Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б11-мм

MAP: Доработка и исправление правил валидатора текста учебных практик и выпускных квалификационных работ

ШМАКОВ Александр Александрович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент кафедры информационно-аналитических систем Чернышев Г. А.

Санкт-Петербург
2023

Оглавление

Введение

Написание выпускной квалификационной работы для студентов университетов имеет исключительное значение в их учебном процессе. Этот процесс предоставляет студентам возможность продемонстрировать усвоенные знания и умения, глубже погрузиться в конкретную область исследования, сформулировать новые идеи в предметной области, а также развить навыки самостоятельной работы и аналитического мышления. Кроме того, умение довести исследование до завершения в виде выпускной работы является важным опытом, который способствует развитию студентов как будущих специалистов в своей области.

Корректно выполненная академическая работа требует тщательной проверки на отсутствие грамматических и пунктуационных ошибок, а также соблюдение установленных стандартов оформления. Однако достижение такого результата требует значительных ресурсов и предполагает выполнение проверки текста в несколько этапов как студентом, так и его научным руководителем. Это требует затрат времени и усилий, которые могли бы быть направлены на улучшение содержания работы.

Приложение Mundane Assignment Police (MAP) [7], реализованное на языке программирования Kotlin, применяется для автоматизированной проверки курсовых и дипломных работ, и на данный момент MAP уже способен выявлять различные нарушения правил в тексте PDF-документов. В ходе данной работы предлагается улучшить и дополнить уже существующие правила проверки текста, реализованные в данном приложении. В частности, автору настоящей работы необходимо реализовать механизм, способный обнаруживать строки текста, содержащие программный код на различных языках программирования. Это необходимо для исключения проверки текста в указанных участках и снижения количества ложных результатов работы приложения.

1 Постановка задачи

Целью работы является доработка и исправление правил валидации текста учебных практик и выпускных квалификационных работ в приложении Mundane Assignment Police с целью уменьшения количества некорректных результатов. Для её выполнения были поставлены следующие задачи:

1. Спроектировать механизм, анализирующий текстовую строку и определяющий вероятность содержания в ней программного кода с помощью простых эвристик.

2. Реализовать спроектированную архитектуру на языке программирования Kotlin и интегрировать в программный код приложения.

3. Собрать датасет из существующих квалификационных работ для тестирования работы реализованного алгоритма.

4. Сравнить результаты работы приложения — количество ложно найденных ошибок до и после внедрения механизма обнаружения строк программного кода.

2 Обзор

2.1 Возникновение потребности в обнаружении программного кода в тексте документов

MAP является довольно мощным и полезным инструментом, который способен обнаруживать большое количество стилистических ошибок при написании академической работы. Работа с PDF-документами в приложении реализована с помощью библиотеки PDFBox, являющейся достаточно удобной и простой в использовании. Однако данный подход влечет за собой некоторые серьезные последствия, а именно огромное количество ложных срабатываний правил валидации.

Такое поведение приложения наблюдается из-за того, что библиотека PDFBox обрабатывает любой текст, который может встретить в документе вне зависимости от его расположения или характера. По этой причине различные листинги также обрабатываются как обычный текст, хотя валидация правил на них не должна осуществляться. Таким образом, такие паттерны кода, как, например, вызов функций или условные операторы, при написании которых используются скобки, могут вызвать ложное срабатывание правил проверки из-за отсутствия пробела перед открывающей скобкой. Приведенный пример является лишь частным случаем, в действительности же подобные ложные срабатывания могут быть обнаружены на тексте таблиц, рисунков, диаграмм, графиков.

2.2 Методы решения

Существует несколько подходов для решения поставленной задачи:

- Использование алгоритмов машинного обучения. Наиболее мощный и многообещающий метод, который с наибольшей вероятностью сможет более корректно определить содержание кода в рассматриваемой строке. Однако данный подход для решения поставленной задачи является очень требовательным к ресурсам, как к

техническим, так и финансовым.

- Поиск паттернов кода по регулярным выражениям. Такой подход может быть потенциально полезен в определении кодовых выражений в строке текста. Однако данный метод ставит перед программистом практически невыполнимую задачу, а именно внесение в базу данных максимально возможного количества регулярных выражений, которые можно встретить в программном коде какого-либо языка программирования.

- Создание набора ключевых слов и обработка текста на их основе. Простой, не подразумевающий комплексной и точной проверки текстовой строки, метод. Выделение в тексте строки ключевых слов само по себе не представляет особой ценности для качественного решения задачи, однако, данный подход возможен к расширению, например, путем добавления дополнительных проверок анализируемой строки на основе данных, собранных о ключевых словах, специальных символах и так далее.

2.3 Существующие реализации решения проблемы

2.3.1 Natural Language Toolkit for Python

Существует большое количество библиотек для языка программирования Python [9], реализующих алгоритмы машинного обучения, которые могут быть использованы для решения задачи выявления программного кода в тексте. Примером таких библиотек являются Scikit-learn [11], TensorFlow [12], PyTorch [10], NLTK [8]. Natural Language Toolkit является наиболее подходящей для поставленной задаче.

Natural Language Toolkit (NLTK) — это популярная библиотека для языка Python, предназначенная для обработки естественного языка. Она содержит множество инструментов и ресурсов для работы с текстом, включая токенизацию, морфологический анализ, синтаксический анализ и машинное обучение.

NLTK может быть использован для анализа текстового документа с целью обнаружения программного кода. Желаемый результат может быть достигнут с помощью NLTK следующим образом:

1. Токенизация: сначала текст документа следует разбить на отдельные слова и другие элементы (токены) с помощью токенизации. NLTK предоставляет инструменты для выполнения этой задачи.

2. Анализ текста: после токенизации можно провести анализ текста, чтобы выделить ключевые слова, которые могут указывать на наличие программного кода. Например, обычно в программном коде используются такие ключевые слова, как “if”, “else”, “for”, “while” и так далее.

3. Использование регулярных выражений: NLTK также поддерживает использование регулярных выражений для поиска специфических паттернов, которые могут соответствовать программному коду, например, строк, которые начинаются с ключевых слов или операторов.

4. Машинное обучение: NLTK также предоставляет инструменты для машинного обучения, которые могут быть использованы для обучения моделей на размеченных данных, чтобы обнаруживать участки текста, которые похожи на программный код.

Библиотека NLTK может быть полезна для базовых проверок и примитивного аналитического анализа строк текста. Однако библиотека не предоставляет средств для комплексных проверок, необходимых для точного выявления программного кода в текстовых строках.

Алгоритмы машинного обучения, реализуемые с помощью данной библиотеки, расходятся с постановкой задачи.

2.3.2 Code Detection Api

Code Detection API [1] — платный API, позволяющий обнаруживать программный код в текстовых документах, используя современные ме-

тоды обработки естественного языка и алгоритмы машинного обучения.

Используя Code Detection API, можно отправлять запросы с текстовым содержанием и получать в ответ информацию о том, содержит ли данный текст программный код, и если содержит, то где и какие участки этот код включает.

Процесс работы Code Detection API включает в себя следующие шаги:

1. Подготовка запроса: Необходимо сформировать запрос к API, включив в него текст, который нужно проанализировать на предмет наличия программного кода.

2. Отправка запроса: Сформированный запрос отправляется на сервер Code Detection API с помощью HTTP-запроса.

3. Анализ и обработка: API производит анализ содержимого запроса, используя методы обработки естественного языка, машинное обучение и другие технологии для выявления программного кода.

4. Возвращение результата: После обработки запроса, API возвращает информацию о наличии и местоположении обнаруженных участков программного кода.

Данный сервис может быть полезным инструментом для обнаружения программного кода в текстовых документах, особенно если необходимо автоматизировать этот процесс на больших объемах текстовых данных. Однако плата в 19$ за каждую тысячу токенов может оказаться бременем для проекта, в котором данный API используется.

2.3.3 Вывод

В результате проведенного обзора не было найдено доступной и соответствующей поставленной задаче готовой реализации, предлагающей полноценное решение задачи обнаружения программного кода в текстовых документах. Метод, реализованный при помощи Natural Language

Toolkit и основанный на машинном обучении, не соответствует поставленной задаче реализации сегментатора текста с использованием простых эвристик. Сервис Code Detection API, в свою очередь, является платным ресурсом, что не представляет интереса в контексте учебной практики и open-source проекта в целом.

3 Метод

В прошлом разделе были указаны недостатки и причины невозможности использования существующих подходов для решения поставленной задачи. По этой причине автором настоящей работы был спроектирован и реализован механизм выявляющий содержание программного кода в строке текста.

3.1 Технический стек

Веб-приложение разрабатывается с использованием следующего технического стека:

- Основным языком программирования, используемым при написании приложения, является Kotlin [4], который широко используется для создания веб-приложений. Средство управления зависимостями и сборки проекта — Maven [5]. Для написания модульных тестов используется фреймворк Kotest [3].
- Для фронтенд-разработки используется фреймворк Vue.js [13].
- JavaScript — используется для разработки клиентской части веб-приложения.
- Python [9] — используется для реализации некоторых алгоритмов машинного обучения, что позволяет повысить функциональность и уровень аналитики веб-приложения.

3.2 Реализация алгоритма выявления строк, содержащих программный код

Согласно поставленной задаче, необходимо создать утилиту, способную выявлять является ли строка текста строкой программного кода, с помощью простых эвристик.

Для решения поставленной задачи были предприняты и выполнены следующие шаги:

1. Составить базу ключевых слов наиболее известных языков программирования, символов, используемых при написании программного кода, символов, с которых потенциально может начинаться строка кода, символов, которые потенциально могут быть единственными в строке кода.

2. Составить датасет программ, написанных на поддерживаемых языках программирования, для последующего подсчета вероятности содержания кода в строке.

3. Написать парсер строк, способный разделить строки на слова и все кодовые символы, собранные в составленной базе слов и символов языков программирования.

4. Составить список свойств, которыми потенциально может обладать строка, содержащая код на языке программирования. Выделить свойства, которые однозначно указывают является строка кодовой или нет.

5. Вычислить частоту (ϕ), с которой ключевые слова и символы встречаются в данной строке, путем деления количества ключевых слов и символов на общее количество слов в строке.

6. Вычислить процент удовлетворения свойствам кодовой строки (ϑ) делением количества выявленных свойств на общее количество свойств.

7. Вычислить и усреднить ϕ и ϑ для датасета (пункт 2), собранного из программ, написанных на поддерживаемых языков программирования. Принять вычисленные значения за необходимую нижнюю границу, через которую должны пройти ϕ и ϑ для того, чтобы быть определенными, как строки, содержащие программный код.

8. Проверить, если в данной строке есть свойства однозначно указывающие на содержание программного кода (Пункт 1). Если такие свойства не найдены, то сравнить значения ϕ и ϑ, полученные для

данной строки, со значениями полученными на основе анализа датасета. Если оба значения переходят или равны минимальному порогу, то строка считается кодовой.

3.3 Составление базы данных и датасета

Для составления базы данных ключевых слов, встречающихся в языках программирования, были выбраны наиболее известные языки программирования такие, как C, C++, C#, Java, Kotlin, JavaScript, Python, Ruby, Scala, OCaml, Haskell, F#, PHP, Pascal, Assembly language, Fortran, Cobol, Rust, Bash, SQL, Algol, Perl. Материал для составления базы был взят из интернет-ресурсов Microsoft Learn [6], Wikipedia [14], а также GitHub [2].

Для составления датасета примеров программ на вышеперечисленных языках программирования были использованы программы из репозитория sample-programs [15].

3.4 UML-диаграмма классов

Для иллюстрации разработанного механизма была создана UML-диаграмма классов (Рисунок 1), отображающая взаимосвязи между различными элементами реализованной функциональности.

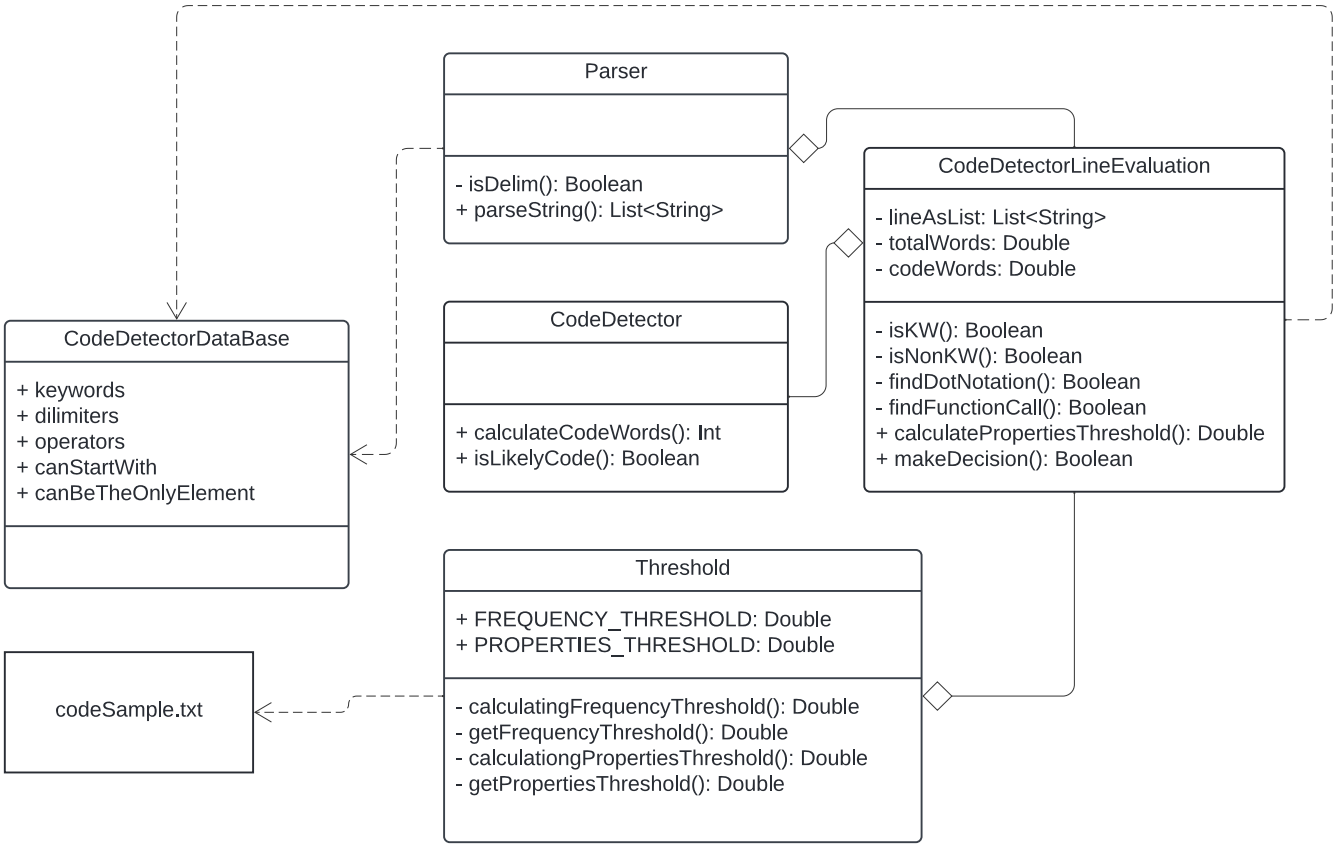

Рис. 1: Диаграмма классов

- Класс CodeDetector являет собой ядро утилиты, объединяющее остальные классы. Метод isLikelyCode() обращается к классу CodeDetectorLineEvaluation, вызывая его метод makeDecision().

- CodeDetectorLineEvaluation — собирает информацию по рассматриваемой строке и устанавливает в соответствующие свойства $true$ или $false$ в зависимости от удовлетворения свойствам. Для того, чтобы метод makeDecision() мог подсчитать вероятность, с которой рассматриваемая строка содержит программный код, метод обращается к классам Parser и Threshold. Результатом метода будет $true$ или $false$ в зависимости от полученных результатов.

- Parser — разбивает строку текста на слова и символы, определенные в CodeDetectorDataBase.

- Threshold — подсчитывает минимальный порог для ϕ (Пункт 5) и ϑ (Пункт 6), используя Parser. Данные для подсчета границ

берутся в датасете codeSample (Пункт 2).

- CodeDetectorDataBase — база данных ключевых слов и символов (Пункт 1).

4 Эксперимент

В результате работы в общей сложности было написано 916 строк полезного кода по статистике GitHub, а также составлен датасет из программ на поддерживаемых утилитой языках программирования на 3150 строк.

Для проверки корректности интеграции механизма обнаружения строк, содержащих программный код, требовалось провести сравнительный анализ результатов работы приложения до и после интеграции данного механизма с целью установить, сократилось ли количество ложных срабатываний правил валидации. Для этой цели обе версии приложения были протестированы на датасете, собранном на основе наличия фрагментов кода в документах. Все работы были взяты из архива учебных практик и выпускных квалификационных работ на сайте кафедры Системного программирования [16].

Ожидается, что версия приложения с реализацией распознавания строк кода:

- Корректно обрабатывает те же входные данные, что и оригинальная версия
- Выводит результат с уменьшенным количеством ложных срабатываний правил.

В Таблице 1 приведены результаты тестирования приложения до и после интеграции механизма распознавания в код приложения. В колонках указаны имя автора работы, количество найденных ошибок до интеграции, количество ошибок после интеграции соответственно и процент снижения ложных срабатываний (ν). Наименьшим выявленным результатом является снижение на 5.5%, в то время как наибольшим — 22.5%. Среднее уменьшение ложных ошибок составляет 12.66%.

Таблица 1

Имя	До интеграции	После интеграции	ν
Bochkarev A.	345	282	18.3%
Beloshapkin M.	147	132	10.2%
Tagiltsev S.	127	120	5.5%
Mischenko S.	71	66	7%
Moskalenko I.	112	97	13.4%
Ploskin A.	102	79	22.5%
Artemenko S.	71	62	12.7%
Gabdrahmanov A.	105	96	8.6
Mihajilov I.	175	155	11.4%
Platonova E.	271	225	17%

Заключение

Механизм выявления содержания программного кода в строке был реализован и интегрирован основную ветку репозитория веб-приложения.

Результаты:

- В ходе проектировки метода распознавания программного кода в тексте PDF-документов, был разработан алгоритм, отвечающий поставленной задаче разработки механизма, основанном на простых эвристиках.
- Для реализации механизма была разработана система классов, отвечающая заданному методу реализации алгоритма, а также соответствующая общему стилю написания кода приложения.
- Была произведена оценка качества результатов первоначальной версии приложения по сравнению с версией, включающей интегрированный механизм распознавания кода. Это было осуществлено путем тестирования на составленном наборе данных, собранном из имеющихся курсовых и выпускных квалификационных работ. Результаты данного анализа подтвердили уменьшение количества ложных срабатываний текстовых правил валидации.

В планах, помимо улучшения, доработки и расширения механизма выявления программного кода, реализовать распознавание в тексте PDF-документов таблиц и рисунков, что потенциально также может улучшить результаты работы приложения и снизить количество ложных срабатываний правил валидации текста.

Код приложения с интегрированным механизмом распознавания срок программного кода доступен по ссылке основной ветки репозитория MAP [7]: https://github.com/Darderion/map.

Список литературы

[1] Code Detection API. — URL: https://codedetectionapi.runtime.dev.

[2] GitHub. — URL: https://github.com/.

[3] Kotest Framework. — URL: https://kotest.io/docs/framework/framework.html.

[4] Kotlin documentation. — URL: https://kotlinlang.org/docs/home.html.

[5] Maven documentation. — URL: https://maven.apache.org/guides/index.html.

[6] Microsoft Learn. — URL: https://learn.microsoft.com.

[7] Mundane Assignment Police. — URL: https://github.com/Darderion/map.

[8] Natural Language Toolkit. — URL: https://www.nltk.org/.

[9] Python documentation. — URL: https://www.python.org/doc/.

[10] Pytorch documentation. — URL: https://pytorch.org/docs/stable/index.html.

[11] Scikit-learn website. — URL: https://scikit-learn.org/stable/.

[12] TensorFlow documentation. — URL: https://www.tensorflow.org/api_docs.

[13] Vue.js documentation. — URL: https://vuejs.org/guide/introduction.html.

[14] Wikipedia. — URL: https://en.wikipedia.org.

[15] sample-programs repository. — URL: https://github.com/TheRenegadeCoder/sample-programs.

[16] Кафедра Системного программирования Математико-механического факультета СПбГУ. — URL: se.math.spbu.ru.