Универсальный механизм первичного поиска повторов в тексте для пакета Duplicate Finder

Глазырин Антон Георгиевич, 21.М07-мм

Научный руководитель: доц. каф. СП, к.ф-м.н. Луцив Д. В.

Рецензент: ген. дир. ООО «Embox» Бондарев А. В.

СПбГУ

6 июня 2023 г.

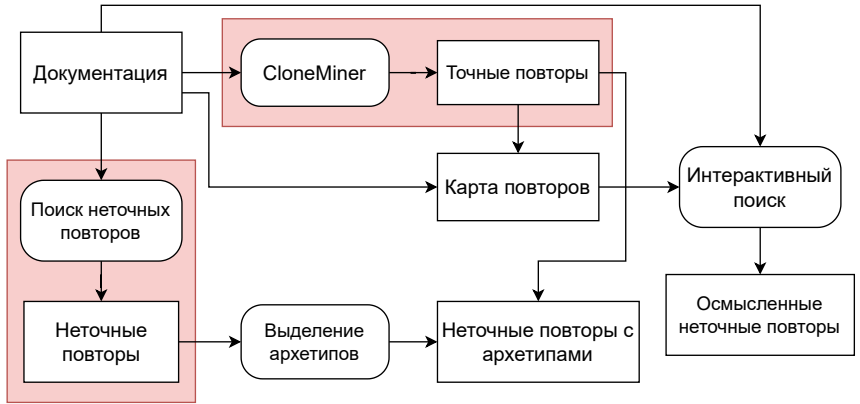
Добрый день, меня зовут Глазырин Антон, и тема моей работы — универсальный механизм первичного поиска повторов в тексте для пакета Duplicate Finder.

Мотивация

- E. Juergens, J. Porubän: ≈10% документации — дублированные фрагменты
- Негативное влияние повторов:
  - Раздувание объема
  - Усложнение модификации
- Управление повторами — улучшение документации

Несколько слов про повторы в документации. Существует ряд исследований, которые показывают, что в среднем достаточно значительная часть документации крупных проектов — около 10% — представляет собой дублированные фрагменты. Хотя повторы в документации сами по себе не являются чем-то плохим, они могут приводить к некоторым негативным последствиям, таким как раздувание объема документации и усложнение внесения изменений, что затрудняет сопровождение. Кроме того, существуют различные методы по улучшению документации, основанные на управлении повторами.

Мотивация: Duplicate Finder



Один из таких методов лежит в основе пакета Duplicate Finder. Этот инструмент является университетской разработкой и предоставляет функционал для улучшения документации за счет поиска повторов. Из-за того, что данный проект разрабатывался и дополнялся в течение длительного промежутка времени многими разными людьми, у него появился ряд проблем с компонентами поиска. Дело в том, что в Duplicate Finder нет своего механизма поиска повторов: для этого используются набор внешних инструментов. Однако, эти инструменты написаны на разных языках, больше не поддерживаются, а самый важный из них — CloneMiner — является закрытой разработкой.

Постановка задачи

Цель — разработка унифицированной подсистемы поиска точных и неточных повторов для Duplicate Finder Toolkit.

- Анализ предметной области
- Определение проблем поиска повторов в DuplicateFinder и требований к новому механизму
- Проектирование конвейера механизма поиска повторов
- Разработка алгоритмов точного и неточного поиска
- Реализация инструмента и его интеграция в DuplicateFinder
- Проведение тестирования разработанного инструмента

Таким образом, целью данной работы является разработка унифицированной подсистемы поиска точных и неточных повторов для Duplicate Finder Toolkit. Для достижения данной цели были поставлены следующие задачи: ...

Существующие решения

- Поиск клонов в ПО:
  - CCFinder
  - CloneMiner
  - Klocwork inSight
  - cpdetector
- Сравнение текстовых документов:
  - Align
  - TxtAlign
- Поиск по образцу:
  - Duplicate Defect Detection
  - Apache Lucene
  - FactorLCS

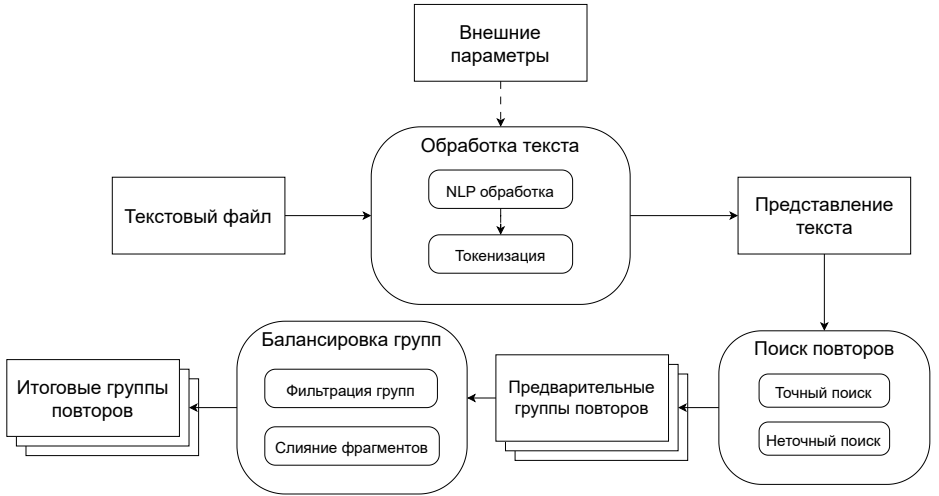
Поиск повторов очень широко распространен в различных сферах, и часто инструменты достаточно сильно заточны под свою конкретную задачу. Среди областей применения наиболее выделяются следующие. Это поиск клонов в исходном коде ПО: такие инструменты как CCFinder или CloneMiner, который как раз и используется в Duplicate Finder, сравнение текстовых документов: самый яркий пример — это проверка на плагиат: такие инструменты как Align и TxtAlign, и поиск похожих вхождений по образцу: например, библиотека для поиска Apache Lucene.

Определение требований

1. Реализация на языке Python
2. Открытая разработка
3. Поиск точных и неточных повторов
4. Универсальность процесса поиска
5. Наличие API и CLI
6. Возможность настройки

На основе анализа проблем поиска повторов в Duplicate Finder определены следующие требования к новому механизму: инструмент должен быть реализован на языке Python для максимальной совместимости с Duplicate Finder, быть открытой разработкой, и иметь возможности как точного так и неточного поиска повторов, сам процесс поиска должен быть универсальным, необходимо наличие API и CLI и должна быть возможность настройки параметров алгоритмов поиска.

Конвейер поиска повторов



На данном слайде представлена схема разработанного конвейера поиска повторов. Он включает в себя 3 основных этапа: предобработку текста для улучшения качества поиска, непосредственно применение алгоритмов поиска повторов, и затем балансировку полученных групп с целью улучшить качество результатов. Далее рассмотрим каждый этап подробнее.

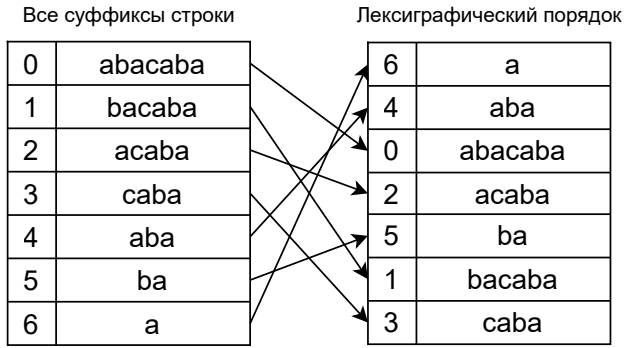
Предобработка текста

Методы NLP:

- Фильтрация спецсимволов
- Удаление стоп слов
- Лемматизация
- Стемминг

Первым шагом является предобработка, которая выполняет 2 задачи: во-первых — преобразование текста в удобный для дальнейшей работы вид, во-вторых — применение ряда методов для облегчения работы алгоритмов поиска. Конкретно — это подходы, широко используемые в NLP — обработке естественного языка. Они включают в себя устранение шума — фильтрацию спецсимволов и удаление стоп слов, а также приведение слов к наиболее унифицированной форме при помощи лемматизаци и стемминга. Кроме того текст токенизируется, и в дальнейшем представляется в виде набора токенов.

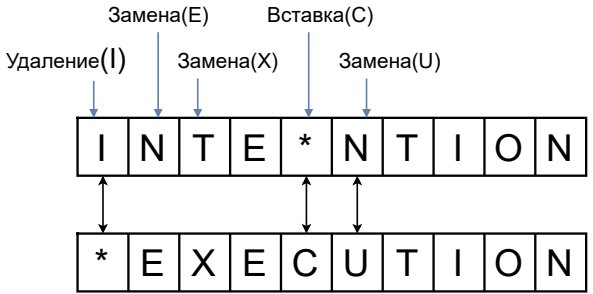
Поиск точных повторов



Далее рассмотрим алгоритмы поиска повторов. Всего были разработаны 3 алгоритма на основе компонент из Duplicate Finder: один для точного поиска и два для неточного. Алгоритм точного поиска основывается на построении суффиксного массива, который представляет собой последовательность суффиксов строки, упорядоченных в лексиграфическом порядке. Суть алгоритма заключается в том, что если в тексте содержатся два одинаковых фрагмента, то суффиксы, начинающиеся с этих фрагментов, будут соседями с суффиксном массиве, таким образом анализируя его можно находить и объединять повторы в группы.

Поиск неточных повторов I

Расстояние Левенштейна:



Первый алгоритм неточного поиска основан на вычисление расстояния Левинштейна между фрагментами — количество операций удаления, замены и вставки нужное, чтобы превратить одну строку в другую. Если расстояние достаточно маленькое — значит фрагменты похожи и являются неточными повторами. Однако вычисление расстояния Левинштейна достаточно трудоемкий процесс, и не имеет смысла проводить этот расчет для сильно различающихся фрагментов.

Поиск неточных повторов I

SimHash:

Hash 1	1	1	0	1	1	0	1	1
Hash 2	1	1	0	0	0	1	1	0
Hash 3	0	1	1	0	1	0	0	1
Result	1	1	0	0	1	0	1	1

Для грубой оценки схожести можно использовать хеширование, в частности — подход SimHash, который позволяет определить хеш коллекции на основе хешей ее составных частей, и затем использовать его для сравнения. Таким образом, фрагменты, имеющие много общих токенов, будут иметь схожие хеши. Суть алгоритма заключается в разбиение текста на фрагменты и попарного их сравнения при помощи этих двух подходов, с последующим объединением найденный пар в группы.

Поиск неточных повторов II

Множества N-грамм:

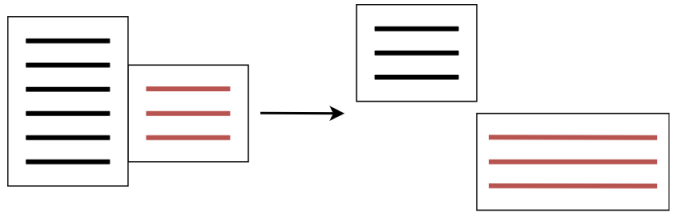
This is Big Data AI Book

Uni-Gram	This	Is	Big	Data	AI	Book
Bi-Gram	This is	Is Big	Big Data	Data AI	AI Book	
Tri-Gram	This is Big	Is Big Data	Big Data AI	Data AI Book		

Второй алгоритм неточного поиска основан на еще одном распространенном подходе для оценки схожести текстов — построении множества N-грам. Имея два таких множества можно вычислить их пересечение, и чем ближе мощность полученного множества к исходному, тем больше эти фрагменты имеют одинаковых текстовых участков. Суть алгоритма заключается в разбиении текста на предложения и объединение их в группы повторов на основе вычисления пересечений их множеств N-грам.

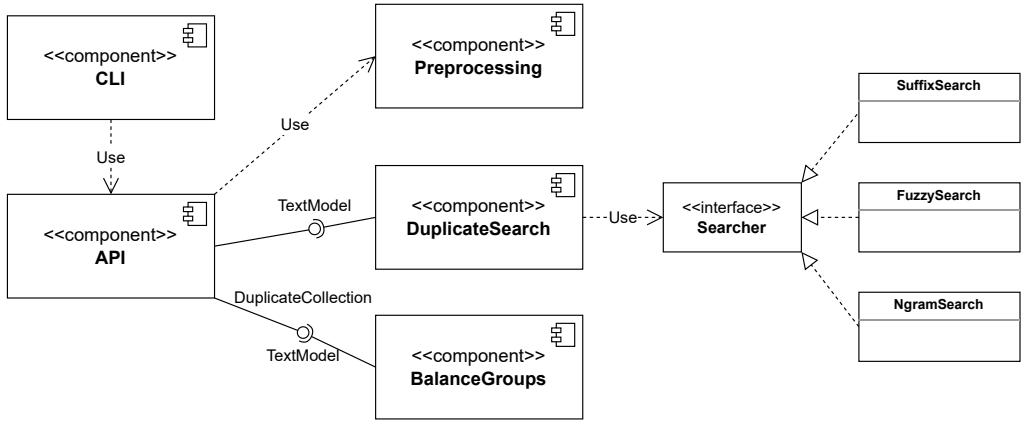
Балансировка групп повторов

- Фильтрация незначимых групп
- Слияние фрагментов



Последним этапом конвейера является балансировка групп повторов. Он включает в себя удаление незначимых групп — которые содержат меньше 2 фрагментов, и слияние фрагментов — если есть группы, все фрагменты которых находятся в исходном тексте рядом с фрагментами другой группы, можно перераспределить эти фрагменты между группами и объединить их, что повышает значимость результатов поиска.

Архитектура



На данном слайде приведена архитектура реализованного инструмента. За каждый этап конвейера отвечает своя компонента, что позволят легко влиять на процесс поиска и добавлять новые алгоритмы.

Тестирование: неточный поиск

Документ	Группы повторов	Средний размер группы	Средняя длина повтора	Покрытие документа
GIMP Manual	574	2,65	13,64	15%
PostgreSQL Manual	464	2,66	17,17	25%
Subversion book	282	2,27	18,93	10%
Zend Framework guide	522	2,32	22,96	16%
Blender Manual	1393	2,48	14,22	16%
Python Requests	23	2,26	20,16	28%

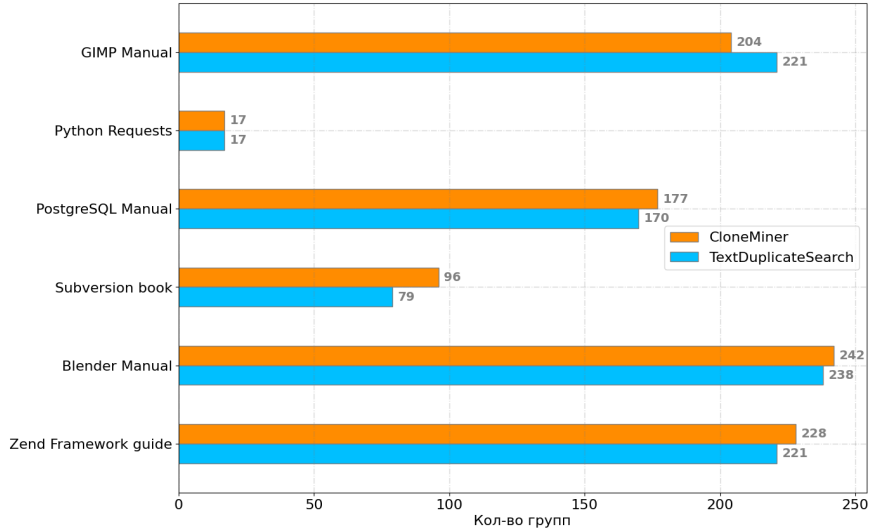
Для тестирования были выбраны документации ряда различных крупных проектов.
Для каждого документа был выполнен поиск точных и неточных повторов.
Результаты приведены на слайде: неточный поиск...

Тестирование: точный поиск

Документ	Группы повторов	Средний размер группы	Средняя длина повтора	Покрытие документа
GIMP Manual	400	2,57	14,59	11%
PostgreSQL Manual	289	2,30	16,31	14%
Subversion book	218	2,17	17,27	7%
Zend Framework guide	557	2,44	16,58	13%
Blender Manual	587	2,33	19,14	9%
Python Requests	24	2,58	18,83	31%

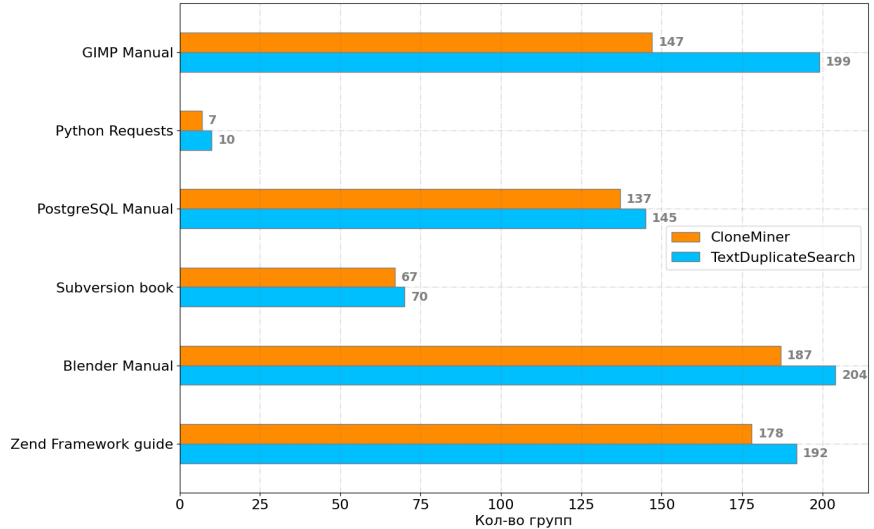
... и точный поиск. Как видно по результатам, инструмент хорошо находит как точные, так и неточные повторы. Кроме того, на практике подтверждается теоретические ожидания: примерно 10% документа составляют дублированные участки. Исключением является Python Requests, которая представляет собой API-документацию и содержит много однообразного кода и описаний. Это также отражает идею, что в некоторых случаях повторы в документации неизбежны и выполняют определенную задачу.

Тестирование: сравнение с CloneMiner



Также была проведена интеграция реализованного инструмента в DuplicateFinder и его сравнение с основным используемым там инструментом CloneMiner. На данном графике показано сколько групп повторов было найдено для документов при использовании каждого инструмента без Duplicate Finder’а.

Тестирование: сравнение с CloneMiner



А на этом — в результате поиска повторов с последующей обработкой DuplicateFinder’ом. Количество групп CloneMiner’а уменьшилось заметно сильнее. Это происходит потому, что CloneMiner создает много групп с пересечениями, которые отфильтровываются при обработке как незначимые.

Тестирование: сравнение с CloneMiner

...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
cid = CaseInsensawitiveDict() cid['Accept'] = 'application/json'
...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
cid = CaseInsensawitiveDict() cid['Accept'] = 'application/json'
...

Например, в случае если есть несколько фрагментов, образующих группу повторов, при этом продолжения некоторых из них также совпадают. Несмотря на их размер, CloneMiner создаст еще одну группу...

Тестирование: сравнение с CloneMiner

...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
cid = CaseInsensawitiveDict() cid['Accept'] = 'application/json'
...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
...
All keys are expected to be strings. The structure remembers the case of the last key to be set, and iter(instance), keys(), items(), iterkeys(), and iteritems() will contain case-sensitive keys. However, querying and contains testing is case insensitive:
cid = CaseInsensawitiveDict() cid['Accept'] = 'application/json'
...

...которая затем будет отфильтрована DuplicateFinder'ом. Разработанный инструмент при этом всегда создает группы без пересечений, и выделит эти продолжения в свою отдельную группу только если они будут достаточного размера.

Заключение

1. Проанализированы основные подходы и средства, которые используются в существующих инструментах для поиска повторов
2. Выявлены требования к новому механизму поиска
3. Спроектирован конвейер для механизма поиска: предобработка текста, применение алгоритмов поиска повторов, балансировка групп повторов
4. Разработаны алгоритмы для точного и неточного поиска повторов на основе использованных в Duplicate Finder инструментов
5. Выполнена реализация инструмента на языке Python с использованием пакета NLTK для предобработки текста, исходный код выложен на GitHub; проведена интеграция с Duplicate Finder
6. Проведено тестирование инструмента на корпусе документов, по результатам работы собрана статистика и проведен ее анализ

Таким образом, были достигнуты следующие результаты.