90	0.46

Таблица 3: Результаты классификатора для обеих моделей на тестовой выборке без окна токенов

Как видно из таблицы, точность определения false positive без использования токенов является очень малой. Далее приведем предсказания обученных на таком датасете моделей для уже известного файла input.c, правильные предсказания для которого известны:

```
predict tree
[0 0 1]
[[0.96153846 0.03846154]
 [0.96153846 0.03846154]
```

```
 [0.01092896 0.98907104]]
predict xgboost
[1 0 0]
[[0.1660918  0.8339082 ]
 [0.793877   0.20612301]
 [0.64596635 0.35403365]]
```

Как видно из листинга, ни одна из моделей не смогла дать адекватного предсказания на реальных данных. Таким образом, предположение о том, что отфильтрованное token-based представление улучшает способность моделей к распознаванию шаблонов, оказалось верным. К схожим выводам можно было прийти из анализа деревьев, полученных в результате обучения.

Поговорим более подробно об анализе деревьев и о том, какую информацию он может дать. В первую очередь, анализ подобных деревьев может быть полезен для разработчиков статических анализаторов в процессе настройки или отладки. В процессе настройки разработчиком статического анализатора, у него имеется большая база данных со статистикой использования анализатора, по которой он смотрит на аномальные поведения и настраивает нужные метрики соответствующим образом. Вручную этот процесс занимает очень долгое время, однако при помощи анализа подобных деревьев может быть ускорен. Поскольку в узлах деревьев содержатся только метрики, которые наиболее существенны при классификации, на их настройку стоит обращать внимание в первую очередь. На кодовой базе Juliet Test Suite такими метриками наряду с token-based представлением были:

1. CWE код ошибки.
2. Количество операторов доступа к элементу массива.
3. Цикломатичческая сложность функции.
4. Количество приведений типов.
5. Количество ключевых слов else.

6. Вложенность функции.

7. Количество прямых доступов к членам структур.

Таким образом, методы, описанные в данной работе, могут быть применены на практике как пользователями статических анализаторов, так и их разработчиками.

6 Заключение FIXME:

Здесь надо перечислить все результаты, полученные в ходе работы. Из текста должно быть понятно, в какой мере решена поставленная задача.

В результате данной работы

- Был разработан инструмент, позволяющий приоритизировать предупреждения статического анализатора
- Был рассмотрен метод быстрой генерации датасета на основе тестовых сюит для статических анализаторов
- Были получены результаты, которые могут заметно упростить процесс разработки статических анализаторов

Список литературы

[1] NSA Center for Assured Software. *Juliet Test Suite*. 2022. URL: https://samate.nist.gov/SARD/test-suites/112 (дата обр. 28.04.2022).

[2] Nathaniel Ayewah и William W. Pugh. "The Google FindBugs fixit". В: *ISSTA '10*. 2010.

[3] Moritz Marc Beller и др. "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software". В: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), с. 470—481.

[4] Al Bessey и др. "A few billion lines of code later". В: *Communications of the ACM* 53 (2010), с. 66—75.

[5] bright-tools. *CCSM repository*. 2022. URL: https://github.com/bright-tools/ccsm (дата обр. 28.04.2022).

[6] Clang. *Clang compiler main page*. 2022. URL: https://clang.llvm.org/ (дата обр. 28.04.2022).

[7] Clang. *Clang Static Analyzer documentation*. 2022. URL: https://clang-analyzer.llvm.org/ (дата обр. 28.04.2022).

[8] Aurélien Delaitre, Vadim Okun и Elizabeth N. Fong. "Of Massive Static Analysis Data". В: *2013 IEEE Seventh International Conference on Software Security and Reliability Companion* (2013), с. 163—167.

[9] Aurelien Delaitre и др. "Evaluating Bug Finders: Test and Measurement of Static Code Analyzers". en. В: Complex FaUlts и Failures in LargE Software Systems (COUFLESS), Firenze, -1, 2015-05-23 2015. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=918370.

[10] Lori Flynn, William Snavely и Zachary Kurtz. "Test Suites as a Source of Training Data for Static Analysis Alert Classifiers". В: (май 2021).

[11] Lori Flynn и др. "Prioritizing alerts from multiple static analysis tools, using classification models". В: (май 2018). DOI: 10.1145/3194095.3194100.

[12] Sarah Heckman и Laurie Williams. “A systematic literature review of actionable alert identification techniques for automated static code analysis”. В: *Information and Software Technology* 53.4 (2011). Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing, с. 363—387. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2010.12.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584910002235.

[13] Ugur Koc и др. “Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools”. В: MAPL 2017 (2017), с. 35—42. DOI: 10.1145/3088525.3088675. URL: https://doi.org/10.1145/3088525.3088675.

[14] David Malcolm. *Static analysis in GCC 10*. 2022. URL: https://gcc.gnu.org/wiki/DavidMalcolm/StaticAnalyzer (дата обр. 28.04.2022).

[15] Michael Pradel и Koushik Sen. “DeepBugs: A Learning Approach to Name-Based Bug Detection”. В: *Proc. ACM Program. Lang.* 2.OOPSLA (окт. 2018). DOI: 10.1145/3276517. URL: https://doi.org/10.1145/3276517.

[16] Zachary Reynolds и др. “Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools”. В: май 2017, с. 55—61. DOI: 10.1109/SER-IP.2017..20.

[17] Joseph R. Ruthruff и др. “Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach”. В: 2018.

[18] Andrey Y. Shedko и др. “Applying probabilistic models to C++ code on an industrial scale”. В: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (2020).

[19] Anshul Tanwar и др. “Assessing Validity of Static Analysis Warnings using Ensemble Learning”. В: (апр. 2021).

[20] Cppcheck Team. *cppcheck static source code analysis tool for C and C++ code*. 2022. URL: http://cppcheck.net/ (дата обр. 21.05.2022).

[21] CWE Team. *Common weakness enumeration: A community-developed dictionary of software weakness types*. 2022. URL: http://cwe.mitre.org/documents/schema/index.html (дата обр. 28.04.2022).

[22] Yandex. *Глава про решающие деревья в учебнике по машинному обучению от ШАД*. 2022. URL: https://ml-handbook.ru/chapters/decision_tree/intro#%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80-%D1%80%D0%B5%D1%88%D0%B0%D1%8E%D1%89%D0%B5%D0%B3%D0%BE-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%B0 (дата обр. 28.04.2022).