Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Леденева Екатерина Юрьевна

Сравнительный анализ алгоритмов вычисления текстовых метрик для документации программного обеспечения

Бакалаврская работа

Научный руководитель:
д. т. н., профессор Кознов Д. В.

Рецензент:
руководитель команды аналитики ООО "Интеллиджей Лабс" Поваров Н. И.

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Ekaterina Ledeneva

Comparative analysis of text distance algorithms for software documentation

Bachelor's Thesis

Scientific supervisor:
Doctor of Engineering, Professor Dmitry Koznov

Reviewer:
Head of Analytics, IntelliJ Labs Co. Ltd. Nikita Povarov

Saint-Petersburg
2020

Оглавление

Введение

Документация играет важную роль в разработке программного обеспечения. Зачастую она имеет большой объём, в течение жизненного цикла разработки в неё вносится множество изменений разными авторами. Без хорошо налаженного процесса разработки и сопровождения в документации накапливаются ошибки, рассогласования и неточности. Поэтому актуальна максимальная автоматизация этого процесса.

При написании документации фрагменты текста часто копируются с последующими изменениями, что приводит к появлению неточных повторов [4, 19]. В технической документации, в отличие от других видов литературы, такие повторы могут быть уместны: текст становится унифицированным, а одна и та же информация выражена одинаковыми словами. Однако при редактировании многократно скопированного фрагмента изменения должны быть распространены на все его копии, чтобы не образовывалось рассогласований. Отслеживать скопированную много раз информацию в больших документах затруднительно, поэтому необходимы инструменты поиска неточных повторов.

Эти инструменты также находят применение в задаче повторного использования фрагментов текста: неточные повторы являются кандидатами на переиспользование [28]. Однако действительно переиспользованы могут быть не все обнаруженные фрагменты, так как похожие тексты не всегда означают одно и то же.

Для написания документации программного обеспечения обычно используются специальные средства. Например, для документирования кода на Java применяется инструмент JavaDoc, позволяющий генерировать HTML-страницы из комментариев исходного кода [20]. Другим примером является DITA — средство структурирования технической документации в виде топиков на основе XML [3]. Все подобные инструменты задают строгий формат документации, который необходимо учитывать при поиске неточных повторов.

Для обнаружения неточных повторов в документации был создан инструмент Duplicate Finder [35], который основан на инструменте по-

иска клонов в программном обеспечении. Однако приведённые в работе [32] результаты экспериментов свидетельствуют о недостаточном качестве поиска.

Поиск повторов основывается на попарном сравнении фрагментов документации. Для этого могут быть применены различные алгоритмы сравнения строк и обработки текста. Так, в [33] используется редакционное расстояние по наибольшей общей подпоследовательности. В той же работе показано, что алгоритм поиска по образцу имеет сложность полинома четвёртой степени от длины образца. Поэтому немаловажным критерием выбора алгоритма попарного сравнения является быстродействие.

В работе [28] приводится сравнение алгоритмов вычисления меры близости строк, но там исследуются топики в DITA-документации в задаче поиска кандидатов на переиспользование. В работе [4] исследуются повторы в JavaDoc-документации, однако не рассматриваются неточные повторы.

Итак, оказывается актуальным исследование существующих алгоритмов для сравнения JavaDoc-комментариев в Java-приложениях.

Постановка задачи

Целью данной работы является сравнительное исследование существующих алгоритмов вычисления сходства текстовых фрагментов для задачи определения сходных JavaDoc-комментариев в индустриальных Java-приложениях. Для достижения этой цели были сформулированы следующие задачи.

- Изучение предметной области и выбор алгоритмов для исследования.
- Разработка и программная реализация инфраструктуры эксперимента.
- Проведение эксперимента и анализ результатов.

1. Обзор

В этом разделе рассматриваются известные форматы технической документации и некоторые исследования на тему поиска неточных повторов в документации, также производится обзор алгоритмов текстовой близости.

1.1. Форматы документации

Для разработки технической документации применяются различные языки разметки и инструменты генерации документации в удобном для чтения формате.

Одни инструменты позволяют создавать "внешнюю" по отношению к исходному коду документацию. Таким инструментом, например, является DITA [3]. В DITA документация генерируется из топиков — целостных по смыслу блоков текста, которые могут содержать ссылки друг на друга, обеспечивая механизм переиспользования. Также к этой группе относятся языки разметки Markdown [13] и Wiki [21].

Другие инструменты используют аннотирование исходного кода для автоматической генерации документации. К таким инструментам относятся Doxygen [5] (поддерживает языки C/C++, Python и другие), Pydoc [22] (для языка Python), JavaDoc [20] и другие. Так как данная работа фокусируется на JavaDoc-комментариях, далее этот инструмент описывается подробнее.

JavaDoc — инструмент генерации документации в HTML-формате из исходного кода Java [20]. Для генерации используются комментарии к классам, полям, методам и пакетам, расположенные непосредственно перед описываемым объектом и заключённые в символы `/**` и `*/`.

Комментарии должны состоять из двух частей: описания и набора тегов. Каждый тег состоит из специального имени, начинающегося с символа `@` (например, `@author`, `@param`, `@return` и т.д.), и строки–значения. В комментарии может содержаться несколько тегов с одинаковым именем.

```
99      /**
100      * Put a diagnostic context value (the <code>val</code> parameter) as identified with the
101      * <code>key</code> parameter into the current thread's diagnostic context map. The
102      * <code>key</code> parameter cannot be null. The <code>val</code> parameter
103      * can be null only if the underlying implementation supports it.
104      *
105      * <p>
106      * This method delegates all work to the MDC of the underlying logging system.
107      *
108      * @param key non-null key
109      * @param val value to put in the map
110      *
111      * @throws IllegalArgumentException
112      *           in case the "key" parameter is null
113      */
114     public static void put(String key, String val) throws IllegalArgumentException {
```

Рис. 1: Пример JavaDoc-комментария

На рис. 1 представлен пример JavaDoc-комментария из проекта Slf4J [29]. Он состоит из описания, двух `@param`-тегов и одного `@throws`-тега.

1.2. Duplicate Finder

Duplicate Finder — это инструмент, созданный на кафедре системного программирования СПбГУ и предназначенный для поиска и анализа неточных повторов в программной документации [6, 35]. Инструмент автоматически находит неточные повторы, объединяет их в группы и предоставляет пользователю визуализацию в виде тепловой карты повторов: по ней можно перейти к конкретным участкам документации для дальнейшей работы. Также выделяются общие и отличающиеся части повторов внутри групп. Затем в интерактивном режиме пользователь может редактировать выделенные группы, изменяя их состав и границы входящих в них повторов.

Для объединения фрагментов текста в группы Duplicate Finder использует наибольшую общую подпоследовательность: отношение длины общей подпоследовательности к длине каждого элемента группы должно превышать порог, задаваемый пользователем.

Duplicate Finder использовался в рамках данного исследования экспертами для полуавтоматического выделения групп неточных повторов

из набора Java-проектов. Эти группы затем были использованы автором данной работы для выделения положительных и отрицательных пар комментариев.

1.3. Исследование Soto и др.

В работе [28] приводится сравнение четырёх алгоритмов вычисления меры близости текстов для поиска кандидатов на переиспользование (определяемых схоже с неточными повторами) в DITA-документации: косинусное расстояние, наибольшая общая подпоследовательность, Google Trigram similarity и Locality-sensitive hashing. В качестве набора данных использовались пары DITA-топиков, часть которых была помечена как кандидаты на переиспользование.

На парах вычислялась мера сходства через один из четырёх алгоритмов, в качестве результата выдавался ранжированный по убыванию сходства список. Для оценки алгоритмов применялись точность на топ-n парах и площадь под графиком зависимости полноты от количества выдаваемых пар. Лучшим алгоритмом по результатам такой оценки оказалась наибольшая общая подпоследовательность.

В исследовании также поднимался вопрос быстродействия. На DITA-документации большого размера длительность исполнения первых трёх алгоритмов превышала сутки, тогда как Locality-sensitive hashing справлялся с задачей за 5 минут, уступая однако остальным алгоритмам в точности.

1.4. Обзор алгоритмов

Алгоритмы вычисления текстовых метрик можно разделить на два класса: синтаксические и семантические [27]. Синтаксические алгоритмы измеряют структурную близость, основываясь на посимвольном сходстве: например, наибольшая общая подпоследовательность или n-граммы символов или строк. Семантические алгоритмы учитывают значения слов, что обычно достигается выделением статистической информации из больших корпусов текста: например, векторное представ-

ление word2vec [7] или сходство на основе n-грамм из корпуса Google Web 1T [11].

В данной работе было решено ограничиться синтаксическими алгоритмами в виду специфики предметной области: неточные повторы получаются многократным копированием и изменением текста, а эти действия являются синтаксическими.

Для исследования были выбраны следующие алгоритмы: наибольшая общая подпоследовательность, косинусное расстояние, расстояние на основе хеширования и расстояние Левенштейна. Первые три алгоритма были взяты из статьи [28], расстояние Левенштейна добавлено в силу его распространённости.

1.4.1. Наибольшая общая подпоследовательность

Наибольшая общая подпоследовательность (LCS, longest common subsequence) — набор подстрок, входящих в обе строки в одинаковом порядке [17]. Как и в [28], мы разрешаем подпоследовательности состоять только их целых слов. Для получения меры сходства из наибольшей общей подпоследовательности достаточно взять отношение удвоенного количества слов в ней к сумме слов в обеих строках:

$$similarity_{LCS}(s_1, s_2) = \frac{2 \cdot |LCS(s_1, s_2)|}{|s_1| + |s_2|}.$$

В данной работе для нахождения наибольшей общей подпоследовательности использована реализация алгоритма из библиотеки difflib [23].

1.4.2. Косинусное расстояние

Косинусное расстояние (COS, cosine similarity) измеряет сходство между текстами как косинус угла между их векторными представлениями [2]. В качестве представления был взят вектор частот слов (term frequency): каждый элемент вектора соответствует уникальному слову, входящему хотя в бы один комментарий из пары, а значением является отношение количества вхождений этого слова в строку к количеству

всех слов в строке:

$$similarity_{COS}(s_1, s_2) = \frac{\overrightarrow{s_1} \cdot \overrightarrow{s_2}}{||\overrightarrow{s_1}|| \cdot ||\overrightarrow{s_2}||}.$$

Реализация алгоритма вычисления косинуса между векторными представлениями взята из библиотеки SciPy [26].

1.4.3. Locality-Sensitive Hashing

Locality-sensitive hashing (LSH) приближённо вычисляет сходство между строками с помощью хеширования [10]. Метод понижает размерность данных, что даёт прирост производительности.

LSH основывается на создании сигнатур. Как предложено в [28], в качестве сигнатур берётся minhash, приближающий коэффициента Жаккара [24]. Исходная строка представляется мешком n-грамм, minhash случайным образом переставляет n-граммы и возвращает минимальный номер n-граммы, присутствующей в строке. Для получения набора сигнатур процедура повторяется несколько раз.

Вычисленные сигнатуры затем группируются в наборы одинаковой длины и хешируются. В данном исследовании было взято 100 сигнатур, сгруппированных по 2. Мера сходства двух строк определяется пропорционально количеству совпавших хешей.

В данном исследовании были использованы реализации minhash и LSH из библиотеки datasketch [30].

1.4.4. Расстояние Левенштейна

Расстояние Левенштейна (LEV, Levenshtein distance) — это минимальное количество вставок, удалений и замен символов, необходимых для преобразования одной строки в другую [17]. Для преобразования расстояния в меру сходства надо нормализовать его максимально возможным значением — максимумом длин двух строк — и вычесть из 1. Это же значение можно получить из отношения длины наибольшей общей подпоследовательности к максимуму длин строк (в отличие от

алгоритма из раздела 1.4.1, здесь наибольшая общая подпоследовательность не ограничивается целыми слова):

$$similarity_{LEV}(s_1, s_2) = 1 - \frac{LEV(s_1, s_2)}{max(|s_1|, |s_2|)} = \frac{LCS(s_1, s_2)}{max(|s_1|, |s_2|)}.$$

Для вычисления расстояния Левенштейна в данной работе используется алгоритм поиска наибольшей общей подпоследовательности из библиотеки difflib [23].

2. Инфраструктура эксперимента

2.1. Подготовка данных

Для сравнения неточных повторов в JavaDoc-комментариях было принято решение использовать целые комментарии. С одной стороны, часто оказываются интересными повторяющиеся фрагменты в разных комментариях, и при этом сами эти комментарии могут оказаться весьма не похожими друг на друга в целом. Так было сделано в [34]. Можно также сравнивать друг с другом отдельные JavaDoc-теги, как это сделано в [4]. Однако оба эти подхода находят слишком мелкие повторы, создавая запутанную общую картину.

Также решение ограничиться сравнением целых комментариев объясняется тем, что, как показали эксперименты, сделанные в рамках данной работы, средняя длина комментария в исследованных Java-приложениях невелика и составляет 44 слова.

Для оценки работы алгоритмов была собрана коллекция пар JavaDoc-комментариев, разбитая на два класса: положительный — с парами, комментарии в которых являются неточными повторами, и отрицательный.

2.1.1. Описание исходных данных

Для создания пар комментариев были использованы четыре известных Java-проекта: GSON [9], JUnit4 [12], Mockito [15] и Slf4J [29]. Выбор именно этих проектов связан с тем, что они использовались в исследовании [4], и в дальнейшем хочется сравнить полученные в нашем исследовании результаты с результатами коллег[1].

Из каждого проекта с помощью инструмента Duplicate Finder экспертами предметной области были выделены группы JavaDoc-комментариев, такие что комментарии внутри одной группы схожи по смыслу. Для выделения групп использовалась не только текстовая близость комментариев, но и схожесть функциональности методов, к кото-

[1]По разным причинам этого не было сделано в данной работе, но планируется в дальнейшем.

```
/**
* Convenience method to get the specified member as a JsonPrimitive element.
*
* @param memberName name of the member being requested.
* @return the JsonPrimitive corresponding to the specified member.
*/
public JsonPrimitive getAsJsonPrimitive(String memberName) {
    ...
```

```
/**
* Convenience method to get the specified member as a JsonArray.
*
* @param memberName name of the member being requested.
* @return the JsonArray corresponding to the specified member.
*/
public JsonArray getAsJsonArray(String memberName) {
    ...
```

```
/**
* Convenience method to get the specified member as a JsonObject.
*
* @param memberName name of the member being requested.
* @return the JsonObject corresponding to the specified member.
*/
public JsonObject getAsJsonObject(String memberName) {
    ...
```

Таблица 1: Группа неточных повторов JavaDoc-комментариев

рым они относятся.

В таблице 1 приведён пример выделенной экспертами группы неточных повторов JavaDoc-комментариев из проекта GSON [9]. Цветом выделены вариативные части элементов группы. Можно заметить, что методы, к которым относятся эти элементы, имеют схожую функциональность и отличаются только типом возвращаемого значения, что отражено и в самих JavaDoc-комментариях.

2.1.2. Схема обработки

По полученным выше группам для каждого проекта были сгенерированы все возможные пары JavaDoc-комментариев. Каждой паре была присвоена метка: если комментарии состояли в одной группе, выделенной экспертами, они считались неточными повторами и попадали в положительный класс, иначе — в отрицательный.

При этом отрицательный класс получился значительно больше положительного. Поэтому из него случайным образом было выделено под-

convenience method to get this element as a primitive short. @return get this element as a primitive short. @throws NumberFormatException if the value contained is not a valid short value. --- convenience method to get this element as a primitive integer. @return get this element as a primitive integer. @throws NumberFormatException if the value contained is not a valid integer.	1
convenience method to get this element as a {@link JsonObject}. If the element is of some other type, a {@link IllegalStateException} will result. Hence it is best to use this method after ensuring that this element is of the desired type by calling {@link #isJsonObject()} first. @return get this element as a {@link JsonObject}. @throws IllegalStateException if the element is of another type. --- convenience method to get this element as a {@link BigDecimal}. @return get this element as a {@link BigDecimal}. @throws NumberFormatException if the value contained is not a valid {@link BigDecimal}.	0
...	…

Таблица 2: Фрагмент размеченного набора данных

множество пар, равное по количеству с положительным классом.

Объединение четырёх наборов размеченных пар, полученных из упомянутых ранее проектов, составило итоговый набор данных из 2638 пар: 364 пары из GSON, 744 пары из JUnit4, 824 пары из Mockito и 706 пар из Slf4J. Фрагмент полученного набора данных представлен в таблице 2.

2.2. Процесс обработки (pipeline)

Для получения меры сходства двух JavaDoc-комментариев можно применить к ним один из алгоритмов вычисления текстовых метрик, рассматривая комментарии как строки. Однако при этом теряется дополнительная информация, которая содержится в комментариях и может быть использована для более качественного сравнения — теговая структура. Более того, некоторые теги можно переставлять местами, при этом текстовые метрики будут выдавать пониженное сходство, в то время как на самом деле сходство сохраняется.

Было принято использовать строковые алгоритмы не только в исходном виде, но создать процесс (pipeline) с дополнительными шагами.

Процесс состоит из шагов, представленных на рис. 2. Кроме обыч-

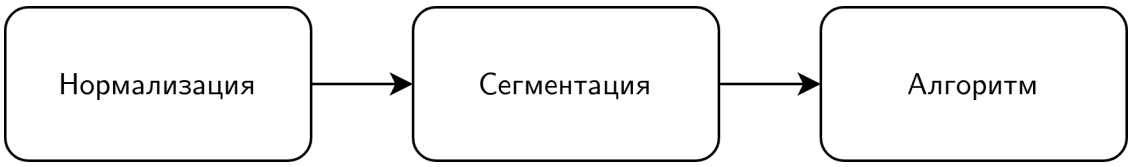

Рис. 2: Процесс обработки

ного применения алгоритма к комментариям как к обычному тексту предусматривается также нормализация и сегментация. Поскольку заранее не было известно, какой вариант работы алгоритма является наилучшим, каждый алгоритм проверялся в следующих сочетаниях: обычный алгоритм, нормализация+алгоритм, сегментация+алгоритм, нормализация+сегментация+алгоритм.

2.2.1. Нормализация

Нормализация используется для того, чтобы оставить от текста только значимую информацию. Нормализация может принимать один из двух видов: частичный и полный.

Частичная нормализация состоит из удаления пунктуации и лемматизации (приведения слова к его словарной форме). Лемматизация осуществляется с помощью WordNet — лексической базы данных английского языка [14]. А именно, с помощью входящей в WordNet библиотеки Morphy, приводящей слова к нормальным формам посредством анализа окончаний и списка слов-исключений для разных частей речи. Если WordNet не смог преобразовать слово (например, это название класса, состоящее из нескольких слитно написанных слов, или слово не на английском языке), оно оставляется как есть.

Полная нормализация добавляет к частичной удаление стоп-слов, взятых из библиотеки NLTK [1]. Этот дополнительный вид нормализации был выделен, так как было замечено, что удаление стоп-слов по-разному влияет на результат разных алгоритмов.

2.2.2. Сегментация

Сегментация позволяет учитывать структуру JavaDoc-комментариев, описанную в разделе 1.1. Комментарии разбиваются на JavaDoc-теги, части описания присваивается пустой тег. Теги, встретившиеся несколько раз (например, `@param`, `@exception` или `@throws`), группируются: такие теги будем называть составными.

Затем к одноимённым тегам двух комментариев применяется один из исследуемых алгоритмов.

В случае составных тегов добавляются следующие действия. Обозначим за I и J множества вхождений тега в первый и второй комментарии соответственно. Построим матрицу $S = (s_{ij})$, где s_{ij} — результат алгоритма на строках $i \in I$ и $j \in J$. Чтобы оптимально сопоставить друг другу элементы I и J и получить итоговое значение, определим следующую задачу линейного программирования:

$$\begin{aligned} &\max_X \overline{S}^T \cdot \overline{X} \\ &\sum_{j \in J} x_{ij} \leq 1 \ (\forall i \in I) \\ &\sum_{i \in I} x_{ij} \leq 1 \ (\forall j \in J) \end{aligned}$$

где $\overline{S}$ и $\overline{X}$ — векторы, полученные из записанных подряд строк соответствующих матриц. Условия этой задачи означают то, что элементы I и J сопоставляются друг с другом с некоторым весом (возможно нулевым) и суммарный вес, относящийся к одному элементу, не может превосходить единицы. Результатом для рассматриваемого составного тега считается $\dfrac{\max\limits_X \overline{S}^T \cdot \overline{X}}{\min\{|I|, |J|\}}$.

В результате описанных действий получается набор чисел d_i, соответствующих каждому присутствующему в обоих комментариях теге. Итоговое сходство определяется как $\dfrac{\sum_i d_i \cdot l_i}{\sum_i l_i}$, где l_i — сумма длин тега в комментариях.

2.3. Подбор порога

После применения одной из конфигураций описанного выше процесса обработки для каждой пары была получена мера сходства — число в промежутке от 0 до 1. Далее необходимо преобразовать это число в ответ, является ли пара неточным повтором, иными словами произвести классификацию.

Для решения этой задачи было решено использовать машинное обучение. Был выбран метод под названием логистическая регрессия [18]. Этот метод позволяет построить линейный классификатор вида $sign(w_0 + \sum_i w_i \cdot x_i)$, где x_i — числовые признаки классифицируемого объекта, w_i — веса, минимизирующие логистическую функцию потерь на обучающей выборке.

Первоначальной идеей было взять два признака для пары комментариев: меру сходства и сумму их длин. Последний признак казался важным в виду эмпирических соображений о том, что для длинных комментариев достаточно небольшой степени сходства, чтобы считать их неточными повторами, тогда как для коротких комментариев сходство должно быть значительным. Однако эксперименты показали, что зависимость классификации от длины слишком мала, поэтому было решено оставить только один признак. Тогда построение классификатора сводится к подбору порога w, такого что относящимися к положительному классу считаются все пары с $x \geq w$, где x — мера сходства.

2.4. Метрики

Для оценки классификации в данной работе были использованы три метрики: F-мера, Accuracy и ROC AUC (реализация их алгоритмов вычисления взята из библиотеки scikit-learn [31]). Так как для применения F-меры и Accuracy необходимо выделение тестового подмножества данных, эта операция осуществляется с помощью J-K-fold кросс-валидации. Для описания метрик введём несколько определений.

Истинно положительными (tp, true positive) будем называть верно классифицированные элементы положительного класса, ложно поло-

жительными (fp, false positive) — ошибочно отнесённые к положительному классу элементы. Аналогично определяются истинно отрицательные (tn, true negative) и ложно отрицательные (fn, false negative) элементы.

2.4.1. J-K-fold кросс-валидация

Данный метод является расширением широко используемой для оценки классификаторов K-fold кросс-валидации [16]. В K-fold кросс-валидации набор данных делится на k частей, классификатор обучается на $k-1$ части (обучающий набор), а к оставшейся части (тестовый набор) применяется одна из метрик (например, F1-мера или Accuracy). Перебирая тестовую часть, генерируется k чисел.

Однако полученные числа могут иметь большую дисперсию, поэтому в статье [16] предлагается j раз по-разному делить набор данных на k частей и брать на каждом шаге среднее k чисел. В данном исследовании были взяты $j = 4$ и $k = 5$.

2.4.2. F-мера

Данная метрика применяется для оценки результатов работы классификатора и позволяет учесть как точность (P, precision), так и полноту (R, recall) в рамках одного значения [2]. Она является гармоническим средним точности и полноты:

$$P = \frac{tp}{tp + fp}, \quad R = \frac{tp}{tp + fn}, \quad F = \frac{2PR}{P + R}.$$

Следует отметить, что эта метрика никак не учитывает tn-элементы, что может привести к неадекватной оценке при несбалансированных классах. Но в нашем наборе данных размеры классов совпадают, поэтому такой ситуации не возникнет.

2.4.3. Accuracy

Эта метрика также применяется для оценки результатов работы классификатора. Она позволяет определить долю правильно классифицированных объектов [2] и считается следующим образом:

$$A = \frac{tp + tn}{tp + fp + fn + tn}.$$

Accuracy является довольно естественным подходом к оценке классификатора, однако она также подвержена влиянию несбалансированности классов.

2.4.4. ROC AUC

Подобно F-мере и Accuracy, ROC AUC является способом определения качества классификатора. ROC-кривая (receiver operating characteristic) — это график, в котором по оси x откладывается доля неправильно классифицированных элементов отрицательного класса (FRP, false positive rate), а по оси y — доля правильно классифицированных элементов положительного класса (TPR, true positive rate, полнота) [8]:

$$TPR = \frac{tp}{tp + fn}, \quad FPR = \frac{fn}{tn + fp}.$$

Каждой точке этого графика соответствует некоторый классификатор. Чем ближе точка к левому верхнему углу, который соответствует идеальному классификатору с $FPR = 0$ и $TPR = 1$, тем классификатор лучше. В нашем случае разные точки порождаются разными порогами, образуя кривую при переборе порогов от 0 до 1.

Для сравнения нескольких ROC-кривых используется AUC (area under curve, площадь под графиком). Эта численная характеристика эквивалентна вероятности того, что классификатор даст случайному положительному элементу большую оценку (в нашем случае оценка — это мера сходства), чем случайному отрицательному.

ROC-кривая не зависит от несбалансированности классов, в отличие

от F-меры и Accuracy.

2.4.5. Методы оценки быстродействия

Для оценки быстродействия алгоритмов можно использовать среднее время работы и дисперсию, однако эти оценки сильно подвержены влиянию выбросов. Поэтому вместо них в данной работе используются квантили, а именно: медиана и 95-й перцентиль [25].

Медиана в нашем случае показывает, сколько времени требуется в среднем на обработку одной пары JavaDoc-комментариев.

Так как длина комментариев в собранном наборе данных сильно варьируется, также полезно знать время обработки крупных комментариев, для чего и используется 95-й перцентиль, отсекающий слишком крупные выбросы. Выбор именно этой квантили обусловлен результатами проведённых экспериментов: с помощью оценки межквартильного размаха было определено, что выбросы в среднем составляют 5% самых длинных комментариев.

2.5. Вопросы экспериментального исследования

Для исследования выбранных алгоритмов и процессов обработки были сформулированы следующие вопросы.

RQ1: Какая конфигурация процесса обработки является лучшей для каждого алгоритма? Т.е. для каждого из исследуемых алгоритмов определяет наилучшую конфигурацию процесса обработки по трём метрикам, описанным в разделе 2.4.

RQ2: Какой алгоритм является наилучшим? RQ2 ранжирует алгоритмы, взятые с полученными в RQ1 конфигурациями, по той же схеме.

RQ3: Насколько хорошо алгоритмы работают по времени? Здесь речь идет об оценке быстродействия алгоритмов с разными процессами обработки. Так как задача обнаружения неточных повторов может возникнуть при разработке инструментов, работающих в реальном времени, менее точный, но более быстрый алгоритм может лучше подойти

для её решения.

RQ4: Как полученные результаты соотносятся с результатами статьи Soto и др.? Здесь предполагается сравнить результаты RQ2 с результатами статьи [28] для совпадающего подмножества исследуемых алгоритмов (LCS, COS, LSH). Это важно в виду того, что в статье рассматривается очень близкая к нашей задача с применением тех же алгоритмов, поэтому хотелось бы проверить, как они себя покажут в ситуации с JavaDoc-документацией. Так как наборы данных отличаются, а также используются разные методы оценки, результаты могут не совпасть.

3. Результаты эксперимента

3.1. Лучшая конфигурация процесса обработки

Результаты оценки классификации с помощью разных алгоритмов и процессов обработки представлены на рис. 3 в виде тепловой карты. Можно заметить, что все три метрики породили схожее ранжирование.

Нормализация не дала хороших результатов. Полная нормализация оказала негативное действие на качество работы алгоритмов, частичная дала незначительный прирост только для LCS и COS без сегментации.

Сегментация улучшила качество классификации в половине случаев. Можно наблюдать стабильное улучшение показателей у COS и LEV. У LSH имеется незначительный прирост по F-мере и Accuracy и ухудшение по ROC AUC, поэтому в целом сегментация не нужна для этого алгоритма. А на LCS сегментация оказала вовсе негативное воздействие.

В результате LCS и LSH лучше использовать без добавления процесса обработки, а COS и LEV — с сегментацией и без нормализации. В целом алгоритмы в исходном виде уже работают с высокой точностью, поэтому влияние процессов обработки на них незначительно.

3.2. Лучший алгоритм

Лучшим алгоритмом оказался LCS без обработки, он лидирует по всем трём метрикам.

Немного хуже качество у LEV с сегментацией, ещё хуже у COS. LSH без обработки показал наихудшие результаты по F-мере и Accuracy, но обогнал COS по ROC AUC, поэтому нельзя с уверенностью сказать, какой из этих алгоритмов менее точен.

3.3. Оценка быстродействия

Для сравнения быстродействия алгоритмов было измерено время работы на каждом элементе набора данных, а затем выделены медиана и 95-й перцентиль. Сводные тепловые карты приведены на рис. 4.

Pipelines \ Алгоритмы	LCS	COS	LEV	LSH
Только алгоритм	96.77%±0.06%	92.85%±0.12%	92.12%±0.05%	92.02%±0.58%
Частичная нормализация	96.77%±0.09%	93.24%±0.03%	88.36%±0.14%	91.52%±0.01%
Полная нормализация	95.88%±0.04%	92.87%±0.04%	89.05%±0.11%	84.86%±0.02%
Сегментация	96.08%±0.09%	95.18%±0.05%	96.50%±0.06%	92.25%±0.09%
Сегм., частичная норм.	96.19%±0.09%	94.95%±0.11%	95.51%±0.08%	91.44%±0.07%
Сегм., полная норм.	96.04%±0.06%	94.95%±0.06%	95.77%±0.08%	83.65%±0.05%

(a) F-мера

Pipelines \ Алгоритмы	LCS	COS	LEV	LSH
Только алгоритм	96.78%±0.11%	92.67%±0.10%	92.29%±0.11%	92.23%±0.44%
Частичная нормализация	96.77%±0.06%	93.12%±0.06%	88.84%±0.12%	92.06%±0.19%
Полная нормализация	95.84%±0.03%	92.82%±0.06%	89.38%±0.11%	86.58%±0.00%
Сегментация	96.00%±0.06%	95.06%±0.06%	96.52%±0.03%	92.42%±0.10%
Сегм., частичная норм.	96.11%±0.11%	94.71%±0.20%	95.44%±0.03%	91.68%±0.04%
Сегм., полная норм.	95.97%±0.03%	94.91%±0.10%	95.63%±0.06%	84.90%±0.03%

(b) Accuracy

Pipelines \ Алгоритмы	LCS	COS	LEV	LSH
Только алгоритм	0.99063	0.98317	0.96362	0.98642
Частичная нормализация	0.99068	0.98451	0.94153	0.98188
Полная нормализация	0.98824	0.97978	0.93498	0.96567
Сегментация	0.98775	0.98561	0.98770	0.97450
Сегм., частичная норм.	0.98814	0.98526	0.98415	0.97732
Сегм., полная норм.	0.98553	0.98283	0.98409	0.95148

(c) ROC AUC

Рис. 3: Оценка качества алгоритмов

Pipelines \ Алгоритмы	LCS	COS	LEV	LSH
Только алгоритм	0.107	0.120	0.385	3.524
Частичная нормализация	4.089	4.120	4.378	7.630
Полная нормализация	4.675	4.526	5.064	7.770
Сегментация	4.296	4.478	4.664	12.589
Сегм., частичная норм.	8.083	8.218	8.701	17.966
Сегм., полная норм.	8.892	8.912	9.413	18.697

(a) Медиана

Pipelines \ Алгоритмы	LCS	COS	LEV	LSH
Только алгоритм	0.259	0.199	1.863	5.467
Частичная нормализация	9.610	9.631	10.591	14.638
Полная нормализация	10.474	10.084	11.138	13.829
Сегментация	6.170	6.355	7.744	36.986
Сегм., частичная норм.	13.208	13.472	15.517	42.175
Сегм., полная норм.	14.378	15.062	16.564	43.988

(b) 95-й перцентиль

Рис. 4: Квантили времени работы алгоритмов в миллисекундах

Среди алгоритмов в исходном виде самым быстрым оказался COS. Примерно с той же скоростью работает LCS, немного медленнее LEV. Самым медленным алгоритмом стал LSH.

Процессы обработки на порядок замедляют работу алгоритмов. При этом, как было выяснено ранее, их положительное влияние на качество классификации незначительно: нормализация не даёт хороших результатов, а сегментация даёт прирост только COS и LEV, и то в среднем всего на 3% по F-мере и Accuracy. Поэтому можно утверждать, что ущерб производительности от процессов обработки сводит на нет их преимущества.

3.4. Сравнение с результатами статьи Soto и др.

Результаты оценки качества классификации с помощью алгоритмов LCS, COS и LSH совпали с результатами статьи [28]: и у нас, и в статье лучшим алгоритмом оказался LCS, а худшим — LSH.

Однако при этом в нашем случае алгоритм LSH не оказался самым быстрым, как это было в [28]. Это связано с тем, что DITA-топики намного длиннее JavaDoc-комментариев, поэтому снижение размерности данных с помощью LSH даёт большой прирост производительности в дальнейших вычислениях, тогда как в нашем случае существенными оказываются затраты на само применение LSH.

3.5. Дискуссия и выводы

Из ответов на четыре вопроса экспериментального исследования можно сделать несколько выводов.

Лучшим алгоритмом для задачи поиска неточных повторов в JavaDoc-документации является LCS. При этом другие рассмотренные алгоритмы ненамного уступают ему в качестве.

Алгоритмы сравнения текста лучше всего использовать в их исходном виде, так как дополнительные шаги приносят мало пользы, но занимают много времени. Слабая эффективность процессов обработки имеет несколько причин. Нормализация удаляет стоп-слова, однако в паре JavaDoc-комментариев зачастую именно они и составляют общую часть, определяющую неточный повтор. Сегментация нужна в первую очередь для того, чтобы учитывать переставленные местами теги, но такая ситуация возникает достаточно редко: в подавляющем большинстве случаев в неточном повторе будет такой же порядок тегов, и тогда с сопоставлением общих фрагментов может справиться и исходный алгоритм (например, LCS).

Рассмотренные алгоритмы ведут себя примерно одинаково на JavaDoc- и DITA-документации с позиции точности, но их быстродействие может зависеть от типа обрабатываемых данных.

Заключение

В ходе данной работы были получены следующие результаты.

- Изучена предметная область.
  - Проблема поиска неточных повторов в документации ПО, подход Duplicate Finder.
- Выбраны следующие алгоритмы для исследования: Longest Common Subsequence (LCS), Cosine Similarity (COS), Levenshtein Distance (LEV), Locality-Sensitive Hashing (LSH). Для них были выбраны следующие готовые реализации: difflib (LCS, LEV), SciPy (COS), datasketch (LSH).
- Разработана и реализована инфраструктура эксперимента.
  - Собран набор из 2638 пар JavaDoc-комментариев из известных open source Java-проектов GSON, JUnit4, Mockito, Slf4J.
  - Создан процесс обработки из нормализации (лемматизация, удаление стоп-слов) и сегментации (потеговое сравнение комментариев с использованием линейного программирования).
  - Выполнена классификация пар с помощью логистической регрессии. Результаты оценены с помощью следующих метрик: F-мера, Accuracy, ROC AUC.
  - Выполнена оценка быстродействия алгоритмов.
  - Инфраструктура реализована на языке Python с использованием библиотек NLTK, SciPy, scikit-learn.
- Выполнен эксперимент и проанализированы результаты.
  - Лучшим алгоритмом оказался LCS.
  - Нормализация и сегментация не дали хороших результатов, но более чем на порядок увеличили время обработки данных.
  - Результаты частично совпали с результатами Soto и др.

Список литературы

[1] Bird Steven, Klein Ewan, Loper Edward. Natural language processing with Python: analyzing text with the natural language toolkit. — O'Reilly Media, Inc., 2009.

[2] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. Introduction to Information Retrieval. — Cambridge University Press, 2008.

[3] Darwin Information Typing Architecture (DITA) Version 1.2. OASIS Standard. — `http://docs.oasis-open.org/dita/v1.2/os/spec/DITA1.2-spec.html`.

[4] Documentation Reuse: Hot or Not? An Empirical Study / Mohamed A. Oumaziz, Alan Charpentier, Jean-Rémy Falleri, Xavier Blanc // Mastering Scale and Complexity in Software Reuse / Ed. by Goetz Botterweck, Claudia Werner. — Cham : Springer International Publishing, 2017. — P. 12–27.

[5] Doxygen. — `http://www.doxygen.nl/`.

[6] Duplicate Finder. — `https://docline.github.io/index.ru`.

[7] Efficient Estimation of Word Representations in Vector Space / Tomas Mikolov, Greg Corrado, Kai Chen, Jeffrey Dean // Proceedings of the International Conference on Learning Representations (ICLR 2013). — 2013. — P. 1–12.

[8] Fawcett Tom. An introduction to ROC analysis // Pattern recognition letters. — 2006. — Vol. 27, no. 8. — P. 861–874.

[9] GSON. — `https://github.com/google/gson`.

[10] Gionis Aristides, Indyk Piotr, Motwani Rajeev. Similarity Search in High Dimensions via Hashing // Proceedings of the 25th International Conference on Very Large Data Bases. — VLDB '99. — Morgan Kaufmann Publishers Inc., 1999. — P. 518–529.

[11] Islam Aminul, Milios Evangelos, Keselj Vlado. Text Similarity Using Google Tri-grams // Advances in Artificial Intelligence. — Vol. 7310 of Lecture Notes in Computer Science. — Springer Berlin Heidelberg, 2012. — P. 312–317.

[12] JUnit4. — https://junit.org/junit4/.

[13] Guidance on Markdown: Design Philosophies, Stability Strategies, and Select Registrations : RFC : 7764 ; Executor: S. Leonard : 2016. — Access mode: https://www.rfc-editor.org/info/rfc7764.

[14] Miller George A. WordNet: An electronic lexical database. — MIT press, 1998.

[15] Mockito. — https://site.mockito.org/.

[16] Moss Henry B, Leslie David S, Rayson Paul. Using JK fold Cross Validation to Reduce Variance When Tuning NLP Models // arXiv preprint arXiv:1806.07139. — 2018.

[17] Navarro Gonzalo. A guided tour to approximate string matching // ACM computing surveys (CSUR). — 2001. — Vol. 33, no. 1. — P. 31–88.

[18] Ng Andrew. CS229 Lecture notes. — P. 16–19.

[19] Nosál' M., Porubän J. Preliminary report on empirical study of repeated fragments in internal documentation // 2016 Federated Conference on Computer Science and Information Systems (FedCSIS). — 2016. — P. 1573–1576.

[20] Oracle. How to Write Doc Comments for the Javadoc Tool. — https://www.oracle.com/technetwork/java/javase/index-137868.html.

[21] Plummer Shawn M, Fox Laurie J. A wiki: One tool for communication, collaboration, and collection of documentation // Proceedings of

the 37th annual ACM SIGUCCS fall conference: communication and collaboration. — 2009. — P. 271–274.

[22] Pydoc. — https://docs.python.org/3/library/pydoc.html.

[23] The Python Standard Library: difflib. — https://docs.python.org/3/library/difflib.html.

[24] Rajaraman Anand, Ullman Jeffrey David. Mining of Massive Datasets. — Cambridge University Press, 2011.

[25] Rice John A. Mathematical statistics and data analysis. — Cengage Learning, 2006.

[26] SciPy. — https://www.scipy.org/.

[27] Sentence Similarity Using Syntactic and Semantic Features for Multi-document Summarization / M. Anjaneyulu, S. S. V. N. Sarma, P. Vijaya Pal Reddy et al. // International Conference on Innovative Computing and Communications / Ed. by Siddhartha Bhattacharyya, Aboul Ella Hassanien, Deepak Gupta et al. — Singapore : Springer Singapore, 2019. — P. 471–485.

[28] Similarity-Based Support for Text Reuse in Technical Writing / Axel J. Soto, Abidalrahman Mohammad, Andrew Albert et al. // Proceedings of the 2015 ACM Symposium on Document Engineering. — DocEng '15. — ACM, 2015. — P. 97–106.

[29] Slf4J. — http://www.slf4j.org/.

[30] datasketch: Big Data Looks Small. — http://ekzhu.com/datasketch/index.html.

[31] scikit-learn: machine learning in Python. — https://scikit-learn.org/.

[32] Задача поиска нечётких повторов при организации повторного использования документации / Д.В. Луцив, Д.В. Кознов,

Х.А. Басит, А.Н. Терехов // Программирование. — 2016. — Vol. 4. — P. 39–49.

[33] Интерактивный поиск неточных повторов в документации программного обеспечения / Д.В. Луцив, Д.В. Кознов, А.А. Шелиховский et al. // Программирование. — 2019. — Vol. 6. — P. 55–66.

[34] Луцив Д. В. Поиск неточных повторов в документации программного обеспечения : дис. — Диссертация на соискание научной степени кандидата физико-математических наук. — Санкт-Петербургский государственный университет. — 2018.

[35] Обнаружение неточных повторов в документации программного обеспечения / Д.В. Луцив, Д.В. Кознов, Г.А. Чернышев et al. // Программирование. — 2018. — Vol. 5. — P. 57–67.