Санкт-Петербургский государственный университет

Кафедра Системного программирования

Соловьев Александр Александрович

Применение инструментов анализа семантики языков к поиску неточных повторов в документации ПО

Курсовая работа

Научный руководитель:
к. ф.-м. н., ст. преп. Луцив Д. В.

Санкт-Петербург
2018

Оглавление

Введение

Сложность программного обеспечения постоянно растет, как следствие, более сложной и громоздкой становится и его документация. Как правило, в процессе её написания появляется множество практически идентичных описаний, которые различаются между собой лишь незначительной спецификой. Однако, помимо фрагментов, порожденных копированием и последующим их изменением, существуют также и те части, которые были написаны независимо, возможно, с использованием разных языковых конструкций, но при этом являются похожими по смыслу.

Зачастую дублирующиеся фрагменты лишь ухудшают качество документации, поэтому их поиск и последующая обработка являются важной задачей. С учетом того, что в реальной жизни описания могут достигать огромных размеров и содержать мегабайты текста, для её решения требуются автоматизированные инструменты. Так или иначе, проблема решения данной задачи поднимается в различных работах. Во многих из них, например, в [3, 8] исследуется применимость техник поиска повторов в исходном коде к их обнаружению в документации, однако предлагаемые решения позволяют обнаруживать лишь идентичные фрагменты текста. Проблема же поиска неточных повторов затрагивается гораздо реже. Для её решения в рамках [4] был представлен алгоритм на основе N-граммной модели, успешно справляющийся с данной задачей. Однако он всё еще не способен находить дубликаты, в которых используются синонимы.

Таким образом, возникает необходимость в установлении семантической связи между словами. Существуют различные способы для решения данной задачи. В частности, успешно с данной задачей помогает справиться инструмент, разработанный инженерами компании Google, — word2vec [13].

Исходя из этого, была поставлена задача исследовать применимость инструментов анализа семантики языков к алгоритму поиска неточных повторов на основе N-граммной модели на примере word2vec.

1. Постановка задачи

Целью данной работы является изучение применимости инструментов анализа семантики языков к алгоритму поиска неточных повторов на основе N-граммной модели. Для достижения данной цели были поставлены следующие задачи:

- реализация алгоритма поиска неточных повторов с использованием информации о семантических связях в языке;
- оптимизация полученного решения;
- проведение эксперементов для оценки эффективности решения.

2. Обзор

2.1. Алгоритм поиска нечетких повторов на основе N-граммной модели

Под N-граммной моделью предложения понимается множество всех его последовательностей, состоящих из N строго следующих друг за другом слов, также называющихся N-граммами [11]. N-граммные модели очень часто используются в различных задачах по анализу текста. Так, в работе [4] на их основе был представлен алгоритм поиска нечетких повторов.

Первоначально, текст разбивается на предложения, по каждому из которых строится множество его триграмм. Затем постепенно конструируются группы дубликатов: для каждого предложения последовательно подыскивается наиболее подходящая группа, в которую добавляется соотвтествующее предложение и его триграммы. Если таковой не существует, создается новая группа. Группа считается подходящей, если в ней содержится по крайней мере половина триграмм соответствующего предложения. По окончании работы алгоритма, группы, в которых находятся по крайней мере два предложения, считаются содержащими почти дубликаты.

На листинге 1 представлен псевдокод данного алгоритма. В нём использованы следующие обозначения:

- $size(A)$ – функция, возвращающая количество элементов в A;
- $intersect(A, B)$ – функция, возвращающая пересечение множеств A и B;
- $sent$ – список предложений в тексте;
- $groups$ – список групп дубликатов;
- $nGrams$ – множество триграмм предложения/группы;
- $group.sent$ – множество предложений, входящих в группую;

- *overlap* – отношение количества триграммов некоторого предложения, входящих в некоторую группу, к общему количеству триграммов в этом предложении.

Algorithm 1: Алгоритм поиска нечетких повторов на основе N-граммной модели. Заимствован из [4]

```
1  for i = 1 to size(sent) do
2      curSent ← sent_i
3      bestOverlap ← 0
4      bestGroup ← NULL
5      for j = 1 to size(groups) do
6          curGroups ← groups_j
7          curIntersect ← intersect(curSent.nGrams, curGroup.nGrams)
8          curOverlap ← size(curIntersect)/size(curSent.nGrams)
9          if curOverlap > bestOverlap then
10             bestOverlap ← curOverlap
11             bestGroup ← curGroup
12         end if
13     end for
14     if bestOverlap < 0.5 then
15         create new group newGroup
16         newGroup.nGrams += curSent.nGrams
17         newGroup.sent += curSent
18     else
19         bestGroup.nGrams += curSent.nGrams
20         bestGroup.sent += curSent
21     end if
22 end for
23 forall the G in groups such that size(G) ← 1 do
24     groups −= G
25 end forall
```

Данный алгоритм был реализован, как часть Duplicate Finder [5], и на момент написания работы отличался от описанния в статье тем, что позволял обнаруживать неточные повторы не только одиночных предложений, но и целых последовательностей предложений.

2.2. word2vec

Существуют различные подходы для определения синтаксической связи слов в языке. Широко известным является векторное представление слов, при котором каждому слову сопоставляется некоторая точка в векторном пространстве, а семантически близкими считаются те словами, которые расположены ближе всего в пространстве [12]. Зачастую для построения такого сопоставления используются нейронные сети.

Так, в работе [6] представлены два метода обучения для построения векторного представления слов, CBOW (Continuous Bag-of-Words) и skip-grams. Оба подхода основаны на идее того, что близкие по смыслу слов, встречаются в одинаковом окружении. CBOW предугадывает слово, основываясь на контексте, skip-grams же, напротив, – контекст, основываясь на слове. Модели, получаемые при использовании данных подходов, хранят семантическую информацию и позволяют отвечать на вопросы отношения между словами. Например, для нахождения слова A, относящегося к слову B, как C относится к D достаточно предпринять следующие шаги:

1. найти векторные представления x_B, x_C, x_D;
2. вычислить вектор $y = x_C - x_D + x_B$;
3. найти вектор x_w, имеющий наибольшое косинусное сходство с вектором y: $w^* = argmax_w \left(\frac{x_w \cdot y}{\|x_w\| \|y\|} \right)$;
4. слово, которому соответствует данный вектор, и должно быть тем самым словом A из постановки задачи.

Данные методы обучения были реализованы в инструменте word2vec [13], который по большому текстовому корпусу строит векторное пространство с использованием одного из вышеуказанных методов обучения.

3. Реализация

Для того, чтобы решение, полученное в ходе данной работы, учитывало семантическую связь между словами, требовалось модифицировать функцию $intersect$, вызывающуюся в строке 7 листинга 1. Данной функции предстояло искать множество таких триграмм предложения $curSent$, для которых существуют некоторые семантически близкие триграммы в группе $curGroup$. В контексте задачи определить семантическую близость пары триграмм можно различными способами. В данном решении оно задавалось следующим образом. Первоначально определялась семантическая близость для слов. Так, пара слов считалась семантически близкой, если их косинусное сходство было больше некоторого ϵ. Пара триграммов же считалась семантически близкой, если пары слов, находящихся на одинаковых позициях, являлись семантически близкими. В дальнейшем оператор, определяющий семантическую близость слов или же триграмм, будет называться оператором семантического подобия. Стоит заметить, что при сохранении словаря в память производится его нормирование, что позволяет вычислять косинусный коэффициент лишь через скалярное произведение векторов:

$$cos(x, y) = \left(\frac{x \cdot y}{\|x\|\|y\|}\right) = \left(\frac{x \cdot y}{1 * 1}\right) = x \cdot y$$

Поскольку в первой реализации были выявлены серьезные недостатки наивного подхода к определению семантической близости языковых конструкций, в ходе работы предлагались различные оптимизации, позволяющие в некоторой степени сглатить эти недостатки. В данной секции будут описаны первая реализация, а также последовавшие за ней оптимизации, будут детально описаны принимавшиеся решения и причины их принятия. Стоит заметить, что некоторые технические решения были обусловлены тем, что работа велась на языке Python.

3.1. Наивная реализация

Первоначальная версия модифицированного алгоритма отличалась от оригинального алгоритма появлением словаря, предназначенного для хранения векторного представления слов, а также введением функции проверки триграмм на семантическую близость. Необходимость хранения знаний о семантики в оперативной памяти была очевидна, поскольку загрузка соответствующих данных при каждом обращении требовала бы слишком большого времени. Для того, чтобы избегать поиска по структуре, хранящей информацию о семантике, при каждом обращении к ней во время работы алгоритма, последняя была разделена на массив, в котором и хранились векторные представления слов, и на словарь, сопоставляющий словам в индексы вышеупомянутого массива. Одновременно с этим, структура, представляющая триграммы, была расширена, чтобы вместе со словами также хранить и соответствующие им индексы. Данные решения позволили получать векторные представления слов за константное время по индексу.

Оператор семантического подобия был определен выше, однако стоит заметить, что некоторые из слов, для которых также требуется определить семантическую близость могут не содержаться в словаре. Подобная ситуация происходит достаточно часто при работе с документацией к программному обеспечению, поскольку в ней зачастую используются некоторые уникальные для неё наименования, например, какие-либо составные имена вроде ”AviToMp4Converter”, которые являются неизвестым для словаря. В связи с этим было решено задать оператор семантического подобия слов таким образом, чтобы в случае, когда пара слова известна для словаря, проверялось косинусное сходство, в противном же случае происходило посимвольное сравнение.

Также стоит заметить, что для более быстрых операций над векторами, необходимых для вычисления косинуса была использована библиотека NumPy[9].

Несмотря на то, что данная реализация способна учитывать семантику, она не является пригодной для использования в реальных усло-

виях, поскольку время её работы крайне велико, что связано с необходимостью частого пересчета векторного произведения при обращении к оператору семантического подобия. Так, при использовании алгоритма на документации SVN перерасчет происходил чуть более 1.7 миллиарда раз, что приводило к огромному времени работы алгоритма.

3.2. Оптимизации

Поскольку первоначальная реализация была абсолютно непригодна к использованию, был принят ряд решений по её оптимизации. Так как большую часть времени алгоритм проводил, пересчитывая векторные произведения, было решено заранее конструировать матрицу семантики, устанавливавшую семантические отношения для всех возможных пар слов, имеющих векторное представление. Поскольку количество элементов полученной матрицы имеет квадратичную зависимость от количества элементов словаря, было решено использовать структуру bitarray для хранения строк матрицы. Для выбора наиболее эффективной реализации данной структуру проводилось сравнение между библиотеками bitarray [1], bitstring [2] и intbitset [7]. Также было предложено несколько реализаций данной структуры, основанных на встроенных типах Python, в которые входили структуры bytes, bytearray, а также список 64-битных элементов. Для сравнения проводился следующий тест:

1. Первоначально случайным образом генерируется список из N 64-битных элементов, после чего все структуры инициализируются данными значениями;

2. Случайным образом заполняется список индексов доступа длины M, что обеспечивает одинаковые тестовые условия для всех структур;

3. Далее у каждой из структур M раз запрашивается доступ к её элементам, на i запрос запрашивается значение, находящееся в ячейке с индексом M_i;

Структура	Время доступа, сек. $N = 10^5, M = 10^5$	Время доступа, сек. $N = 10^5, M = 10^6$
bitarray	0.0296	0.286
bitstring	0.1724	1.767
intbitset	8.2427	83.02
Структура на bytes	0.0352	0.339
Структура на bytearray	0.0340	0.367
Структура на списке	0.0357	0.364

Таблица 1: Сравнение реализаций bitarray

4. По времени работы, необходимому для M запросов, определялась эффективность конкретной структуры.

Также для сокращения необходимого размера вместо хранения целой матрицы, решено было хранить лишь нижнюю треугольную матрицу, что допустимо, поскольку матрица является симетричной. Последний факт является следствием симметричности оператора семантического подобия.

Получившееся решение уже показывало гораздо более хорошую производительность, новым узким местом решения стал расчет семантической матрицы. Однако данный факт не является проблемой, ведь можно легко заметить, что содержимое матрицы зависит исключительно от векторного представления слов и может быть получено единожды, после чего напрямую использовано в алгоритме. Так конструирование и сохранение необходимых для работы алгоритма данных были вынесено в отдельный скрипт, а сам алгоритм строил словарь по результату работы вышеуказанного скрипта. Данное решение позволило значительно уменьшить общее время работы алгоритма, чуть более подробно об этом будет сказано в разделе 4.2, посвященном производительности полученного решения.

4. Эксперименты

Для проведения экспериментов использовался подход GQM [10]. Были сформулированы следующие вопросы:

1. Как изменилось количество найденных групп, действительно содержащих дубликаты, а также количество ошибок первого рода?
2. Как изменилась производительность алгоритма?

Для исследования были выбраны четыре документа, на которых исследовалась эффективность в работе, посвященной оригинальному алгоритму:

- DocBook 4 Definitive Guide (DocBook);
- Linux Kernel Documentation (LKD);
- Version Control with Subversion(SVN);
- Zend Framework Documentation (Zend).

Также стоит заметить, что в экспериментах использовалось векторное представление для словаря в 71 тысячу слов.

4.1. Эффективность обнаружения дубликатов

Для ответа на первый вопрос вручную проводилось сравнение результатов оригинального алгоритма и полученного решения. Результаты работы алгоритмов вручную распределялись в следующие группы:

- бессмысленные – количество групп, не содержащих какого-либо осмысленного текста, например, разметку;
- ошибки первого рода – количество групп с осмысленным текстом, не содержащих дубликаты;
- группы с дубликатами – количество групп, содержащих дубликаты.

Сравнение значений в последних двух группах для алгоритмов и позвляет ответить на сформулированный вопрос, группа бессмысленных же для этого не требуется, поэтому не представлена в таблице.

Документ	Оригинальный алгоритм	Предложенное решение	Количество новых групп
DocBook	32	33	1
LKD	74	76	2
SVN	182	186	4
ZEND	452	459	7

Таблица 2: Обнаруженные группы с дубликатами

Документ	Оригинальный алгоритм	Предложенное решение	Количество новых групп
DocBook	24	24	0
LKD	41	43	2
SVN	84	92	8
ZEND	239	246	7

Таблица 3: Обнаруженные группы с ошибками первого рода

Данные, представленные в таблицах 2 и 3 позволяют дать ответ на первый вопрос: полученное решение позволяют найти несколько больше групп с дубликатами, однако одновременно с этим будет допущено и большее количество ошибок первого рода.
Так, проблема ошибок первого рода является для полученного решения является гораздо более актуальной. Её разрешение и ослабление требований для оператора семантической близости могут значительно повысить эффективность обнаружения дубликатов данным решением.

4.2. Производительность

Для ответа на второй вопрос было замерено время работы обоих алгоритмов на тестовых данных Результаты представлены в таблице 4. Эксперименты проводились на машине со следующей конфигурацией:

- CPU – Intel Core i7-6700 @ 3.4 GHz x 4
- RAM – DDR4 2666 MHz, 16 Gb
- OS – Ubuntu 16.04 xenial (x64)

Документ	Размер, Кб	Время работы алгоритма 1	Время работы алгоритма 2	Отношение времени работы
DocBook	463	1м. 4с.	20м. 33с.	19.2
LKD	595	50с.	14м. 6с.	16.9
SVN	1101	3м. 56с.	69м. 50с.	17.8
Zend	1646	15м. 13с.	312м. 4с.	20.5

Таблица 4: Сравнение времени работы оригинального алгоритма и полученного в ходе работы решения

Полученные результаты позволяют ответить на второй вопрос: время работы нового алгоритма увеличилось в 16-21 раз. Таким образом, данное решение имеет смысл применять лишь при необходимости очень глубокого анализа документации, а проблема производительности требует дальнейшего исследования.

5. Заключение

В ходе данной работы были достигнуты следующие результаты:

- реализован алгоритм поиска неточных повторов с использованием информации о семантических связях в языке;
- полученное решение было оптимизировано;
- были проведены эксперементы для оценки эффективности решения.

Полученное решение размещено в открытом доступе в репозитории на GitHub.[1]

У данной реализации имеется ряд недостатков, наличие которых не позволяет использовать его в реальных условиях. Так, можно выделить два основных направления развития:

1. Требуется разрешить проблему ошибок первого рода. Данная задача актуальна, как для решения, полученного в данной работе, так и для оригинального алгоритма.
2. Время работы полученной реализации в десятки раз превышает время работы взятого за основу алгоритма, поэтому требуется дальнейшая оптимизация решения.

[1] `https://github.com/Soulflayer1337/N-gram_dup_finder_with_word2vec`

Список литературы

[1] Bitarray. — URL: https://github.com/ilanschnell/bitarray (online; accessed: 28.09.2018).

[2] Bitstring. — URL: https://github.com/scott-griffiths/bitstring (online; accessed: 13.10.2018).

[3] Can clone detection support quality assessments of requirements specifications? / Elmar Jürgens, Florian Deissenboeck, Martin Feilkas et al. — 2010. — 01. — Vol. 2. — P. 79–88.

[4] Discovering Near Duplicate Text in Software Documentation / L.D. Kanteev, Yurii Kostyukov, Dmitry Luciv et al. — 2017. — 01. — Vol. 29. — P. 303–314.

[5] DocLine. — URL: http://www.math.spbu.ru/user/kromanovsky/docline/index_en.html (online; accessed: 27.09.2018).

[6] Efficient Estimation of Word Representations in Vector Space / Tomas Mikolov, Kai Chen, Gregory S. Corrado, Jeffrey Dean // CoRR. — 2013. — Vol. abs/1301.3781.

[7] IntBitSet. — URL: https://github.com/inveniosoftware/intbitset (online; accessed: 13.10.2018).

[8] Nosáľ Milan, Porubän Jaroslav. Preliminary Report on Empirical Study of Repeated Fragments in Internal Documentation // Proceedings of the 2016 Federated Conference on Computer Science and Information Systems / Ed. by M. Ganzha, L. Maciaszek, M. Paprzycki. — Vol. 8 of Annals of Computer Science and Information Systems. — IEEE, 2016. — P. 1573–1576.

[9] NumPy. — URL: http://www.numpy.org/ (online; accessed: 27.09.2018).

[10] R. Basili Victor, Caldiera Gianluigi, Rombach Dieter. The goal question metric approach // Encyclopedia of Software Engineering. — 1994. — 01. — Vol. 1.

[11] Syntactic Clustering of the Web / Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, Geoffrey Zweig // Computer Networks. — 1997. — Vol. 29. — P. 1157–1166.

[12] Turney Peter, Pantel Patrick. From Frequency to Meaning: Vector Space Models of Semantics. — 2010. — 03. — Vol. 37. — P. 141–188.

[13] Word2Vec. — URL: `https://github.com/tmikolov/word2vec` (online; accessed: 23.09.2018).