Обработка естественного языка на Java

Ричард Риз

ДМК ИЗДАТЕЛЬСТВО

УДК 004.438Java
ББК 32.973.2
Р49

Риз Р.
Р49 Обработка естественного языка на Java / пер. с англ. А. В. Снастина. – М.: ДМК Пресс, 2016. – 264 с.: ил.

ISBN 978-5-97060-331-4

Обработка естественного языка (Natural Language Procession — NLP) представляет собой важную область разработки прикладного ПО и, с учетом современных задач ИТ, в будущем эта важность будет только возрастать. Уже сейчас наблюдается рост потребности в приложениях, работающих с естественными языками на основе NLP-методик.

В двнной книге рассматриваются способы организации автоматической обработки текста с применением таких методик, как полнотекстовый поиск, правильное распознавание имен, кластеризация, классификация, извлечение информации и составление аннотаций. Концепции обработки естественного языка излагаются таким образом, что даже читатели, не обладающие знаниями об этой технологии и о методах статистического анализа, смогут понять их.

УДК 004.438Java
ББК 32.973.2

ISBN 978-1-78439-179-9 (анг.)
ISBN 978-5-97060-331-4 (рус.)

Содержание

Глава 2

Поиск фрагментов текста

Процедура поиска фрагментов текста обычно рассматривается с точки зрения разделения текста на отдельные элементы (токены) и в некоторых случаях – дополнительной их обработки. Дополнительная обработка может включать такие операции, как стемминг, лемматизация, удаление *шумовых слов* (*стоп-слов* – *stopwords*), уточнение синонимов и преобразование символов (букв) текста в нижний регистр.

Для начала рассмотрим несколько способов токенизации средствами стандартного пакета Java. Иногда достаточно выполнить относительно простую токенизацию, и в этом случае нет необходимости загружать и подключать NLP-библиотеки. Тем не менее следует помнить, что возможности таких методик ограничены. Затем обсудим специализированные методики, поддерживаемые NLP-библиотеками. Конкретные примеры позволят продемонстрировать практическое их применение и получаемые результаты. А в заключение приведем краткое сравнение рассматриваемых методик.

Существует множество специализированных токенизаторов (tokenizers). Например, в проекте Apache Lucene поддерживаются токенизаторы для разных языков и типов документов. Класс `WikipediaTokenizer` представляет токенизатор для обработки документов в формате Википедии, а класс `ArabicAnalyzer` позволяет работать с текстами на языках арабской группы. К сожалению, все возможные варианты обработки невозможно показать в рамках одной книги.

Также будут продемонстрированы возможности обучения некоторых типов токенизаторов для работы со специализированными текстами. Это необходимо, когда встречается текст в новом формате и/или с новой тематикой. Такой подход позволяет воспользоваться существующим токенизатором, а не писать новый.

Далее будут показаны возможности применения некоторых типов токенизаторов для поддержки таких операций, как стемминг, лемма-

тизация и удаление шумовых слов. В качестве особого случая токенизации рассматривается классификация по частям речи, но эта тема более подробно освящена в главе 5 «Определение частей речи».

Части или фрагменты текста

Существует несколько способов разделения фрагментов текста на категории. Например, основное внимание может быть уделено объектам на уровне символов, таким как знаки пунктуации, с учетом игнорирования или, наоборот, развертывания стяженных форм слов. На уровне отдельных слов возможно выполнение следующих операций:

- определение морфем с помощью стемминга и/или лемматизации;
- развертывание сокращений и аббревиатур;
- выделение числовых элементов.

Правильное разделение слов по знакам пунктуации возможно не всегда, поскольку иногда знаки пунктуации являются частью слова (чаще в европейских языках): can't (английский), s'est (французский), c'è (итальянский) и т. д. Кроме того, может потребоваться объединение нескольких слов в осмысленные словосочетания. Границы предложений тоже могут оказывать определенное воздействие, так как далеко не всегда нужно группировать слова из разных предложений.

В этой главе мы сосредоточимся главным образом на процессе токенизации, а также на нескольких специализированных методиках, таких как стемминг, но не будем стремиться показать, каким образом они используются в других задачах обработки естественного языка. Отложим эту тему до следующих глав.

Что такое токенизация

Токенизация (*tokenization*) – это процесс разделения текста на более простые элементы. Для большинства текстов в качестве элементов рассматриваются отдельные слова. Разделение на элементы, или токены, выполняется с учетом определенного набора так называемых *символов-разделителей* (*delimiters*). Чаще всего разделителями служат пробельные символы. В языке Java пробельные символы определяются с помощью метода `isWhitespace()` из класса `Character`. Краткое описание этих символов см. в табл. 2.1. Иногда возникает необходимость в использовании другого набора разделителей. Например, при

делении текста на абзацы прочие пробельные символы могут стать помехой для выполнения основной задачи.

Таблица 2.1. Обозначение и описание пробельных символов в языке Java

Символ	Описание
Символ пробела в кодировке Unicode	(space_separator, line_separator или paragraph_separator) (пробел, разделитель строк или разделитель абзацев)
`\t`	U+0009 горизонтальная табуляция
`\n`	U+000A перевод строки
`\u000B`	U+000B вертикальная табуляция
`\f`	U+000C перевод страницы
`\r`	U+000D возврат каретки (к началу строки)
`\u001C`	U+001C разделитель файлов
`\u001D`	U+001D разделитель групп
`\u001E`	U+001E разделитель записей
`\u001F`	U+001F разделитель элементов (модулей, блоков)

Сложность процесса токенизации определяется несколькими важными факторами:

- *язык*: в каждом языке есть свои трудности. Пробельные символы часто применяются в качестве разделителей, но при работе с текстом на китайском языке, где пробелы не используются, потребуется другой комплект разделителей;
- *формат текста*: весьма часто текст хранится или предъявляется в разнообразных форматах. Текст в формате HTML или в любом другом формате разметки делает процесс токенизации более сложным, по сравнению с обработкой простого текста;
- *шумовые слова* (*стоп-слова*): часто используемые слова (а также служебные части речи), вероятнее всего, не представляют особой важности для решения некоторых задач обработки естественного языка, например для общих методик поиска. Такие слова называют шумовыми словами, или стоп-словами (*stopwords*). Шумовые слова удаляют, когда они не влияют на выполнение текущей NLP-задачи. К шумовым словам могут относиться неопределенный артикль «a», союз «and» и местоимение «she», в русском языке – союзы «но», «а», «даже», наречие «опять» и т. п.;
- *развертывание некоторых элементов*: в некоторых случаях необходимо развернуть содержание сокращений и аббревиатур,

чтобы при дальнейшей обработке текста получить более точные результаты. Например, если при поиске одним из образцов является слово «machine», то замена аббревиатуры IBM на ее развернутое значение International Business Machines может оказаться полезной;

- *регистр символов*: регистр букв (верхний или нижний), составляющих слово, в некоторых случаях может иметь важное значение, например помогает правильно определять имена собственные. При идентификации частей текста приведение всех букв к одному регистру может оказаться полезным для упрощения процедур поиска;
- *стемминг и лемматизация*: эти процессы изменяют формы слов, приводя их к базовой, начальной форме.

Удаление шумовых слов позволяет формировать более компактные алфавитные и прочие указатели (индексы) и словари, а также ускоряет сам процесс индексации. Но иногда шумовые слова могут иметь значение, поэтому некоторые механизмы поиска их не удаляют. Например, при поиске абсолютно точного совпадения с заданным образцом удаление шумовых слов даст неполный результат. Кроме того, решение задачи распознавания и идентификации именованных объектов часто зависит от шумовых слов, содержащихся в именах. Успех поиска пьесы Шекспира «Ромео и Джульетта» зависит от наличия в образце союза «и», который при обычном поиске рассматривался бы как шумовое, незначимое слово.

В настоящее время составлено множество списков, определяющих шумовые слова. Зачастую классификация слова как шумового напрямую зависит от предметной области задачи. Один из списков шумовых слов представлен на сайте http://www.ranks.nl/stopwords. В нем перечислены несколько категорий шумовых слов английского языка, а также списки шумовых слов для других языков. По адресу http://www.textfixer.com/resources/common-english-words.txt предлагается отформатированный список шумовых слов английского языка, разделенных запятыми.

В табл. 2.2 приводится список из десяти наиболее часто встречающихся шумовых слов, составленный в Стэнфордском университете (http://library.stanford.edu/blogs/digital-library-blog/2011/12/stopwords-searchworks-be-or-not-be).

Таблица 2.2. Десять наиболее часто встречающихся шумовых слов английского языка

Шумовое слово	Частота появлений
the	7578
of	6582
and	4106
in	2298
a	1137
to	1033
for	695
on	685
an	289
with	231

Мы будем рассматривать в основном методики, используемые для токенизации текстов на английском языке, при обработке которых в качестве разделителей применяются пробел и другие пробельные символы.

Синтаксический разбор (парсинг) тесно связан с токенизацией. Оба процесса выполняют идентификацию фрагментов (элементов) текста, но при синтаксическом разборе добавляется еще определение частей речи и их взаимосвязей фрагментов друг с другом.

Использование токенизаторов

Результат токенизации можно использовать для последующего выполнения простых задач, таких как проверка правописания и простые виды поиска. Кроме того, токенизация нужна для решения зависимых от нее задач: определение частей речи, определение границ предложений и классификация. В большинстве последующих глав рассматриваются задачи, требующие предварительной токенизации.

Часто процесс токенизации представляет собой лишь первый этап в последовательности задач. Многоэтапность подразумевает использование конвейеров, и этот подход будет продемонстрирован ниже, в разделе «Использование конвейера». Поэтому токенизаторы должны выдавать качественные результаты для следующего этапа. Если токенизатор плохо выполняет свою работу, это отрицательно сказывается на результатах всех задач в конвейере.

В языке Java предлагается несколько разных токенизаторов и методик токенизации, для поддержки которых предназначен набор базовых классов. Следует обратить внимание, что некоторые из этих

классов устарели, и их применение не рекомендуется. Кроме того, существует ряд библиотек NLP, специализированных для решения как простых, так и сложных задач токенизации. Эти методики описываются в следующих двух разделах. Сначала рассматриваются базовые классы Java, а затем специализированные библиотеки NLP.

Простые токенизаторы языка Java

Ниже перечислены некоторые классы Java, поддерживающие простую токенизацию:

- `Scanner`;
- `String`;
- `BreakIterator`;
- `StreamTokenizer`;
- `StringTokenizer`.

Эти классы предоставляют лишь ограниченную поддержку, тем не менее необходимо хорошо понимать, как их использовать. Для выполнения некоторых задач этих классов вполне достаточно. Зачем применять более трудную для понимания и менее эффективную методику, если базовый класс Java способен сделать ту же работу? Далее мы рассмотрим поддержку процесса токенизации каждым из упомянутых классов.

Классы `StreamTokenizer` и `StringTokenizer` не следует использовать для новых разработок. Часто лучше применять метод `split()` класса `String`. Но знать о классах-токенизаторах необходимо, чтобы не теряться при встрече с ними на практике.

Использование класса Scanner

Класс `Scanner` используется для чтения данных из источника текста, которым может быть стандартное устройство ввода или файл. Класс предлагает простую в применении методику токенизации.

По умолчанию в качестве разделителей принимаются пробельные символы. Экземпляр класса `Scanner` можно создать с помощью множества разных конструкторов. Конструктор в следующем примере принимает простую строку. Метод `next()` извлекает очередной токен из потока ввода. Токены сохраняются в отдельном списке и затем выводятся:

```
Scanner scanner = new Scanner("Let's pause, and then reflect.");
List<String> list = new ArrayList<>();
while(scanner.hasNext()) {
```

```
    String token = scanner.next();
    list.add(token);
}
for(String token : list) {
    System.out.println(token);
}
```

Этот пример выведет следующий результат:

```
Let's
pause,
and
then
reflect.
```

Даже такая простая реализация имеет свои недостатки. Если понадобится идентифицировать и разделить стяженные формы слов, примером которых может служить первый токен, предложенная реализация с этим не справится. Кроме того, последнее слово предложения было возвращено с присоединенной к нему точкой.

Определение разделителя

Если вам не подходит разделитель, принятый по умолчанию, его можно изменить с помощью нескольких методов, перечисленных в табл. 2.3. Краткое описание позволяет получить представление о возможностях этих методов.

Таблица 2.3. Методы определения разделителей в языке Java

Метод	Действие
`useLocale`	Использует локаль для установки соответствующего разделителя по умолчанию
`useDelimiter`	Устанавливает разделитель в соответствии с заданной строкой или шаблоном
`useRadix`	Определяет основание системы счисления, используемой при работе с числами
`skip`	Позволяет пропустить вводимые данные, совпадающие с заданным шаблоном, и игнорировать разделители
`findInLine`	Выполняет поиск следующего совпадения с заданным шаблоном без учета разделителей

Ниже демонстрируется применение метода `useDelimiter()`. Если в примере из предыдущего раздела непосредственно перед циклом `while` вставить следующую инструкцию, в качестве разделителей будут использоваться только пробел, запятая и точка.

```
scanner.useDelimiter("[ ,.]");
```

Измененный пример выведет следующий результат. Пустая строка соответствует найденному разделителю – запятой. Возврат пустой строки в качестве токена является нежелательным побочным эффектом, проявляющимся в данном примере:

```
Let's
pause

and
then
reflect
```

Здесь метод `useDelimiter()` принимает шаблон, заданный в виде строки. Квадратные скобки определяют класс символов. Это регулярное выражение, соответствующее любому одиночному символу из трех заданных. Полное описание использования шаблонов и регулярных выражений в языке Java можно найти в официальной документации: http://docs.oracle.com/javase/8/docs/api/. Список разделителей по умолчанию (пробельные символы) можно восстановить с помощью метода `reset()`.

Использование метода split()

Пример использования метода `split()` из класса `String` рассматривался в главе 1 «Основы обработки естественного языка». Здесь он приведен еще раз для удобства:

```
String text = "Mr. Smith went to 123 Washington avenue.";
String tokens[] = text.split("\\s+");
for(String token : tokens) {
    System.out.println(token);
}
```

Результат:

```
Mr.
Smith
went
to
123
Washington
avenue.
```

Метод `split()` также принимает регулярное выражение в качестве аргумента. Если переменной `text` присвоить строку из примера в предыдущем разделе – «Let's pause, and then reflect.», результат будет точно таким же, как при использовании класса `Scanner`.

Также существует перегруженная версия метода split(), которая принимает дополнительный целочисленный аргумент, определяющий, сколько раз указанный шаблон должен быть применен к обрабатываемому тексту. То есть дополнительный параметр позволяет прекратить токенизацию после нахождения заданного количества совпадений с шаблоном в тексте.

В классе Pattern тоже есть метод split(), который выполняет разделение своего аргумента на токены, используя шаблон, указанный при создании объекта типа Pattern.

Использование класса BreakIterator

Другой подход к процессу токенизации основан на использовании класса BreakIterator, поддерживающего определение целочисленных позиций границ различных элементов текста. В этом разделе демонстрируется его применение для поиска слов.

В классе имеется единственный защищенный (protected) конструктор, выбираемый по умолчанию. Для создания экземпляра класса воспользуемся его статическим методом getWordInstance(). Это перегруженный метод, другая версия которого принимает объект типа Locale. Рассматриваемый класс предлагает несколько методов для получения значений границ элементов (см. табл. 2.4). Кроме того, единственное поле DONE класса используется для хранения значения границы, найденной последней.

Таблица 2.4. Методы класса BreakIterator для определения границ элементов

Метод	Краткое описание
first	Возвращает значение самой первой найденной границы в тексте
next	Возвращает значение границы, следующей за текущей
previous	Возвращает значение границы, предшествующей текущей
setText	Связывает строку текста с экземпляром класса BreakIterator

Чтобы продемонстрировать практическое применение класса BreakIterator, сначала создадим его экземпляр и определим строку текста для обработки:

```
BreakIterator wordIterator = BreakIterator.getWordInstance();
String text = "Let's pause, and then reflect.";
```

Затем строка текста передается созданному экземпляру, и определяется самая первая граница слова в тексте:

```
wordIterator.setText(text);
int boundary = wordIterator.first();
```

Далее следует цикл, в котором определяются начальная и конечная границы каждого найденного слова и сохраняются в переменных `begin` и `end` соответственно. Каждая пара границ и определяемый ими фрагмент текста (слово) выводятся на экран.

После обнаружения самой последней границы цикл завершается:

```
while(boundary != BreakIterator.DONE) {
    int begin = boundary;
    System.out.print(boundary + "-");
    boundary = wordIterator.next();
    int end = boundary;
    if(end == BreakIterator.DONE) break;
    System.out.println(boundary + " ["
    + text.substring(begin, end) + "]");
}
```

Квадратные скобки используются для более четкого выделения элементов текста:

```
0-5 [Let's]
5-6 [ ]
6-11 [pause]
11-12 [,]
12-13 [ ]
13-16 [and]
16-17 [ ]
17-21 [then]
21-22 [ ]
22-29 [reflect]
29-30 [.]
```

Эта методика достаточно хороша для выделения простых токенов.

Использование класса StreamTokenizer

Класс `StreamTokenizer` из пакета `java.io` предназначен для токенизации потока ввода. Он не столь гибок, как класс `StringTokenizer`, рассматриваемый в следующем разделе, поскольку является более «старым» классом. Обычно экземпляр класса `StreamTokenizer` создается на основе файла и обрабатывает текст из этого файла, но можно создать экземпляр и с помощью строки текста.

Класс использует метод `nextToken()` для возврата очередного токена из потока ввода. Токен возвращается в виде целочисленного значения, которое соответствует типу токена. С учетом типа токен может быть обработан различными способами.

Поля класса StreamTokenizer перечислены в табл. 2.5.

Таблица 2.5. Типы и краткое описание полей класса StreamTokenizer

Поле	Тип данных	Краткое описание
nval	double	Содержит числовое значение, если текущий токен является числом
sval	String	Если текущий токен является словом, то содержит этот токен
TT_EOF	static int	Константа для определения конца потока ввода
TT_EOL	static int	Константа для определения конца строки
TT_NUMBER	static int	Константа, обозначающая токен-число
TT_WORD	static int	Константа, обозначающая токен-слово
ttype	int	Тип прочитанного токена

В приведенном ниже примере сначала создается экземпляр токенизатора, затем объявляется переменная isEOF, используемая для завершения цикла. Метод nextToken() возвращает тип токена. Исходя из типа, числовые и строковые токены выводятся по-разному:

```
try {
    StreamTokenizer tokenizer = new StreamTokenizer(
          newStringReader("Let's pause, and then reflect."));
    boolean isEOF = false;
    while(!isEOF) {
        int token = tokenizer.nextToken();
        switch(token) {
            case StreamTokenizer.TT_EOF:
                isEOF = true;
                break;
            case StreamTokenizer.TT_EOL:
                break;
            case StreamTokenizer.TT_WORD:
                System.out.println(tokenizer.sval);
                break;
            case StreamTokenizer.TT_NUMBER:
                System.out.println(tokenizer.nval);
                break;
            default:
                System.out.println((char)token);
        }
    }
} catch(IOException ex) {
    // Обработка исключения.
}
```

После выполнения получим следующий результат:

```
Let
'
```

Это совсем не то, что ожидалось. Обработка выполнена неправильно потому, что данный токенизатор использует одиночные и двойные кавычки для обозначения цитат в тексте (кавычки внутри кавычек). Но поскольку парная кавычка, закрывающая цитату, не обнаружена, остаток строки интерпретировался как цитируемый текст и не был обработан.

Метод `ordinaryChar()` позволяет определять символы, которые должны восприниматься как обычные символы в тексте. Одиночную кавычку и запятую можно сделать обычными символами следующим образом:

```
tokenizer.ordinaryChar('\'');
tokenizer.ordinaryChar(',');
```

Если эти инструкции добавить в предыдущий пример и выполнить его, мы получим следующий результат:

```
Let
'
s
pause
,
and
then
reflect.
```

Теперь апостроф не влияет на правильность обработки. Два знака пунктуации, объявленные обычными символами, интерпретируются как разделители слов и возвращаются как токены. Кроме того, токенизатор предоставляет метод `whitespaceChars()`, определяющий символы, которые при обработке будут интерпретироваться как пробельные.

Использование класса StringTokenizer

Класс `StringTokenizer` находится в пакете `java.util`. Он более гибок, чем класс `StreamTokenizer`, и предназначен для обработки строк из любых источников. Конструктор класса принимает обрабатываемую строку в качестве параметра и использует метод `nextToken()` для возврата токена. Метод `hasMoreTokens()` возвращает `true`, если в потоке

ввода есть еще токены. Следующий пример демонстрирует процесс обработки:

```
StringTokenizer st = new StringTokenizer(
    "Let's pause, and then reflect.");
while(st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Этот пример выводит следующее:

```
Let's
pause,
and
then
reflect.
```

Конструктор имеет перегруженные версии, позволяющие переопределять разделители и указывать, должны ли разделители возвращаться как токены.

Проблемы производительности при выполнении токенизации штатными средствами Java

При использовании методик токенизации, реализованных только базовыми средствами языка Java, следует уделить немного внимания эффективности их применения. Иногда измерение производительности затруднено различными факторами, влияющими на выполнение кода. Неплохое сравнение методик токенизации, реализованных штатными средствами Java, размещено на сайте http://stackoverflow.com/questions/5965767/performance-of-stringtokenizer-class-vs-split-method-in-java. При решении задач токенизации метод `indexOf()` показал наилучшие результаты.

Прикладные программные интерфейсы NLP для токенизации

В этом разделе будут продемонстрированы различные методики, использующие библиотеки OpenNLP, Stanford и LingPipe. Мы ограничимся рассмотрением именно этих программных интерфейсов, несмотря на то что существуют и другие бесплатные библиотеки. Изучая приведенные примеры, вы получите более или менее полное представление о доступных методиках.