Ричард Рид

Обработка естественного языка на Java

Richard M Reese

Natural Language Processing with Java

Explore various approaches to organize and extract useful text from unstructured data using Java

[PACKT] PUBLISHING
open source
community experience distilled
BIRMINGHAM – MUMBAI

Ричард Риз

Обработка естественного языка на Java

Исследование разных подходов к организации
и извлечению полезной текстовой информации
из неструктурированных данных
с использованием Java

ДМК Пресс
издательство
Москва, 2016

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

Об авторе

Ричард Риз (Richard M Reese) имеет опыт работы не только в промышленности, но и в науке. В течение 17 лет он работал в отрасли телефонной связи и аэрокосмической индустрии, выполняя разные обязанности, включая исследовательскую и конструкторскую деятельность, разработку программного обеспечения, руководство группами разработчиков и преподавание. В настоящее время Ричард преподает в Тарлтонском государственном университете (Tarleton State University), где применяет свой богатый практический опыт для усовершенствования курсов обучения.

Ричард написал несколько книг по языкам программирования Java и C. Он излагает материал в лаконичном и понятном стиле. Среди его книг можно отметить «EJB 3.1 Cookbook», книги о новых функциональных возможностях Java 7 и 8, книгу о сертификации разработчика на языке Java, книгу о jMonkey Engine, а также книгу об использовании указателей в языке C.

«Я благодарен моей дочери Дженнифер за многочисленные замечания, поправки и дополнения. Ее вклад в создание данной книги невозможно переоценить».

О рецензентах

Сурьяпракаш С. В. (Suryaprakash C. V.) занимается задачами обработки естественного языка с 2009 года. По окончании учебного заведения получил степень бакалавра по физике, затем в аспирантуре специализировался в области разработки компьютерных приложений. Позже получил возможность продолжить карьеру в наиболее привлекательной для него области – обработке естественного языка.

В настоящее время Сурьяпракаш является ведущим исследователем в компании Senseforth Technologies.

> «Я благодарен коллегам за постоянную поддержку во всем. Их помощь была весьма существенной и при составлении данной рецензии».

Эван Демпси (Evan Dempsey) – программист из Уотерфорда (Ирландия). В редкие часы, когда он отвлекается от языка Python (который использует не только для заработка, но и с большим удовольствием), Эван любит посидеть с кружкой знаменитого ирландского пива, заняться программированием на Common Lisp и почитать о новейших достижениях в области машинного обучения (machine learning). Он также участвует в нескольких проектах с открытым исходным кодом.

Анил Оманвар (Anil Omanwar) – весьма энергичная личность с огромным интересом к новейшим направлениям в технологиях и исследованиях. Обладает более чем 8-летним опытом исследовательской работы в области когнитивных (интеллектуальных) вычислений. Обработка естественного языка, машинное обучение, визуальное представление информации и интеллектуальный анализ текстов – главные сферы его исследовательского интереса.

Анил является экспертом в анализе эмоциональной окраски текста (sentiment analysis), составлении и анализе опросных листов, кластеризации текстов (документов) и извлечении информации (связных фраз) в разнообразных сферах знаний, таких как науки о жизни (биология, медицина, антропология и т. п.), производство, розничная торговля, электронная коммерция, представительство и работа с клиентами, банковское дело, социальные медиакоммуникации.

В настоящее время Анил активно сотрудничает с лабораториями обработки естественного языка и IBM Watson в корпорации IBM. Тема его исследований – автоматизация важнейших ручных операций и помощь экспертов в соответствующих областях знаний для оптимизации функциональных возможностей систем «человек–машина».

Свое свободное время Анил посвящает общественной работе, пешеходному туризму, фотографии и путешествиям. Он всегда готов заняться решением любой, самой сложной технической задачи.

Амитабх Шарма (Amitabh Sharma) – профессиональный программист. В его активе множество промышленных приложений в сферах телекоммуникации и бизнес-аналитики. Профессиональные интересы Амитабха – сервис-ориентированные архитектуры, хранение данных и такие языки программирования, как Java, Python и др.

Предисловие

Обработка текстов на естественных языках (Natural Language Processing, NLP) используется для решения обширного класса задач, включая поддержку механизмов поиска, аннотирования и классификации текстов на веб-страницах, а также для внедрения методов машинного обучения с целью решения таких нетривиальных задач, как распознавание речи и анализ запросов. Технология обработки естественного языка наиболее эффективна при работе с документами, содержащими полезную информацию.

Обработка текстов на естественных языках применяется для расширения функциональных возможностей приложений, например для упрощения ввода пользователем исходных данных и преобразования текста в более удобные формы. Суть технологии NLP состоит в обработке текстов на естественном языке, взятых из самых разнообразных источников. При этом используется последовательность ключевых операций обработки естественного языка для преобразования исходного текста или извлечения из него полезной информации.

В этой книге подробно рассматриваются ключевые операции NLP, которые с большой вероятностью можно встретить в NLP-приложениях. Для каждой операции, рассматриваемой в книге, сначала дается описание задачи и области ее применения. Далее перечисляются все нюансы, определяющие сложность задачи, чтобы читатель смог лучше понять данную задачу в целом. Затем следуют многочисленные примеры решений на языке Java, а также примеры использования прикладных программных интерфейсов (API) поддержки обработки естественного языка.

Краткий обзор содержания книги

Глава 1 «Основы обработки естественного языка» описывает области применения обработки естественного языка и важное значение этой технологии. Методики NLP, используемые в этой главе, объясняются на простых практических примерах.

Глава 2 «Поиск фрагментов текста» в основном посвящена операциям разделения потока текста на смысловые фрагменты (фразы, слова, символы и т. п.)[1]. Это первый этап подготовки к решению более

[1] Такие операции обозначаются термином токенизация (tokenization). – *Прим. перев.*

сложных задач NLP. Также описываются программные интерфейсы Java для разделения текста на фрагменты и поиска по образцам.

Глава 3 «Поиск предложений» демонстрирует важную роль еще одной задачи обработки естественного языка – определение границ предложений. Этот этап обработки предшествует многим другим NLP-задачам, в которых текстовые элементы не должны выходить за границы предложений, включая гарантии вхождения всех фраз в одно предложение, а также частичную поддержку распознавания и анализа речи.

Глава 4 «Поиск людей и именованных объектов» посвящена тому, что в широком смысле обычно обозначают термином «распознавание и идентификация именованных объектов» (named-entity recognition). Это задача идентификации людей, местоположений и прочих подобных объектов в тексте. Данная методика является подготовительным этапом для обработки запросов и операций поиска.

Глава 5 «Определение частей речи» рассказывает, как определяются части речи грамматических элементов текста, такие как существительные и глаголы. Идентификация этих элементов является важным этапом в процессе определения общего смыслового значения исследуемого текста и установления смысловых связей внутри текста.

Глава 6 «Классификация текстов и документов» наглядно показывает необходимость классификации текста для таких задач, как выявление спама и анализ эмоциональной окраски текста (sentiment analysis). Подробно описывает методики NLP, которые обеспечивают поддержку классификации текстов.

Глава 7 «Использование синтаксического анализатора (парсера) для выделения взаимосвязей» рассматривает деревья синтаксического анализа. Дерево синтаксического анализа используется для многих целей, в том числе для извлечения информации. Оно содержит данные об отношениях между своими элементами. Для наглядной демонстрации процесса в главе приведен пример реализации простого запроса.

Глава 8 «Комплексные методики» рассматривает способы извлечения данных из документов различных типов, таких как PDF и файлы, созданные в MS Word. Здесь показано, как объединить NLP-методики, описанные в предыдущих главах, в своеобразный «конвейер» для решения более крупномасштабных задач.

Что нужно для чтения этой книги

Для демонстрации работы методик обработки естественного языка используется Java SDK 7. Потребуются также разнообразные NLP

API, которые нетрудно будет найти и скачать. Наличие интегрированной среды разработки (IDE) не обязательно, но желательно.

Для кого эта книга

Книга будет полезна опытным разработчикам на Java, интересующимся технологиями обработки естественного языка. Для ее чтения не требуется предварительное знакомство с NLP.

Соглашения, принятые в книге

В книге вы обнаружите несколько стилей текста, которые позволяют выделять различные виды информации. Ниже приведены примеры этих стилей и описание их смысла.

Ключевые слова языка программирования, имена классов, переменных, методов и т. п. оформляются моноширинным шрифтом: «Метод `keyset` возвращает набор всех ключей аннотации, в текущий момент содержащихся в объекте `Annotation`».

Имена таблиц баз данных, каталогов, файлов, расширения файлов, новые термины и важные замечания оформлены курсивом: «Для демонстрации применения POI воспользуемся файлом с именем *TestDocument.pdf*».

Блоки программного кода и листинги оформлены моноширинным шрифтом:

```
for (int index = 0; index < sentences.length; index++) {
    String tokens[] = tokenizer.tokenize(sentences[index]);
    Span nameSpans[] = nameFinder.find(tokens);
    for(Span span : nameSpans) {
        list.add("Sentence: " + index
            + " Span: " + span.toString() + " Entity: "
            + tokens[span.getStart()]);
    }
}
```

Моноширинным шрифтом также оформляются ввод пользователя и вывод результатов работы программ на экран:

```
Sentence: 0 Span: [0..1) person Entity: Joe
Sentence: 0 Span: [7..9) person Entity: Fred
Sentence: 2 Span: [0..1) person Entity: Joe
```

Так оформляются предупреждения или важные примечания.

Так оформляются советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и «Packt» очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей кни-

ги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Основы обработки естественного языка

Обработка естественного языка (*Natural Language Processing, NLP*) – это обширная область ИТ, связанная с использованием компьютеров для анализа естественных языков, к которой относятся такие дисциплины, как распознавание и обработка речи, выделение смысловых отношений, категоризация документов, а также реферирование и аннотирование текста. Но все эти виды анализа основаны на относительно небольшой группе базовых методик: разделение потока текста на фрагменты – *токенизация* (*tokenization*), определение границ предложений, классификация и выделение отношений (между элементами текста). Именно эти основополагающие методики и являются главными темами данной книги. Мы начнем с подробного рассмотрения обработки естественного языка в целом, попробуем разобраться, почему она столь важна, и определим области ее применения.

Для поддержки задач обработки естественного языка существует множество доступных инструментальных средств. Мы сосредоточим внимание на использовании языка программирования Java и различных *прикладных программных интерфейсов* (*Application Programming Interface, API*) на языке Java поддержки NLP. В этой главе будут кратко описаны основные программные интерфейсы, включая OpenNLP (Apache), библиотеки Stanford NLP, LingPipe и GATE.

Затем будут обсуждаться основные методики обработки естественного языка, рассматриваемые в данной книге. С помощью одного из прикладных программных интерфейсов будут представлены и продемонстрированы сущность и примеры использования этих методик. Многие из этих методик используют так называемые модели. Модели похожи на наборы правил, применяемых для выполнения, например, такой задачи, как разделение текста на фрагменты, и обычно представлены в виде классов, экземпляры которых хранятся в файлах.

Завершается глава кратким описанием процесса подготовки данных для поддержки задач NLP.

В целом проблему обработки текстов на естественных языках нельзя назвать простой. Несмотря на то что некоторые ее задачи решаются относительно легко, многие другие могут потребовать применения изощренных методик. В этой книге сделана попытка предоставить всю необходимую фундаментальную информацию, чтобы читатель смог лучше понять, какие методики доступны и применимы для каждой конкретно поставленной задачи.

Обработка естественного языка представляет собой весьма обширную и сложную область ИТ. В данной книге рассматривается лишь небольшая ее часть. Мы сосредоточимся на ключевых задачах NLP, которые могут быть реализованы на языке программирования Java. На протяжении всей книги будет демонстрироваться практическое применение методик NLP с использованием Java SE SDK и других библиотек, таких как OpenNLP и Stanford NLP. Для работы с конкретными библиотеками в проект должны быть включены специальные JAR-файлы. Описание библиотек можно найти в разделе «Обзор инструментальных средств обработки естественного языка», где также приведены ссылки на эти библиотеки. Примеры, приводимые в книге, были разработаны с помощью IDE NetBeans 8.0.2. Перед сборкой проектов необходимо добавить требуемые JAR-файлы в категорию **Библиотеки** (Libraries) в диалоговом окне **Свойства проектов** (Projects Properties).

Что такое обработка естественного языка

Формальное определение обработки текстов на естественных языках часто представлено в таком изложении: область исследований, использующая информатику (computer science), искусственный интеллект и понятия формальной лингвистики для анализа естественного языка. Менее формальное определение полагает, что это набор инструментальных средств для извлечения содержательной и полезной информации из источников, сформированных на естественном языке, таких, например, как веб-страницы и текстовые документы.

Характеристики «содержательная» и «полезная» означают, что информация должна обладать некоторой коммерческой ценностью, хотя часто такая информация используется также для решения научных теоретических проблем. Наиболее очевидно это проявляется в поддержке поисковых механизмов. Пользовательский запрос обра-

батывается с применением методик обработки естественного языка, чтобы сгенерировать страницу результатов, наиболее удобную для пользователя. Современные механизмы поиска в этом отношении обладают весьма впечатляющими достижениями. Кроме того, методики обработки естественного языка нашли применение в автоматизированных системах помощи (подсказки) и в поддержке сложных систем запросов, например в известном проекте Watson корпорации IBM.

При работе с любым языком часто встречаются термины *«синтаксис»* (*syntax*) и *«семантика»* (*semantics*). Синтаксис языка определяет правила, управляющие правильной структурой предложения. В английском языке, например, общая структурная схема предложения начинается с подлежащего (существительное), за которым следует сказуемое (глагол) и далее – дополнение: «Tim hit the ball» (Тим ударил по мячу). Изменение порядка слов в предложении в английском языке (и в некоторых других) считается неправильным и не употребляется: «Hit ball Tim»[1]. Несмотря на то что синтаксические правила английского языка не столь строги, как их аналоги для компьютерных языков, мы вправе ожидать, что предложение будет соответствовать основной структурной схеме.

Семантика предложения – это его смысл. Люди, говорящие по-английски, понимают смысл предложения «Tim hit the ball». И все же как английский, так и другие естественные языки иногда могут быть неоднозначными, поэтому смысл предложения можно правильно определить только из его контекста. Далее мы увидим, как использовать различные методики машинного обучения для определения смысла текста.

В процессе чтения вы встретите множество лингвистических терминов, которые помогут лучше понимать естественные языки и сформируют небольшой специализированный словарь, полезный для описания различных методик обработки естественного языка. Вы узнаете, как разделить текст на отдельные элементы и как провести классификацию этих элементов.

Вообще говоря, рассматриваемые методики используются для улучшения приложений, то есть для того, чтобы приложения стали более полезными. Область применения обработки естественного язы-

---

[1] В этом отношении правила русского языка более свободны и зачастую позволяют произвольно менять порядок слов в предложении – например, знаменитые фразы из букваря: «Мама мыла раму» – «Раму мыла мама» и т. п. – *Прим. перев.*

ка охватывает широкий диапазон: от относительно простых случаев до «технологий завтрашнего дня». В книге показаны примеры, демонстрирующие простые методики, которых вполне достаточно для решения некоторых практических задач, а также библиотеки и классы, обладающие более развитыми функциональными свойствами, позволяющими справиться с комплексными сложными проблемами.

Для чего используется обработка естественного языка

Обработка естественного языка используется во многих научных и прикладных дисциплинах для решения разнообразных типов задач. Анализ текста выполняется и при вводе пользователем данных, состоящих из нескольких слов, и при формировании интернет-запроса для поиска многочисленных документов, которые необходимо обработать. В последние годы наблюдается лавинообразный рост объема доступных неструктурированных данных, представленных в таких формах, как блоги, твиты и прочие социальные сети. Технология обработки естественного языка является наиболее подходящим инструментом для анализа этого типа информации.

Машинное обучение и анализ текста часто используются для расширения функциональных возможностей приложений. Вот краткий список областей применения обработки естественного языка:

- *поиск*: идентификация заданных элементов текста. Может быть простым, например поиск всех вхождений заданного имени в документе, или выполняться с использованием синонимов, а также различного и даже неверного (неточного) написания для поиска объектов, более или менее совпадающих с исходной строкой-образцом;
- *машинный перевод*: обычно предполагает перевод с одного естественного языка на другой;
- *аннотирование*, или *реферирование* (*summation*): для абзацев, статей, документов или подшивок документов может потребоваться аннотирование (реферирование). NLP-технология успешно справляется с этой задачей;
- *распознавание и идентификация именованных объектов* (*Named Entity Recognition, NER*): имеется в виду извлечение из текста имен людей, названий географических и прочих объектов. Как правило, эта методика используется в сочетании с другими за-

дачами обработки естественного языка, например с обработкой запросов;

- *группирование информации*: это важный этап обработки – на основе исходных текстовых данных создается набор категорий, отображающих содержание документа. Вам наверняка встречались веб-сайты, где информация достаточно удобно организована по категориям, а список этих категорий размещен в левой части страницы;
- *морфологическая разметка* (*Parts of Speech Tagging, POST*): при решении данной задачи текст разбивается на различные грамматические элементы: существительные, глаголы и т. п. Это удобно для дальнейшего анализа текста;
- *анализ эмоциональной окраски текста* (*sentiment analysis*): с помощью этой методики можно определять, какие чувства люди испытывают и как относятся к фильмам, книгам и многим другим вещам (в том числе и к предлагаемым товарам). С коммерческой точки зрения полезно знать, как покупатели воспринимают тот или иной продукт, применяя автоматизированную методику;
- *ответы на вопросы*: этот тип обработки был наглядно продемонстрирован, когда программа Watson, разработанная корпорацией IBM, выиграла телеигру Jeopardy[1]. Но область применения этой методики вовсе не ограничивается игровыми шоу, она используется во многих областях человеческой деятельности, например в медицине;
- *распознавание речи*: человеческую речь анализировать трудно. Большинство достижений в этой области являются результатами применения технологии обработки естественного языка;
- *генерация текстов на естественном языке* (*Natural Language generation, NLG*): это процесс генерации связного текста из данных или из специализированного источника данных (знаний), например из базы данных. Методика позволяет автоматизировать создание отчетов и информационных сводок, таких как метеорологические бюллетени или результаты медицинских обследований.

В задачах обработки естественного языка часто используются методики машинного обучения. В обобщенном виде суть этого подхода такова: сначала модель обучается выполнению поставленной задачи,

[1] В России аналогичная телеигра называется «Своя игра». – *Прим. перев.*

проверяется корректность выбранной модели, затем эта модель применяется для решения реальной задачи. Более подробно мы рассмотрим этот процесс в разделе «Основы использования моделей в обработке естественного языка» ниже.

Трудности обработки естественного языка

Как отмечалось выше, проблему обработки естественного языка нельзя назвать простой. Трудности возникают по ряду объективных причин. Например, существуют сотни естественных языков, в каждом из которых имеются свои синтаксические правила. Слова могут иметь разный смысл в зависимости от контекста употребления. Ниже мы подробнее остановимся на некоторых наиболее важных факторах, затрудняющих обработку естественного языка.

Даже на уровне отдельных символов встречаются некоторые трудности. Например, всегда следует учитывать кодировку, используемую в конкретном документе. Текст может храниться в различных кодировках: ASCII, UTF-8, UTF-16 или Latin-1. Также необходимо принимать во внимание и другие факторы, например зависимость текста от регистра букв. Особые виды обработки могут потребоваться для знаков пунктуации и для чисел. Иногда приходится отдельно обрабатывать использование значков, отражающих эмоции (комбинации символов или специальные символы), гиперссылок, повторяющихся знаков пунктуации (... или ---), расширений файлов и имен пользователей, содержащих точки. Большинство этих задач решается с помощью предварительной обработки текста, которая будет обсуждаться в разделе «Подготовка данных» ниже.

Под делением текста на фрагменты или элементы обычно подразумевается представление текста в виде последовательности слов. В этом случае слова обозначаются термином «*лексический элемент*», «*лексема*», или просто «*токен*» (*token*), а процесс деления текста – «*токенизация*» (*tokenization*). Этот процесс не вызывает особых затруднений в языках, использующих пробельные символы для разделения слов, но в языках, подобных китайскому, это сделать гораздо труднее, поскольку иероглифы могут обозначать и слоги, и целые слова.

Слова и морфемы могут быть обозначены метками, идентифицирующими соответствующую часть речи. *Морфема* (*morpheme*) – это минимальная значимая часть слова (текста). Примерами морфем являются приставки, корни и суффиксы слов. При работе с отдельными

словами как с элементами текста часто приходится определять синонимы, аббревиатуры, акронимы и правильность написания.

Определение основы слова представляет собой еще одну задачу, требующую практического решения. Термин *стемминг* (*stemming*) обозначает процесс определения основы слова по его различным формам (причем основа не всегда совпадает с корнем слова). Например, для слов «замедлить», «замедленный», «замедляющий» основой является «замедл». Пример из английского языка: слова «walking», «walked», «walks» имеют основу «walk». Поисковые механизмы часто используют стемминг для улучшения качества ответов на запросы.

Со стеммингом тесно связана *лемматизация* (*lemmatization*), процесс определения начальной (основной) формы слова, обозначаемой термином *«лемма»* (*lemma*). Например, для слова «действующий» основой является «действ», но его лемма – «действовать» (в английском языке слово «operating» имеет основу «oper», но лемму «operate»). Лемматизация – это более сложный и тонкий процесс, чем стемминг. Для определения леммы используются словарь форм и морфологические методики. В некоторых случаях это позволяет выполнить более точный анализ текста.

Слова объединяются в словосочетания и предложения. Определение границ предложений тоже может быть связано с некоторыми трудностями, хотя на первый взгляд кажется, что достаточно лишь найти точку, обозначающую конец предложения. Но точки могут встречаться и внутри предложения, например после сокращенных слов г. или н. э., а также при использовании британского формата записи чисел 12.834.

Часто необходимо знать, какие слова в предложении являются существительными, а какие – глаголами. Иногда приходится определять отношения между словами. Например, установление *кореферентности* (*референциального тождества – coreference resolution*) определяет взаимоотношения между конкретными словами в одном или в нескольких предложениях. Рассмотрим следующие предложения:

«Город велик, но прекрасен. Он занимает всю долину».
(«The city is large but beautiful. It fills the entire valley».)

Здесь слово «он» («it») кореферентно (референциально тождественно) слову «город» («city»). Если слово может иметь несколько смысловых значений, для определения его смысла в данном конкретном случае может потребоваться выполнение операции *разрешения лексической многозначности* (*word sense disambiguation, WSD*). Иногда

это связано с определенными трудностями. Например, «Джон вернулся домой» («John went back home»).

Что именно здесь подразумевает слово «дом» («home»): строение, город, какое-либо место или нечто другое? В некоторых случаях смысл слова можно определить из контекста, в котором оно употребляется. Например, «Джон вернулся домой. Дом его находился в конце тупика» («John went back home. It was situated at the end of a culde-sac»).

Несмотря на все перечисленные трудности, технология обработки естественного языка в большинстве случаев способна достаточно уверенно справляться со своими задачами, поэтому весьма полезна во многих областях. Например, по твитам покупателей можно выполнить анализ эмоциональной окраски текста и получить возможность предложить недовольным клиентам варианты компенсации, в том числе какие-либо бесплатные товары. Медицинские документы легко поддаются аннотированию, что позволяет выделять наиболее важные части текста и увеличить производительность их обработки.

Аннотирование, или реферирование (summarization) – это процесс составления краткого описания различных фрагментов текста. Фрагменты могут состоять из нескольких предложений, абзацев, включать в себя целый документ или даже несколько документов. Задачей аннотирования (реферирования) может быть определение тех предложений, которые выражают основной смысл фрагмента, определение предпосылок, помогающих понять содержание фрагмента, или поиск заданных элементов во фрагментах. Зачастую для выполнения этих задач (и задачи аннотирования в целом) очень важен общий контекст обрабатываемого текста.

Обзор инструментальных средств обработки естественного языка

Существует много общедоступных инструментальных средств, поддерживающих технологию обработки естественного языка. Некоторые из них входят в комплект Java SE SDK, но их возможности ограничены достаточно узким кругом простейших задач. Другие библиотеки, например OpenNLP (Apache) и LingPipe, предоставляют более мощную и усовершенствованную поддержку методик обработки естественного языка.

Низкоуровневая поддержка в языке Java включает классы для работы со строками: `String`, `StringBuilder` и `StringBuffer`. Эти классы содержат методы, выполняющие операции поиска, сопоставления и замены текстовых фрагментов. *Регулярные выражения* (*regular expressions*) применяют специализированные методы определения со-

впадения подстрок. Java предлагает богатый выбор инструментальных средств для использования регулярных выражений.

Ранее уже отмечалось, что для разделения текста на отдельные элементы используются механизмы *токенизации*. Java поддерживает и этот вид обработки:

- метод `split()` в классе `String`;
- класс `StreamTokenizer`;
- класс `StringTokenizer`.

Кроме того, для языка Java написано множество библиотек/API для поддержки обработки естественного языка. Неполный список таких библиотек приводится в табл. 1.1. Большинство из них являются проектами с открытым исходным кодом, но существуют также и коммерческие API и библиотеки. Мы будем рассматривать API с открытыми исходными кодами.

Таблица 1.1. Список API на языке Java, поддерживающих обработку естественного языка

API	URL
Apertium	http://www.apertium.org/
General Architecture for Text Engineering	http://gate.ac.uk/
Learning Based Java	http://cogcomp.cs.illinois.edu/page/software_view/LBJ
LinguaStream	http://www.linguastream.org/
LingPipe	http://alias-i.com/lingpipe/
Mallet	http://mallet.cs.umass.edu/
MontyLingua	http://web.media.mit.edu/~hugo/montylingua/
Apache OpenNLP	http://opennlp.apach.org/
UIMA	http://uima.apache.org/
Stanford Parser	http://nlp.stanford.edu/software

Многие из упомянутых выше API обработки естественного языка позволяют объединять задачи в *конвейеры* (*pipeline*), своеобразные последовательности шагов (операций), для достижения определенных целей обработки текста. Примерами инструментов, поддерживающих конвейеры, являются GATE и Apache UIMA.

В следующих разделах мы подробнее познакомим вас с некоторыми NLP API, дадим краткий обзор их возможностей и для каждого API приведем список полезных ссылок.

Apache OpenNLP

Проект Apache OpenNLP предназначен для решения общих задач обработки естественного языка и будет широко использоваться в данной книге. Он состоит из нескольких компонентов, которые выполняют собственные задачи, формируют модели для обучения и обеспечивают поддержку тестирования этих моделей. Основная методика, принятая в OpenNLP, состоит в создании экземпляра модели для конкретной задачи и последующем выполнении методов этой модели для решения задачи.

Например, следуюший пример выполняет разбивку простой строки на элементы. Для корректной работы кода в нем должна быть реализована обработка исключений `FileNotFoundException` и `IOException`. Блок попытки получения ресурса создает экземпляр потока ввода `FileInputStream` и открывает файл *en-token.bin*. Этот файл содержит модель, обученную на выборке с текстом на английском языке:

```
try (InputStream is = new FileInputStream(
        new File(getModeDir(), "en-token.bin"))) {
    // Здесь вставляется код для токенизации текста.
} catch(FileNotFoundException ex) {
    ...
} catch(IOException ex)
    ...
}
```

Затем на основе этого файла внутри блока `try` создается экземпляр класса `TokenizerModel`. Далее создается экземпляр класса `Tokenizer`, как показано ниже:

```
TokenizerModel model = new TokenizerModel(is);
Tokenizer tokenizer = new TokenizerME(model);
```

После этого применяется метод `tokenize()`, которому передается обрабатываемый текст. Метод возвращает массив объектов `String`:

```
String tokens[] = tokenizer.tokenize("He lives at 1511 W. Randolph.");
```

Оператор `for-each` последовательно выводит все элементы, полученные в результате обработки текста. Квадратные скобки используются для более четкого выделения элементов:

```
for (String a : tokens) {
  System.out.print("[" + a + "] ");
}
System.out.println();
```

При выполнении будет выведен следующий результат:

```
[He] [lives] [at] [1511] [W.] [Randolph] [.]
```

В данном случае модель правильно определила, что элемент `W.` – это сокращение, а последняя точка является отдельным элементом, обозначающим конец предложения.

Во многих примерах в этой книге мы будем пользоваться OpenNLP API. Ссылки на разделы проекта OpenNLP приведены в табл. 1.2.

Таблица 1.2. Ссылки на разделы проекта OpenNLP

Раздел проекта OpenNLP	Ссылка на раздел веб-сайта
Домашная страница	https://opennlp.apache.org/
Документация	https://opennlp.apache.org/documentation.html
Документация в формате Javadoc	http://nlp.stanford.edu/nlp/javadoc/javanlp/index.html
Загрузка файлов библиотеки	https://opennlp.apache.org/cgi-bin/download.cgi
Wiki	https://cwiki.apache.org/confluence/display/OPENNLP/Index%3bjsessionid=32B408C73729ACCCDD071D9EC354FC54

Stanford NLP

Рабочая группа Stanford NLP Group проводит исследования в области обработки естественного языка и предоставляет инструменты для решения задач NLP. Одним из таких наборов инструментальных средств является Stanford CoreNLP. Группа предлагает также другие инструменты, такие как Stanford Parser, Stanford POS Tagger, Stanford Classifier. Инструменты Stanford поддерживают английский и китайский языки, обеспечивают решение основных задач обработки естественного языка, включая токенизацию и распознавание именованных объектов.

Инструменты Stanford выпускаются на условиях лицензии GPL, но их использование в коммерческих приложениях запрещено – для этого необходимо приобрести специальную коммерческую лицензию. Библиотека Stanford NLP хорошо организована и поддерживает все основные функции обработки естественного языка.

Группа Stanford обеспечивает поддержку нескольких методик токенизации. Мы воспользуемся классом `PTBTokenizer` для демонстрации возможностей этой NLP-библиотеки. Конструктор, показанный

ниже, принимает объект Reader, аргумент LexedTokenFactory<T> и строку, определяющую некоторые дополнительные параметры.

LexedTokenFactory – это интерфейс, реализация которого выполняется классами CoreLabelTokenFactory и WordTokenFactory. Первый сохраняет позиции начального и конечного символов элемента, а второй просто возвращает элемент в виде строки без какой-либо информации о его расположении. По умолчанию используется класс WordTokenFactory.

Следующий пример использует класс CoreLabelTokenFactory. Он создает объект StringReader из заданной строки текста. В последнем аргументе передаются дополнительные параметры, но в данном примере они отсутствуют, поэтому передается null. Реализация интерфейса Iterator выполняется классом PTBTokenizer, что позволяет применить методы hasNext() и next() для вывода элементов:

```
PTBTokenizer ptb = new PTBTokenizer(
new StringReader("He lives at 1511 W. Randolph."),
new CoreLabelTokenFactory(), null);
while(ptb.hasNext()) {
  System.out.println(ptb.next());
}
```

Пример выведет следующий результат:

```
He
lives
at
1511
W.
Randolph
.
```

В дальнейшем мы будем активно использовать библиотеку Stanford NLP. В табл. 1.3 приводится список ссылок на пакеты инструментов Stanford. Ссылки на документацию и на файлы для загрузки можно без труда найти на соответствующих сайтах.

Таблица 1.3. Инструментальные средства Stanford NLP

Stanford NLP	Ссылка на раздел веб-сайта
Домашняя страница	http://nlp.stanford.edu/index.shtml
CoreNLP	http://nlp.stanford.edu/software/corenlp.shtml#Download
Parser	http://nlp.stanford.edu/software/lex-parser.shtml
POS Tagger	http://nlp.stanford.edu/software/tagger.shtml
Список рассылки java-nlp-user	https://mailman.stanford.edu/mailman/listinfo/java-nlp-user

LingPipe

Библиотека LingPipe состоит из набора инструментов для выполнения общих задач обработки естественного языка и поддерживает обучение и тестирование моделей. Существуют две версии LingPipe – свободная и коммерческая. Использование свободной версии подразумевает некоторые ограничения.

Для демонстрации практического применения LingPipe рассмотрим пример разбивки текста на элементы с помощью класса `Tokenizer`. Сначала объявляются два списка: один – для хранения элементов, второй – для хранения пробельных символов:

```
List<String> tokenList = new ArrayList<>();
List<String> whiteList = new ArrayList<>();
```

Загрузка исходных кодов примеров
Скачать файлы с примерами программного кода для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Далее объявляется строка с исходным текстом:

```
String text = "A sample sentence processed \nby \tthe LingPipe tokenizer.";
```

Затем создается экземпляр класса `Tokenizer`. Как показано в следующем фрагменте, статический метод `tokenizer()` используется для создания экземпляра класса `Tokenizer` на основе класса-фабрики `IndoEuropean`:

```
Tokenizer tokenizer = IndoEuropeanTokenizerFactory.INSTANCE.
tokenizer(text.toCharArray(), 0, text.length());
```

Далее вызовом метода `tokenize()` этого класса производится заполнение списков, объявленных ранее:

```
tokenizer.tokenize(tokenList, whiteList);
```

Для вывода элементов текста используется оператор `for-each`:

```
for(String element : tokenList) {
  System.out.print(element + " ");
}
System.out.println();
```

Этот пример выведет следующие результаты:

```
A sample sentence processed by the LingPipe tokenizer
```

В табл. 1.4 приводится список ссылок на разные разделы проекта LingPipe.

Таблица 1.4. Ссылки на разделы проекта LingPipe

LingPipe	Ссылка на раздел веб-сайта
Домашняя страница	http://alias-i.com/lingpipe/index.html
Учебные пособия	http://alias-i.com/lingpipe/demos/tutorial/read-me.html
JavaDocs	http://alias-i.com/lingpipe/docs/api/index.html
Файлы загрузки	http://alias-i.com/lingpipe/web/install.html
Ядро библиотеки	http://alias-i.com/lingpipe/web/download.html
Модели	http://alias-i.com/lingpipe/web/models.html

GATE

General Architecture for Text Engineering (*GATE*) – это набор инструментов на Java, разработанный в университете города Шеффилд (Англия) (University of Sheffield, England) и поддерживающий многие языки и типы задач NLP. GATE также можно использовать в качестве конвейера для NLP-обработки.

API поддерживает совместную работу с GATE Developer, инструментом просмотра документов, который выводит текст с аннотациями. Это удобно при изучении документа, содержащего выделенные аннотации. Также доступен GATE Mimur – инструмент для индексирования и поиска в тексте, собранном из различных источников. Применение GATE для решения многих задач обработки естественного языка требует написания небольшого объема исходного кода. Для встраивания функциональных возможностей GATE непосредственно в код приложения используется пакет GATE Embedded. Ссылки на разделы проекта GATE приведены в табл. 1.5.

Таблица 1.5. Ссылки на разделы проекта GATE

GATE	Ссылка на раздел веб-сайта
Домашняя страница	https://gate.ac.uk/
Документация	https://gate.ac.uk/documentation.html
JavaDocs	http://jenkins.gate.ac.uk/job/GATE-Nightly/javadoc/
Файлы загрузки	https://gate.ac.uk/download
Wiki	http://gatewiki.sf.net/

UIMA

Организация по развитию стандартов структурированной информации (*Organization for the Advancement of Structured Information Standards, OASIS*) – это консорциум, основной задачей которого является развитие технологий электронной коммерции и бизнес-про-

ектов в области ИТ. OASIS разработал стандарт *архитектуры управления неструктурированной информацией* (*Unstructured Information Management Architecture, UIMA*) как базовую рабочую среду для конвейеров обработки естественного языка. Этот стандарт поддерживается инструментальной библиотекой Apache UIMA.

Несмотря на то что стандарт UIMA в основном ориентирован на поддержку создания конвейеров, он также позволяет описывать последовательности шаблонов проектирования, различные варианты представления данных и роли пользователей для анализа текста. Ссылки на разделы проекта Apache UIMA приведены в табл. 1.6.

Таблица 1.6. Ссылки на разделы проекта Apache UIMA

Apache UIMA	Ссылка на раздел веб-сайта
Домашняя страница	https://uima.apache.org/
Документация	https://uima.apache.org/documentation.html
JavaDocs	https://uima.apache.org/d/uimaj-2.6.0/apidocs/index.html
Файлы загрузки	https://uima.apache.org/downloads.cgi
Wiki	https://cwiki.apache.org/confluence/display/UIMA/Index

Обзор задач обработки текста

Существует достаточно большое разнообразие задач обработки естественного языка, но мы сосредоточимся лишь на ограниченном их подмножестве. Ниже представлены краткие обзорные характеристики задач, которые будут более подробно рассматриваться в последующих главах:

- поиск фрагментов текста;
- поиск предложений;
- поиск людей и неодушевленных именованных объектов;
- определение частей речи;
- классификация текстов и документов;
- выделение взаимоотношений;
- комплексные методики обработки.

Для достижения поставленной цели многие задачи используются в комплексе. В данной книге мы неоднократно будем наблюдать примеры такого объединения. Например, разбивка текста на элементы часто применяется в качестве начального этапа для многих других задач, поэтому можно сказать, что это основной, фундаментальный этап.

Поиск фрагментов текста

Текст можно разделить на элементы разных типов – слова, предложения, абзацы и т. д. Существует несколько способов классификации таких элементов. Говоря о частях текста в данной книге, мы подразумеваем слова, иногда обозначаемые специальным термином «*лексема*», или «*токен*» (*token*). *Морфология* (*morphology*) изучает структуру и формы слов. В дальнейшем мы часто будем пользоваться морфологическими терминами. А сейчас просто перечислим некоторые основные способы классификации слов:

- *простые слова* – слова в обычном, общем смысле, как, например, 14 слов в данном предложении;
- *морфемы* (*morphemes*) – минимальные значимые части слов, которые сохраняют смысловое значение. Например, в слове «лисий» морфемой является «лис» (в слове «bounded» морфема «bound»). Морфемами могут быть корни, приставки и суффиксы, например в слове «пересказывать» можно выделить морфемы «пере» (приставка), «сказ» (корень) и «ыва» (суффикс), а в английском слове «bounded» суффикс «ed» также представляет собой морфему;
- *приставка/суффикс* – приставка располагается перед корнем слова, суффикс – после корня слова, но перед его окончанием (если оно есть). Например, в слове «перевозчик» приставкой является «пере», а суффиксом – «чик». В английском слове «graduation» суффикс «ation» присоединяется к редуцированной форме слова «graduate»;
- *синонимы* – слова, различные по звучанию, но тождественные или близкие по смыслу, а также синтаксические и грамматические конструкции, совпадающие по значению. В качестве примеров можно привести пары слов «путь» – «дорога», «огромный» – «колоссальный», в английском языке: «small» – «tiny». Работа с синонимами требует разрешения лексической многозначности;
- *сокращения слов* – в текстах часто применяются сокращения, например вместо «господин Чернов» пишут «г-н Чернов», вместо «Mister Smith» – «Mr. Smith»;
- *акронимы и аббревиатуры* – весьма широко используются во многих областях деятельности, в том числе и в ИТ. Представляют собой комбинации начальных букв (одной или нескольких) слов, входящих в сокращаемое словосочетание, например из FORmula TRANslation составлено название языка програм-

мирования Fortran. Сокращения могут быть рекурсивными – GNU («GNU is Not Unix»). Даже в этой книге активно используется аббревиатура NLP;

- *стяжение*, или *стяженная форма* (*контракция – contraction*) – слияние двух смежных гласных в один гласный или в дифтонг, которое иногда бывает чисто фонетическим, но может стать и частью морфологии языка. Например, в русском языке формы глаголов «приду» и «приму» возникли путем стяжения гласных: при-иду → приду, при-иму → приму. В английском языке стяжения встречаются несколько чаще: cannot → can't, I will → I'll, got to → gotta и т. д.;
- *числа* – особые слова, обычно состоящие только из цифр. Более сложные варианты написания чисел могут содержать десятичную точку (запятую), специальные символы, используемые в научной нотации, показатель степени числа или обозначение системы счисления.

Определение перечисленных выше частей (элементов) очень важно для решения других задач обработки естественного языка. Например, чтобы найти границы предложения, необходимо разделить его на отдельные элементы и определить, какие из этих элементов обозначают конец предложения.

Процесс разделения текста на элементы обозначается термином «*токенизация*» (*tokenization*). Результатом этого процесса является поток или последовательность элементов – лексем или *токенов*[1] (*tokens*). Элементы текста, по которым определяются границы токенов, называют *разделителями* (*delimiters*). В большинстве текстов на русском, английском и других европейских языках разделителями являются пробельные символы, к которым обычно относятся пробелы, символы табуляции и символы перехода на новую строку.

Токенизация может быть простой или сложной (комплексной). Ниже приводится простой метод токенизации с использованием метода `split()` из класса `String`. Сначала объявляется строка с текстом, который нужно обработать:

```
String text = "Mr. Smith went to 123 Washington avenue.";
```

В метод `split()` передается регулярное выражение, определяющее разделители, по которым будет разбит текст. В данном случае выра-

[1] Эти термины часто используют как взаимозаменяемые, хотя, строго говоря, между ними существует некоторое различие. – *Прим. перев.*

жение представляет собой строку "\\s+", то есть в качестве разделителей принимается один или несколько пробельных символов:

```
String tokens[] = text.split("\\s+");
```

Оператор for-each используется для вывода результатов в виде отдельных элементов-токенов:

```
for(String token : tokens) {
  System.out.println(token);
}
```

Пример выведет следующий результат:

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

Более подробно процесс токенизации будет рассматриваться в главе 2 «Поиск фрагментов текста».

Поиск предложений

На первый взгляд в процедуре определения отдельного предложения нет ничего сложного. Как в русском языке, так и в английском достаточно найти символ, завершающий предложение, – точку, вопросительный или восклицательный знак. Но в главе 3 «Поиск предложений» мы увидим, что этот процесс не всегда так прост, как кажется. Например, поиск конца предложения может быть затруднен наличием внутри предложения точек, обозначающих сокращенные слова: «г. Белов» или «ул. Садовая, д. 12, кв. 5» («Dr. Smith» или «204 SW. Park Street»).

Процесс поиска конца предложения в сложных случаях называется *разрешением границ предложений* (*Sentence Boundary Disambiguation, SBD*). Это более существенная проблема в русском или английском языках, чем в таких языках, как китайский или японский, в которых существуют четко определенные, однозначные разделители предложений.

Определение границ предложений необходимо по многим причинам. Некоторые задачи обработки естественного языка, такие как *морфологическая разметка* (*POS tagging*) и идентификация (извлечение) объектов, работают с отдельными предложениями. Прило-

жениям, организованным по схеме «вопрос–ответ», также требуется разбивка на отдельные предложения. Для корректного решения этих и подобных задач без правильного определения границ предложений никак не обойтись.

Следующий пример показывает, как разделить текст на предложения с помощью класса `DocumentPreprocessor` из библиотеки Stanford. Этот класс генерирует список отдельных предложений либо из обычного текста, либо из XML-документа. Класс реализует интерфейс `Iterable`, позволяющий использовать список непосредственно в операторе for-each.

Сначала объявляется строка с предложением:

```
String paragraph = "The first sentence. The second sentence.";
```

На основе этой строки создается объект класса `StringReader`, поддерживающий простые методы типа `read()`, который затем передается конструктору `DocumentPreprocessor`:

```
Reader reader = new StringReader(paragraph);
DocumentPreprocessor documentPreprocessor =
new DocumentPreprocessor(reader);
```

Теперь объект `DocumentPreprocessor` хранит предложения абзаца. Следующая строка кода создает список строк, куда будут записаны найденные предложения:

```
List<String> sentenceList = new LinkedList<String>();
```

Далее обрабатывается каждый элемент, найденный объектом `documentPreprocessor`, то есть для каждого элемента составляется список объектов типа `HasWord`, как показано ниже. Элементы `HasWord` – это объекты, представляющие отдельные слова. Экземпляр класса `StringBuilder` используется для формирования предложения, включающего каждый элемент, ранее добавленный в список `hasWordList`. По окончании формирования предложения оно добавляется в список `sentenceList`:

```
for(List<HasWord> element : documentPreprocessor) {
  StringBuilder sentence = new StringBuilder();
  List<HasWord> hasWordList = element;
  for(HasWord token : hasWordList) {
     sentence.append(token).append(" ");
  }
  sentenceList.add(sentence.toString());
}
```

Наконец, оператор `for-each` выводит каждое найденное предложение в отдельной строке:

```
for(String sentence : sentenceList) {
  System.out.println(sentence);
}
```

Данный пример выведет следующее:

```
The first sentence .
The second sentence .
```

Более подробно процесс разрешения границ предложений (SBD) будет рассматриваться в главе 3 «Поиск предложений».

Поиск людей и прочих именованных объектов

Механизмы поиска вполне приемлемо выполняют свою работу, соответствующую потребностям большинства пользователей. Эти механизмы часто используются для поиска адресов предприятий или чтобы узнать время начала сеанса в кинотеатре. Любой текстовый процессор может выполнить простейший поиск и найти заданное слово или фразу в тексте. Но эта задача может усложниться, если потребуется учесть дополнительные условия поиска, скажем понадобится найти не только заданное слово, но и возможные его синонимы или объекты, напрямую связанные с заданной темой.

Например, предположим, что мы находимся на некотором вебсайте с целью покупки нового ноутбука. Новый ноутбук никому не помешает. Поисковый механизм на этом сайте помогает найти именно ту модель ноутбука, которая обладает всеми требуемыми характеристиками. Зачастую поиск выполняется на основе предварительного анализа информации, получаемой от продавца. Как правило, такой анализ требует обработки текста для выделения полезной информации, немедленно предоставляемой покупателю.

Информация может быть представлена в форме нескольких разделов или категорий, обычно расположенных в левой части вебстраницы. Например, разделы для ноутбуков могут включать такие категории, как Ultrabook, Chromebook, или классификацию по размеру жесткого диска, как показано на рис. 1.1, где изображен фрагмент веб-страницы Amazon.

Некоторые виды поиска могут быть очень простыми. Например, в классе `String` и в родственных ему классах имеются такие методы, как `indexOf()` и `lastIndexOf()`, определяющие местоположение заданного образца в объекте класса `String`, то есть в строке. В следующем

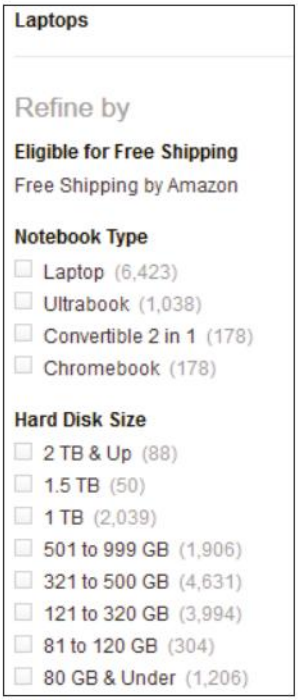


Рис. 1.1. Пример представления информации для пользователя

простом примере метод `indexOf()` возвращает индекс (он же номер позиции) начала заданного образца поиска:

```
String text = "Mr. Smith went to 123
Washington avenue.";
String target = "Washington";
int index = text.indexOf(target);
System.out.println(index);
```

Этот фрагмент выведет следующее:

```
22
```

Такой подход применим только для решения самых простых задач.

При поиске в тексте часто используется прием на основе структуры данных, называемой *инвертированным*, или *обратным*, *индексом* (*inverted index*). В процессе поиска выполняются токенизация текста и идентификация целевых слов и выражений вместе с номерами их позиций. Затем целевые слова и выражения сохраняются в инвертированном индексе. При поиске конкретного слова или выражения выполняется проход по инвертированному индексу и извлекается информация о позиции искомого объекта. Эта процедура выполняется быстрее, чем поиск заданного образца по всему тексту каждый раз, когда это необходимо. Поиск по инвертированному индексу часто применяется в базах данных, в специализированных информационных системах и в механизмах поиска.

При ответах на запросы, такие как «Адреса хороших ресторанов в Бостоне», приходится выполнять более сложные виды поиска. Для ответа на такой запрос необходимо выполнить распознавание/определение объектов, выделить значимые слова (словосочетания), провести семантический анализ, чтобы установить точный смысл запроса, и затем выполнить собственно поиск и ранжирование найденных ответов.

Чтобы наглядно показать процесс поиска именованных объектов, используем связку из объекта `Tokenizer` и класса `TokenNameFinderModel` из библиотеки OpenNLP. Поскольку применение этой методики может вызвать исключение `IOException`, для его обработки следует за-

ключить данный фрагмент кода в блок try-catch. В том же блоке размещается и массив строк, содержащий исследуемые предложения:

```
try {
    String[] sentences = {
        "Tim was a good neighbor. Perhaps not as good a Bob " +
        "Haywood, but still pretty good. Of course Mr. Adam " +
        "took the cake!" };
    // Здесь вставляется код для поиска имен в заданном тексте.
  } catch(IOException ex) {
    ex.printStackTrace();
}
```

Перед обработкой предложений необходимо разделить текст на элементы (провести токенизацию). Для этого создается экземпляр класса Tokenizer:

```
Tokenizer tokenizer = SimpleTokenizer.INSTANCE;
```

Для идентификации предложений нужно использовать соответствующую модель, чтобы исключить элементы, которые могут помешать определению границ предложений. Мы будем использовать класс TokenNameFinderModel на основе модели в файле *en-ner-person.bin*. Экземпляр класса TokenNameFinderModel создается из указанного файла следующим образом:

```
TokenNameFinderModel model = new TokenNameFinderModel(
new File("C:\\OpenNLP Models", "en-ner-person.bin"));
```

Класс NameFinderME непосредственно выполняет поиск заданного имени. Экземпляр этого класса создается с помощью ранее созданного экземпляра TokenNameFinderModel, как показано ниже:

```
NameFinderME finder = new NameFinderME(model);
```

Оператор for-each в следующем фрагменте обрабатывает каждое предложение. Метод tokenize() разделяет предложение на элементы, метод find() возвращает массив объектов типа Span, хранящих начальную и конечную позиции (индекс) имен, найденных методом find():

```
for(String sentence : sentences) {
   String[] tokens = tokenizer.tokenize(sentence);
   Span[] nameSpans = finder.find(tokens);
   System.out.println(Arrays.toString(
   Span.spansToString(nameSpans, tokens)));
}
```

Описанный выше пример выведет следующий результат:

```
[Tim, Bob Haywood, Adam]
```

Поиск и распознавание имен являются основной темой главы 4 «Поиск людей и неодушевленных именованных объектов».

Определение частей речи

Еще один способ классификации элементов текста выполняется на уровне предложения. Предложение может быть разделено на отдельные слова или словосочетания по таким категориям, как существительные, глаголы, наречия и предлоги. Определению частей речи все (или почти все) научились еще в школе. Кроме того, мы узнали, что не следует завершать предложение предлогом или союзом, а также много других полезных вещей.

Определение *частей речи* (*Parts of speech, POS*) имеет важное значение для других задач, таких как определение отношений в тексте и его общего смысла. Классификация по частям речи называется *синтаксическим анализом*, или *парсингом* (*parsing*). Определение частей речи выполняется для улучшения качества данных, передаваемых в другие части конвейера обработки.

Внутренние операции процесса определения частей речи могут быть достаточно сложными. К счастью, большая часть этих операций скрыта от нас и заключена в классы и методы. Для демонстрации определения частей речи мы воспользуемся двумя классами из библиотеки OpenNLP. Нам также потребуется модель для определения частей речи. Для создания экземпляра такой модели мы используем класс `POSModel` и файл *en-pos-maxent.bin*, как показано ниже:

```
POSModel model = new POSModelLoader().load(
    new File("../OpenNLP Models/", "en-pos-maxent.bin"));
```

Фактическое определение частей речи будет выполнять класс `POSTaggerME`. Экземпляр этого класса создается на основе ранее определенной модели:

```
POSTaggerME tagger = new POSTaggerME(model);
```

Далее объявляется строка с обрабатываемым текстом:

```
String sentence = "POS processing is useful for enhancing the " +
   "quality of data sent to other elements of a pipeline.";
```

Разделим текст на элементы по пробельным символам:

```
String tokens[] = WhitespaceTokenizer.INSTANCE.tokenize(sentence);
```

Определим части речи для всех элементов с помощью метода `tag()` и сохраним результаты в массиве строк:

```
String[] tags = tagger.tag(tokens);
```

Затем выведем выделенные элементы и соответствующие обозначения частей речи следующим образом:

```
for(int i=0; i < tokens.length; i++) {
    System.out.print(tokens[i] + "[" + tags[i] + "] ");
}
```

Данный пример должен вывести такой результат:

```
POS[NNP] processing[NN] is[VBZ] useful[JJ] for[IN] enhancing[VBG]
the[DT] quality[NN] of[IN] data[NNS] sent[VBN] to[TO] other[JJ]
elements[NNS] of[IN] a[DT] pipeline.[NN]
```

За каждым элементом текста следует аббревиатура в квадратных скобках, обозначающая соответствующую часть речи. Например, NNP обозначает существительное, имя собственное (proper noun). Значения всех аббревиатур будут описаны в главе 5 «Определение частей речи», в которой данная тема будет рассматриваться во всех подробностях.

Классификация текстов и документов

Целью классификации обычно является присваивание меток фрагментам, найденным в текстах и документах. Если метки известны, то есть имеют имена, процесс называют *классификацией* (*classification*). Если метки неизвестны, не имеют имен, процесс называют *кластеризацией* (*clustering*).

К задачам обработки естественного языка относится также *категоризация* (*categorization*). Это процесс причисления некоторого элемента текста к одной из нескольких возможных групп (категорий). Например, военные летательные аппараты можно разделить по следующим категориям: истребитель, бомбардировщик, штурмовик, разведчик, транспорт, спасатель.

Классификаторы могут быть организованы в соответствии с типом результата. Часто встречается бинарная, или двоичная, классификация, результаты которой представлены в виде ответов «да/нет». Обычно этот тип классификации используется для поддержки фильтров спама. Другие типы подразумевают получение результата, разделенного на несколько возможных категорий.

Классификация, в отличие от многих других задач NLP, требует нескольких этапов обработки, о чем мы более подробно поговорим в разделе «О моделях обработки естественного языка» данной главы, хотя из-за сложности и многоэтапности этого процесса продемон-

стрировать его на кратком примере не представляется возможным. В главе 6 «Классификация текстов и документов» процедура классификации рассматривается во всех подробностях и приводится развернутый пример ее применения.

Выделение взаимоотношений

Процесс определения взаимоотношений определяет смысловые отношения, существующие в тексте. Например, мы понимаем, что в предложении «Смысл и цель жизни вполне очевидны» («The meaning and purpose of life is plain to see») главная мысль (тема) выражена словами «Смысл и цель жизни» («The meaning and purpose of life») и связана с последующим словосочетанием, в котором утверждается, что объекты главной мысли «вполне очевидны» («plain to see»).

Люди хорошо справляются с определением взаимосвязей объектов, по крайней мере тех, что лежат на поверхности, но при выявлении более глубоких взаимоотношений могут возникать существенные затруднения. Еще более сложной задачей становится использование компьютера для определения взаимоотношений. Зато компьютеры способны обрабатывать большие объемы данных для обнаружения взаимоотношений, не очевидных для человека, или для выявления взаимосвязей, которые невозможно найти вручную за сколько-нибудь приемлемое время.

Взаимоотношения могут быть самого разного вида, такие как место расположения некоторого объекта, отношение двух людей друг к другу, взаимоотношения частей системы и, наконец, кто несет ответственность за все это. Определение взаимоотношений позволяет сформировать исходные данные для выполнения ряда других задач: создание баз знаний, анализ тенденций в экономике и коммерции, сбор разведданных и т. д. Процесс определения взаимоотношений иногда называют *интеллектуальным анализом текста* (*text analytics* или *text mining*).

Существует несколько методик определения взаимоотношений. Они подробно рассматриваются в главе 7 «Использование синтаксического анализатора (парсера) для выделения взаимоотношений». Здесь мы покажем применение одной из этих методик для определения взаимоотношений в предложении, воспользовавшись классом `StanfordCoreNLP` из библиотеки Stanford NLP. Данный класс поддерживает конвейер, в котором определяются и применяются к тексту так называемые аннотаторы. *Аннотатор* (*annotator*) можно считать отдельной операцией. После создания вышеупомянутого

класса аннотаторы добавляются в него с помощью объекта Properties из пакета java.util.

Сначала создается экземпляр класса Properties, затем в него добавляются аннотаторы:

```
Properties properties = new Properties();
properties.put("annotators", "tokenize, ssplit, parse");
```

Здесь задействованы три аннотатора, определяющих операции для выполнения. В данном случае это минимальный набор операций, требуемых для синтаксического разбора текста: токенизация (аннотатор tokenize), разделение элементов (токенов) по предложениям (аннотатор ssplit) и синтаксический анализ (аннотатор parse).

Далее создается экземпляр класса StanfordCoreNLP, использующий переменную, ссылающуюся на набор свойств:

```
StanfordCoreNLP pipeline = new StanfordCoreNLP(properties);
```

После этого создается экземпляр класса Annotation, конструктору которого передается обрабатываемый текст:

```
Annotation annotation = new Annotation(
    "The meaning and purpose of life is plain to see.");
```

Метод annotate() объекта pipeline применяется для обработки объекта annotation. Далее вызывается метод prettyPrint() для вывода результатов обработки:

```
pipeline.annotate(annotation);
pipeline.prettyPrint(annotation, System.out);
```

Описанный выше код введет следующий результат:

```
Sentence #1 (11 tokens):
The meaning and purpose of life is plain to see.

[Text=The CharacterOffsetBegin=0 CharacterOffsetEnd=3 PartOfSpeech=DT]
[Text=meaning CharacterOffsetBegin=4 CharacterOffsetEnd=11
PartOfSpeech=NN]
[Text=and CharacterOffsetBegin=12 CharacterOffsetEnd=15 PartOfSpeech=CC]
[Text=purpose CharacterOffsetBegin=16 CharacterOffsetEnd=23
PartOfSpeech=NN]
[Text=of CharacterOffsetBegin=24 CharacterOffsetEnd=26 PartOfSpeech=IN]
[Text=life CharacterOffsetBegin=27 CharacterOffsetEnd=31
PartOfSpeech=NN]
[Text=is CharacterOffsetBegin=32 CharacterOffsetEnd=34 PartOfSpeech=VBZ]
[Text=plain CharacterOffsetBegin=35 CharacterOffsetEnd=40
PartOfSpeech=JJ]
[Text=to CharacterOffsetBegin=41 CharacterOffsetEnd=43 PartOfSpeech=TO]
```

```
[Text=see CharacterOffsetBegin=44 CharacterOffsetEnd=47 PartOfSpeech=VB]
[Text=. CharacterOffsetBegin=47 CharacterOffsetEnd=48 PartOfSpeech=.]

(ROOT
  (S
    (NP
      (NP (DT The) (NN meaning)
        (CC and)
        (NN purpose))
      (PP (IN of)
        (NP (NN life))))
    (VP (VBZ is)
      (ADJP (JJ plain)
        (S
          (VP (TO to)
            (VP (VB see))))))
    (. .)))

root(ROOT-0, plain-8)
det(meaning-2, The-1)
nsubj(plain-8, meaning-2)
conj_and(meaning-2, purpose-4)
prep_of(meaning-2, life-6)
cop(plain-8, is-7)
aux(see-10, to-9)
xcomp(plain-8, see-10)
```

В первой части результатов содержится исходный текст с отдельными элементами (токенами) и обозначениями частей речи. Далее следует древовидная структура, отображающая организацию предложения. В последней части показаны взаимоотношения между элементами на грамматическом уровне. Рассмотрим следующий пример:

```
prep_of(meaning-2, life-6)
```

Данная запись показывает, как предлог «of» используется для установления взаимосвязи между словами «meaning» и «life». Эта информация полезна для решения многих задач упрощения текста.

Комплексные методики обработки

Как уже отмечалось, на практике решения задач обработки естественного языка часто состоят из решений нескольких элементарных NLP-задач, объединяемых в конвейер для получения требуемых результатов. Пример использования такого конвейера мы видели в предыдущем разделе.

Поскольку большинство решений используют конвейеры, мы представим несколько примеров применения конвейеров в главе 8 «Комплексные методики».

О моделях обработки естественного языка

Вне зависимости от конкретной решаемой задачи обработки естественного языка и от используемого набора инструментальных средств существует общая последовательность этапов, или шагов, решения любой задачи. В данном разделе приводится краткое описание этих этапов. При чтении книги и изучении представленных в ней методик вы убедитесь, что последовательность этапов решения задач все время повторяется с небольшими вариациями. Понимание необходимости соблюдения порядка следования шагов поможет в дальнейшем быстрее и проще осваивать предлагаемые методики.

Основные этапы (шаги) решения любой задачи обработки естественного языка включают:

- определение задачи (типа задачи);
- выбор модели;
- создание и обучение модели;
- проверка модели;
- практическое использование модели.

В последующих разделах дается краткое описание каждого этапа решения.

Определение задачи (типа задачи)

Важно хорошо понимать сущность задачи, которую необходимо решить. Хорошее понимание представляет собой основу для разработки решения, состоящего из последовательности определенных шагов. На каждом таком шаге решается одна из элементарных задач NLP.

Например, предположим, что необходимо найти ответ на вопрос: «Кто сейчас является мэром Парижа?» («Who is the mayor of Paris?») Для этого нужно провести синтаксический разбор вопроса, то есть определить все части речи в нем, выяснить сущность, смысл вопроса, классификацию элементов вопроса, и, наконец, воспользоваться базой знаний, созданной с помощью других NLP-методик, для ответа на поставленный вопрос.

Другие задачи могут в большей или меньшей степени отличаться от описанной выше. Иногда требуется лишь разделить текст на компоненты, чтобы связать текст с какой-либо категорией. Например, анализ описания некоторого товара, предлагаемого поставщиком, может выполняться с целью определения предполагаемых категорий

этого товара. Анализ описания автомобиля позволяет распределять модели по таким категориям, как седан, спорткар, внедорожник или компактткар.

Представление обо всех доступных методиках решения элементарных задач обработки естественного языка существенно облегчает их подбор для решения конкретной практической задачи, стоящей перед вами.

Выбор модели

Многие задачи, которые мы будем рассматривать, основаны на моделях. Например, чтобы разделить документ на отдельные приложения, требуется соответствующий алгоритм. Но даже самые лучшие методики определения границ предложений не всегда способны работать правильно. Поэтому возникает необходимость в разработке моделей, исследующих элементы текста и использующих полученную информацию для обнаружения действительных границ предложений.

Точность модели может зависеть от характерных свойств обрабатываемого текста. Модель, успешно определяющая границы предложений в исторических документах, может не подойти для обработки медицинских текстов.

Существует множество готовых моделей, которые можно использовать непосредственно для решения задач обработки естественного языка. Рассматривая конкретную решаемую задачу, мы получаем возможность сделать обоснованный выбор наилучшей модели для нее. В некоторых случаях требуется обучение новой модели. При этом часто приходится принимать компромиссное решение, отдавая предпочтение либо точности, либо скорости. Хорошее знание предметной области задачи и требований к качеству результатов решения позволяет выбрать наиболее подходящую модель.

Создание и обучение модели

Обучение модели – это процесс выполнения некоторого алгоритма с набором данных, описывающих выбранную модель, и последующая проверка модели. В некоторых ситуациях текст требуется обрабатывать совершенно иначе, чем было показано выше. Например, модели, обученные на статьях журналистского стиля, вряд ли будут успешно обрабатывать твиты. Это означает, что существующие модели не всегда способны правильно обрабатывать новые типы данных. В таких случаях необходимо создать и обучить новую модель.

Для обучения моделей мы часто будем использовать так называемые «контрольные» данные, то есть данные, для которых заранее известен правильный результат. Например, при обучении модели определения частей речи в исходных данных все элементы (существительные, глаголы и т. д.) будут уже промаркированы. В процессе обучения эта информация будет использоваться для формирования и уточнения данной модели. Такой набор данных называется *корпусом текстов* (*corpus*)[1].

Проверка модели

После создания, обучения и уточнения модели необходимо проверить ее на контрольном наборе данных. Часто для проверки используется тестовый набор данных с известным правильным результатом. Сравнивая фактический результат применения модели к этим данным с правильным результатом, можно оценить качество модели. Нередко для обучения используется только часть корпуса, а другая часть используется для проверки.

Практическое использование модели

Использование модели означает просто применение обученной и проверенной модели непосредственно к поставленной задаче. Подробности зависят от выбранной модели. В данной главе уже приводились примеры практического применения моделей, например в разделе «Определение частей речи», где использовалась модель, взятая из файла *en-pos-maxent.bin*.

Подготовка данных

Важным этапом в процессе обработки естественного языка являются поиск и подготовка данных. Сюда входят данные для обучения и данные, которые будут обрабатываться. При этом необходимо учитывать несколько факторов. Здесь мы сосредоточимся на поддержке, предоставляемой языком программирования Java для работы с различными наборами символов.

Всегда следует учитывать, как представлены символы в тексте. Несмотря на то что в данной книге основное внимание уделяется обработке текстов на английском языке, мы должны помнить, что в дру-

[1] В лингвистике корпус – подобранная и обработанная по определенным правилам совокупность текстов, используемых в качестве базы для исследования языка. – *Прим. перев.*

гих языках могут быть поставлены специфические задачи, присущие только этим языкам. Различия существуют не только в способах кодировки символов, но и в направлении чтения текста. Например, в японском языке текст читается по столбцам, справа налево.

Кроме того, существует несколько разных стандартов кодировки символов: ASCII, Latin, Unicode и т. д. Краткие характеристики наиболее распространенных кодировок приведены в табл. 1.7. В частности, Unicode представляет собой самую сложную и прогрессивную схему кодировки символов.

Таблица 1.7. Краткое описание наиболее распространенных стандартов кодировки символов

Стандарт кодировки	Описание
ASCII	Для кодирования символов используются 128 числовых значений (от 0 до 127)
Latin	Существует несколько вариантов кодировки Latin, в которой используются 256 числовых значений. Варианты содержат различные комбинации умляута (ö) и прочих диакритических знаков с другими символами (буквами). Разнообразие версий Latin объясняется их предназначением для разных языков индоевропейской группы, в том числе для турецкого и эсперанто
Big5	Двухбайтовая кодировка для набора символов (иероглифов) китайского языка
Unicode	Существуют три версии кодировки Unicode: UTF-8, UTF-16 и UTF-32, – использующие 1, 2 и 4 байта соответственно. Эта схема кодировки позволяет представить символы всех известных в настоящее время языков, включая и такие «фантастические» (искусственно созданные) языки, как клингонский (см. сериал «Стар Трек») и эльфийский

Java предоставляет возможности обработки всех этих схем кодировки. Компилятор `javac` имеет ключ командной строки `-encoding`, позволяющий задать используемую схему кодировки. Следующая командная строка определяет схему кодировки `Big5`:

```
javac -encoding Big5
```

Обработка символов поддерживается на уровне элементарного типа данных `char`, класса `Character` и некоторых других классов и интерфейсов, перечисленных в табл. 1.8.

Таблица 1.8. Символьные типы в языке Java

Символьный тип	Описание
`char`	Элементарный тип данных
`Character`	Класс-обертка для типа `char`
`CharBuffer`	Класс обеспечивает поддержку буфера символов типа `char` и предоставляет методы для записи/извлечения символов в (из) буфер(а), а также для операций с последовательностями символов
`CharSequence`	Интерфейс, реализуемый классами `CharBuffer`, `Segment`, `String` и `StringBuilder`. Поддерживает защищенный от записи (read-only) доступ к последовательности символов типа `char`

Кроме того, Java предоставляет ряд классов и интерфейсов для поддержки строк. Некоторые из них перечислены в табл. 1.9. Эти классы и интерфейсы мы будем использовать в примерах. Классы `String`, `StringBuffer` и `StringBuilder` обладают схожими возможностями, отличаясь только способностью изменять исходную строку поддержкой выполнения в многопоточном режиме выполнения. Интерфейс `CharacterIterator` и класс `StringCharacterIterator` предоставляют методы для обхода последовательностей символов.

Класс `Segment` представляет фрагмент текста.

Таблица 1.9. Классы и интерфейсы для работы со строками в языке Java

Класс/Интерфейс	Описание
`String`	Неизменяемая строка
`StringBuffer`	Представляет изменяемую строку. Обеспечивает безопасность в многопоточном режиме
`StringBuilder`	Совместим с классом `StringBuffer`, но не обеспечивает безопасности в многопоточном режиме
`Segment`	Представляет фрагмент текста в виде символьного массива. Обеспечивает быстрый доступ к отдельным символам и группам символов в массиве
`CharacterIterator`	Определяет итератор для текста. Поддерживает проход по тексту в обоих направлениях
`StringCharacterIterator`	Класс, обеспечивающий реализацию интерфейса `CharacterIterator` для строк типа `String`

При чтении данных из файлов необходимо учитывать их формат. Часто данные извлекаются из источников с аннотированными словами. Например, текст в веб-страницах размечен с помощью HTML-

тегов, которые далеко не всегда имеют отношение к анализу текста и обычно должны удаляться.

Тип многоцелевых расширений интернет-почты (Multipurpose Internet Mail Extensions, MIME) определяет формат файла-источника. Наиболее часто используются типы файлов, перечисленные в табл. 1.10. Удаление или изменение разметки данных, взятых из файла, выполняется либо средствами языка Java, либо с помощью специализированных программ. Некоторые NLP API предоставляют инструментальные средства для работы со специальными форматами файлов.

Таблица 1.10. Наиболее распространенные форматы файлов

Формат файла	MIME-тип	Описание
Текст	plain/text	Простой текстовый файл
Документ, созданный текстовым процессором из пакета Office	application/msword application/vnd.oasis.opendocument.text	Microsoft Office Open Office
PDF	application/pdf	Переносимый формат документа (Portable Document Format) компании Adobe
HTML	text/html	Веб-страницы
XML	text/xml	Расширяемый язык разметки (eXtensible Markup Language)
Файл базы данных	Нет соответствующего MIME-типа	Данные могут быть представлены в различных форматах

Многие библиотеки NLP предполагают, что данные предварительно подготовлены, то есть из них удалены теги разметки. Если это не так, подготовку данных следует провести своими силами, чтобы не получить неожиданных и ошибочных результатов.

Резюме

В этой главе был сделан общий обзор технологии обработки естественного языка и областей ее применения, а также отмечено, что эта технология используется во многих сферах деятельности для решения разнообразных задач: от простого поиска до сложной многоступенчатой классификации. Кратко представлена поддержка технологии обработки естественного языка как штатными средствами языка программирования Java (обработка строк), так и отдельно разработанными библиотеками, специализированными для решения NLP-задач. Рассматривались общие элементарные задачи обработки естественно-

го языка, и для каждой задачи был продемонстрирован пример исходного кода. Кроме того, были описаны в общих чертах процессы обучения, проверки и практического использования моделей.

С помощью данной книги будут заложены прочные основы для применения на практике основных задач обработки естественного языка с помощью и простых, и более сложных методик. Вы увидите, что некоторые задачи требуют только простых методик, и в этом случае уверенного владения ими более чем достаточно. В других ситуациях возникает необходимость в более сложных, комплексных методиках. Как бы то ни было, вы научитесь выбирать наиболее подходящий инструмент и соответствующую методику для решения поставленной задачи.

В следующей главе мы будем подробно рассматривать процесс разделения текста на элементы (токенизации) и как его можно использовать для поиска определенных частей текста.

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.

Для демонстрации этих методик мы будем использовать строковую переменную с именем paragraph. Строка содержит символ перехода на новую строку, который в реальном тексте может встретиться в самых неожиданных местах. Переменная определяется так:

```
private String paragraph = "Let's pause, \nand then reflect.";
```

Использование класса Tokenizer из библиотеки OpenNLP

В библиотеке OpenNLP имеется интерфейс Tokenizer, реализацию которого обеспечивают три класса: SimpleTokenizer, TokenizerME и WhitespaceTokenizer. Интерфейс поддерживает два метода:

- tokenize() – принимает обрабатываемую строку и возвращает массив токенов, представленных в виде отдельных строк;
- tokenizePos() – принимает строку и возвращает массив объектов типа Span. Класс Span используется для определения начального и конечного смещений токенов относительно начала исходной строки.

В следующих разделах рассматривается применение каждого из вышеупомянутых классов.

Использование класса SimpleTokenizer

По имени класса SimpleTokenizer очевидно, что он выполняет простую токенизацию текста. Поле INSTANCE используется для создания экземпляра данного класса, как показано в коде ниже. Затем вызывается метод tokenize() с аргументом paragraph, определенным выше, и выводятся обнаруженные токены:

```
SimpleTokenizer simpleTokenizer = SimpleTokenizer.INSTANCE;
String tokens[] = simpleTokenizer.tokenize(paragraph);
for(String token : tokens) {
    System.out.println(token);
}
```

Этот пример выводит следующее:

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

При использовании этого токенизатора знаки пунктуации возвращаются как отдельные токены.

Использование класса WhitespaceTokenizer

Имя класса `WhitespaceTokenizer` позволяет понять, что в качестве разделителей он использует пробельные символы. В следующем примере в одну инструкцию объединено создание экземпляра данного класса и вызов метода `tokenize()`, потоком ввода для которого является все та же переменная `paragraph`. Вывод токенов осуществляется с помощью оператора `for-each`:

```
String tokens[] =
WhitespaceTokenizer.INSTANCE.tokenize(paragraph);
for(String token : tokens) {
   System.out.println(token);
}
```

Результат выполнения:

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

Несмотря на то что этот класс не поддерживает разделения стяженных форм слов, отделение знаков пунктуации и прочих подобных элементов текста, он может быть полезен в некоторых приложениях. В классе `WhitespaceTokenizer` также имеется метод `tokenizePos()`, возвращающий числовые значения границ токенов.

Использование класса TokenizerME

Для токенизации класс `TokenizerME` использует методику *максимизации энтропии* (*Maximum Entropy – maxent – ME*) и статистическую модель. Модель максимизации энтропии, или maxent-модель, применяется для определения взаимосвязей между данными, в нашем случае – взаимосвязей в тексте. Некоторые источники текстов, такие как разнообразные социальные сети, практически не имеют какого-либо определенного формата, содержат огромное количество жаргонных и просторечных выражений, а также специальные символы, например комбинации символов, выражающих эмоции. Статистический токенизатор на основе maxent-модели улучшает качество токенизации.

В данной книге невозможно привести подробное описание модели максимизации энтропии из-за ее сложности. Заинтересованным читателям для начала можно предложить статью в Википедии http://en.wikipedia.org/w/index.php?title=Multinomial_logistic_regression&redirect=no.

Реализация maxent-модели скрыта в классе `TokenizerModel`. Этот класс используется для создания экземпляра токенизатора. Модель требует обязательного предварительного обучения. Следующий пример создает экземпляр токенизатора на основе модели в файле *en-token.bin*. Затем модель проходит обучение на простом английском тексте.

Местоположение файла модели возвращает метод `getModelDir()`, это необходимо для включения модели в работу. Возвращаемое значение зависит от фактического местоположения моделей в вашей системе. Многие из моделей, используемых в книге, можно найти на сайте http://opennlp.sourceforge.net/models-1.5/.

После создания экземпляра класса `FileInputStream` он передается конструктору `TokenizerModel`. Метод `tokenize()` генерирует массив строк-токенов. После этого токены выводятся в цикле:

```
try {
    InputStream modelInputStream = new FileInputStream(
        new File(getModelDir(), "en-token.bin"));
    TokenizerModel model = new TokenizerModel(modelInputStream);
    Tokenizer tokenizer = new TokenizerME(model);
    String tokens[] = tokenizer.tokenize(paragraph);
    for(String token : tokens) {
        System.out.println(token);
    }
} catch(IOException ex) {
    // Обработка исключения.
}
```

Результат выполнения кода:

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

Использование токенизатора из библиотеки Stanford

Процесс токенизации поддерживается несколькими классами в библиотеке Stanford NLP. Ниже перечислены некоторые из них:

- класс PTBTokenizer;
- класс DocumentPreprocessor;
- класс StanfordCoreNLP как конвейер.

Примеры в следующих разделах снова будут использовать строку paragraph, которая была определена ранее.

Использование класса PTBTokenizer

Этот класс имитирует работу токенизатора *Penn Treebank 3* (*PTB*) (http://www.cis.upenn.edu/~treebank/). Он отличается от PTB некоторыми дополнительными возможностями и поддержкой символов Unicode. Класс PTBTokenizer включает несколько устаревших конструкторов, но по умолчанию предполагается использование конструктора с тремя аргументами. Этот конструктор принимает объект Reader, аргумент LexedTokenFactory<T> и строку с дополнительными параметрами.

Реализация интерфейса LexedTokenFactory выполняется классами CoreLabelTokenFactory и WordTokenFactory. Первый поддерживает сохранение номеров позиций начального и конечного символов токена, второй просто возвращает токен в виде строки без какой-либо информации о позициях его символов. По умолчанию применяется класс WordTokenFactory. Мы рассмотрим использование обоих классов.

В следующем примере рассматривается применение класса CoreLabelTokenFactory. Он создает экземпляр класса StringReader на основе строки paragraph. Последний аргумент, предназначенный для передачи дополнительных параметров, в данном примере имеет значение null. Реализация интерфейса Iterator в классе PTBTokenizer позволяет применять методы hasNext() и next() для вывода токенов.

```
PTBTokenizer ptb = new PTBTokenizer(new StringReader(paragraph),
    new CoreLabelTokenFactory(), null);
while(ptb.hasNext()) {
    System.out.println(ptb.next())
}
```

Результат выполнения выглядит так:

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

Точно такой же результат можно получить при использовании класса `WordTokenFactory`:

```
PTBTokenizer ptb = new PTBTokenizer(new StringReader(paragraph),
    new WordTokenFactory(), null);
```

Богатыми возможностями класса `CoreLabelTokenFactory` можно воспользоваться с помощью третьего аргумента конструктора `PTBTokenizer`. Передаваемые в нем параметры настройки позволяют управлять режимом работы токенизатора. Например, можно установить режим обработки кавычек, способ интерпретации многоточий, также выбрать режим проверки правописания по правилам британского английского или американского английского языка. Полный список параметров настройки можно найти на сайте http://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/process/PTBTokenizer.html.

В следующем примере объект `PTBTokenizer` создается с использованием переменной `ctf`, имеющей тип `CoreLabelTokenFactory`, кроме того, в третьем аргументе передается параметр настройки `invertible=true`, позволяющий работать с объектом `CoreLabel`, возвращающим начальную и конечную позиции каждого токена:

```
CoreLabelTokenFactory ctf = new CoreLabelTokenFactory();
PTBTokenizer ptb = new PTBTokenizer(new StringReader(paragraph),
    ctf, "invertible=true");
while(ptb.hasNext()) {
    CoreLabel cl = (CoreLabel)ptb.next();
    System.out.println(cl.originalText() + " (" +
        cl.beginPosition() + "-" + cl.endPosition() + ")");
}
```

Ниже приводится результат выполнения этого кода. Числовые значения в скобках обозначают начальную и конечную позиции каждого токена:

```
Let (0-3)
's (3-5)
pause (6-11)
, (11-12)
and (14-17)
then (18-22)
reflect (23-30)
. (30-31)
```

Использование класса DocumentPreprocessor

Класс `DocumentPreprocessor` выполняет токенизацию данных, поступающих из потока ввода. Кроме того, он реализует интерфейс

Iterable, упрощая перемещение по обрабатываемой последовательности. Данный токенизатор поддерживает обработку простого текста и данных в формате XML.

Для демонстрации воспользуемся классом StringReader, который инициализируется строковой переменной paragraph:

```
Reader reader = new StringReader(paragraph);
```

Затем создается экземпляр класса DocumentPreprocessor:

```
DocumentPreprocessor documentPreprocessor =
      new DocumentPreprocessor(reader);
```

Класс DocumentPreprocessor реализует интерфейс Iterable<java.util.List<HasWord>>. Интерфейс HasWord определяет два метода для работы со словами: setWord() и word(). Второй возвращает слово как отдельную строку. В следующем примере объект класса DocumentPreprocessor разделяет исходный текст на предложения, сохраняемые в виде списка List<HasWord>. Объект Iterator используется для извлечения очередного предложения из списка, после чего отдельные токены выводятся в цикле for-each:

```
Iterator<List<HasWord>> it = documentPreprocessor.iterator();
while(it.hasNext()) {
    List<HasWord> sentence = it.next;
    for(HasWord token : sentence) {
        System.out.println(token);
    }
}
```

При выполнении этот пример дает следующий результат:

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

Использование конвейера

В этом разделе используется класс StanfordCoreNLP, представленный в главе 1 «Основы обработки естественного языка». Но здесь применяется более простая строка-аннотатор. Как показано ниже, сначала создается объект Properties, а затем ему передаются аннотаторы tokenize и ssplit.

Аннотатор tokenize указывает, что должна быть выполнена токенизация, а ssplit отвечает за разделение предложений:

```
Properties properties = new Properties();
properties.put("annotators", "tokenize, ssplit");
```

Далее создаются экземпляры классов StanfordCoreNLP и Annotation:

```
StanfordCoreNLP pipeline = new StanfordCoreNLP(properties);
Annotation annotation = new Annotation(paragraph);
```

Метод annotate() выполняет токенизацию текста, а метод prettyPrint() выводит отдельные токены:

```
pipeline.annotate(annotation);
pipeline.prettyPrint(annotation, System.out);
```

В результате на экран выводятся некоторые статистические данные, за которыми следуют токены с обозначениями их позиций в исходном тексте:

```
Sentence #1 (8 tokens):

Let's pause,

and then reflect.

[Text=Let CharacterOffsetBegin=0 CharacterOffsetEnd=3]
[Text='s CharacterOffsetBegin=3 CharacterOffsetEnd=5]
[Text=pause CharacterOffsetBegin=6 CharacterOffsetEnd=11]
[Text=, CharacterOffsetBegin=11 CharacterOffsetEnd=12]
[Text=and CharacterOffsetBegin=14 CharacterOffsetEnd=17]
[Text=then CharacterOffsetBegin=18 CharacterOffsetEnd=22]
[Text=reflect CharacterOffsetBegin=23 CharacterOffsetEnd=30]
[Text=. CharacterOffsetBegin=30 CharacterOffsetEnd=31]
```

Использование токенизаторов из библиотеки LingPipe

Библиотека LingPipe включает несколько токенизаторов. В этом разделе будет показано использование класса IndoEuropeanTokenizerFactory. В последующих разделах демонстрируются другие способы токенизации библиотекой LingPipe. Поле INSTANCE представляет экземпляр токенизатора для группы индоевропейских языков. Метод tokenizer() возвращает экземпляр класса Tokenizer, создаваемый на основе обрабатываемого текста, как показано ниже:

```
char text[] = paragraph.toCharArray();
TokenizerFactory tokenizerFactory =
IndoEuropeanTokenizerFactory.INSTANCE;
Tokenizer tokenizer = tokenizerFactory.tokenizer(text, 0,
```

```
text.length);
for(String token : tokenizer) {
    System.out.println(token);
}
```

Результат выполнения:

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

Эти токенизаторы поддерживают токенизацию простого, обычного текста. В следующем разделе будет показано, как обучить токенизатор для работы со специализированным текстом.

Обучение токенизатора поиску заданных элементов текста

Необходимость обучения токенизатора возникает, когда встречается текст, который невозможно правильно обработать стандартными токенизаторами. Но не следует спешить с созданием нового специализированного токенизатора, потому что можно создать модель, которая в полной мере соответствует данному конкретному случаю.

Чтобы показать, как создать такую модель, будем читать специально подготовленные данные из файла и на этих данных обучать модель. Данные хранятся в виде последовательностей слов, разделенных пробелами и полями <SPLIT>. Поле <SPLIT> предоставляет дополнительную информацию о том, как должны определяться токены. Оно помогает распознавать границы между числами, такими как *23.6*, и знаками пунктуации, такими как точки и запятые. Обучающие данные хранятся в файле *training-data.train* и имеют следующий вид (это два предложения из текущего абзаца на английском языке):

```
These fields are used to provide further information about how tokens
should be identified<SPLIT>.
They can help identify breaks between numbers<SPLIT>, such as
23.6<SPLIT>, punctuation characters such as commas<SPLIT>.
```

Конечно, эти данные не являются специализированным, сложным текстом, они лишь позволяют наглядно показать, как разметить обучающий текст, и продемонстрировать процесс обучения модели.

Для создания модели используется перегруженный метод train() из класса TokenizerME библиотеки OpenNLP. Последние два параметра метода требуют более подробного объяснения. Для определения взаимосвязей между элементами текста используется модель максимизации энтропии (maxent).

Можно определить, сколько раз модель должна распознать признак, чтобы включить его в модель. Такие признаки можно считать характеристиками модели. Количество итераций означает, сколько раз необходимо выполнить процедуру обучения для определения всех параметров модели. Некоторые параметры класса TokenME приведены в табл. 2.6.

Таблица 2.6. Некоторые параметры класса TokenME

Тип параметра	Краткое описание
String	Код используемого языка
ObjectStream <TokenSample>	Параметр типа ObjectStream содержит обучающие данные
boolean	Если передается значение true, алфавитно-цифровые данные игнорируются
int	Определяет, сколько раз признак должен быть обработан
int	Определяет количество итераций обучения модели

Следующий пример начинается с определения объекта BufferedOutputStream, который будет использован для хранения новой модели. Некоторые из методов в этом примере могут генерировать исключения, поэтому необходимо обеспечить обработку последних в блоках catch:

```
BufferedOutputStream modelOutputStream = null;
try {
    ...
} catch(UnsupportedEncodingException ex) {
    // Обработка данного исключения.
} catch(IOException) {
    // Обработка данного исключения.
}
```

Экземпляр класса ObjectStream создается с использованием класса PlainTextByLineStream. В качестве аргументов конструктор принимает имя файла с обучающими данными и схему кодировки символов. Созданный объект используется для создания другого экземпляра ObjectStream для объектов типа TokenSample. Эти объекты содержат

текст с включенной в него информацией о позициях, занимаемых токенами:

```
ObjectStream<String> lineStream = new PlainTextByLineStream(
    new FileInputStream("training-data.train"), "UTF-8");
ObjectStream<TokenSample> sampleStream =
    new TokenSampleStream(lineStream);
```

Теперь можно воспользоваться методом `train()`, как показано ниже. Здесь указывается, что обрабатывается текст на английском языке. Алфавитно-цифровая информация игнорируется. Признак должен быть обработан не менее 5 раз, а количество итераций устанавливается равным 100.

```
TokenizerModel model = TokenizerME.train("en", sampleStream, true, 5, 100);
```

Описание параметров метода `train()` приводится в табл. 2.7.

Таблица 2.7. Описание параметров метода train()

Параметр	Описание
Код языка	Строка, определяющая используемый естественный язык
Образцы	Образец текста
Оптимизация обработки алфавитно-цифровых данных	Если передается значение `true`, алфавитно-цифровые данные пропускаются
Пороговое количество обработок признака	Определяет, сколько раз признак должен быть обработан
Количество итераций	Определяет количество итераций обучения модели

В следующем фрагменте создается поток вывода, затем обученная модель записывается в файл *mymodel.bin*. После этого модель готова к практическому применению:

```
BufferedOutputStream modelOutputStream = new BufferedOutputStream(
    new FileOutputStream(new File("mymodel.bin")));
model.serialize(modelOutputStream);
```

Здесь мы не будем подробно обсуждать полученные результаты. Тем не менее очень важно понаблюдать за процессом обучения, промежуточные результаты которого приведены ниже. Последний раздел сокращен – из него удалена большая часть шагов для экономии места:

```
Indexing events using cutoff of 5

Dropped event F:[p=2, s=3.6,, p1=2, p1_num, p2=bok, p1f1=23, f1=3,
f1_num, f2=., f2_eos, f12=3.]
```

```
Dropped event F:[p=23, s=.6, p1=3, p1_num, p2=2, p2_num, p21=23,
p1f1=3., f1=., f1_eos, f2=6, f2_num, f12=.6]
Dropped event F:[p=23., s=6,, p1=., p1_eos, p2=3, p2_num, p21=3.,
p1f1=.6, f1=6, f1_num, f2=,, f12=6,]
  Computing event counts... done. 27 events
  Indexing... done.
Sorting and merging events... done. Reduced 23 events to 4.
Done indexing.
Incorporated indexed data for training... done.
  Number of Event Tokens: 4
      Number of Outcomes: 2
    Number of Predicates: 4
...done.
Computing model parameters...
Performing 100 iterations.
  1: ...loglikelihood=-15.942385152878742  0.8695652173913043
  2: ...loglikelihood=-9.223608340603953  0.8695652173913043
  3: ...loglikelihood=-8.222154969329086  0.8695652173913043
  4: ...loglikelihood=-7.885816898591612  0.8695652173913043
  5: ...loglikelihood=-7.674336804488621  0.8695652173913043
  6: ...loglikelihood=-7.494512270303332  0.8695652173913043
Dropped event T:[p=23.6, s=,, p1=6, p1_num, p2=., p2_eos, p21=.6,
p1f1=6,, f1=,, f2=bok]
  7: ...loglikelihood=-7.327098298508153  0.8695652173913043
  8: ...loglikelihood=-7.1676028756216965  0.8695652173913043
  9: ...loglikelihood=-7.014728408489079  0.8695652173913043
...
100: ...loglikelihood=-2.3177060257465376  1.0
```

В следующем примере показано практическое применение обученной модели с использованием методики, описанной в разделе «Использование класса TokenizerME». Различаются лишь применяемые модели:

```
try {
    paragraph = "A demonstration of how to train a tokenizer.";
    InputStream modelIn = new FileInputStream(
        new File(".", "mymodel.bin"));
    TokenizerModel model = new TokenizerModel(modelIn);
    Tokenizer tokenizer = new TokenizerME(model);
    String tokens[] = tokenizer.tokenize(paragraph);
    for(String token : tokens) {
        System.out.println(token);
    }
} catch(IOException ex) {
    ex.printStackTrace();
}
```

Ниже показан результат:

```
A
demonstration
```

```
of
how
to
train
a
tokenizer
.
```

Сравнение токенизаторов

В табл. 2.8 приводится краткое сравнение результатов работы токенизаторов из библиотек NLP. Токены, сгенерированные на основе одного и того же текста `"Let's pause, \nand then reflect."`, размещены под именем каждого токенизатора. Следует помнить, что приведенные результаты получены при простых вариантах использования рассматриваемых классов. В примеры не включались дополнительные параметры, которые, несомненно, повлияли бы на способ генерации токенов, поскольку преследовалась основная цель – показать тип выводимого результата, ожидаемого при выполнении простого кода, который использует простые данные.

Таблица 2.8. Сравнение результатов работы токенизаторов из библиотек NLP

Simple Tokenizer	Whitespace Tokenizer	Tokenizer-ME	PTB-Tokenizer	Document Preprocessor	IndoEuropean Tokenizer-Factory
Let	Let's	Let	Let	Let	Let
‘	pause,	‘s	‘s	‘s	‘
s	and	pause	pause	pause	s
pause	then	,	,	,	pause
,	reflect.	and	and	and	,
and		then	then	then	and
then		reflect	reflect	reflect	then
reflect		.	.	.	reflect
.					.

Нормализация

Нормализация – это процесс преобразования произвольного списка слов в более однородную последовательность. Такое преобразование может выполняться в ходе подготовки текста к дальнейшей обработке. Кроме преобразования слов в стандартный формат, возможны и другие операции с данными, но при условии, что они не оказывают от-

рицательного влияния на весь процесс обработки в целом. Например, перевод букв всех слов в нижний регистр упрощает процесс поиска.

Процесс нормализации может улучшить процедуру поиска совпадений с образцом в тексте. Например, специальный термин «modem router» может быть записан несколькими способами: «modem and router», «modem & router», «modem/router», «modem-router». Нормализация этого словосочетания, то есть перевод в общеупотребительную форму, существенно упрощает поиск и предоставление правильной информации потенциальному покупателю.

В то же время важно понимать, что нормализация способна оказать и отрицательное влияние на решение задачи обработки естественного языка. В тех случаях, когда регистр букв важен, преобразование в нижний регистр, вероятнее всего, снизит корректность процедур поиска и достоверность их результатов.

К операциям нормализации относятся следующие действия:

- преобразование символов (букв) в нижний регистр;
- развертывание аббревиатур и сокращений;
- удаление шумовых слов;
- стемминг и лемматизация.

В этом разделе мы рассмотрим перечисленные методики, за исключением процедуры развертывания аббревиатур и сокращений, так как она имеет много общего с методикой, используемой для удаления шумовых слов, только аббревиатуры не удаляются, а заменяются соответствующей полной версией.

Преобразование букв в нижний регистр

Преобразование символов (букв) текста в нижний регистр – это простая операция, позволяющая улучшить качество результатов поиска. Для такого преобразования можно воспользоваться штатными средствами языка Java, например методом `toLowerCase()` класса `String`, или обратиться к библиотекам NLP, например к классу `LowerCaseTokenizerFactory` из библиотеки LingPipe. Вот пример применения метода `toLowerCase()`:

```
String text = "A Sample string with acronyms, IBM, and UPPER "
  + "and lowercase letters.";
String result = text.toLowerCase();
System.out.println(result);
```

Результат выполнения:

```
a sample string with acronyms, ibm, and upper and lowercase letters.
```

Применение класса LowerCaseTokenizerFactory из библиотеки LingPipe будет показано в разделе «Нормализация с использованием конвейера» ниже.

Удаление шумовых слов

Существует несколько способов удаления шумовых слов. Самый простой состоит в создании класса, содержащего шумовые слова, предназначенные для удаления. Некоторые библиотеки NLP также предоставляют поддержку удаления шумовых слов. Первый способ будет продемонстрирован ниже на примере создания простого класса StopWords. Затем будет реализована методика на основе специализированного класса EnglishStopTokenizerFactory из библиотеки LingPipe.

Создание класса StopWords

Процесс удаления шумовых слов включает просмотр потока токенов, сравнение их со списком шумовых слов и удаление совпадений из входного потока. Для демонстрации этой методики будет создан простой класс, поддерживающий основные операции, перечисленные в табл. 2.9.

Таблица 2.9. Описание методов класса StopWords

Конструктор/Метод	Описание
Конструктор по умолчанию	Используется для установки набора шумовых слов по умолчанию
Конструктор с одним аргументом	Используется для установки набора шумовых слов из заданного файла
addStopWord	Добавляет новое шумовое слово во внутренний список класса
removeStopWord	Принимает массив слов и возвращает новый массив, в котором удалены шумовые слова

В следующем примере создается класс StopWords с двумя внутренними переменными. Переменная defaultStopWords – это массив со списком шумовых слов по умолчанию. Переменная stopWords имеет тип HashSet и используется для хранения шумовых слов в процессе обработки:

```
public class StopWords {
    private String[] defaultStopWords = {"i", "a", "about", "an", "are",
      "as", "at", "be", "by", "com", "for", "from", "how", "in", "is",
      "it", "of", "on", "or", "that", "the", "this", "to", "was",
```

```
      "what", "when", "where", "who", "will", "with"};

    private static HashSet stopWords = new HashSet();
    ...
}
```

В этом классе определяются два конструктора, предназначенные для заполнения списка типа `HashSet`:

```
public StopWords() {
    stopWords.addAll(Arrays.asList(defaultStopWords));
}

public StopWords(String fileName) {
    try {
        BufferedReader bufferedReader = new BufferedReader(
                new FileReader(fileName));
        while(bufferedReader.ready()) {
            stopWords.add(bufferedReader.readLine());
        }
    } catch(IOException ex) {
        ex.printStackTrace();
    }
}
```

Метод `addStopWord()` позволяет расширять список шумовых слов:

```
public void addStopWord(String word) {
    stopWords.add(word);
}
```

Метод `removeStopWords()` используется для удаления шумовых слов из обрабатываемого текста. В нем создается временный список `ArrayList`, содержащий слова, переданные в метод. Цикл `for` используется для удаления шумовых слов из этого списка. Метод `contains()` позволяет проверить, является ли рассматриваемое слово шумовым, перед удалением. Затем выполняется преобразование списка `ArrayList` в массив строк, который возвращается из метода как результат. Все это происходит в следующем фрагменте кода:

```
public String[] removeStopWords(String[] words) {
    ArrayList<String> tokens =
        new ArrayList<String>(Arrays.asList(words));
    for(int i=0; i < tokens.size(); i++) {
        if(stopWords.contains(tokens.get(i))) {
           tokens.remove(i);
        }
    }
    return (String[])tokens.toArray(
        new String[tokens.size());
}
```

Ниже демонстрируется практическое применение класса StopWords. Сначала создается экземпляр этого класса вызовом конструктора по умолчанию. Затем определяются токенизатор на основе класса SimpleTokenizer из библиотеки OpenNLP и переменная, содержащая обрабатываемый текст:

```
StopWords stopWords = new StopWords();
SimpleTokenizer simpleTokenizer = SimpleTokenizer.INSTANCE;
String paragraph = "A simple approach is to create a class "
    + "to hold and remove stopwords.";
```

Выполняется токенизация предложенного текста, и результат передается в метод removeStopWords(). После этого выводится измененный список:

```
String tokens[] = simpleTokenizer.tokenize(paragraph);
String list[] = stopWords.removeStopWords(tokens);
for(String word : list) {
    System.out.println(word);
}
```

Ниже приводится результат выполнения. Артикль «A» не был удален, потому что записан в верхнем регистре, а наш класс не выполняет преобразования регистров букв.

```
A
simple
approach
create
class
hold
remove
stopwords
.
```

Использование библиотеки LingPipe для удаления шумовых слов

В библиотеке LingPipe имеется класс EnglishStopTokenizerFactory, которым мы воспользуемся для обнаружения и удаления шумовых слов. Список шумовых слов для работы этого класса можно найти в документации: http://alias-i.com/lingpipe/docs/api/com/aliasi/tokenizer/EnglishStopTokenizerFactory.html. В список включены такие слова, как: «a», «was», «but», «he», «for».

Конструктор класса EnglishStopTokenizerFactory требует экземпляр класса TokenizerFactory. Его метод tokenizer() используется для обра-

ботки списка слов и удаления шумовых слов. Сначала определяется обрабатываемая строка текста:

```
String paragraph = "A simple approach is to create a class "
    + "to hold and remove stopwords.";
```

Далее создается экземпляр класса TokenizerFactory на основе класса IndoEuropeanTokenizerFactory. Затем созданный объект factory используется как аргумент при создании экземпляра EnglishStopTokenizerFactory:

```
TokenizerFactory factory = IndoEuropeanTokenizerFactory.INSTANCE;
factory = new EnglishStopTokenizerFactory(factory);
```

Текст в переменной paragraph обрабатывается с использованием класса Tokenizer из библиотеки LingPipe и метода фабрики tokenizer(), который принимает массив типа char, начальный индекс и длину массива:

```
Tokenizer tokenizer = factory.tokenizer(paragraph.toCharArray(),
    0, paragraph.length());
```

Оператор for-each позволяет выполнить обход обработанного списка:

```
for(String token : tokenizer) {
    System.out.println(token);
}
```

Результат точно такой же, как при использовании класса StopWords:

```
A
simple
approach
create
class
hold
remove
stopwords
.
```

Обратите внимание, что и в этом случае артикль «A» не был удален, хотя является шумовым словом. Причина та же – при обработке не выполняется преобразование символов в нижний регистр, а в списке шумовых слов артикль хранится в нижнем регистре 'a'. В итоге шумовое слово не было обработано. Этот недостаток будет исправлен в подразделе «Нормализация с использованием конвейера» далее в текущей главе.

Использование стемминга

При поиске *основы слова* (*stem*) удаляются все приставки и суффиксы, а оставшаяся часть классифицируется как основа. Определение основы необходимо при решении задач, в которых самым важным является обнаружение похожих слов. Например, поставлена задача поиска всех вхождений слова «book» в любых формах. Заданную последовательность символов могут содержать слова «books», «booked», «bookings», «bookmark» и т. п. Поэтому удобнее определить основу слова и искать вхождения этой основы в документ. Во многих случаях такой подход может улучшить качество поиска.

Инструмент поиска основы (stemmer) может дать такую основу, которая не является обычным, общеупотребительным словом. Например, для слов «bounties», «bounty», «bountiful» может быть принято решение определить основу как «bounti». Это также может оказаться полезным при поиске.

Со стеммингом тесно связан процесс лемматизации (lemmatization), который определяет начальные (основные) формы слов, обозначаемые термином «лемма» (lemma), из которых обычно составляются словари. Лемматизация также может улучшать качество некоторых видов поиска. Стемминг часто считается более простой методикой, поскольку для получения основы от слова «отсекаются» его начальные и конечные части (приставки и суффиксы).

Лемматизация представляет собой более сложную и тонкую методику, в соответствии с которой определяется морфологическое или словарное значение (и форма) токена, то есть собственно лемма. Например, слово «having» имеет основу «hav», но лемму «have». Могут встречаться и более сложные случаи: у слов «was» и «been» совершенно разные основы, но одна и та же лемма «be».

Часто лемматизация использует больше вычислительных ресурсов, чем стемминг. Оба этих процесса являются полноправными компонентами технологии обработки естественного языка, а обоснованность их применения и КПД частично определяются самой решаемой задачей.

Использование инструмента стемминга Porter Stemmer

Porter Stemmer – это широко используемый инструмент стемминга для текстов на английском языке. Более подробную информацию о нем можно получить на сайте http://tartarus.org/martin/PorterStemmer/. Процесс определения основы слова по методике Porter Stemmer состоит из пяти шагов.

Несмотря на то что библиотека Apache OpenNLP 1.5.3 не содержит класса `PorterStemmer`, исходный код этого класса можно скачать здесь: https://svn.apache.org/repos/asf/opennlp/trunk/opennlp-tools/src/main/

java/opennlp/tools/stemmer/PorterStemmer.java – и включить в свой проект.

В следующем примере показана работа объекта класса `PorterStemmer` с массивом слов. Организовать поток ввода можно без труда из любого другого источника. Сначала создается экземпляр класса `PorterStemmer`, затем его метод `stem()` применяется к каждому слову из массива:

```
String words[] = {"bank", "banking", "banks", "banker", "banked",
    "bankart"};
PorterStemmer ps = new PorterStemmer();
for(String word : words) {
    String stem = ps.stem(word);
    System.out.println("Word: " + word + "   Stem: " + stem);
}
```

После выполнения получим следующий результат:

```
Word: bank   Stem: bank
Word: banking   Stem: bank
Word: banks   Stem: bank
Word: banker   Stem: banker
Word: banked   Stem: bank
Word: bankart   Stem: bankart
```

Последнее слово используется в сочетании со словом «lesion» – это «симптом Банкарта» (Bankart lesion), описывающий повреждение плечевого сустава и не имеющий ничего общего с предыдущими словами в массиве. В список он включен, только чтобы показать, что при поиске основы слова используются лишь общеупотребительные приставки и суффиксы.

Некоторые другие методы класса `PorterStemmer`, которые могут оказаться полезными, перечислены в табл. 2.10.

Таблица 2.10. Методы класса PorterStemmer

Метод	Описание
`add`	Добавляет символ типа `char` в конец текущей рассматриваемой основы слова
`stem`	Метод, вызванный без аргументов, возвращает значение `true`, если обнаружена другая основа
`reset`	Восстанавливает начальное состояние стеммера, что позволяет использовать другое слово

Стемминг с использованием библиотеки LingPipe

Класс `PorterStemmerTokenizerFactory` используется для поиска основ слов средствами библиотеки LingPipe. В следующем примере

обрабатывается тот же массив, что и в предыдущем разделе. Класс IndoEuropeanTokenizerFactory необходим для выполнения предварительной токенизации, после чего начинает работу объект типа PorterStemmer. Эти классы определяются в первую очередь:

```
TokenizerFactory tokenizerFactory =
IndoEuropeanTokenizerFactory.INSTANCE;
TokenizerFactory porterFactory =
    new PorterStemmerTokenizerFactory(tokenizerFactory);
```

Далее объявляется массив для хранения основ слов. Мы повторно используем массив words, объявленный в предыдущем разделе. Каждое слово обрабатывается отдельно. Выполняется токенизация слова, и его основа сохраняется в массиве stems, как показано ниже. Слова и их основы выводятся последовательно:

```
String[] stems = new String[words.length];
for(int i=0; i < words.length; i++) {
    Tokenization tokenizer = new Tokenization(words[i], porterFactory);
    stems = tokenizer.tokens();
    System.out.print("Word: " + words[i]);
    for(String stem : stems) {
        System.out.println("   Stem: " + stem);
    }
}
```

Этот пример выводит следующее:

```
Word: bank   Stem: bank
Word: banking   Stem: bank
Word: banks   Stem: bank
Word: banker   Stem: banker
Word: banked   Stem: bank
Word: bankart   Stem: bankart
```

Итак, на примерах было продемонстрировано применение методики Porter Stemmer, реализованной в библиотеках OpenNLP и LingPipe. Следует отметить, что существуют другие типы инструментов стемминга, в том числе библиотека NGrams и различные методики, объединяющие вероятностный и алгоритмический подходы.

Использование лемматизации

Лемматизация поддерживается несколькими библиотеками NLP. В этом разделе будет показано выполнение лемматизации с использованием классов StanfordCoreNLP и OpenNLPLemmatizer. Процесс лемматизации определяет основную, или начальную, форму слова, то есть

лемму. Лемму можно считать словарной формой слова. Например, для слова «was» леммой является форма «be».

Использование класса StanfordLemmatizer

Покажем процесс лемматизации с применением конвейера на основе класса `StanfordCoreNLP`. Сначала создадим объект-конвейер с четырьмя аннотаторами, включая аннотатор `lemma`:

```
StanfordCoreNLP pipeline;
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, pos, lemma");
pipeline = new StanfordCoreNLP(props);
```

Смысл этих и некоторых других аннотаторов кратко описан в табл. 2.11.

Таблица 2.11. Краткое описание аннотаторов, используемых при создании конвейера

Аннотатор	Выполняемая операция
`tokenize`	Токенизация
`ssplit`	Разделение предложений
`pos`	Расстановка тегов, соответствующих частям речи
`lemma`	Лемматизация
`ner`	Распознавание и идентификация именованных объектов
`parse`	Синтаксический анализ (парсинг)
`dcoref`	Установление кореферентности (референциального тождества)

Переменная `paragraph` используется при вызове конструктора класса `Annotation`, затем выполняется метод `annotate()`:

```
String paragraph = "Similar to stemming is Lemmatization. "
    + "This is the process of finding its lemma, its form "
    + "as found in a dictionary.";
Annotation document = new Annotation(paragraph);
pipeline.annotate(document);
```

Теперь необходимо выполнить обход предложений и токенов в этих предложениях. Методы `get()` классов `Annotation` и `CoreMap` возвращают значения заданного типа. Если значения заданного типа отсутствуют, возвращается `null`. Упомянутые классы будут использоваться для получения списка лемм.

Сначала возвращается список предложений, затем каждое слово из каждого предложения обрабатывается для определения соответст-

вующей ему леммы. Для хранения предложений и лемм необходимо объявить списки:

```
List<CoreMap> sentences =
    document.get(SentencesAnnotation.class);
List<String> lemmas = new LinkedList<>();
```

Два оператора `for-each` позволяют организовать последовательный обход предложений, чтобы заполнить список лемм. После завершения всех итераций выводится сформированный список лемм:

```
for(CoreMap sentence : sentences) {
    for(CoreLabel word : sentence.get(TokensAnnotation.class)) {
        lemmas.add(word.get(LemmaAnnotation.class));
    }
}

System.out.print("[");
for(String element : lemmas) {
    System.out.print(element + " ");
}
System.out.println("]");
```

Результат выполнения примера будет таким:

```
[similar to stem be lemmatization . this be the process of find its lemma
, its form as find in a dictionary . ]
```

Если сравнить результат с исходным текстом, то можно по достоинству оценить проделанную работу:

```
Similar to stemming is Lemmatization. This is the process of finding its
lemma, its form as found in a dictionary.
```

Поддержка лемматизации в библиотеке OpenNLP

Библиотека OpenNLP также обеспечивает поддержку лемматизации с использованием класса `JWNLDictionary`. Конструктор этого класса принимает строку, содержащую путь к файлам словарей, предназначенных для определения корней и начальных форм слов. Мы будем использовать словарь WordNet, разработанный в Принстонском университете (Princeton University) (wordnet.princeton.edu). В действительности словарь представляет собой набор файлов, хранящихся в одном каталоге. Файлы содержат списки слов и соответствующих им корней. В примере ниже используется словарь https://code.google.com/p/xssm/downloads/detail?name=SimilarityUtils.zip&can=2&q=.

В классе JWNLDictionary имеется метод getLemmas(), принимающий обрабатываемое слово. Второй параметр метода определяет часть речи, соответствующую переданному слову. Для получения точных результатов очень важно правильно определять часть речи для каждого обрабатываемого слова.

В следующем фрагменте создается экземпляр класса JWNLDictionary с передачей ему пути, заканчивающегося каталогом \\dict\\. Это местоположение словаря. Кроме того, определяется текст, подлежащий обработке. Конструктор может сгенерировать исключения IOException и JWNLException, поэтому код необходимо поместить в блок try-catch:

```
try {
    dictionary = new JWNLDictionary("...\\dict\\");
    paragraph = "Eat, drink, and be merry, for life is but a dream";
    ...
} catch(IOException | JWNLException ex) {
    // Обработка исключения.
}
```

Все дальнейшие инструкции следует добавлять после инициализации строковой переменной paragraph в блоке try. Сначала выполним токенизацию строки с помощью экземпляра класса WhitespaceTokenizer по методике, описанной в разделе «Использование класса WhitespaceTokenizer». Затем передадим в метод getLemmas() каждый токен и пустую строку вместо обозначения части речи. После этого выведем исходный токен и его лемму:

```
String tokens[] =
    WhitespaceTokenizer.INSTANCE.tokenize(paragraph);
for(String token : tokens) {
    String[] lemmas = dictionary.getLemmas(token, "");
    for(String lemma : lemmas) {
        System.out.println("Token: " + token + "   Lemma: " + lemma);
    }
}
```

Выполнение приведенного выше примера дает следующий результат:

```
Token: Eat,   Lemma: at
Token: drink,   Lemma: drink
Token: be   Lemma: be
Token: life   Lemma: life
Token: is   Lemma: is
Token: is   Lemma: i
Token: a   Lemma: a
Token: dream   Lemma: dream
```

В целом процесс лемматизации прошел нормально, за исключением обработки токена «is», из которого получилось две леммы, причем вторая лемма неверна. Это наглядно показывает важность использования правильного признака части речи для каждого токена. Можно было бы передавать методу `getLemmas()` один или несколько тегов, обозначающих части речи, во втором аргументе. Но при этом сразу возникает вопрос: как правильно определить части речи? Эта тема будет подробно рассматриваться в главе 5 «Определение частей речи».

Сокращенный список тегов, обозначающих части речи в английском языке, приводится в табл. 2.12. Этот список взят с сайта https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html. Полный список под названием «The University of Pennsylvania (Penn) Treebank Tagset» можно найти на сайте http://www.comp.leeds.ac.uk/ccalas/tagsets/ upenn.html.

Таблица 2.12. Сокращенный список тегов, обозначающих части речи в английском языке

Тег	Описание
JJ	Прилагательное
NN	Существительное в единственном числе или не имеющее формы множественного числа
NNS	Существительное во множественном числе
NNP	Имя собственное, единственное число
NNPS	Имя собственное, множественное число
POS	Притяжательное окончание (конечная часть слова)
PRP	Личное местоимение
RB	Наречие
RP	Частица
VB	Глагол в основной, начальной форме
VBD	Глагол прошедшего времени
VBG	Глагол, герундий или причастие настоящего времени

Нормализация с применением конвейера

В этом разделе мы объединим несколько методик нормализации с помощью конвейера. Для демонстрации процесса расширим пример из раздела «Использование библиотеки LingPipe», в котором удалялись шумовые слова. Для нормализации текста добавим еще две «фабрики»: `LowerCaseTokenizerFactory` и `PorterStemmerTokenizerFactory`.

Фабрика `LowerCaseTokenizerFactory` добавляется перед созданием экземпляра `EnglishStopTokenizerFactory`, а фабрика `PorterStemmerTokenizerFactory` – после его создания, как показано ниже:

```
paragraph = "A simple approach is to create a class "
     + "to hold and remove stopwords.";
TokenizerFactory factory =
    IndoEuropeanTokenizerFactory.INSTANCE;
factory = new LowerCaseTokenizerFactory(factory);
factory = new EnglishStopTokenizerFactory(factory);
factory = new PorterStemmerTokenizerFactory(factory);
Tokenizer tokenizer =
    factory.tokenizer(paragraph.toCharArray(), 0, paragraph.length());
for(String token : tokenizer) {
    System.out.println(token);
}
```

Этот пример выведет следующий результат:

```
simpl
approach
creat
class
hold
remov
stopword
.
```

В полученном списке все шумовые слова удалены и приведены только основы оставшихся слов.

Резюме

В этой главе были продемонстрированы различные методики токенизации и нормализации текста. Сначала рассматривалась простейшая методика токенизации на основе классов языка Java: метод `split()` класса `String` и класс `StringTokenizer`. Такой подход вполне оправдан, если принято решение отказаться от использования библиотек NLP.

Далее было показано выполнение токенизации с помощью библиотек OpenNLP, Stanford и LingPipe. Здесь были обнаружены различия в выполнении токенизации, а также разнообразные дополнительные функциональные возможности, предлагаемые этими библиотеками. Было проведено краткое сравнение результатов их работы.

Глава завершилась описанием процесса нормализации текста, в который обычно включаются преобразование символов (букв) в ниж-

ний регистр, развертывание сокращений и аббревиатур, удаление шумовых слов, стемминг и лемматизация. Продемонстрирована реализация этих методик как с помощью штатных средств языка Java, так и с применением библиотек NLP.

В следующей главе будут рассматриваться темы, связанные с определением границ предложений с использованием различных библиотек NLP.

Глава 3

Поиск предложений

Разделение текста на предложения также называют *разрешением границ предложений* (*Sentence Boundary Disambiguation, SBD*). Этот процесс необходим для решения многих других задач обработки естественного языка, требующих проведения анализа внутри предложений, например определение частей речи и анализ словосочетаний выполняются в пределах предложения.

В этой главе будут подробно описаны причины затруднений при разрешении границ предложений. Затем рассматриваются некоторые методики с применением штатных средств языка Java, способные решить задачу поиска предложений в некоторых случаях, после чего мы перейдем к использованию моделей из различных библиотек NLP. Особое внимание будет уделено методикам обучения и проверки моделей, предназначенных для определения границ предложений. Для улучшения качества этого процесса можно добавлять вспомогательные правила, но подобный подход работает только до определенного момента. После этого модель необходимо обучить обработке как общих, так и особых ситуаций. В заключительной части главы рассматриваются именно такие модели и их практическое применение.

Процесс разрешения границ предложений

Процесс разрешения границ предложений зависит от конкретного языка и чаще всего связан с определенными трудностями. Общие методики поиска границ предложений предполагают использование набора правил или обучение модели решению данной задачи. Самый простой набор правил для поиска границ предложений приведен ниже. Конец предложения считается найденным, если:

- текст завершается точкой, вопросительным или восклицательным знаком;
- точке не предшествует какое-либо сокращение или после точки не располагается цифровой символ.

Эти правила хорошо работают для большинства предложений, но все-таки не для всех. Например, не всегда можно без затруднений определить, что слово перед точкой является сокращением, а некоторые последовательности символов, такие как многоточие, легко перепутать с обычными точками.

В большинстве механизмов поиска не уделяется особого внимания проблеме разрешения границ предложений, для них главное – это токены и их расположение в запросе. Инструменты определения частей речи и другие задачи обработки естественного языка, ориентированные на извлечение данных, гораздо чаще работают с отдельными предложениями. Определение границ предложений помогает отбрасывать «ложные» словосочетания, которые могут казаться осмысленными, если не учитываются границы предложений. Например, рассмотрим такую последовательность предложений:

«The construction process was over. The hill where the house was built was short».

Если выполнить поиск словосочетания «over the hill» без учета границ предложений, в приведенном выше тексте искомый образец будет найден, несмотря на то что с точки зрения пользователя данный результат абсолютно не соответствует критериям поиска.

В данной главе многие примеры применения методик разрешения границ предложений будут использовать приведенный ниже текст, состоящий из трех простых предложений, за которыми следует более сложное:

```
private static String paragraph = "When determining the end of "
    + "sentences we need to consider several factors. Sentences may "
    + "end with exclamation marks! Or possibly question marks? Within "
    + "sentences we may find numbers like 3.14159, abbreviations "
    + "such as found in Mr. Smith, and possibly ellipses either within "
    + a sentence..., or at the end of a sentence...";
```

Затруднения при разрешении границ предложений

Разделение текста на предложения представляет собой трудную задачу по ряду причин:

- часто пунктуация бывает неоднозначной;
- сокращения в большинстве случаев содержат точки;

- предложения могут быть размещены внутри других предложений – прямая речь или цитаты в кавычках;
- в некоторых особых типах текста, например в твитах и чат-сеансах, может потребоваться обработка символов перехода на новую строку или символов завершения сообщений (высказываний).

Лучшей иллюстрацией неоднозначности пунктуации может служить точка. Чаще всего точка применяется для обозначения конца предложения. Но она может встречаться в ряде других контекстов: в сокращениях, числовых значениях, адресах электронной почты и многоточиях. Другие знаки пунктуации, такие как вопросительный и восклицательный знаки, помимо своей основной функции, могут использоваться в цитируемых фрагментах, заключенных в кавычки, а также в узкоспециализированных текстах, например в исходном коде программы, включенном в документ.

Точка используется в следующих случаях:

- для обозначения конца предложения;
- для обозначения сокращения;
- для обозначения сокращения и конца предложения одновременно;
- в многоточиях;
- в многоточиях и для обозначения конца предложения одновременно;
- в прямой речи и в цитатах, включенных в другие предложения в кавычках или в скобках.

Большинство предложений заканчивается точкой, поэтому их идентификация проста. Несколько труднее определить границы предложения, если оно заканчивается сокращением. Следующее предложение содержит сокращения с использованием точек:

«Mr. and Mrs. Smith went to the ball».

В следующих двух предложениях сокращение встречается в конце предложения:

«He was an agent of the CIA».
«He was an agent of the C.I.A.».

Во втором предложении после каждой буквы аббревиатуры стоит точка. Подобное написание встречается не часто, тем не менее необходимо учитывать и такой вариант.

Другое затруднение возникает при попытке определить, является ли слово аббревиатурой. Нельзя считать аббревиатурами абсолютно все последовательности букв в верхнем регистре. Возможно, пользователь случайно ввел слово заглавными буквами или намеренно выделил его как особо важное. А как быть в случае, если во время предварительной обработки все символы были переведены в нижний регистр? Кроме того, некоторые аббревиатуры состоят из букв и верхнего, и нижнего регистров. Иногда для правильной обработки используют список допустимых аббревиатур, но зачастую применяемые в конкретном тексте аббревиатуры относятся к определенной предметной области, поэтому весьма специфичны.

Дополнительные сложности создают многоточия. Они могут встречаться в виде последовательности из трех точек или в виде одного символа (Extended ASCII 0x85 или Unicode U+2026). В Unicode, кроме горизонтального многоточия (U+2026), определено еще и вертикальное многоточие (U+22EE), а также особая форма представления вертикального и горизонтального многоточий (U+FE19). Помимо этого, существует HTML-кодировка. В языке Java используется формат записи `\uFE19`. Все перечисленные варианты кодировок свидетельствуют о том, что перед анализом текста необходимо выполнить тщательную предварительную обработку.

Следующие два предложения демонстрируют возможные варианты использования многоточия:

«And then there was ... one».
(«И наконец, остался... один».)

«And the list goes on and on and ...»
(«А список все продолжается и продолжается и...»)

Второе предложение завершается многоточием. В некоторых случаях, как предлагает справочное руководство MLA (http://www.mla-handbook.org/fragment/public_index), можно воспользоваться квадратными скобками, чтобы различать добавленное многоточие (пропуск фрагмента цитаты) от многоточия, являющегося частью исходного текста:

«The people [...] used various forms of transportation [...]» (Young 73)
(«Люди [...] использовали разнообразные формы транспортных средств [...]» (Юнг 73))

Часто встречаются предложения, расположенные внутри других предложений, например прямая речь:

The man said, «That's not right».
(Он сказал: «Это несправедливо».)

Еще одно затруднение создают восклицательные и вопросительные знаки, хотя встречаются они реже, чем точки. Вероятность появления восклицательного знака не в конце, а в середине предложения невелика. Восклицательный знак может быть неотъемлемой частью слова, например «Yahoo!» как название известной компании. Кроме того, несколько восклицательных знаков (или комбинация восклицательных и вопросительных знаков) используются для усиления эмоциональности высказывания, например «Best wishes!!» («С наилучшими пожеланиями!!»). Это может привести к ошибочному обнаружению нескольких предложений там, где в действительности находится лишь одно.

Правила разрешения границ предложений в классе HeuristicSentenceModel библиотеки LingPipe

Для разрешения границ предложений могут применяться разные наборы правил. Класс `HeuristicSentenceModel` из библиотеки LingPipe использует собственные правила, определяющие токены, которые могут служить признаками конца предложения. Здесь мы рассмотрим эти токены более подробно, как пример, какими должны быть правила разрешения границ предложений.

Для разрешения границ предложений в классе `HeuristicSentenceModel` определены три набора токенов и два флага:

- *возможные завершающие токены*: набор токенов, которые могут завершать предложения;
- *невозможные предпоследние токены*: набор токенов, которые не могут находиться перед завершающим токеном предложения;
- *невозможные начальные токены*: набор токенов, которые не могут находиться в начале предложения;
- *парность скобок*: этот флаг определяет, что предложение не может завершиться без соблюдения соответствия открывающих и закрывающих скобок;
- *принудительная установка границы*: этот флаг указывает, что конечный токен в потоке ввода должен интерпретироваться как конец предложения, даже если он не входит в список возможных завершающих токенов.

При проверке парности скобок рассматриваются круглые () и квадратные [] скобки. Но если текст бессвязный, отрывочный, это правило будет работать очень плохо. Наборы токенов по умолчанию приведены в табл. 3.1.

Таблица 3.1. Наборы токенов по умолчанию в классе HeuristicSentenceModel

Возможные завершающие токены	Невозможные предпоследние токены	Невозможные начальные токены
.	Любая одиночная буква	закрывающая скобка
..	Личные имена, профессиональные звания, титулы и т. п.	,
!	Запятые, двоеточия и кавычки	;
?	Общепринятые сокращения и аббревиатуры	:
»	Обозначения направлений	-
«	Коды авиакомпаний	--
).	Обозначения времени, месяца и т. п.	---
	Названия политических партий США	%
	Двухбуквенные обозначения штатов США (ME или IN)	«
	Термины, относящиеся к перевозке (доставке) грузов	
	Сокращения в почтовых адресах	

Несмотря на перечисленные правила, используемые в классе `HeuristicSentenceModel` из библиотеки LingPipe, нет никаких препятствий для применения тех же правил в любой другой реализации разрешения границ предложений.

Эвристический подход к разрешению границ предложений не всегда дает столь же точный результат, как другие методики. Тем не менее он вполне применим в некоторых предметных областях и часто работает быстрее и потребляет меньше памяти.

Простые средства разрешения границ предложений в языке Java

Для обработки простых текстов иногда достаточно поддержки, обеспечиваемой штатными средствами языка Java. Существуют два варианта реализации разрешения границ предложений: с помощью регу-

лярных выражений и с использованием класса `BreakIterator`. Ниже мы рассмотрим оба подхода.

Использование регулярных выражений

Регулярные выражения иногда трудны для понимания. Простые регулярные выражения обычно не вызывают затруднений, но с увеличением сложности они становятся все менее и менее понятными при чтении. Это один из недостатков регулярных выражений при использовании их для разрешения границ предложений.

Здесь будут рассматриваться два регулярных выражения. Первое – достаточно простое, но не обеспечивает хорошего качества результатов и показывает, что для некоторых предметных областей решение может быть очень простым. Второе выражение более изощренное и лучше справляется с задачей.

В следующем примере создается класс регулярного выражения, обнаруживающий совпадение с точкой, вопросительным и восклицательным знаками. Для разделения текста на предложения используется метод `split()` класса `String`:

```
String simple = "[.?!]";
String[] splitString = paragraph.split(simple);
for(String string : splitString) {
    System.out.println(string);
}
```

Этот пример выводит следующее:

```
When determining the end of sentences we need to consider several factors
 Sentences may end with exclamation marks
 Or possibly question marks
 Within sentences we may find numbers like 3
14159, abbreviations such as found in Mr
 Smith, and possibly ellipses either within a sentence..., or at the end
of a sentence...
```

Как и предполагалось, метод разбивает текст скорее на группы символов (а не на предложения) по заданным трем знакам препинания, не обращая внимания, входят ли они в состав числового значения или сокращения.

Рассмотрим второй вариант, который дает более приемлемые результаты. Регулярное выражение взято из примера с сайта http://stackoverflow.com/questions/5553410/regular-expression-match-a-sentence и немного изменено. Здесь используется класс `Pattern`, компилирующий это регулярное выражение:

```
[^.!?\s][^.!?]*(?:[.!?](?![’"]?\s|$)[^.!?]*)*[.!?]?[’"]?(?=\s|$)
```

Комментарий в следующем фрагменте кода объясняет каждую часть применяемого регулярного выражения:

```
Pattern sentencePattern = Pattern.compile(
    "# Поиск конца предложения по знакам пунктуации или по символу конца
     # строки.\n"
    + "[^.!?\\s]   # Первый символ - не знак пунктуации и не пробельный
                   # символ.\n"
    + "[^.!?]*     # Любые символы, кроме трех знаков пунктуации.\n"
    + "(?:         # Группа для организации цикла.\n"
    + "   [.!?]    # Особый случай: допустимы внутренние знаки пунктуации,
                   # если\n"
    + "   (?![’\"]?\\s|$  # далее не следуют пробельные символы или
                          # конец строки.\n"
    + "   [^.!?]* # Любые символы, кроме трех знаков пунктуации.\n"
    + ")*          # Любое число таких групп, включая нулевое (звезда
                   # Клини)\n"
    + "[.!?]?      # Необязательный конечный знак пунктуации (один).\n"
    + "[’\"]?      # Необязательная закрывающая кавычка (одиночная
                   # или двойная).\n"
    + "(?=\\s|$)",
    Pattern.MULTILINE | Pattern.COMMENTS);
```

Другое представление этого регулярного выражения можно получить с помощью графического инструмента на сайте http://regexper.com/. На рис. 3.1 показана подробная схема регулярного выражения, помогающая понять, как оно работает.

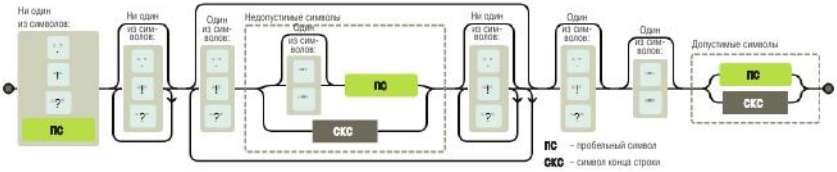


Рис. 3.1. Графическая схема используемого регулярного выражения

Метод `matcher()` обрабатывает ранее определенную строковую переменную `paragraph`, и затем следует вывод результатов:

```
Matcher matcher = sentencePattern.matcher(paragraph);
while(matcher.find()) {
    System.out.println(matcher.group());
}
```

Итог работы показан ниже. Здесь сохранены завершающие символы предложений, но осталась нерешенной проблема с сокращениями:

```
When determining the end of sentences we need to consider several factors.
Sentences may end with exclamation marks!
Or possibly question marks?
Within sentences we may find numbers like 3.14159, abbreviations such as
found in Mr.
Smith, and possibly ellipses either within a sentence..., or at the end
of a sentence...
```

Использование класса BreakIterator

Класс `BreakIterator` может использоваться для определения различных типов границ в тексте: между символами, между словами, между предложениями или между строками. При создании конкретных экземпляров класса `BreakIterator` применяются специализированные методы:

- для символов – метод `getCharacterInstance()`;
- для слов – метод `getWordInstance()`;
- для строк – метод `getLineInstance()`.

Необходимость определения границ между символами может возникнуть, например, при обработке символов, составленных из нескольких Unicode-символов, таких как ü, который иногда формируется из двух – `\u0075`(u) и `\u00a8`(¨). В этом случае класс будет определять подобные типы символов. Более подробно определение границ между символами описано в руководстве http://docs.oracle.com/javase/tutorial/i18n/text/char.html.

С помощью класса `BreakIterator` также можно решить задачу определения конца предложения. Он использует курсор, указывающий на текущую границу. Для перемещения курсора по тексту вперед и назад предлагаются методы `next()` и `previous()` соответственно. В классе определен только один закрытый конструктор по умолчанию. Для создания экземпляра класса `BreakIterator`, способного находить конец предложения, используется статический метод `getSentenceInstance()`, как показано ниже:

```
BreakIterator sentenceIterator = BreakIterator.getSentenceInstance();
```

Этот метод имеет перегруженную версию, принимающую объект типа `Locale`:

```
Locale currentLocale = new Locale("en", "US");
BreakIterator sentenceIterator =
    BreakIterator.getSentenceInstance(currentLocale);
```

После создания экземпляра класса BreakIterator метод setText() связывает обрабатываемый текст с итератором:

```
sentenceIterator.setText(paragraph);
```

Экземпляр BreakIterator определяет границы с помощью набора методов и полей. Все методы возвращают целочисленные значения, описанные в табл. 3.2.

Таблица 3.2. Краткое описание методов и полей класса BreakIterator

Метод	Описание
first	Возвращает первую границу в тексте
next	Возвращает границу, следующую за текущей границей
previous	Возвращает границу, предшествующую текущей транице
DONE	Константа со значением –1 (обозначающая, что в тексте больше не найдено границ)

Чтобы использовать итератор, необходимо сначала найти самую первую границу с помощью метода first(), затем в цикле вызывать метод next() для определения последующих границ. Процесс завершается, когда возвращается значение DONE. В следующем примере демонстрируется описанная методика, использующая экземпляр sentenceIterator, объявленный выше:

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

Выполнив этот пример, мы получим следующее:

```
0-75 [When determining the end of sentences we need to consider several
factors. ]
75-117 [Sentences may end with exclamation marks! ]
117-146 [Or possibly question marks? ]
146-233 [Within sentences we may find numbers like 3.14159, abbreviations
such as found in Mr. ]
233-319 [Smith, and possibly ellipses either within a sentence..., or at
the end of a sentence...]
319-
```

Описанная выше методика работает для простых предложений, но с более сложными предложениями не справляется.

Регулярные выражения и класс `BreakIterator` имеют определенные ограничения. С их помощью можно обрабатывать только текст, состоящий из относительно простых предложений. В случаях, когда текст имеет более сложную структуру, лучше воспользоваться библиотеками NLP, о чем и пойдет речь в следующем разделе.

Использование библиотек NLP API

Существует множество библиотек NLP с классами, поддерживающими разрешение границ предложений. Некоторые основаны на наборах правил, другие используют модели, которые можно обучать с помощью обычных и специализированных текстов. Для решения задачи определения границ предложений здесь будут рассматриваться классы из библиотек OpenNLP, Stanford и LingPipe.

Обсуждению процесса обучения моделей посвящен раздел «Обучение модели определителя предложений Sentence Detector» ниже. Модели с узкой специализацией необходимы для обработки профессионально ориентированных текстов, например медицинских или юридических документов.

Использование библиотеки OpenNLP

Для разрешения границ предложений библиотека OpenNLP использует модели. Экземпляр класса `SentenceDetectorME` создается на основе модели из соответствующего файла. Метод `sentDetect()` возвращает найденные предложения, а информацию об их местоположении возвращает метод `sentPosDetect()`.

Использование класса SentenceDetectorME

Модель загружается из файла с помощью класса `SentenceModel`. Затем на основе загруженной модели создается экземпляр класса `SentenceDetectorME` и вызывается метод `sentDetect()` для разрешения границ предложений. Метод возвращает массив строк, каждый элемент которого содержит найденное предложение.

В следующем примере демонстрируется описанный выше процесс. При открытии файла *en-sent.bin* с требуемой моделью используется блок попытки захвата ресурсов (try-with-resources). Затем обрабатывается строка `paragraph`. После этого перехватываются исключения

типа IO (ввод-вывод), если это необходимо. Оператор for-each позволяет вывести найденные предложения по отдельности:

```
try(InputStream is = new FileInputStream(
       new File(getModelDir(), "en-sent.bin"))) {
    SentenceModel model = new SentenceModel(is);
    SentenceDetectorME detector = new SentenceDetectorME(model);
    String sentences[] = detector.sentDetect(paragraph);
    for(String sentence : sentences) {
        System.out.println(sentence);
    }
} catch(FileNotFoundException ex) {
    // Обработка исключения.
} catch(IOException ex) {
    // Обработка исключения.
}
```

Выполнив этот пример, мы получим следующее:

```
When determining the end of sentences we need to consider several
factors.
Sentences may end with exclamation marks!
Or possibly question marks?
Within sentences we may find numbers like 3.14159, abbreviations such as
found in Mr.Smith, and possibly ellipses either within a sentence..., or
at the end of a sentence...
```

Текст в переменной paragraph обработан правильно – и простые, и сложные предложения. Но обрабатываемый текст далеко не всегда правильно отформатирован. В качестве примера рассмотрим текст, в котором встречаются лишние пробелы перед точкой, завершающей предложение, и отсутствие пробелов там, где они должны быть, то есть между предложениями. Подобные случаи «плохого» форматирования часто встречаются в задачах анализа чат-сеансов:

```
paragraph = "  This sentence starts with spaces and ends with "
    + "spaces . This sentence has no spaces between the next "
    + "one.This is the next one.";
```

Для данного текста мы получим следующий результат:

```
This sentence starts with spaces and ends with spaces .
This sentence has no spaces between the next one.This is the next one.
```

Пробелы перед первым предложением удалены, но ненужные пробелы в конце предложения остались. Начало третьего предложения определить не удалось, и оно оказалось «приклеенным» к концу второго предложения.

Метод getSentenceProbabilities() возвращает массив значений с плавающей точкой двойной точности, представляющих степень до-

стоверности границ предложений, найденных во время последнего вызова метода sentDetect(). Если в примере после оператора for-each, выводящего найденные предложения, добавить следующий код:

```
double probabilities[] = detector.getSentenceProbabilities();
for(double probability : probabilities) {
    System.out.println(probability);
}
```

для того же текста после списка предложений будут выведены следующие значения:

```
0.9841708738988814
0.908052385070974
0.9130082376342675
1.0
```

Эти числа говорят о весьма высокой степени достоверности разрешения границ предложений.

Использование метода sentPosDetect

В классе SentenceDetectorME имеется метод PosDetect(), возвращающий объекты типа Span для каждого предложения. Воспользуемся примером из предыдущего подраздела и внесем в него два изменения: заменим метод sentDetect() на sentPosDetect() и добавим оператор for-each для вывода результатов:

```
Span spans[] = sdetector.sentPosDetect(paragraph);
for(Span span : spans) {
    System.out.println(span);
}
```

Ниже показаны результаты обработки первоначального текста в переменной paragraph. Объекты Span содержат информацию о границах найденных строк, преобразованную методом toString():

```
[0..74)
[75..116)
[117..145)
[146..317)
```

Класс Span предоставляет множество вспомогательных методов. В следующем фрагменте используются методы getStart() и getEnd() для вывода предложений, определяемых найденными выше границами:

```
for(Span span : spans) {
    System.out.println(span + "["
        + paragraph.substring(span.getStart(), span.getEnd()) + "]");
}
```

Теперь пример выводит также сами найденные предложения:

```
[0..74) [When determining the end of sentences we need to consider
several factors.]
[75..116) [Sentences may end with exclamation marks!]
[117..145) [Or possibly question marks?]
[146..317) [Within sentences we may find numbers like 3.14159,
abbreviations such as found in Mr.Smith, and possibly ellipses either
within a sentence..., or at the end of a sentence...]
```

Краткие характеристики других методов класса `Span` приведены в табл. 3.3.

Таблица 3.3. Некоторые методы класса Span

Метод	Описание
`contains`	Перегружаемый метод, который определяет, содержится ли в заданном целевом объекте другой объект типа `Span` или соответствующий ему индекс позиции
`crosses`	Определяет, перекрываются ли два интервала
`length`	Возвращает длину интервала
`startsWith`	Определяет, начинается ли интервал с позиции заданного интервала

Использование библиотеки Stanford API

Библиотека Stanford API поддерживает несколько методик определения границ предложений. В этом разделе будет рассматриваться обработка текста с применением следующих классов:

- `PTBTokenizer`;
- `DocumentPreprocessor`;
- `StanfordCoreNLP`.

Несмотря на то что все эти классы обеспечивают разрешения границ предложений, каждый применяет собственный подход к реализации этого процесса.

Использование класса PTBTokenizer

Класс `PTBTokenizer` использует правила разрешения границ предложений и предлагает набор параметров токенизации. Конструктор этого класса принимает три параметра:

- экземпляр класса `Reader` с обрабатываемым текстом;
- объект, реализующий интерфейс `LexedTokenFactory`;
- строку параметров токенизации.

Перечисленные параметры позволяют определить исходный текст, используемый токенизатор и параметры, которые могут потребоваться при обработке заданного потока текста.

В следующем примере создается экземпляр класса `StringReader`, предназначенный для хранения текста. Класс `CoreLabelTokenFactory` используется с параметрами токенизации, которые в данном примере отсутствуют, поэтому представлены значением `null`:

```
PTBTokenizer ptb = new PTBTokenizer(new StringReader(paragraph),
    new CoreLabelTokenFactory(), null);
```

Класс `WordToSentenceProcessor` используется при создании экземпляра класса `List`, в котором будут сохраняться предложения и их токены. Метод `process()` принимает токены от экземпляра `PTBTokenizer` для создания списка `List`, как показано ниже:

```
WordToSentenceProcessor wtsp = new WordToSentenceProcessor();
List<List<CoreLabel>> sents = wtsp.process(ptb.tokenize());
```

Содержимое списка типа `List` можно вывести несколькими способами. В следующем фрагменте вызывается метод `toString()` класса `List`, который выводит список, заключенный в квадратные скобки, а его элементы разделяются запятыми:

```
for(List<CoreLabel> sent : sents) {
    System.out.println(sent);
}
```

Этот пример выводит следующее:

```
[When, determining, the, end, of, sentences, we, need, to, consider,
several, factors, .]
[Sentences, may, end, with, exclamation, marks, !]
[Or, possibly, question, marks, ?]
[Within, sentences, we, may, find, numbers, like, 3.14159, ,,
abbreviations, such, as, found, in, Mr., Smith, ,, and, possibly,
ellipses, either, within, a, sentence, ..., ,, or, at, the, end, of, a,
sentence, ...]
```

Ниже показан другой подход, который выводит каждое найденное предложение в отдельной строке:

```
for(List<CoreLabel> sent : sents) {
    for(CoreLabel element : sent) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```

Он выводит следующее:

```
When determining the end of sentences we need to consider several factors .
Sentences may end with exclamation marks !
Or possibly question marks ?
Within sentences we may find numbers like 3.14159 , abbreviations such as
found in Mr. Smith , and possibly ellipses either within a sentence ...
, or at the end of a sentence ...
```

Если необходимы только позиции слов и предложений, можно воспользоваться методом `endPosition()`, как показано ниже:

```
for(List<CoreLabel> sent : sents) {
    for(CoreLabel element : sent) {
        System.out.print(element.endPosition() + " ");
    }
    System.out.println();
}
```

Эта версия примера выведет следующий результат. Последнее число в каждой строке обозначает позицию границы текущего предложения:

```
4 16 20 24 27 37 40 45 48 57 65 73 74
84 88 92 97 109 115 116
119 128 138 144 145
152 162 165 169 174 182 187 195 196 210 215 218 224 227 231 237 238 242
251 260 267 274 276 285 287 288 291 294 298 302 305 307 316 317
```

Следующий фрагмент выведет первый элемент каждого предложения и его позицию:

```
for(List<CoreLabel> sent : sents) {
    System.out.println(sent.get(0) + " " + sent.get(0).beginPosition());
}
```

Результат:

```
When 0
Sentences 75
Or 117
Within 146
```

Чтобы получить последние элементы предложений, можно применить следующий прием. Здесь для вывода последнего элемента и его позиции используется размер списка:

```
for(List<CoreLabel> sent : sents) {
    int size = sent.size();
    System.out.println(sent.get(size-1) + " "
        + sent.get(size-1).endPosition());
}
```

Этот вариант выведет следующее:

```
. 74
! 116
? 145
... 317
```

Конструктору класса `PTBTokenizer` можно передать дополнительные параметры в третьем строковом аргументе. Параметры в этой строке отделяются друг от друга запятыми, как показано ниже:

```
"americanize=true,normalizeFractions=true,asciiQuotes=true"
```

Некоторые дополнительные параметры описаны в табл. 3.4.

Таблица 3.4. Дополнительные параметры для конструктора PTBTokenizer

Параметр	Описание
`invertible`	Определяет необходимость предварительного сохранения токенов и пробельных символов, чтобы можно было восстановить исходную строку
`tokenizeNLs`	Определяет, что символы концов строк (перехода на новую строку) должны интерпретироваться как токены
`amerikanize`	Если равен `true`, слова в британском написании будут переписаны в американском написании
`normalizeAmpersandEntity`	Определяет преобразование XML-символа &amp в амперсанд
`normalizeFractions`	Определяет преобразование символов дробей, таких как ½, в развернутый формат (1/2)
`asciiQuotes`	Определяет преобразование символов кавычек в более простые символы ' и "
`unicodeQuotes`	Определяет преобразование символов кавычек в Unicode-символы из диапазона от U+2018 до U+201D

В следующем примере демонстрируется применение приведенной выше строки дополнительных параметров:

```
paragraph = "The colour of money is green. Common fraction "
    + "characters such as ½ are converted to the long form 1/2. "
    + "Quotes such as "cat" are converted to their simpler form.";
ptb = new PTBTokenizer(
    new StringReader(paragraph), new CoreLabelTokenFactory(),
    "americanize=true,normalizeFractions=true,asciiQuotes=true");
wtsp = new WordToSentenceProcessor();
sents = wtsp.process(ptb.tokenize());
for(List<CoreLabel> sent : sents) {
```

```
    for(CoreLabel element : sent) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```

Результат работы этого примера:

```
The color of money is green .
Common fraction characters such as 1/2 are converted to the long form 1/2 .
Quotes such as " cat " are converted to their simpler form .
```

Слово, написанное в соответствии с британскими правилами правописания, «colour», было преобразовано в его американский вариант. Символ дроби ½ был развернут в три символа: 1/2. В последнем предложении особые формы кавычек были переведены в более простую форму.

Использование класса DocumentPreprocessor

Когда создается экземпляр класса `DocumentPreprocessor`, используется параметр типа `Reader`, чтобы получить список предложений. Кроме того, этот класс реализует интерфейс `Iterable` для поддержки простого перемещения по списку.

В следующем примере ранее определенная переменная `paragraph` используется для создания объекта `StringReader`, который, в свою очередь, нужен для создания экземпляра `DocumentPreprocessor`:

```
Reader reader = new StringReader(paragraph);
DocumentPreprocessor dp = new DocumentPreprocessor(reader);
for(List sentence : dp) {
    System.out.println(sentence);
}
```

Этот пример выведет следующий результат:

```
[When, determining, the, end, of, sentences, we, need, to, consider,
several, factors, .]
[Sentences, may, end, with, exclamation, marks, !]
[Or, possibly, question, marks, ?]
[Within, sentences, we, may, find, numbers, like, 3.14159, ,,
abbreviations, such, as, found, in, Mr., Smith, ,, and, possibly,
ellipses, either, within, a, sentence, ..., ,, or, at, the, end, of, a,
sentence, ...]
```

По умолчанию для токенизации потока ввода используется `PTBTokenizer`. Метод `setTokenizerFactory()` предназначен для определения другого токенизатора. Могут оказаться полезными и другие методы, перечисленные в табл. 3.5.

Таблица 3.5. Некоторые методы класса DocumentPreprocessor

Метод	Описание
`setElementDelimiter`	Аргумент метода определяет XML-элемент. Будет обрабатываться только текст, содержащийся внутри заданных элементов
`setSentenceDelimiter`	Процессор интерпретирует строковый аргумент этого метода как разделитель предложений
`setSentenceFinalPuncWords`	Аргумент в виде массива строк определяет окончания разделителей предложений
`setKeepEmptySentences`	При работе с моделями, учитывающими пробельные символы, значение `true` в аргументе этого метода позволяет сохранять пустые предложения

Класс `DocumentPreprocessor` может обрабатывать обычный текст и XML-документы.

Для демонстрации обработки документов в формате XML создадим простой XML-файл с именем *XMLText.xml*, содержащий следующие данные:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"?>
<document>
    <sentences>
        <sentence id="1">
            <word>When</word>
            <word>the</word>
            <word>day</word>
            <word>is</word>
            <word>done</word>
            <word>we</word>
            <word>can</word>
            <word>sleep</word>
            <word>.</word>
        </sentence>
       <sentence id="2">
            <word>When</word>
            <word>the</word>
            <word>morning</word>
            <word>comes</word>
            <word>we</word>
            <word>can</word>
            <word>wake</word>
            <word>.</word>
        </sentence>
        <sentence id="3">
            <word>After</word>
            <word>that</word>
            <word>who</word>
```

```
            <word>knows</word>
            <word>.</word>
        </sentence>
    </sentences>
</document>
```

Для обработки будет повторно использован предыдущий пример. Но вместо переменной `paragraph` будет открыт файл *XMLText.xml*, и во втором аргументе конструктору класса `DocumentPreprocessor` будет передано значение `DocumentPreprocessor.DocType.XML`, указывающее, что процессор должен интерпретировать входной поток как XML-текст. Кроме того, указывается, что должны обрабатываться только элементы внутри тега `<sentence>`:

```
try {
    Reader reader = new FileReader("XMLText.xml");
    DocumentPreprocessor dp = new DocumentPreprocessor(
        reader, DocumentPreprocessor.DocType.XML);
    dp.setElementDelimiter("sentence");
    for(List sentence : dp) {
        System.out.println(sentence);
    }
} catch(FileNotFoundException ex) {
    // Обработка исключения.
}
```

Результат работы этого примера показан ниже:

```
[When, the, day, is, done, we, can, sleep, .]
[When, the, morning, comes, we, can, wake, .]
[After, that, who, knows, .]
```

Убрать все лишние символы, мешающие чтению, можно с помощью объекта типа `ListIterator`, как показано ниже:

```
for(List sentence : dp) {
    ListIterator list = sentence.listIterator();
    while(list.hasNext()) {
        System.out.print(list.next() + " ");
    }
    System.out.println();
}
```

Результат представлен ниже:

```
When, the, day, is, done, we, can, sleep, .
When, the, morning, comes, we, can, wake, .
After, that, who, knows, .
```

Если в первоначальной версии примера опустить элемент-разделитель, каждое слово выводилось бы на отдельной строке:

```
[When]
[the]
[day]
[is]
[done]
...
[who]
[knows]
[.]
```

Использование класса StanfordCoreNLP

Класс `StanfordCoreNLP` поддерживает определение границ предложений с использованием аннотатора `ssplit`. В следующем примере задействованы аннотаторы `tokenize` и `ssplit`. Здесь создается конвейер, и к нему применяется метод `annotate()`, которому передается переменная `paragraph`:

```
Properties properties = new Properties();
properties.put("annotators", "tokenize, ssplit");
StanfordCoreNLP pipeline = new StanfordCoreNLP(properties);
Annotation annotation = new Annotation(paragraph);
pipeline.annotate(annotation);
```

Этот пример выведет очень большой объем информации. Здесь показан только результат для первого предложения:

```
Sentence #1 (13 tokens):
When determining the end of sentences we need to consider several
factors.
[Text=When CharacterOffsetBegin=0 CharacterOffsetEnd=4]
[Text=determining CharacterOffsetBegin=5 CharacterOffsetEnd=16]
[Text=the CharacterOffsetBegin=17 CharacterOffsetEnd=20]
[Text=end CharacterOffsetBegin=21 CharacterOffsetEnd=24]
[Text=of CharacterOffsetBegin=25 CharacterOffsetEnd=27]
[Text=sentences CharacterOffsetBegin=28 CharacterOffsetEnd=37]
[Text=we CharacterOffsetBegin=38 CharacterOffsetEnd=40]
[Text=need CharacterOffsetBegin=41 CharacterOffsetEnd=45]
[Text=to CharacterOffsetBegin=46 CharacterOffsetEnd=48]
[Text=consider CharacterOffsetBegin=49 CharacterOffsetEnd=57]
[Text=several CharacterOffsetBegin=58 CharacterOffsetEnd=65]
[Text=factors CharacterOffsetBegin=66 CharacterOffsetEnd=73]
[Text=. CharacterOffsetBegin=73 CharacterOffsetEnd=74]
```

Другое решение состоит в использовании метода `xmlPrint()`, позволяющего получить результат в XML-формате, который весьма часто применяется для представления искомой информации. Альтернативная версия показана ниже, но в ней необходимо предусмотреть обработку исключения `IOException`:

```
try {
    pipeline.xmlPrint(annotation, System.out);
} catch(IOException ex) {
    // Обработка исключения.
}
```

В целях экономии места ниже показана только часть результата:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="CoreNLP-to-HTML.xsl" type="text/xsl"?>
<root>
  <document>
    <sentences>
      <sentence id="1">
        <tokens>
          <token id="1">
            <word>When</word>
            <CharacterOffsetBegin>0</CharacterOffsetBegin>
            <CharacterOffsetEnd>4</CharacterOffsetEnd>
          </token>
...
          <token id="34">
            <word>...</word>
            <CharacterOffsetBegin>316</CharacterOffsetBegin>
            <CharacterOffsetEnd>317</CharacterOffsetEnd>
          </token>
        </tokens>
      </sentence>
    </sentences>
  </document>
</root>
```

Использование библиотеки LingPipe

Для разрешения границ предложений в библиотеке LingPipe используется иерархия классов, показанная на рис. 3.2. В основе этой иерархии лежит базовый класс `AbstractSentenceModel`, главную роль в котором играет перегруженный метод `boundaryIndices()`, возвращающий массив целочисленных значений, где каждый элемент соответствует позиции границы предложения.

Ниже в иерархии расположен производный класс `HeuristicSentenceModel`, использующий наборы токенов: возможные завершающие токены, невозможные предпоследние токены и невозможные начальные токены, которые рассматривались ранее, в разделе «Правила разрешения границ предложений в классе HeuristicSentenceModel библиотеки LingPipe» выше.

Классы `IndoEuropeanSentenceModel` и `MedlineSentenceModel` являются производными от класса `HeuristicSentenceModel`. Они уже обучены

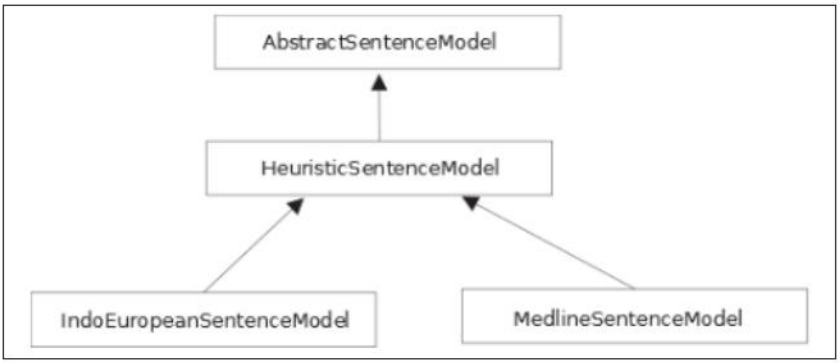


Рис. 3.2. Иерархия классов для разрешения границ предложений в библиотеке LingPipe

для обработки обычных текстов на английском языке и специализированных медицинских текстов соответственно. Мы рассмотрим оба этих класса.

Использование класса IndoEuropeanSentenceModel

Модель `IndoEuropeanSentenceModel` используется для обработки обычных текстов на английском языке. Конструктор класса принимает два аргумента, которые определяют:

- должен ли конечный токен завершать процесс обработки;
- следует ли соблюдать парность скобок.

Конструктор по умолчанию не воспринимает конечный токен как признак завершения обработки и не следит за парностью скобок. Эта модель предложения должна использоваться в сочетании с токенизатором. Ниже показан пример применения конструктора по умолчанию класса `IndoEuropeanSentenceModel`:

```
TokenizerFactory TOKENIZER_FACTORY =
IndoEuropeanTokenizerFactory.INSTANCE;
SentenceModel sentenceModel = new IndoEuropeanSentenceModel();
```

Создается токенизатор, и вызывается его метод `tokenize()` для заполнения двух списков:

```
List<String> tokenList = new ArrayList<>();
List<String> whiteList = new ArrayList<>();
Tokenizer tokenizer = TOKENIZER_FACTORY.tokenizer(
    paragraph.toCharArray(), 0,paragraph.length());
tokenizer.tokenize(tokenList, whiteList);
```

Метод `boundaryIndices()` возвращает массив целочисленных значений границ и принимает два аргумента – массивы типа `String` с токенами и пробельными символами. Выше метод `tokenize()` создал для этих элементов два списка типа `List`. Теперь необходимо преобразовать списки в соответствующие массивы:

```
String[] tokens = new String[tokenList.size()];
String[] whites = new String[whiteList.size()];
tokenList.toArray(tokens);
whiteList.toArray(whites);
```

После этого можно воспользоваться методом `boundaryIndices()` и вывести позиции границ предложений:

```
int[] sentenceBoundaries =
sentenceModel.boundaryIndices(tokens, whites);
for(int boundary : sentenceBoundaries) {
    System.out.println(boundary);
}
```

В результате выводятся три значения границ предложений:

```
12
19
24
```

Для вывода самих предложений предназначен следующий фрагмент. Позиции пробелов указываются с единичным смещением относительно текущего токена:

```
int start = 0;
for(int boundary : sentenceBoundaries) {
    while(start <= boundary) {
        System.out.print(tokenList.get(start)
            + whiteList.get(start+1));
        start++;
    }
    System.out.println();
}
```

Этот фрагмент выведет следующий результат:

```
When determining the end of sentences we need to consider several factors.
Sentences may end with exclamation marks!
Or possibly question marks?
```

Но здесь пропущено последнее предложение. Это произошло из-за того, что оно заканчивается многоточием. Если в конец последнего предложения добавить точку, недостаток будет устранен:

```
When determining the end of sentences we need to consider several factors.
Sentences may end with exclamation marks!
Or possibly question marks?
Within sentences we may find numbers like 3.14159, abbreviations such as
found in Mr. Smith, and possibly ellipses either within a sentence …, or
at the end of a sentence….
```

Использование класса SentenceChunker

Другая методика разрешения границ предложений предполагает использование класса `SentenceChunker`. Конструктор этого класса требует наличия объектов типа `TokenizerFactory` и `SentenceModel`, которые создаются в следующем фрагменте кода:

```
TokenizerFactory tokenizerfactory =
IndoEuropeanTokenizerFactory.INSTANCE;
SentenceModel sentenceModel = new IndoEuropeanSentenceModel();
```

Экземпляр `SentenceChunker` создается с использованием двух только что созданных объектов:

```
SentenceChunker sentenceChunker =
    new SentenceChunker(tokenizerfactory, sentenceModel);
```

Класс `SentenceChunker` реализует интерфейс `Chunker` с методом `chunk()`, который возвращает объект, в свою очередь реализующий интерфейс `Chunking`. Этот объект определяет фрагменты («chunks») текста с помощью последовательности символов типа `CharSequence`.

Метод `chunk()` использует массив символов и индексы в этом массиве для определения части текста, которая требует обработки. Объект типа `Chunking` возвращается следующим образом:

```
Chunking chunking = sentenceChunking.chunk(
    paragraph.toCharArray(), 0, paragraph.length());
```

Объект `chunking` необходим для достижения двух целей. Во-первых, его метод `chunkSet()` позволяет получить набор объектов типа `Chunk`. После этого будет сформирована строка, содержащая все исходные предложения:

```
Set<Chunk> sentences = chunking.chunkSet();
String slice = chunking.charSequence().toString();
```

В объекте типа `Chunk` сохраняются значения смещений символов, находящихся на границах предложений. Для вывода предложений воспользуемся методами `start()` и `end()`, работающими со строкой `slice`, как показано ниже. Каждый элемент, переменная цикла

sentence, содержит границы текущего предложения. Эта информация используется для вывода предложений из фрагмента slice:

```
for(Chunk sentence : sentences) {
    System.out.println("[" + slice.substring(sentence.start(),
      sentence.end()) + "]");
}
```

Результат выполнения примера приведен ниже. Здесь также наблюдается проблема с предложениями, заканчивающимися многоточием, поэтому перед обработкой текста необходимо добавить точку в конец последнего предложения.

```
[When determining the end of sentences we need to consider several
factors.]
[Sentences may end with exclamation marks!]
[Or possibly question marks?]
[Within sentences we may find numbers like 3.14159, abbreviations such as
found in Mr. Smith, and possibly ellipses either within a sentence …, or
at the end of a sentence….]
```

Класс IndoEuropeanSentenceModel дает вполне приемлемые результаты при работе с обычными текстами на английском языке, но не всегда способен успешно обрабатывать узкоспециализированные тексты. Далее мы рассмотрим класс MedlineSentenceModel, специально обученный для обработки медицинских текстов.

Использование класса MedlineSentenceModel

Специализированная модель предложений в библиотеке LingPipe использует *MEDLINE* – крупное собрание биомедицинской литературы, которое хранится в XML-формате и поддерживается Национальной медицинской библиотекой США (United States National Library of Medicine – http://www.nlm.nih.gov/).

Класс MedlineSentenceModel из библиотеки LingPipe применяется для разрешения границ предложений. Эта модель обучена на данных MEDLINE. Сначала выполняется токенизация текста с целью его разделения на токены и пробельные символы. Затем используется модель MEDLINE для определения предложений.

В следующем примере демонстрируется применение этой модели для обработки абзаца специализированного текста, взятого из статьи http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3139422/:

```
paragraph = "HepG2 cells were obtained from the American Type "
    + "Culture Collection (Rockville, MD, USA) and were used "
    + "only until passage 30. They were routinely grown at 37°C "
```

```
    + "in Dulbecco's modified Eagle's medium (DMEM) containing "
    + "10 % fetal bovine serum (FBS), 2mM glutamine, 1mM sodium "
    + "pyruvate, and 25 mM glucose (Invitrogen, Carlsbad, CA, "
    + "USA) in a humidified atmosphere containing 5% CO2. For "
    + "precursor and 13C-sugar experiments, tissue culture treated "
    + "polystyrene 35 mm dishes (Corning Inc, Lowell, MA, USA) were "
    + "seeded with 2 × 106 cells and grown to confluency in DMEM.";
```

В примере ниже применяется класс `SentenceChunker`, как и в предыдущем разделе. Разница лишь в том, что здесь он использует класс `MedlineSentenceModel`:

```
TokenizerFactory tokenizerfactory =
    IndoEuropeanTokenizerFactory.INSTANCE;
MedlineSentenceModel sentenceModel =
    new MedlineSentenceModel();
SentenceChunker sentenceChunker =
    new SentenceChunker(tokenizerfactory, sentenceModel);
Chunking chunking = sentenceChunker.chunk(
    paragraph.toCharArray(), 0, paragraph.length());
Set<Chunk> sentences = chunking.chunkSet();
String slice = chunking.charSequence().toString();
for(Chunk sentence : sentences) {
    System.out.println("["
        + slice.substring(sentence.start(), sentence.end())
        + "]");
}
```

Этот пример выведет следующий результат:

```
[HepG2 cells were obtained from the American Type Culture Collection
(Rockville, MD, USA) and were used only until passage 30.]

[They were routinely grown at 37°C in Dulbecco's modified Eagle's medium
(DMEM) containing 10 % fetal bovine serum (FBS), 2mM glutamine, 1mM
sodium pyruvate, and 25 mM glucose (Invitrogen, Carlsbad, CA, USA) in a
humidified atmosphere containing 5% CO2.]

[For precursor and 13C-sugar experiments, tissue culture treated
polystyrene 35 mm dishes (Corning Inc, Lowell, MA, USA) were seeded with
2 × 106 cells and grown to confluency in DMEM.]
```

Эта модель лучше справляется с медицинскими текстами, чем другие модели.

Обучение модели SentenceDetector

Для демонстрации процесса обучения воспользуемся классом `SentenceDetectorME` из библиотеки OpenNLP. В этом классе имеется статический метод `train()`, работающий с образцами предложений,

найденными в файле. Метод возвращает готовую модель, которая обычно сериализуется и сохраняется в файле для дальнейшего использования.

Для моделей подготовлены специально аннотированные данные, позволяющие однозначно определять границы предложений. Часто для обучения лучше подходит текстовый файл большого размера. Часть такого файла используется непосредственно для обучения, а остаток –для проверки работы модели после обучения.

Файл для обучения модели из библиотеки OpenNLP состоит из предложений, размещенных на отдельных строках. Обычно требуется от 10 до 20 предложений, чтобы устранить ошибки при обработке. Процесс обучения будет показан на примере файла с именем *sentence.train*, в котором содержится текст из главы 5 книги «20 тысяч лье под водой» Жюля Верна (Jules Verne). Текст этой главы размещен на сайте проекта «Гутенберг» (http://www.gutenberg.org/files/164/164-h/164-h.htm#chap05). Файл *sentence.train* можно скачать с сайта www.packtpub.com, а также с сайта www.dmkpress.com или www.дмк.рф – в разделе «Читателям – Файлы к книгам».

Объект `FileReader` используется, чтобы открыть файл. Он передается в вызов конструктора `PlainTextByLineStream()` потока ввода. Поток ввода передает содержимое файла построчно. Объект потока передается в вызов конструктора `SentenceSampleStream` объекта, преобразующего строки-предложения в объекты типа `SentenceSample`, которые хранят позицию начала каждого предложения. Весь описанный процесс реализуется, как показано ниже, где все инструкции заключены в блок `try` для перехвата и обработки возможных исключений:

```
try {
    ObjectStream<String> lineStream =
        new PlainTextByLineStream(new FileReader("sentence.train"));
    ObjectStream<SentenceSample> sampleStream =
        new SentenceSampleStream(lineStream);
    ...
} catch(FileNotFoundException ex) {
    // Обработка исключения.
} catch(IOException ex) {
    // Обработка исключения.
}
```

Далее можно применить метод `train()`:

```
SentenceModel model = SentenceDetectorME.train("en",
    sampleStream, true, null,
    TrainingParameters.defaultParams());
```

Результатом этого метода является обученная модель. Краткое описание его параметров приводится в табл. 3.6.

Таблица 3.6. Описание параметров метода train()

Параметр	Описание
`"en"`	Определяет, что обрабатывается текст на английском языке
`sampleStream`	Поток обучающего текста
`true`	Определяет необходимость использования видимых конечных токенов предложения
`null`	Словарь сокращений (в данном случае не нужен)
`TrainingParameters.defaultParams()`	Определяет, что должны использоваться параметры обучения, принятые по умолчанию

Далее создается объект потока вывода `OutputStream` для сохранения обученной модели в файле *modelFile*. Это позволит повторно использовать данную модель в других приложениях.

```
OutputStream modelStream =
    new BufferedOutputStream(new FileOutputStream("modelFile"));
model.serialize(modelStream);
```

Результат обучения приводится ниже. В целях экономии места показана лишь небольшая часть итераций. По умолчанию определено выполнение 100 итераций, а также пороговое значение 5 количества событий, соответствующих обнаружению границы предложения:

```
Indexing events using cutoff of 5
  Computing event counts... done. 93 events
  Indexing... done.
Sorting and merging events... done. Reduced 93 events to 63.
Done indexing.
Incorporating indexed data for training... done.
  Number of Event Tokens: 63
    Number of Outcomes:   2
    Number of Predicates: 21
...done.
Computing model parameters ...
Performing 100 iterations.
  1: ... loglikelihood=-64.4626877920749   0.9032258064516129
  2: ... loglikelihood=-31.11084296202819   0.9032258064516129
  3: ... loglikelihood=-26.418795734248626   0.9032258064516129
  4: ... loglikelihood=-24.327956749903198   0.9032258064516129
  5: ... loglikelihood=-22.766489585258565   0.9032258064516129
  6: ... loglikelihood=-21.46379347841989   0.9139784946236559
  7: ... loglikelihood=-20.356036369911394   0.9139784946236559
  8: ... loglikelihood=-19.406935608514992   0.9139784946236559
```

```
  9: ... loglikelihood=-18.58725539754483   0.9139784946236559
 10: ... loglikelihood=-17.873030559849326   0.9139784946236559
  ...
 99: ... loglikelihood=-7.214933901940582   0.978494623655914
100: ... loglikelihood=-7.183774954664058   0.978494623655914
```

Использование обученной модели

Теперь можно воспользоваться обученной моделью. Для демонстрации практического ее применения ниже приводится пример кода, реализующий методику из раздела «Использование класса SentenceDetectorME» (см. выше в этой главе):

```
try(InputStream is = new FileInputStream(
       new File(getModelDir(), "modelFile"))) {
    SentenceModel model = new SentenceModel(is);
    SentenceDetectorME detector = new SentenceDetectorME(model);
    String sentences[] = detector.sentDetect(paragraph);
    for(String sentence : sentences) {
        System.out.println(sentence);
    }
} catch(FileNotFoundException ex) {
    // Обработка исключения.
} catch(IOException ex) {
    // Обработка исключения.
}
```

Этот фрагмент выведет следующий результат:

```
When determining the end of sentences we need to consider several factors.
Sentences may end with exclamation marks!
Or possibly question marks?
Within sentences we may find numbers like 3.14159, abbreviations such as
found in Mr.
Smith, and possibly ellipses either within a sentence …, or at the end
of a sentence…
```

Модель не совсем правильно обработала последнее предложение, что объясняется несоответствием между образцом текста, на котором обучалась модель, и текстом, к которому модель применялась. Это подчеркивает важность правильного выбора данных для обучения. В противном случае ошибки будут сказываться на результатах задач, следущих далее в конвейере.

Вычисление характеристик модели с помощью класса SentenceDetectorEvaluator

Как отмечалось выше, часть файла с образцом текста была предназначена для проверки и определения характеристик модели, поэто-

му мы можем использовать класс SentenceDetectorEvaluator для вычисления характеристик данной модели. С этой целью возьмем из файла *sentence.train* десять последних предложений и поместим их в файл *evalSample*, который и будет источником данных для расчета характеристик. В следующем примере повторно используются переменные lineStream и sampleStream для создания потока объектов типа SentenceSample на основе содержимого файла *evalSample*:

```
lineStream = new PlainTextByLineStream(
    new FileReader("evalSample"));
sampleStream = new SentenceSampleStream(lineStream);
```

Экземпляр класса SentenceDetectorEvaluator создается с использованием ранее созданного детектора на основе класса SentenceDetectorME. Во втором аргументе конструктору можно передать объект типа SentenceDetectorEvaluationMonitor, который в данном примере не нужен, поэтому мы передаем значение null. Затем вызывается метод evaluate():

```
SentenceDetectorEvaluator sentenceDetectorEvaluator =
    new SentenceDetectorEvaluator(detector, null);
sentenceDetectorEvaluator.evaluate(sampleStream);
```

Метод getFMeasure() возвращает экземпляр класса FMeasure, представляющий величины характеристик, по которым можно оценить качество модели:

```
System.out.println(sentenceDetectorEvaluator.getFMeasure());
```

При выполнении выводится следующий результат:

```
Precision: 0.8181818181818182
Recall: 0.9
F-Measure: 0.8571428571428572
```

Здесь *Precision* (*точность*) определяет долю правильно определенных предложений в обработанном тексте, а *recall* (*полнота*) – это отношение количества найденных предложений к общему количеству предложений в тексте. *F-Measure* – это *F-мера*, или *мера Ван Ризбергена*, объединяющая точность и полноту в одной усредненной величине. F-мера определяется как взвешенное гармоническое среднее точности и полноты и позволяет оценить, насколько хорошо в целом работает модель. Для токенизации и задачи разрешения границ предложений лучше поддерживать значение точности выше отметки 90 процентов (то есть выше 0.9).

Резюме

В этой главе рассматривались многие проблемы, создающие трудности при определении границ предложений. В первую очередь это точки в числовых значениях, в сокращениях и в некоторых аббревиатурах. Кроме того, затруднения могут быть вызваны использованием многоточий и вложенных кавычек (прямая речь, цитаты и т. п.).

В языке Java поддерживается пара методик определения границ предложений. На конкретных примерах было показано применение регулярных выражений и класса `BreakIterator` для решения этой задачи. Описанные методики пригодны для работы с простыми предложениями, но для более сложных текстов результаты их работы нельзя признать удовлетворительными.

Подробно рассматривалось использование различных NLP API. Некоторые из рассматриваемых средств обрабатывают текст, опираясь на набор правил, другие используют модели. Кроме того, на примерах был показан процесс обучения модели и вычисление ее характеристик.

Тема следующей главы – поиск людей и именованных объектов в тексте.

Глава 4

Поиск людей и именованных объектов

Процесс поиска людей и именованных объектов часто обозначается специальным термином «*распознавание и идентификация именованных объектов*» (*Named Entity Recognition, NER*). Многие объекты, например люди, достопримечательности и т. п., связаны с конкретными категориями и имеют имена или названия, позволяющие их идентифицировать. Самая простая и очевидная категория – «люди» с их персональными именами. Наиболее часто принимаются во внимание следующие типы (категории) объектов, обладающих именами и названиями:

- люди;
- географические объекты и достопримечательности;
- организации;
- денежные единицы;
- время и дата;
- URL.

Поиск имен, достопримечательностей и прочих именованных объектов в документах представляет собой весьма важную задачу обработки естественного языка, решаемую во многих случаях: выполнение простого поиска, обработка запросов, работа со ссылками, устранение неоднозначности текста и определение смыслового значения текста. Например, иногда методика распознавания именованных объектов позволяет находить только объекты, принадлежащие к конкретной категории. Используя категории, можно сузить область поиска, исключив из нее другие типы объектов. Результаты распознавания именованных объектов применяются и для решения других задач обработки естественного языка, таких как морфологическая разметка текста и разрешение перекрестных ссылок.

Процесс распознавания и идентификации именованных объектов подразделяется на две задачи:

- определение объектов;
- классификация объектов.

Определение объектов подразумевает поиск местоположения (позиции) объекта в тексте. После определения позиции объекта необходимо выяснить, к какому типу объектов он относится. Результаты этих двух задач можно использовать для решения других NLP-задач: поиск, определение смыслового значения текста и т. п. Например, часто возникает необходимость идентификации имен и названий, упоминаемых в кинофильме или в обзоре литературы, и поиске других кинофильмов и/или книг, в которых также встречаются эти имена и названия. Получение информации о названиях мест может помочь в поиске ссылок на ближайшие организации, предоставляющие необходимые услуги.

Трудности, возникающие при распознавании и идентификации именованных объектов

Как и другие задачи обработки естественного языка, распознавание и идентификацию именованных объектов не всегда можно осуществить легко и просто. Токенизация текста выделяет все его компоненты, но правильно определить их смысл иногда бывает очень сложно. Использование имен собственных не устраняет проблему полностью из-за неоднозначности естественного языка. Например, вполне корректные имена Penny и Faith могут использоваться как обозначение денежной единицы и веры в высоком религиозном смысле соответственно[1]. Кроме того, можно встретить слова, подобные Georgia, которое означает название страны или штата, или имя человека.

Некоторые словосочетания могут вызвать существенные затруднения. В выражении «Metropolitan Convention and Exhibit Hall» каждое слово само по себе является корректным объектом. Если предметная область точно определена, можно без затруднений составить список допустимых именованных объектов, который будет весьма полезен при обработке.

---

[1] В русском языке подобная неоднозначность характерна для имен Вера, Надежда, Любовь. – *Прим. перев.*

Обычно распознавание и идентификация именованных объектов выполняются внутри предложений, иначе можно ошибочно обнаружить словосочетание, начинающееся в одном предложении, а заканчивающееся в другом, что приводит к появлению абсолютно неправильного объекта. Например, рассмотрим следующий фрагмент текста:

«Bob went south. Dakota went west».
(«Боб ушел на юг. Дакота ушла на запад».)

В русском языке также возможны подобные ситуации:

«Восточнее Москвы расположен город Владимир. Балашов – гораздо дальше к югу».

Если не учитывать границы предложений, в рассматриваемых фрагментах неожиданно возникают кажущиеся правильными, но неуместные (а может быть, и нежелательные) в данном контексте именованные объекты «South Dakota» (географический объект) и «Владимир Балашов» (имя человека).

Особые формы текста – адреса URL, электронной почты и особые форматы чисел – также могут вызывать затруднения при их определении. Если вспомнить о разнообразии форм записи таких объектов, идентификация становится еще более сложной. Например, при записи телефонных номеров могут использоваться (или не использоваться) круглые скобки. Группы цифр в телефонных номерах могут быть разделены дефисами, пробелами или другими символами. Нужно ли при поиске учитывать формат международных телефонных номеров?

Все эти факторы требуют применения тщательно разработанных методик распознавания и идентификации именованных объектов.

Методики распознавания именованных объектов

Существует несколько методик распознавания именованных объектов. Некоторые используют регулярные выражения, другие основаны на сопоставлении с предварительно сформированным словарем. Гибкость и мощь регулярных выражений позволяют во многих случаях выделить именованные объекты из текста. Словарь именованных объектов предназначен для поиска совпадающих токенов в обрабатываемом тексте.

Еще одна широко распространенная методика распознавания именованных объектов предполагает использование предварительно обученных моделей. Конкретные модели зависят от типа искомых объектов и от языка, на котором составлен исследуемый текст. Модели, которые хорошо работают в одной предметной области, скажем, при поиске имен и названий в веб-страницах, могут вообще не работать или выдавать удручающие результаты в других предметных областях, как, например, при обработке статей из медицинских журналов.

При обучении модели используется аннотированный (специально размеченный) блок текста, в котором четко идентифицированы требуемые объекты. Качество обучения модели оценивается количественно, с помощью следующих характеристик:

- *точность (precision)* – процентное отношение количества объектов, точно совпадающих с образцом, к общему количеству найденных объектов;
- *полнота (recall)* – процентное отношение количества объектов, точно совпадающих с образцом, к общему количеству объектов в исследуемом корпусе текстов, сформированном для обучения;
- *объединенная характеристика эффективности* – среднее гармоническое точности и полноты, называемое *сбалансированной мерой*, или *F1-мерой*, вычисляемой по формуле: F1 = 2 × точность × полнота / (точность + полнота)[1].

Все перечисленные характеристики будут использоваться в дальнейшем для оценки качества работы моделей.

Задача распознавания именованных объектов также известна как идентификация объектов и поверхностный синтаксический анализ. *Поверхностный синтаксический анализ* (*chunking*, или *shallow parsing*) – это упрощенный анализ текста, при котором определяются такие его фрагменты, как группы существительных, глаголы, глагольные группы и т. п., но без разбора их внутренней структуры и без уточнения их роли в предложении. С точки зрения человека более привычно разделять предложение на осмысленные фрагменты. Эти фрагменты формируют структуру, по которой мы определяем смысловое значение предложения. В процессе распознавания именованных объектов будут создаваться небольшие фрагменты текста (группы), такие как «Queen of England». Но внутри этих фрагмен-

[1] Это частный случай F-меры, или меры Ван Ризбергена, упоминавшейся в конце предыдущей главы. – *Прим. перев.*

тов (групп) могут располагаться другие объекты, в рассматриваемом примере это «England».

Списки и регулярные выражения

Одна из самых простых методик идентификации именованных объектов состоит в использовании списков «стандартных» имен объектов в сочетании с регулярными выражениями. Именованные объекты часто называют именами собственными. Список стандартных имен объектов может содержать названия штатов (областей), общеупотребительные имена людей, названия месяцев, часто упоминаемые географические названия и т. п. Географические справочники, фактически являющиеся алфавитными списками географических объектов, могут стать источниками для составления списка географических названий. Но при этом следует учесть, что сопровождение подобных списков почти наверняка потребует больших затрат времени. Кроме того, списки могут зависеть от языка, на котором они составляются, и от национальных (местных) форматов представления информации. Внесение изменений и дополнений в такой список может стать весьма утомительным занятием. Описанный подход будет продемонстрирован ниже, в разделе «Использование класса ExactDictionaryChunker».

При идентификации именованных объектов могут оказаться полезными регулярные выражения. Их синтаксис обеспечивает достаточную гибкость для точного определения искомого объекта. Но у гибкости регулярных выражений есть обратная сторона – трудность их понимания и сопровождения. В этой главе будет рассматриваться несколько вариантов использования регулярных выражений.

Статистические классификаторы

Статистические классификаторы позволяют определить, является ли слово начальным элементом объекта, последующим элементом объекта или вообще не относится к объекту и само по себе не является объектом. Образец текста размечается тегами, выделяющими объекты. После разработки классификатора его можно обучить на различных наборах данных для отдельных предметных областей. Недостаток этой методики состоит в необходимости привлечения квалифицированного специалиста для разметки образца текста, что отнимает немало времени. Кроме того, обучение на специализированных образцах текста сильно зависит от предметной области.

Мы рассмотрим несколько методик распознавания и идентификации именованных объектов. Начнем с применения регулярных выражений для поиска заданных объектов.

Использование регулярных выражений для распознавания и идентификации именованных объектов

Для идентификации именованных объектов в документе можно использовать регулярные выражения. Ниже описываются два обобщенных способа решения этой задачи:

- использование регулярных выражений, поддерживаемых в языке Java. Это вполне подходит для случаев, когда объекты относительно просты, а форма их постоянна;
- использование классов, специально предназначенных для применения регулярных выражений. Этот способ будет продемонстрирован на примере класса `RegExChunker` из библиотеки LingPipe.

Работа с регулярными выражениями хороша тем, что для каждой поставленной задачи не нужно «изобретать колесо». Существует множество сайтов, где можно найти библиотеки сформированных и тщательно протестированных регулярных выражений для решения практически любых задач. Одна из таких библиотек находится по адресу http://regexlib.com/Default.aspx, и именно из нее мы возьмем несколько регулярных выражений для примеров.

В большинстве примеров с применением регулярных выражений будет использоваться строковая переменная со следующим текстом:

```
private static String regularExpressionText =
    "He left his e-mail address (rgb@colorworks.com) and his "
    + "phone number,800-555-1234. We believe his current address is "
    + "100 Washington Place, Seattle, CO 12345-1234. I understand you "
    + "can also call at 123-555-1234 between 8:00 AM and 4:30 most "
    + "days. His URL is http://example.com and he was born on February "
    + "25, 1954 or 2/25/1954.";
```

Использование регулярных выражений в языке Java для поиска объектов

Демонстрацию способов применения регулярных выражений начнем с нескольких простых примеров. Сначала определим переменную

с простым регулярным выражением для определения одного из возможных форматов записи телефонных номеров:

```
String phoneNumberRE = "\\d{3}-\\d{3}-\\d{4}";
```

Приведенный ниже код используется для тестирования простых регулярных выражений в этом и в следующих примерах. Метод `compile()` класса `Pattern` принимает регулярное выражение и компилирует его в объект типа `Pattern`. Затем можно применить метод `matcher()` к целевому тексту, чтобы получить в результате объект типа `Matcher`. Этот объект позволяет многократно находить совпадения с регулярным выражением:

```
Pattern pattern = Pattern.compile(phoneNumberRE);
Matcher matcher = pattern.matcher(regularExpressionText);
while(matcher.find()) {
    System.out.println(matcher.group() + " [" + matcher.start() + ":"
        + matcher.end() + "]");
}
```

Метод `find()` возвращает значение `true`, если найдено совпадение. Метод `group()` возвращает текст, совпадающий с регулярным выражением. Методы `start()` и `end()` позволяют получить позиции совпадающего фрагмента в обрабатываемом тексте.

После выполнения кода будет выведен следующий результат:

```
800-555-1234 [68:80]
123-555-1234 [196:208]
```

Другие регулярные выражения применяются аналогичным образом. Примеры показаны в табл. 4.1. В третьем столбце представлены результаты работы приведенного выше кода с использованием соответствующих регулярных выражений из второго столбца.

Таблица 4.1. Примеры применения регулярных выражений для поиска именованных объектов

Объект	Регулярное выражение	Выводимый результат
URL	`\\b(https?|ftp|file|ldap)://[-A-Za-z0-9+&@#/%?=~_|!:,.;]*[-A-Za-z0-9+&@#/%=~_|]`	`http://example.com [256:274]`
ZIP-код	`[0-9]{5}(\\-?[0-9]{4})?`	`12345-1234 [150:160]`
Адрес e-mail	`[a-zA-Z0-9'._%+-]+@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,4}`	`rgb@colorworks.com [27:45]`
Время	`(([0-1]?[0-9])|([2][0-3])):([0-5]?[0-9])(:([0-5]?[0-9]))?`	`8:00 [217:221]` `4:30 [229:233]`

Окончание табл. 4.1

Объект	Регулярное выражение	Выводимый результат
Дата	((0?[13578]|10|12)(-|\\/)(([1-9])|(0[1-9])|([12])([0-9]?)|(3[01]?))(-|\\/)((19)([2-9])(\\d{1})|(20)([01])(\\d{1})|([8901])(\\d{1}))|(0?[2469]|11)(-|\\/)(([1-9])|(0[1-9])|([12])([0-9]?)|(3[0]?))(-|\\/)((19)([2-9])(\\d{1})|(20)([01])(\\d{1})|([8901])(\\d{1})))	2/25/1954 [315:324]

Можно было бы воспользоваться многими другими регулярными выражениями, но приведенные выше примеры демонстрируют основную методику. На примере поиска даты можно видеть, что регулярные выражения могут быть весьма сложными.

При использовании регулярных выражений достаточно часто происходят пропуск некоторых объектов и ошибочный вывод других элементов текста, не являющихся в действительности искомыми объектами. Например, если заменить исходный текст следующим выражением:

```
regularExpressionText =
    "(888)555-1111 888-SEL-HIGH 888-555-2222-J88-W3S";
```

при выполнении примера будет получен такой результат:

```
888-555-2222 [27:39]
```

То есть пропущены два первых номера телефонов, но ошибочно выведена часть индекса отрасли промышленности, интерпретированная как номер телефона.

Кроме того, имеется возможность выполнять поиск по нескольким регулярным выражениям одновременно с помощью оператора |. В следующей инструкции этот оператор используется для объединения трех регулярных выражений. Предполагается, что выражения объявлены в соответствии с типами объектов из табл. 4.1:

```
Pattern pattern = Pattern.compile(phoneNumberRE + "|"
    + timeRE + "|" + emailRegEx);
```

Если эту комбинацию регулярных выражений применить к исходному тексту в переменной `regularExpressionText`, объявленной в начале предыдущего раздела, результат будет таким:

```
rgb@colorworks.com [27:45]
800-555-1234 [68:80]
```

```
123-555-1234 [196:208]
8:00 [217:221]
4:30 [229:233]
```

Использование класса RegExChunker из библиотеки LingPipe

Класс `RegExChunker` применяет поверхностный синтаксический анализ для поиска именованных объектов в тексте, используя для их представления регулярные выражения. Метод `chunk()` этого класса возвращает объект типа `Chunking`, который мы уже использовали в предыдущих примерах.

Конструктор класса `RegExChunker` принимает три аргумента:

- регулярное выражение – тип `String`;
- тип именованного объекта или его категория – тип `String`;
- числовая оценка – тип `double`.

Работа этого класса будет показана на примере регулярного выражения, представляющего время. Регулярное выражение то же самое, что и в разделе «Использование регулярных выражений в языке Java для поиска объектов» текущей главы. Сначала создается экземпляр класса `Chunker`:

```
String timeRE =
   "(([0-1]?[0-9])|([2][0-3])):([0-5]?[0-9])(:([0-5]?[0-9]))?";
       Chunker chunker = new RegExChunker(timeRE, "time", 1.0);
```

Метод `chunk()` используется совместно с методом `displayChunkSet()`, как показано ниже:

```
Chunking chunking = chunker.chunk(regularExpressionText);
Set<Chunk> chunkSet = chunking.chunkSet();
displayChunkSet(chunker, regularExpressionText);
```

Определение метода `displayChunkSet()` показано ниже. Метод `chunkSet()` возвращает коллекцию типа `Set`, состоящую из экземпляров класса `Chunk`. Мы можем использовать разные методы для вывода заданных частей:

```
public void displayChunkSet(Chunker chunker, String text) {
    Chunking chunking = chunker.chunk(text);
    Set<Chunk> set = chunking.chunkSet();
    for(Chunk chunk : set) {
        System.out.println("Type: " + chunk.type() + " Entity: ["
             + text.substring(chunk.start(), chunk.end())
             + "] Score: " + chunk.score());
    }
}
```

Результат работы примера:

```
Type: time Entity: [8:00] Score: 1.0
Type: time Entity: [4:30] Score: 1.0+95
```

Мы можем также определить простой класс, содержащий строку с регулярным выражением, что позволит использовать его в других случаях. Ниже определяется класс `TimeRegexChunker`, поддерживающий поиск объектов, обозначающих время:

```
public class TimeRegexChunker extends RegExChunker {
    private final static String TIME_RE =
      "(([0-1]?[0-9])|([2][0-3])):([0-5]?[0-9])(:([0-5]?[0-9]))?";
    private final static String CHUNK_TYPE = "time";
    private final static double CHUNK_SCORE = 1.0;

    public TimeRegexChunker() {
        super(TIME_RE, CHUNK_TYPE, CHUNK_SCORE);
    }
}
```

Чтобы воспользоваться этим классом, нужно заменить в примере выше инструкцию инициализации объекта `chunker` следующей строкой:

```
Chunker chunker = new TimeRegexChunker();
```

Результат будет точно таким же, как при выполнении первой версии примера.

Использование библиотек NLP

Процесс распознавания и идентификации именованных объектов будет демонстрироваться с использованием библиотек OpenNLP, Stanford API и LingPipe. Каждая библиотека предлагает свою методику, позволяющую достаточно успешно решать задачу поиска именованных объектов в тексте. Во всех примерах в данном разделе будет обрабатываться следующий текст, содержащийся в массиве строк:

```
String sentences[] = {"Joe was the last person to see Fred. ",
  "He saw him in Boston at McKenzie's pub at 3:00 where he "
  + "paid $2.45 for an ale. ",
  "Joe wanted to go to Vermont for the day to visit a cousin who "
  + "works at IBM, but Sally and he had to look for Fred"};
```

Использование библиотеки OpenNLP для поиска именованных объектов

Поиск именованных объектов с помощью библиотеки OpenNLP мы продемонстрируем на примере класса `TokenNameFinderModel`. Кроме того, мы покажем процедуру вычисления вероятности правильной идентификации найденного объекта.

Обобщенный подход предполагает преобразование исходного текста в последовательность токенизированных предложений, создание экземпляра класса `TokenNameFinderModel` и применение метода `find()` для идентификации искомых объектов в тексте.

В следующем примере демонстрируется использование класса `TokenNameFinderModel`. Сначала обработаем одно простое предложение, затем перейдем к группе предложений. Отдельное предложение определено ниже:

```
String sentence = "He was the last person to see Fred.";
```

В файлах *en-token.bin* и *en-ner-person.bin* содержатся модели для токенизатора и для поиска имен соответственно. Объект типа `InputStream` для чтения этих файлов создается в блоке попытки захвата ресурсов, как показано ниже:

```
try(InputStream tokenStream = new FileInputStream(
        new File(getModelDir(), "en-token.bin"));
        InputStream modelStream = new FileInputStream(
            new File(getMOdelDir(), "en-ner-person.bin"));) {
    ...
} catch(Exception ex) {
    // Обработка исключений.
}
```

Внутри блока `try` создаются объекты типа `TokenizerModel` и `Tokenizer`:

```
TokenizerModel tokenModel = new TokenizerModel(tokenStream);
Tokenizer tokenizer = new TokenizerME(tokenModel);
```

Далее создается экземпляр класса `NameFinderME` с использованием модели поиска имен людей:

```
TokenNameFinderModel entityModel =
    new TokenNameFinderModel(modelStream);
NameFinderME nameFinder = new NameFinderME(entityModel);
```

Теперь мы можем воспользоваться методом `tokenize()` для токенизации текста и методом `find()` для идентификации имен людей в этом

тексте. Метод find() принимает токенизированный массив строк String и возвращает массив объектов Span, как показано ниже:

```
String tokens[] = tokenizer.tokenize(sentence);
Span nameSpans[] = nameFinder.find(tokens);
```

Работа с классом Span подробно рассматривалась в главе 3 «Поиск предложений». В ней содержится информация о позициях найденных именованных объектов. Сами строки с именами хранятся в массиве tokens.

В цикле for выводятся все имена людей, найденные в исходном предложении. Информация о позиции и соответствующее имя выводятся на отдельных строках:

```
for(int i=0; i < nameSpans.length; i++) {
    System.out.println("Span: " + nameSpans[i].toString());
    System.out.println("Entity: " + tokens[nameSpans[i].getStart()]);
}
```

Обработка одного предложения дает следующий результат:

```
Span: [7..9) person
Entity: Fred
```

Но гораздо чаще приходится работать с несколькими предложениями, поэтому мы продолжим работу с ранее определенным массивом строк sentences. Простой цикл for из предыдущего фрагмента заменяется более сложной конструкцией. Метод tokenize() вызывается для обработки каждого предложения, затем выводится информация о найденном объекте:

```
for(String sentence : sentences) {
    String tokens[] = tokenizer.tokenize(sentence);
    Span nameSpans[] = nameFinder.find(tokens);
    for(int i=0; i < nameSpans.length; i++) {
        System.out.println("Span: " + nameSpans[i].toString());
        System.out.println("Entity: "
            + tokens[nameSpans[i].getStart()]);
    }
    System.out.println();
}
```

Результат выполнения показан ниже. После двух имен, найденных в первом предложении, вставлена пустая строка, поскольку второе предложение имен не содержит:

```
Span: [0..1) person
Entity: Joe
Span: [7..9) person
```

```
Entity: Fred

Span: [0..1) person
Entity: Joe
Span: [19..20) person
Entity: Sally
Span: [26..27) person
Entity: Fred
```

Вычисление точности идентификации именованного объекта

Объект класса `TokenNameFinderModel` при идентификации именованных объектов в тексте вычисляет вероятность правильной идентификации найденного объекта. Эта информация доступна с помощью метода `probs()`, как показано ниже. Метод возвращает массив чисел двойной точности, соответствующих элементам массива `nameSpans`:

```
double[] spanProbs = nameFinder.probs(nameSpans);
```

Эту строку нужно добавить в предыдущий пример, сразу после вызова метода `find()`. Затем в самом конце тела вложенного цикла `for` добавить следующую инструкцию:

```
System.out.println("Probability: " + spanProbs[i]);
```

При выполнении измененного примера будет выведен следующий результат. В полях *Probability* (вероятность, правдоподобие) указана степень достоверности идентификации каждого найденного именованного объекта. Например, для первого объекта данная модель определила 80.529 процента достоверности, что «Joe» является именем человека:

```
Span: [0..1) person
Entity: Joe
Probability: 0.8052914774025202
Span: [7..9) person
Entity: Fred
Probability: 0.9042160889302772

Span: [0..1) person
Entity: Joe
Probability: 0.9620970782763985
Span: [19..20) person
Entity: Sally
Probability: 0.964568603518126
Span: [26..27) person
Entity: Fred
Probability: 0.990383039618594
```

Использование других типов именованных объектов

Библиотека OpenNLP поддерживает и другие модели поиска, перечисленные в табл. 4.2. Указанные здесь модели можно скачать с сайта http://opennlp.sourceforge.net/models-1.5/. Префикс `en` означает, что модели работают с английским языком, а `ner` – это выполняемая задача, то есть распознавание и идентификация именованных объектов.

Таблица 4.2. Модели для поиска именованных объектов, поддерживаемые библиотекой OpenNLP

Модели поиска для английского языка	Имя файла
Модель поиска географических названий и мест	`en-ner-location.bin`
Модель поиска наименований денежных единиц	`en-ner-money.bin`
Модель поиска названий организаций	`en-ner-organization.bin`
Модель поиска обозначений процентных отношений	`en-ner-percentage.bin`
Модель поиска имен людей	`en-ner-person.bin`
Модель поиска обозначений времени	`en-ner-time.bin`

Если в примере из предыдущего раздела использовать файлы других моделей, можно наблюдать, как они работают с ранее определенным набором предложений:

```
InputStream modelStream = new FileInputStream(
    new File(getModelDir(), "en-ner-time.bin"));
```

При использовании модели en-ner-money.bin необходимо увеличивать на единицу индексы в массиве `tokens`. Если этого не сделать, всегда будет возвращаться только знак доллара.

Результаты работы различных моделей показаны в табл. 4.3.

Таблица 4.3. Результаты работы некоторых моделей из библиотеки OpenNLP

Модель	Выведенный результат

`en-ner-location.bin`	`Span: [4..5) location`
	`Entity: Boston`
	`Probability: 0.8656908776583051`
	`Span: [5..6) location`
	`Entity: Vermont`
	`Probability: 0.9732488014011262`

`en-ner-money.bin`	`Span: [14..16) money`
	`Entity: 2.45`
	`Probability: 0.7200919701507937`

`en-ner-organization.bin`	`Span: [16..17) organization`
	`Entity: IBM`
	`Probability: 0.9256970736336729`

Окончание табл. 4.3

Модель	Выведенный результат
en-ner-time.bin	The model was not able to detect time in this text sequence (Модель не смогла найти обозначение времени в данном тексте)

Поиск обозначений времени в предложенном тексте с помощью модели *en-ner-time.bin* не дал никаких результатов. Это говорит о том, что данная модель не сочла правдоподобным ни один из объектов, претендующих на обозначение времени, в обрабатываемом тексте.

Одновременная обработка нескольких типов объектов

Имеется возможность одновременной обработки нескольких типов именованных объектов. Для этого в цикле необходимо создать экземпляры класса `NameFinderME` для каждой модели и последовательно применять модели к отдельным предложениям, сохраняя найденные объекты.

Этот процесс демонстрируется в следующем примере. Блок `try` из предыдущего примера должен быть переписан так, чтобы экземпляр `InputStream` создавался внутри блока:

```
try {
    InputStream tokenStream = new FileInputStream(
        new File(getModelDir(), "en-token.bin"));
    TokenizerModel tokenModel = new TokenizerModel(tokenStream);
    Tokenizer tokenizer = new TokenizerME(tokenModel);
    ...
} catch(Exception ex) {
    // Обработка исключений.
}
```

Внутри блока `try` определяется массив строк для хранения имен файлов моделей. Мы будем использовать модели поиска имен людей, географических названий и организаций:

```
String modelNames[] = {"en-ner-person.bin",
    "en-ner-location.bin", "en-ner-organization.bin"};
```

Для хранения найденных этими моделями объектов создается экземпляр списка `ArrayList`:

```
ArrayList<String> list = new ArrayLIst();
```

Оператор `for-each` используется для поочередной загрузки моделей и создания соответствующего экземпляра класса `NameFinderME`:

```
for(String name : modelNames) {
    TokenNameFinderModel entityModel = new TokenNameFinderModel(
        new FileInputStream(new File(getModelDir(), name)));
    NameFinderME nameFinder = new NameFinderME(entityModel);
    ...
}
```

В предыдущих примерах мы не задумывались над тем, чтобы определить, в каких предложениях были найдены именованные объекты. Сделать это нетрудно, нужно лишь заменить цикл `for-each` простым циклом `for`, чтобы отслеживать индексы предложений, как показано в следующем примере, где используется целочисленная переменная `index` для обхода предложений. В остальном код работает так же, как раньше:

```
for(int index = 0; index < sentences.length; index++) {
    String tokens[] = tokenizer.tokenize(sentences[index]);
    Span nameSpans[] = nameFinder.find(tokens);
    for(Span span : nameSpans) {
        list.add("Sentence: " + index + " Span: " + span.toString()
            + " Entity: " + tokens[span.getSpan()]);
    }
}
```

После этого выводятся все найденные объекты:

```
for(String element : list) {
    Syste.out.println(element);
}
```

Результат приведен ниже:

```
Sentence: 0 Span: [0..1) person Entity: Joe
Sentence: 0 Span: [7..9) person Entity: Fred
Sentence: 2 Span: [0..1) person Entity: Joe
Sentence: 2 Span: [19..20) person Entity: Sally
Sentence: 2 Span: [26..27) person Entity: Fred
Sentence: 1 Span: [4..5) location Entity: Boston
Sentence: 1 Span: [5..6) location Entity: Vermont
Sentence: 2 Span: [16..17) organization Entity: IBM
```

Использование библиотеки Stanford API для поиска именованных объектов

Для распознавания и идентификации именованных объектов будет использован класс `CRFClassifier`, который поддерживает реализацию модели, именуемой *линейной последовательностью условно случайных полей* (*Conditional Random Field, CRF*)[1].

[1] CRF представляет собой отдельный класс методов статистического моделирования. – *Прим. перев.*

Демонстрацию применения класса CRFClassifier начнем с объявления строки, определяющей имя файла классификатора:

```
String model = getModelDir() +
    "\\english.conll.4class.distsim.crf.ser.gz";
```

Затем создается классификатор, использующий заданную модель:

```
CRFClassifier<CoreLabel> classifier =
    CRFClassifier.getClassifierNoExceptions(model);
```

Метод classify() принимает одну строку, содержащую обрабатываемый текст. Чтобы воспользоваться ранее определенным массивом sentences, необходимо преобразовать его содержимое в строку:

```
String sentence = "";
for(String element : sentences) {
    sentence += element;
}
```

После этого метод classify() можно применить к полученной строке текста:

```
List<List<CoreLabel>> entityList = classifier.classify(sentence);
```

В результате возвращается список List, состоящий из списков List с объектами типа CoreLabel. Класс CoreLabel представляет отдельное слово и дополнительную информацию о нем. Внутренний список составлен из таких слов. Во внешнем цикле for-each, в следующем фрагменте, переменная internalList представляет ссылку на одно предложение из текста. Во внутреннем цикле for-each выводится каждое слово из внутреннего списка. Метод word() возвращает слово, а метод get() – его тип.

Слова и их типы выводятся, как показано ниже:

```
for(List<CoreLabel> internalList : entityList) {
    for(CoreLabel coreLabel : internalList) {
        String word = coreLabel.word();
        String category = coreLabel.get(
            CoreAnnotations.AnswerAnnotation.class);
        System.out.println(word + ":" + category);
    }
}
```

Поскольку каждое слово выводится в отдельной строке, ниже приводится только часть полученного результата. Буква «O» обозначает категорию «Other» (Прочее):

```
Joe:PERSON
was:O
the:O
```

```
last:O
person:O
to:O
see:O
Fred:PERSON
.:O
He:O
...
look:O
for:O
Fred:PERSON
```

Чтобы отсечь все слова, не являющиеся именованными объектами, нужно заменить вызов метода `println()` следующей конструкцией, позволяющей исключить категорию «Other» (Прочее):

```
if(!"O".equals(category)) {
    System.out.println(word + ":" + category);
}
```

После этого результаты выводятся в более простой форме:

```
Joe:PERSON
Fred:PERSON
Boston:LOCATION
McKenzie:PERSON
Joe:PERSON
Vermont:LOCATION
IBM:ORGANIZATION
Sally:PERSON
Fred:PERSON
```

Использование библиотеки LingPipe для поиска именованных объектов

В разделе «Использование регулярных выражений для распознавания и идентификации именованных объектов» выше мы уже применяли средства поддержки регулярных выражений из библиотеки LingPipe. Здесь мы покажем, как работают модели именованных объектов и класс `ExactDictionaryChunker` для решения задачи распознавания и идентификации именованных объектов.

Использование моделей именованных объектов из библиотеки LingPipe

Библиотека LingPipe предлагает несколько моделей именованных объектов, которые можно использовать для поверхностного анализа текста. В каждом специальном файле содержится сериализованный объект, который можно прочитать и применить к тексту. Все объекты

реализуют интерфейс `Chunker`. Результаты поверхностного синтаксического анализа возвращаются в наборе объектов типа `Chunking`, которые соответствуют найденным именованным объектам.

Список моделей для распознавания и идентификации именованных объектов приведен в табл. 4.4. Файлы моделей можно скачать с сайта http://alias-i.com/lingpipe/web/models.html.

Таблица 4.4. Список моделей для распознавания и идентификации именованных объектов из библиотеки LingPipe

Тематика текста	Корпус текстов	Имя файла
Новости на английском языке	MUC-6	`ne-en-news-muc6.AbstractCharLmRescoringChunker`
Генетика на английском языке	GeneTag	`ne-en-bio-genetag.HmmChunker`
Геномика (раздел молекулярной генетики) на английском языке	GENIA	`ne-en-bio-genia.TokenShapeChunker`

Воспользуемся моделью из файла *ne-en-news-muc6.AbstractCharLmRescoringChunker*, чтобы продемонстрировать практическое применение класса, упомянутого в имени этого файла. Начнем с блока `try-catch`, позволяющего перехватывать и обрабатывать исключения. Файл открывается, а затем связанный с ним поток передается в метод `readObject()` класса `AbstractExternalizable` для создания экземпляра класса `Chunker`. Этот метод читает сериализованную модель из файла:

```
try {
    File modelFile = new File(getModelDir(),
        "ne-en-news-muc6.AbstractCharLmRescoringChunker");
    Chunker chunker =
        (Chunker)AbstractExternalizable.readObject(modelFile);
    ...
} catch(IOException | ClassNotFoundException ex) {
    // Обработка исключения.
}
```

Интерфейсы `Chunker` и `Chunking` предоставляют методы для работы с набором элементов, полученных в результате поверхностного синтаксического анализа текста. Метод `chunk()` возвращает объект, реализующий интерфейс `Chunking`. Следующий код выводит элементы, найденные в каждом предложении обрабатываемого текста:

```
for(int i=0; i < sentences.length; ++i) {
    Chunking chunking = chunker.chunk(sentences[i]);
    System.out.println("Chunking=" + chunking);
}
```

Результат его работы выглядит следующим образом:

```
Chunking=Joe was the last person to see Fred. : [0-3:PERSON@-Infinity,
31-35:ORGANIZATION@-Infinity]
Chunking=He saw him in Boston at McKenzie's pub at 3:00 where he paid
$2.45 for an ale. : [14-20:LOCATION@-Infinity, 24-32:PERSON@-Infinity]
Chunking=Joe wanted to go to Vermont for the day to visit a cousin who
works at IBM, but Sally and he had to look for Fred : [0-3:PERSON@-
Infinity, 20-27:ORGANIZATION@-Infinity, 71-74:ORGANIZATION@-Infinity,
109-113:ORGANIZATION@-Infinity]
```

Для извлечения конкретных фрагментов информации можно также воспользоваться методами класса `Chunk`, как показано ниже. Оператор `for` из предыдущего примера заменяется на оператор `for-each`, в теле которого вызывается метод `displayChunkSet()`, разработанный в разделе «Использование класса RegExChunker из библиотеки LingPipe» выше:

```
for(String sentence : sentences) {
    displayChunkSet(chunker, sentence);
}
```

Результат показан ниже. Но следует отметить, что тип именованного объекта не во всех случаях определен правильно.

```
Type: PERSON Entity: [Joe] Score: -Infinity
Type: ORGANIZATION Entity: [Fred] Score: -Infinity
Type: LOCATION Entity: [Boston] Score: -Infinity
Type: PERSON Entity: [McKenzie] Score: -Infinity
Type: PERSON Entity: [Joe] Score: -Infinity
Type: ORGANIZATION Entity: [Vermont] Score: -Infinity
Type: ORGANIZATION Entity: [IBM] Score: -Infinity
Type: ORGANIZATION Entity: [Fred] Score: -Infinity
```

Использование класса ExactDictionaryChunker

Класс `ExactDictionaryChunker` предоставляет простой способ создания словаря именованных объектов и их типов. В дальнейшем такой словарь можно использовать для поиска именованных объектов в тексте. Имена сохраняются в объекте типа `MapDictionary`, после чего класс `ExactDictionaryChunker` можно применять для извлечения фрагментов текста, совпадающих с элементами словаря.

Интерфейс `AbstractDictionary` поддерживает основные операции с именами, категориями и оценками достоверности. Оценки достоверности используются в процессе поиска совпадений. Классы `MapDictionary` и `TrieDictionary` осуществляют реализацию интерфейса `AbstractDictionary`. Класс `TrieDictionary` хранит информацию в спе-

циальной символьной *Trie-структуре*[1]. Такой подход требует меньшего объема памяти, что очень важно. В нашем примере будет показано применение класса `MapDictionary`.

В первую очередь объявляется экземпляр словаря типа `MapDictionary`:

```
private MapDictionary<String> dictionary;
```

В словарь будут включены имена объектов, представляющих интерес при поиске. Созданную модель словаря необходимо инициализировать, чем и занимается приведенный ниже метод `initializeDictionary()`. Используемый при инициализации конструктор `DictionaryEntity()` принимает три аргумента следующих типов:

- имя объекта – тип `String`;
- категория объекта – тип `String`;
- оценка достоверности (score) для данного объекта – тип `double`.

Оценка достоверности нужна при определении совпадений. Далее в словарь добавляются несколько целевых объектов:

```
private static void initializeDictionary() {
    dictionary = new MapDictionary<String>();
    dictionary.addEntry(
        new DictionaryEntry<String>("Joe", "PERSON", 1.0));
    dictionary.addEntry(
        new DictionaryEntry<String>("Fred", "PERSON", 1.0));
    dictionary.addEntry(
        new DictionaryEntry<String>("Boston", "PLACE", 1.0));
    dictionary.addEntry(
        new DictionaryEntry<String>("pub", "PLACE", 1.0));
    dictionary.addEntry(
        new DictionaryEntry<String>("Vermont", "PLACE", 1.0));
    dictionary.addEntry(
        new DictionaryEntry<String>("IBM", "ORGANIZATION", 1.0));
    dictionary.addEntry(
        new DictionaryEntry<String>("Sally", "PERSON", 1.0));
}
```

Созданный словарь будет использовать экземпляр класса `ExactDictionaryChunker`, конструктор которого принимает аргументы следующих типов:

- `Dictionary<String>` – словарь, содержащий искомые объекты;
- `TokenizerFactory` – токенизатор, используемый при поверхностном синтаксическом анализе;

---

[1] Trie-структура, или бор – разновидность дерева поиска; от англ. *reTRIEval* – выБОРка. Часто используются термины «нагруженное дерево» и «префиксное дерево». – *Прим. перев.*

- boolean – если передано значение true, инструмент поверхностного синтаксического анализа должен возвращать все совпадения;
- boolean – если передано значение true, при поиске совпадений необходимо учитывать регистр символов.

Совпадения могут быть перекрывающимися. Например, в словосочетании «The First National Bank» объект «bank» может быть найден и как отдельное слово, и как часть фразы, заданной для поиска. Третий параметр определяет, нужно ли возвращать все обнаруженные совпадения в подобных случаях.

Далее инициализируется словарь. Создается экземпляр класса ExactDictionaryChunker с использованием токенизатора Indo-European, определяется необходимость возврата всех найденных совпадений без учета регистра символов в токенах:

```
initializeDictionary();
ExactDictionaryChunker dictionaryChunker =
    new ExactDictionaryChunker(dictionary,
        IndoEuropeanTokenizerFactory.INSTANCE, true, false);
```

Затем объект dictionaryChunker последовательно применяется к каждому предложению, как показано ниже. Кроме того, мы снова воспользуемся методом displayChunkSet(), разработанным в разделе «Использование класса RegExChunker из библиотеки LingPipe» выше:

```
for(String sentence : sentences) {
    System.out.println("\nTEXT=" + sentence);
    displayChunkSet(dictionaryChunker, sentence);
}
```

Выполнив пример, получим следующий результат:

```
TEXT=Joe was the last person to see Fred.
Type: PERSON Entity: [Joe] Score: 1.0
Type: PERSON Entity: [Fred] Score: 1.0

TEXT=He saw him in Boston at McKenzie's pub at 3:00 where he paid $2.45
for an ale.
Type: PLACE Entity: [Boston] Score: 1.0
Type: PLACE Entity: [pub] Score: 1.0

TEXT=Joe wanted to go to Vermont for the day to visit a cousin who works
at IBM, but Sally and he had to look for Fred
Type: PERSON Entity: [Joe] Score: 1.0
Type: PLACE Entity: [Vermont] Score: 1.0
Type: ORGANIZATION Entity: [IBM] Score: 1.0
```

```
Type: PERSON Entity: [Sally] Score: 1.0
Type: PERSON Entity: [Fred] Score: 1.0
```

Для текста в данном примере работа выполнена успешно, но для обработки реальных текстов потребуется очень большой словарь, создание которого связано с огромными трудозатратами.

Обучение модели

Для демонстрации процесса обучения модели будет использоваться библиотека OpenNLP. Файл для обучения содержит текст, который должен соответствовать обязательным требованиям:

- наличие в тексте пометок, обозначающих именованные объекты;
- каждое предложение должно быть размещено на отдельной строке.

В нашем случае файл *en-ner-person.train* содержит следующий обучающий текст:

```
<START:person> Joe <END> was the last person to see <START:person> Fred
<END>.
He saw him in Boston at McKenzie's pub at 3:00 where he paid $2.45 for
an ale.
<START:person> Joe <END> wanted to go to Vermont for the day to visit a
cousin who works at IBM, but <START:person> Sally <END> and he had to
look for <START:person> Fred <END>.
```

Некоторые методы в рассматриваемом примере могут сгенерировать исключения, поэтому основная часть кода заключена в блок `try` с попыткой захвата ресурсов и созданием потока вывода для записи обученной модели в файл, как показано ниже:

```
try(OutputStream modelOutputStream = new BufferedOutputStream(
       new FileOutputStream(new File("modelFile")));) {
    ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

Внутри блока `try` создается объект типа `OutputStream<String>`, использующий класс `PlainTextByLineStream`. Конструктор принимает экземпляр `FileInputStream` и возвращает каждую строку в виде объекта типа `String`. Источником входных данных является файл *en-ner-person.train*. Аргумент `"UTF-8"` определяет кодировку исходного текста:

```
ObjectStream<String> lineStream = new PlainTextByLineStream(
    new FileInputStream("en-ner-person.train"), "UTF-8");
```

Объект lineStream содержит строки, размеченные специальными тегами, определяющими именованные объекты в тексте. Последние необходимо преобразовать в объекты NameSample, чтобы модель можно было обучить. Преобразование выполняется с помощью класса NameSampleDataStream, как показано ниже. Объект типа NameSample хранит имена объектов, найденные в исходном тексте:

```
ObjectStream<NameSample> sampleStream =
    new NameSampleDataStream(lineStream);
```

После этого можно вызвать метод train():

```
TokenNameFinderModel model = NameFinderME.train(
    "en", "person", sampleStream,
    Collections.<String, Object>emptyMap(), 100, 5);
```

Смысл аргументов метода train() описан в табл. 4.5.

Таблица 4.5. Краткое описание аргументов метода train()

Параметр	Описание
"en"	Код используемого языка
"person"	Тип (категория) именованного объекта
sampleStream	Обрабатываемые данные
null	Ресурсы
100	Количество итераций
5	Пороговое ограничение

Затем модель сериализуется в выходной файл:

```
model.serialize(modelOutputStream);
```

Результат работы примера приведен ниже. В целях экономии места протокол итераций существенно сокращен: показана только наиболее важная информация о процессе обучения модели:

```
Indexing events using cutoff of 5
  Computing event counts... done. 53 events
  Indexing... done.
Sorting and merging events... done. Reduced 53 events to 46.
Done indexing.
Incorporating indexed data for training... done.
  Number of Event Tokens: 46
      Number of Outcomes: 2
    Number of Predicates: 34
...done.
Computing model parameters ...
Performing 100 iterations.
```

```
  1: ... loglikelihood=-36.73680056967707  0.05660377358490566
  2: ... loglikelihood=-17.499660626361216  0.9433962264150944
  3: ... loglikelihood=-13.216835449617108  0.9433962264150944
  4: ... loglikelihood=-11.461783667999262  0.9433962264150944
  5: ... loglikelihood=-10.380239416084963  0.9433962264150944
  6: ... loglikelihood=-9.570622475692486  0.9433962264150944
  7: ... loglikelihood=-8.919945779143012  0.9433962264150944
...
 99: ... loglikelihood=-3.513810438211968  0.9622641509433962
100: ... loglikelihood=-3.507213816708068  0.9622641509433962
```

Оценка характеристик модели

Характеристики модели можно вычислить с помощью класса `TokenNameFinderEvaluator`. В процессе вычисления используется специально размеченный образец текста. Для этого простого примера создан файл *en-ner-person.eval*, в котором содержится следующий текст:

```
<START:person> Bill <END> went to the farm to see <START:person> Sally
<END>.
Unable to find <START:person> Sally <END> he went to town.
There he saw <START:person> Fred <END> who had seen <START:person> Sally
<END> at the book store with <START:person> Mary <END>.
```

Следующий код вычисляет характеристики модели. Обученная в предыдущем разделе модель здесь используется как аргумент конструктора класса `TokenNameFinderEvaluator`. Экземпляр класса `NameSampleDataStream` создается на основе файла с образцом текста. Вычисления выполняет метод `evaluate()` класса `TokenNameFinderEvaluator`:

```
TokenNameFinderEvaluator evaluator =
    new TokenNameFinderEvaluator(new NameFinderME(model));
lineStream = new PlainTextByLineStream(
    new FileInputStream("en-ner-person.eval"), "UTF-8");
sampleStream = new NameSampleDataStream(lineStream);
evaluator.evaluate(sampleStream);
```

Чтобы определить, насколько успешно модель справилась с обработкой контрольных данных, вызывается метод `getFMeasure()`. После этого выводятся результаты:

```
FMeasure result = evaluator.getMeasure();
System.out.println(result.toString());
```

Ниже показаны полученные оценки: точность, полнота и F-мера (мера Ван Ризбергена), по которым видно, что только для половины именованных объектов были найдены точные совпадения. Полнота

представляет собой процентное отношение количества правильно определенных объектов к общему количеству релевантных объектов в исследуемом корпусе текстов. Общая эффективность представлена сбалансированной мерой, или F1-мерой, вычисляемой как среднее гармоническое точности и полноты по формуле: *F1 = 2 × Точность × Полнота / (Точность + Полнота)*.

```
Precision: 0.5
Recall: 0.25
F-Measure: 0.3333333333333333
```

Для создания модели более высокого качества необходимо использовать намного больший объем данных для обучения и вычисления оценок эффективности. Здесь же важнее всего было продемонстрировать основную методику, применяемую для обучения и расчета характеристик модели распознавания и идентификации именованных объектов.

Резюме

Процесс распознавания и идентификации именованных объектов делится на две задачи: определение и последующая классификация объектов. Наиболее часто используются такие категории, как имена людей, географические названия и названия организаций. Эта важная задача реализуется во многих приложениях для поддержки поиска, обработки ссылок и определения смыслового значения текста. Результаты распознавания именованных объектов применяются и для решения других задач обработки естественного языка.

Мы рассмотрели несколько методик распознавания и идентификации именованных объектов. Регулярные выражения поддерживаются как базовыми классами языка Java, так и библиотеками NLP. Эта методика хорошо подходит для многих приложений, к тому же в свободном доступе имеется огромное количество библиотек регулярных выражений, готовых к употреблению.

Методики, основанные на словарях, также имеют право на существование и неплохо работают в некоторых приложениях. Но следует отметить, что для формирования словарей иногда требуются весьма существенные трудозатраты. Для демонстрации этой методики использовался класс `MapDictionary` из библиотеки LingPipe.

В процессе распознавания и идентификации именованных объектов также применяется обучение моделей. Этот подход был показан

на примере класса `NameFinderME` из библиотеки OpenNLP. Процесс обучения, продемонстрированный здесь, очень похож на примеры обучения моделей из предыдущих глав.

В следующей главе рассматривается задача определения частей речи, таких как существительные, прилагательные и предлоги.

Глава 5

Определение частей речи

В предыдущей главе мы выполняли поиск таких элементов текста, как имена людей, географические названия и названия организаций. В текущей главе мы рассмотрим процесс определения *частей речи* (*Parts Of Speech*, *POS*). Это элементы, которые в английском, русском и многих других языках классифицируются как грамматические категории: существительные, глаголы, прилагательные и т. д. Сразу отметим важность контекста слова для определения его грамматического типа.

Мы рассмотрим процесс разметки (tagging), который подразумевает размещение тегов, соответствующих частям речи, и является ключевым в определении частей речи. Кратко обоснуем его важность и уделим внимание различным факторам, создающим трудности при определении частей речи. Затем для демонстрации процесса разметки по частям речи используем различные библиотеки NLP. В заключение будет показано обучение модели, предназначенной для обработки специализированного текста.

Процесс разметки

Разметка (*tagging*) – это процесс связывания некоторого конкретного описания с токеном или с фрагментом текста. Такое описание называется *тегом* (*tag*). *Разметка по частям речи* (*POS tagging*)[1] – это процесс присваивания токену тега, соответствующего конкретной части речи. Обычно это теги, обозначающие существительное (noun), глагол (verb) и прилагательное (adjective).

Например, рассмотрим следующее предложение:

---

[1] В русской Википедии используется термин «морфологическая разметка». – *Прим. перев.*

«The cow jumped over the moon».
(«Корова перепрыгнула через луну».)

Для большинства простых примеров в начале данной главы мы будем показывать результат разметки по частям речи, используя инструмент разметки из библиотеки OpenNLP, рассматриваемый ниже, в разделе «Использование инструментов разметки по частям речи из библиотеки OpenNLP». Если применить этот инструмент к предыдущему примеру, мы получим следующий результат. Обратите внимание, что за каждым словом следует символ слэша, затем тег, соответствующий части речи. Теги будут расшифрованы несколько позже:

```
The/DT cow/NN jumped/VBD over/IN the/DT moon./NN
```

В зависимости от контекста слову может быть поставлено в соответствие более одного тега. Например, слово «saw» может быть существительным или глаголом. В русском языке слово «были» представляет либо существительное (множественное число от «быль»), либо глагол (форма прошедшего времени множественного числа от «быть»). Если слово может быть отнесено к разным категориям, для вероятностного определения правильной категории используется информация о его положении в тексте, о словах, окружающих его, и т. п. Например, если имеется последовательность: проблемному слову предшествует определяющее слово, а за ним следует существительное, – проблемному слову назначается тег «прилагательное».

В общем виде процесс разметки состоит из токенизации текста, расстановки тегов для токенов, не вызывающих сомнений, и разрешения неоднозначности прочих тегов. Для идентификации частей речи (разметки) применяются алгоритмы двух основных типов:

- *алгоритмы, основанные на правилах* (*rule-based algorithms*): инструменты разметки на основе правил используют набор правил и словарь слов с соответствующими возможными тегами. Правила начинают действовать, когда слово связано с несколькими тегами. Достаточно часто в правилах учитываются предыдущее и/или последующее слово в контексте для выбора тега;
- *стохастические алгоритмы* (*stochastic algorithms*): стохастические инструменты разметки основаны либо на марковских моделях, либо на механизме поиска *Cue*, использующем деревья решений или принцип максимальной энтропии. Марковские модели – это конечные автоматы, в которых каждое состояние имеет два варианта распределения вероятностей, а цель работы марковских моделей – определение оптимальной последо-

вательности тегов в исследуемом предложении. Кроме того, используются *скрытые марковские модели* (*Hidden Markov Models, HMM*), для которых характерны неизвестные («невидимые») переходы между состояниями.

Инструмент разметки на основе принципа максимальной энтропии использует статистические данные для определения частей речи и часто обращается к корпусу текстов для обучения модели. Корпус текстов представляет собой набор слов, отмеченных тегами, соответствующих частям речи. Корпусы текстов существуют для многих языков. Их разработка требует огромных трудозатрат. Наиболее часто для английского языка используются корпусы Penn Treebank (http://www.tis.upenn.edu/~treebank/) и Brown Corpus (http://www.essex.ac.uk/linguistics/external/clmt/w3c/corpus_ling/content/corpora/list/private/brown/brown.html).

Приведем пример из корпуса текстов Penn Treebank, иллюстрирующий разметку текста тегами, соответствующими частям речи:

```
Well/UH what/WP do/VBP you/PRP think/VB about/IN the/DT idea/NN of/IN
,/, uh/UH ,/, kids/NNS having/VBG to/TO do/VB public/JJ service/NN
work/NN for/IN a/DT year/NN ?/.
```

Как правило, в английском языке выделяют девять основных частей речи: существительное (noun), глагол (verb), артикль (article), прилагательное (adjective), предлог (preposition), местоимение (pronoun), наречие (adverb), союз (conjunction) и междометие (interjection). В русском языке нет артиклей, но в дополнение к перечисленным частями речи также считаются имя числительное, причастие, деепричастие и частица. Но при более полном и тщательном анализе зачастую требуются дополнительные категории и подкатегории. В общей сложности определено около 150 различных частей речи. В некоторых случаях может возникать необходимость создания новых тегов. Краткий список тегов, обозначающих части речи, приведен в табл. 5.1. Этими тегами мы будем пользоваться чаще всего на протяжении всей главы.

Таблица 5.1. Теги, обозначающие части речи, встречающиеся наиболее часто

Тег части речи	Описание
NN	Существительное в единственном числе или не имеющее формы множественного числа
DT	Определяющее слово (в том числе артикль)
VB	Глагол в основной, «словарной» форме

Окончание табл. 5.1

Тэг части речи	Описание
VBD	Глагол, форма прошедшего времени
VBZ	Глагол, третье лицо, единственное число, настоящее время
IN	Предлог или подчинительный союз
NNP	Имя собственное, единственное число
TO	to
JJ	Прилагательное

Более полный список приводится в табл. 5.2. С небольшими изменениями этот список взят с сайта https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html. Самый полный список тегов The University of Pennsylvania (Penn) Treebank Tagset находится здесь: http://www.comp.leeds.ac.uk/cclass/tagsets/upenn.html. *Набор тегов* обозначается специальным термином *tag set*.

Таблица 5.2. Расширенный набор тегов, обозначающих части речи

Тег	Описание	Тег	Описание
CC	Соединительный союз	PRP$	Притяжательное местоимение
CD	Количественное числительное	RB	Наречие
DT	Определяющее слово	RBR	Наречие, сравнительная степень
EX	There с глаголами существования и движения	RBS	Наречие, превосходная степень
FW	Иностранное слово	RP	Частица
IN	Предлог или подчинительный союз	SYM	Символ
JJ	Прилагательное	TO	to
JJR	Прилагательное, сравнительная степень	UH	Междометие
JJS	Прилагательное, превосходная степень	VB	Глагол, основная, «словарная» форма
LS	Маркер элемента списка	VBD	Глагол, прошедшее время
MD	Модальный грамматический элемент	VBG	Глагол, герундий или причастие настоящего времени
NN	Существительное, единственное число или не имеющее формы множественного числа	VBN	Глагол, причастие прошедшего времени
NNS	Существительное, множественное число	VBP	Глагол, кроме формы третьего лица единственного числа настоящего времеми
NNP	Имя собственное, единственное число	VBZ	Глагол, третье лицо, единственное число, настоящее время

Окончание табл. 5.2

Тэг	Описание	Тэг	Описание
NNPS	Имя собственное, множественное число	WDT	Вопросительное определяющее слово
PDT	Особый тип определяющего слова, количественная характеристика (но не числительное: все, несколько и т. п.)	WP	Вопросительное местоимение
POS	Притяжательное окончание	WP$	Вопросительное притяжательное местоимение
PRP	Личное местоимение	WRB	Вопросительное наречие

Разработка корпуса текстов вручную требует немалых трудозатрат, поэтому для автоматизации этого процесса были созданы специальные статистические методики. Достаточно много корпусов текстов представлено в свободном доступе. Одним из первых был Brown Corpus (http://clu.uni.no/icame/manuals/BROWN/INDEX.HTM). Среди более новых корпусов можно выделить British National Corpus (http://www.natcorp.ox.ac.uk/corpus/index.xml), содержащий более 100 миллионов слов, и American National Corpus (http://www.anc.org/). Список корпусов текстов можно найти в Википедии: http://en.wikipedia.org/wiki/List_of_text_corpora.

Важное значение инструментов разметки по частям речи

Правильная разметка предложения может способствовать улучшению качества выполнения последующих задач обработки текста. Если мы точно знаем, что слово «sue» является глаголом, а не существительным, это существенная помощь в определении взаимосвязей между токенами. Идентификация частей речи, словосочетаний, частей сложных предложений и всех взаимосвязей между ними называется *синтаксическим анализом* (*parsing*). Синтаксический анализ кардинально отличается от токенизации, когда элементы-«слова» выделяются без учета их смыслового значения.

Разметка по частям речи используется как основа для многих последующих задач NLP, таких как анализ вопросов/запросов и эмоциональной окраски текста. Некоторые сайты социальных сетей часто желают знать настроения и чувства своих клиентов. Методика индексации текстов часто пользуется данными разметки по частям речи. При обработке и распознавании речи теги могут помочь в определении правильного произношения слов.

Трудности в идентификации частей речи

При разметке по частям речи для любого естественного языка возникают затруднения, вызываемые несколькими причинами. С большинством английских слов связаны два и более тегов, обозначающих части речи. Обращение к словарю не всегда решает проблему определения части речи для конкретного слова. Например, смысловое значение слов «bill» и «force» зависит от их контекста. В следующем предложении показано их использование и в качестве существительных, и в качестве глаголов.

«Bill used the force to force the manager to tear the bill in two».
(«Билл силой заставил управляющего разорвать счет пополам».)

В русском языке тоже встречаются подобные ситуации:

«Мама мыла грязнулю, не жалея мыла».

Применение инструмента разметки из библиотеки OpenNLP к предложению на английском языке позволяет получить следующий результат:

```
Bill/NNP used/VBD the/DT force/NN to/TO force/VB the/DT manager/NN to/TO
tear/VB the/DT bill/NN in/IN two./PRP$
```

Использование *смс-языка* (*textese* или *sms-language*), то есть сочетания разных форм текста, включая сокращения, хэш-теги, значки, выражающих эмоции, жаргонные выражения, при обмене информацией в коммуникационных средах, таких как твиты, чаты и т. п., увеличивает сложность расстановки тегов в предложениях. Например, следующее сообщение совсем не просто разметить тегами, соответствующими частям речи:

«AFAIK she H8 cth! BTW had a GR8 tym at the party BBIAM».

Если «расшифровать» это сообщение, то получим следующий текст:

«As far as I know, she hates cleaning the house! By the way, had a great time at the party. Be back in a minute».
(«Насколько я знаю, она ненавидит домашнюю уборку! Кстати, классная была вечеринка. Вернусь через минуту».)

Если применить инструмент из библиотеки OpenNLP к исходному сообщению, результат разметки будет выглядеть так:

```
AFAIK/NNS she/PRP H8/CD cth!/.
BTW/NNP had/VBD a/DT GR8/CD tym/NN at/IN the/DT party/NN BBIAM./.
```

Далее, в разделе «Использование класса MaxentTagger для разметки смс-языка», мы покажем, как правильно обрабатывать сообщения на смс-языке с помощью средств библиотеки LingPipe. Краткий список типичных выражений на смс-языке приведен в табл. 5.3.

Таблица 5.3. Аббревиатуры, наиболее часто используемые в смс-языке

Обычное выражение	СМС-аббревиатура	Обычное выражение	СМС-аббревиатура
As far as I know (Насколько я знаю)	AFAIK	By the way (Кстати)	BTW
Away from keyboard ((Я) не за компьютером)	AFK	You're on your own (Под твою ответственность; По твоей инициативе)	YOYO
Thanks (Спасибо)	THNX или THX	As soon as possible (Как можно быстрее)	ASAP
Today (Сегодня)	2day	What do you mean by that (Что ты имеешь в виду под этим)	WDYMBT
Before (До; перед)	B4	Be back in a minute (Вернусь через минуту)	BBIAM
See you (Пока; увидимся)	C U	Can't (Не могу; невозможно)	CNT
Haha	hh	Later (Позже)	l8R
Laughing out loud (Хохочу во весь голос)	LOL	On the other hand (С другой стороны)	OTOH
Rolling on the floor laughing (Катаюсь по полу от смеха)	ROFL или ROTFL	I don't know (Не знаю)	IDK
Great (Отлично; классно)	GR8	Cleaning the house (Домашняя уборка)	CTH
At the moment (В данное время; сейчас)	ATM	In my humble opinion (По моему скромному мнению)	IMHO

Существует несколько словарей смс-языка, но самый крупный расположен по адресу http://www.ukrainecalling.com/textspeak.aspx.

В разметке по частям речи важным этапом является токенизация. Если разделение на токены выполнено неправильно, вероятнее всего, в дальнейшем будут получены ошибочные результаты. Кроме того, возможно возникновение еще нескольких проблем:

- если текст переведен в нижний регистр, такие слова, как, например, «sam», могут означать и имя человека, и аббревиатуру SAM – System for Award Management (www.sam.gov)[1];
- необходимо принимать во внимание стяженные формы, такие как «can't», и учитывать, что вместо апострофа могут использоваться и другие символы;
- несмотря на то что словосочетания, подобные «vice versa», могут интерпретироваться как единый элемент, это еще и имя музыкальной группы в Англии, название романа и журнала;
- нельзя оставлять без внимания слова, содержащие дефис, например «first-cut» и «prime-cut», поскольку их смысловое значение отличается от смысла при раздельном использовании этих слов;
- некоторые слова содержат цифровые символы, например iPhone 5S;
- кроме того, необходимо обрабатывать специальные последовательности символов, такие как URL и адреса электронной почты.

Затруднения могут вызывать некоторые слова, заключенные в кавычки или скобки. Рассмотрим следующий пример:

«Whether “Blue” was correct or not (it’s not) is debatable».

Здесь «Blue» может обозначать синий цвет или предположительно псевдоним (ник) человека. Инструмент разметки при обработке этого предложения выдает следующий результат:

```
Whether/IN "Blue"/NNP was/VBD correct/JJ or/CC not/RB (it's/JJ not)/NN
is/VBZ debatable/VBG
```

Использование библиотек NLP API

Процесс разметки по частям речи будет демонстрироваться с использованием библиотек OpenNLP, Stanford API и LingPipe. В каждом примере обрабатывается предложение, приведенное ниже. Это первое предложение из пятой главы («Погоня вслепую») романа Жюля Верна «20 000 лье под водой» в английском переводе (Jules Verne, «Twenty Thousands Leagues Under the Sea», Chapter 5, At A Venture):

```
private String[] sentence = {"The", "voyage", "of", "the",
    "Abraham", "Lincoln", "was", "for", "a", "long", "time", "marked",
    "by", "no", "special", "incident."};
```

[1] SAM – система управления базами данных правительственных госзаказов частным предприятиям. – *Прим. перев.*

Текст для обработки не всегда можно представить в такой форме. Иногда предложение будет иметь вид единой строки:

```
String theSentence = "The voyage of the Abraham Lincoln was for a "
    + "long time marked by no special incident.";
```

Может потребоваться преобразовать строку в массив строк. Существует множество методик преобразования строки в массив слов. Метод `tokenizeSentence()` выполняет эту операцию следующим образом:

```
public String[] tokenizeSentence(String sentence) {
    String words[] = sentence.split("\S+");
    return words;
}
```

Следующий фрагмент демонстрирует применение данного метода:

```
String words[] = tokenizeSentence(theSentence);
for(String word : words) {
    System.out.print(word + " ");
}
System.out.println();
```

Результат очевиден:

```
The voyage of the Abraham Lincoln was for a long time marked by no
special incident.
```

В качестве альтернативного решения можно воспользоваться готовым токенизатором, например классом `WhitespaceTokenizer` из библиотеки OpenNLP:

```
String words[] =
    WhitespaceTokenizer.INSTANCE.tokenize(theSentence);
```

Использование инструментов разметки по частям речи из библиотеки OpenNLP

Библиотека OpenNLP предлагает несколько классов для разметки по частям речи. Для простой разметки воспользуемся классом `POSTaggerME`, а для поверхностного синтаксического анализа возьмем класс `ChunkerME`. Поверхностный синтаксический анализ предполагает объединение в группы связанных слов в соответствии с их типами. Это помогает точнее определить структуру предложения. Кроме того, рассмотрим создание и практическое использование экземпляра словаря `POSDictionary`.

Использование класса POSTaggerME для разметки по частям речи

Класс `POSTaggerME` из библиотеки OpenNLP использует принцип максимальной энтропии для обработки тегов. Инструмент разметки определяет тип тега по самому слову и его контексту. С любым обрабатываемым словом может быть связано несколько тегов. Для выбора конкретного тега применяется вероятностная модель.

Модели частей речи загружаются из файла. Чаще других используется модель *en-pos-maxent.bin*, основанная на наборе тегов Penn TreeBank. Разнообразные предварительно обученные POS-модели для библиотеки OpenNLP можно найти на сайте http://opennlp.sourceforge.net/models-1.5/.

Начнем с блока `try-catch`, предназначенного для обработки любых исключений ввода-вывода `IOException`, которые могут быть сгенерированы при загрузке модели, как показано ниже.

В данном примере используется модель из файла *en-pos-maxent.bin*:

```
try(InputStream modelIn = new FileInputStream(
    new File(getModelDir(), "en-pos-maxent.bin"));) {
    ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

Далее создаются экземпляры классов `POSModel` и `POSTaggerMe`:

```
POSModel model = new POSModel(modelIn);
POSTaggerME tagger = new POSTaggerME(model);
```

Теперь можно применить метод `tag()` объекта `tagger` для обработки текста:

```
String tags[] = tagger.tag(sentence);
```

Слова и соответствующие им теги выводятся в одной строке:

```
for(int i=0; i < sentence.length; i++) {
    System.out.print(sentence[i] + "/" + tags[i] + " ");
}
```

Результат приведен ниже. После каждого слова указан его тип (тег части речи):

```
The/DT voyage/NN of/IN the/DT Abraham/NNP Lincoln/NNP was/VBD for/IN a/
DT long/JJ time/NN marked/VBN by/IN no/DT special/JJ incident./NN
```

В любом предложении словам может быть присвоено более одного тега части речи. Метод `topKSequences()` возвращает набор последовательностей, определенных на основе вероятности правильного назначения тегов. В следующем фрагменте метод `topKSequences()` применяется к переменной `sentence`, и затем выводится результат:

```
Sequence topSequences[] = tagger.topKSequences(sentence);
for(int i=0; i < topSequences.length; i++) {
    System.out.println(topSequences[i];
}
```

В результатах ниже первое числовое значение представляет взвешенную оценку, а в квадратных скобках приведена оцениваемая последовательность тегов:

```
-0.5563571615737618 [DT, NN, IN, DT, NNP, NNP, VBD, IN, DT, JJ, NN, VBN,
IN, DT, JJ, NN]
-2.9886144610050907 [DT, NN, IN, DT, NNP, NNP, VBD, IN, DT, JJ, NN, VBN,
IN, DT, JJ, .]
-3.771930515521527 [DT, NN, IN, DT, NNP, NNP, VBD, IN, DT, JJ, NN, VBN,
IN, DT, NN, NN]
```

Проверьте и убедитесь, что вы правильно включили в программу определение класса `Sequence`. Для рассматриваемого примера необходимо использовать инструкцию `import opennlp.tools.util.Sequence`.

Класс `Sequence` содержит несколько методов, кратко описанных в табл. 5.4.

Таблица 5.4. Описание методов класса Sequence

Методы	Описание
`getOutcomes`	Возвращает список строк, представляющих теги для обрабатываемого предложения
`getProbs`	Возвращает массив переменных типа `double`, представляющих вероятность правильного определения каждого тега в последовательности
`getScore`	Возвращает взвешенное среднее значение оценки для обрабатываемого предложения

В следующем фрагменте демонстрируется применение некоторых из перечисленных выше методов. В каждой последовательности будут выведены теги и соответствующие вероятности правильного определения, разделенные символом слэша:

```
for(int i=0; i < topSequences.length; i++) {
    List<String> outcomes = topSequences[i].getOutcomes();
```

```
    double probabilities[] = topSequences[i].getProbs();
    for(int j=0; j < outcomes.size(); j++) {
        System.out.printf("%s/%5.3f ", outcomes.get(j),
        probabilities[j]);
    }
    System.out.println();
}
System.out.println();
```

Ниже приводится результат работы. Каждая пара строк представляет одну последовательность с учетом переноса обработанного текста на новую строку:

```
DT/0.992 NN/0.990 IN/0.989 DT/0.990 NNP/0.996 NNP/0.991 VBD/0.994
in/0.996 dt/0.996 jj/0.991 nn/0.994 vbn/0.860 in/0.985 dt/0.960 jj/0.919
NN/0.832
DT/0.992 NN/0.990 IN/0.989 DT/0.990 NNP/0.996 NNP/0.991 VBD/0.994
IN/0.996 DT/0.996 JJ/0.991 NN/0.994 VBN/0.860 IN/0.985 DT/0.960 JJ/0.919
./0.073
DT/0.992 NN/0.990 IN/0.989 DT/0.990 NNP/0.996 NNP/0.991 VBD/0.994
IN/0.996 DT/0.996 JJ/0.991 NN/0.994 VBN/0.860 IN/0.985 DT/0.960 NN/0.073
NN/0.419
```

Использование средств поверхностного синтаксического анализа из библиотеки OpenNLP

Процесс поверхностного синтаксического анализа подразумевает разделение предложения на фрагменты, или синтаксические элементы, которые отмечаются соответствующими тегами. Чтобы наглядно показать этот процесс, воспользуемся классом `ChunkerME`, работающим с моделью, загружаемой в экземпляр класса `ChunkerModel`. Метод `chunk()` класса `ChunkerME` непосредственно выполняет поверхностный синтаксический анализ текста. Кроме того, демонстрируется применение метода `chunkAsSpans()`, возвращающего информацию об интервалах, которые охватывают найденные синтаксические элементы. Это позволяет оценить размер каждого элемента и определить компоненты, из которых он состоит.

Модель для экземпляра класса `POSTaggerME` читается из файла *en-pos-maxent.bin*. Этот экземпляр используется для разметки текста точно так же, как в разделе «Использование класса POSTaggerME для разметки по частям речи» текущей главы. Кроме того, нам потребуется файл *en-chunker.bin* для создания экземпляра класса `ChunkerModel`, используемого объектом `ChunkerME`.

Все перечисленные модели создаются с помощью потоков ввода, как показано в следующем примере.

Процедуры открытия и закрытия файлов включены в блок try, чтобы появилась возможность обработки вероятных исключений:

```
try(InputStream posModelStream = new FileInputStream(
        getModelDir() + "\\en-pos-maxent.bin");
    InputStream chunkerStream = new FileInputStream(getModelDir()
        + "\\en-chunker.bin");) {
    ...
} catch(IOException ex) {
    // Обработка исключений.
}
```

В следующем фрагменте создается и применяется инструмент разметки tagger, идентифицирующий части речи в заданном предложении. Затем выводятся предложение и все присвоенные теги:

```
POSModel model = new POSModel(posModelStream);
POSTaggerME tagger = new POSTaggerME(model);

String tags[] = tagger.tag(sentence);
for(int i=0; i < tags.length; i++) {
    System.out.print(sentence[i] + "/" + tags[i] + " ");
}
System.out.println();
```

Выполнение кода дает следующий результат, приведенный, чтобы стало понятно, как работает *синтаксический анализатор* (*chunker*):

```
The/DT voyage/NN of/IN the/DT Abraham/NNP Lincoln/NNP was/VBD for/IN a/
DT long/JJ time/NN marked/VBN by/IN no/DT special/JJ incident./NN
```

Далее создается экземпляр класса ChunkerModel, использующий свой поток ввода. На его основе создается экземпляр класса ChunkerME, после чего вызывается метод chunk(), как показано ниже. Метод chunk() принимает токены предложения и соответствующие им теги для создания массива строк. Каждая строка содержит информацию о токене и его синтаксические данные:

```
ChunkerModel chunkerModel = new ChunkerModel(chunkerStream);
ChunkerME chunkerME = new ChunkerME(chunkerModel);
String result[] = chunkerME.chunk(sentence, tags);
```

Затем выводится каждый токен из массива results и связанный с ним синтаксический тег:

```
for(int i=0; i < result.length; i++) {
    System.out.println("[" + sentence[i] + "] " + result[i]);
}
```

Результат работы примера приводится ниже. Каждый токен заключен в квадратные скобки, за ним следует тег синтаксической группы. Используемые теги кратко описаны в табл. 5.5.

Таблица 5.5. Описание тегов синтаксических групп, используемых в рассматриваемом примере

Первая часть тега	
B	Начало тега
I	Продолжение тега
E	Конец тега
Вторая часть тега	
NP	Синтаксическая группа существительного
VP	Синтаксическая группа глагола

Некоторые слова объединяются в синтаксические группы, например «The voyage» и «the Abraham Lincoln».

```
[The] B-NP
[voyage] I-NP
[of] B-PP
[the] B-NP
[Abraham] I-NP
[Lincoln] I-NP
[was] B-VP
[for] B-PP
[a] B-NP
[long] I-NP
[time] I-NP
[marked] B-VP
[by] B-PP
[no] B-NP
[special] I-NP
[incident.] I-NP
```

Если потребуется более подробная информация о синтаксических группах, можно воспользоваться методом `chunkAsSpans()` класса `ChunkerME`. Метод возвращает массив объектов типа `Span`. Каждый такой объект представляет отдельный интервал, занимаемый синтаксической группой, найденной в обрабатываемом тексте.

Класс `ChunkerME` предоставляет ряд других полезных методов. Здесь будет показано применение методов `getType()`, `getStart()` и `getEnd()`. Метод `getType()` возвращает вторую часть тега синтаксической группы, методы `getStart()` и `getEnd()` возвращают, соответственно, начальную и конечную позиции интервала, занимаемого токеном в исход-

ном массиве sentence. Метод length() возвращает длину интервала, выражаемую в количестве содержащихся в нем токенов.

В следующем фрагменте кода вызывается метод chunkAsSpans(), принимающий массивы sentence и tags. Затем выводится полученный массив spans. Внешний цикл for обрабатывает за одну итерацию один объект Span и выводит его характеристики. Внутренний цикл for позволяет вывести текст в обрабатываемом интервале, заключенный в квадратные скобки:

```
Span[] spans = chunkerME.chunkAsSpans(sentence, tags);
for(Span span : spans) {
    System.out.print("Type: " + span.getType() + " - "
        + " Begin: " + span.getStart()
        + " End: " + span.getEnd()
        + " Length: " + span.length() + "  [");
    for(int j = span.getStart(); j < span.getEnd(); j++) {
        System.out.print(sentence[j] + " ");
    }
    System.out.println("]");
}
```

Полученный результат позволяет увидеть тип синтаксической группы (выделенного интервала), ее положение в массиве sentence, размер в токенах и собственно фрагмент текста, относящийся к данной группе:

```
Type: NP - Begin: 0 End: 2 Length: 2  [The voyage ]
Type: PP - Begin: 2 End: 3 Length: 1  [of ]
Type: NP - Begin: 3 End: 6 Length: 3  [the Abraham Lincoln ]
Type: VP - Begin: 6 End: 7 Length: 1  [was ]
Type: PP - Begin: 7 End: 8 Length: 1  [for ]
Type: NP - Begin: 8 End: 11 Length: 3  [a long time ]
Type: VP - Begin: 11 End: 12 Length: 1  [marked ]
Type: PP - Begin: 12 End: 13 Length: 1  [by ]
Type: NP - Begin: 13 End: 16 Length: 3  [no special incident. ]
```

Использование класса POSDictionary

Словарь тегов определяет теги, допустимые для каждого слова. Это помогает исключить случаи неправильного присваивания тегов. Кроме того, некоторые алгоритмы поиска выполняются быстрее, поскольку нет необходимости рассматривать другие теги с меньшей характеристикой вероятности.

В этом разделе рассматриваются следующие операции:

- получение словаря тегов;
- определение тегов, связанных с каждым словом;

- изменение тегов для некоторых слов;
- добавление нового словаря тегов в новую фабрику инструмента разметки.

Как и в предыдущем примере, здесь используется блок попытки захвата ресурсов для открытия потоков ввода, чтения POS-модели и последующего создания новой модели и фабрики инструмента разметки:

```
try(InputStream modelIn = new FileInputStream(
        new File(getModelDir(), "en-pos-maxent.bin"));) {
    POSModel model = new POSModel(modelIn);
    POSTaggerFactory posTaggerFactory = model.getFactory();
    ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

Получение словаря тегов

Для создания экземпляра `POSTaggerFactory` используется метод `getFactory()` класса `POSModel`. Метод `getTagDictionary()` класса `POSTaggerFactory` позволяет создать экземпляр `TagDictionary`, как показано здесь:

```
MutableTagDictionary tagDictionary =
    (MutableTagDictionary)posTaggerFactory.getTagDictionary();
```

Интерфейс `MutableTagDictionary` расширяет интерфейс `TagDictionary`. Интерфейс `TagDictionary` предоставляет метод `getTags()`, а интерфейс `MutableTagDictionary` добавляет метод `put()`, позволяющий добавлять теги в словарь. Эти интерфейсы реализует класс `POSDictionary`.

Определение тегов, связанных с конкретным словом

Для получения тегов, связанных с заданным словом, используется метод `getTags()`, возвращающий массив тегов, представленных в виде строк. Затем выполняется вывод тегов:

```
String tags[] = tagDictionary.getTags("force");
for(String tag : tags) {
    System.out.print("/" + tag);
}
System.out.println();
```

В результате выводятся следующие теги:

```
/NN/VBP/VB
```

Это означает, что слово «force» может интерпретироваться тремя разными способами.

Изменение тегов для заданного слова

Метод `put()` интерфейса `MutableTagDictionary` позволяет добавлять теги к слову. Метод принимает два аргумента: само слово и список новых тегов для него – и возвращает массив прежних тегов.

В следующем примере старые теги заменяются новыми. Затем выводится список старых тегов.

```
String oldTags[] = tagDictionary.put("force", "newTag");
for(String tag : oldTags) {
    System.out.print("/" + tag);
}
System.out.println();
```

При выполнении выводятся старые теги для слова «force»:

```
/NN/VBP/VB
```

Эти теги были заменены новым тегом, что показано ниже, в блоке вывода текущих тегов для заданного слова:

```
tags = tagDictionary.getTags("force");
for(String tag : tags) {
    System.out.print("/" + tag);
}
System.out.println();
```

Результат вполне ожидаем:

```
/newTag
```

Чтобы вернуть изначально присвоенные теги, необходимо создать массив строк со старыми и новыми тегами, затем передать этот массив во втором аргументе методу `put()`, как показано ниже:

```
String newTags[] = new String[tags.length+1];
for(int i=0; i < tags.length; i++) {
    newTags[i] = tags[i];
}
newTags[tags.length] = "newTag";
oldTags = tagDictionary("force", newTags);
```

Если еще раз вывести список текущих тегов, можно увидеть, что старые теги восстановлены и к ним добавлен новый тег:

```
/NN/VBP/VB/newTag
```

При добавлении тегов будьте внимательны и располагайте теги в правильном порядке, поскольку порядок расположения тегов влияет на очередность их присваивания.

Добавление нового словаря тегов

В экземпляр класса `POSTaggerFactory` можно добавить новый словарь тегов. Этот процесс демонстрируется на примере создания нового объекта `POSTaggerFactory` с последующим добавлением объекта `tagDictionary`, разработанного выше. Сначала создадим новую фабрику с помощью конструктора по умолчанию, как показано ниже. Затем вызовем метод `setTagDictionary()` этой новой фабрики.

```
POSTaggerFactory newFactory = new POSTaggerFactory();
newFactory.setTagDictionary(tagDictionary);
```

Чтобы убедиться, что словарь действительно добавлен, опять выведем теги для слова «force»:

```
tags = newFactory.getTagDictionary().getTags("force");
for(String tag : tags) {
    System.out.print("/" + tag);
}
System.out.println();
```

После добавления нового словаря со словом «force» будут связаны следующие теги:

```
/NN/VBP/VB/newTag
```

Создание словаря из файла

Если потребуется новый словарь, можно для этого сформировать XML-файл, содержащий все требуемые слова и соответствующие им теги, а затем создать словарь из этого файла. Библиотека OpenNLP поддерживает описанный подход с помощью метода `create()` класса `POSDictionary`.

XML-файл состоит из корневого элемента `dictionary`, включающего в себя последовательность элементов `entry`. В элементе `entry` используется атрибут `tags`, определяющий теги для словарного слова. Само слово включено в элемент `entry` как вложенный элемент `token`. Простой пример для двух слов, описанных в файле *dictionary.txt*, приведен ниже:

```
<dictionary case_sensitive="false">
    <entry tags="JJ VB">
        <token>strong</token>
    </entry>
    <entry tags="NN VBP VB">
        <token>force</token>
    </entry>
</dictionary>
```

Для создания словаря вызывается метод `create()`, использующий поток ввода, как показано ниже:

```
try(InputStream dictionaryIn =
      new FileInputStream(new File("dictionary.txt"));) {
    POSDictionary dictionary = POSDictionary.create(dictionaryIn);
    ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

В классе `POSDictionary` имеется метод `iterator()`, возвращающий объект-итератор. Его метод `next()` возвращает каждое слово из словаря в виде строки. Этими методами можно воспользоваться для вывода содержимого словаря:

```
Iterator<String> iterator = dictionary.iterator();
while(iterator.hasNext()) {
    String entry = iterator.next();
    String tags[] = dictionary.getTags(entry);
    System.out.print(entry + " ");
    for(String tag : tags) {
        System.out.print("/" + tag);
    }
    System.out.println();
}
```

Результат выполнения вполне ожидаем:

```
strong /JJ/VB
force /NN/VBP/VB
```

Использование инструментов разметки по частям речи из библиотеки Stanford

В этом разделе рассматриваются две методики разметки по частям речи, поддерживаемые библиотекой Stanford API. Первая использует класс `MaxentTagger`. По его имени можно понять, что для определения частей речи он применяет принцип максимальной энтропии. Мы также задействуем этот класс для демонстрации работы модели, предназначенной для обработки текста на смс-языке. Второй подход основан на конвейере с аннотаторами. Инструменты разметки для английского языка пользуются набором тегов Penn Treebank English POS.

Использование класса MaxentTagger

Класс `MaxentTagger` использует для разметки модели, связанные с API и размещающиеся в файлах с расширением *.tagger*. Существуют моде-

ли для английского, китайского, арабского, французского и немецкого языков. Модели для английского языка перечислены ниже. Префикс *wsj* означает, что модели основаны на базе текстов бизнес-издания Wall Street Journal. Другие части имен обозначают методики, применяемые для обучения моделей, которые здесь не рассматриваются:

- *wsj-0-18-bidirectional-distsim.tagger*;
- *wsj-0-18-bidirectional-nodistsim.tagger*;
- *wsj-0-18-caseless-left3words-distsim.tagger*;
- *wsj-0-18-left3words-distsim.tagger*;
- *wsj-0-18-left3words-nodistsim.tagger*;
- *english-bidirectional-distsim.tagger*;
- *english-caseless-left3words-distsim.tagger*;
- *english-left3words-distsim.tagger*.

Следующий пример читает последовательность предложений из файла. Затем каждое предложение обрабатывается, и демонстрируются различные способы доступа к словам и тегам.

Блок `try` с попыткой захвата ресурсов позволяет перехватывать и обрабатывать вероятные исключения при вводе-выводе. Для создания экземпляра класса `MaxentTagger` используется файл *wsj-0-18-bidirectional-distsim.tagger*.

Экземпляр списка `List`, состоящего из списков объектов `HasWord`, создается с помощью метода `tokenizeText()` класса `MaxentTagger`. Предложения читаются из файла *sentences.txt*. Интерфейс `HasWord` представляет отдельные слова и содержит методы `setWord()` и `word()`. Второй метод возвращает слово в виде строки. Каждое предложение представлено экземпляром списка `List`, содержащего объекты `HasWord`:

```
try {
    MaxentTagger tagger = new MaxentTagger(getModelDir()
        + "\\wsj-0-18-bidirectional-distsim.tagger");
    List<List<HasWord>> sentences = MaxentTagger.tokenizeText(
        new BufferedReader(new FileReader("sentence.txt")));
    ...
} catch(FileNotFoundException ex) {
    // Обработка исключения.
}
```

Файл *sentence.txt* содержит первые четыре предложения из пятой главы («Погоня вслепую») романа Жюля Верна «20 000 лье под водой» в английском переводе (Jules Verne, «Twenty Thousands Leagues Under the Sea», Chapter 5, At A Venture):

```
The voyage of the Abraham Lincoln was for a long time marked by no
special incident.
```

```
But one circumstance happened which showed the wonderful dexterity of
Ned Land, and proved what confidence we might place in him.
The 30th of June, the frigate spoke some American whalers, from whom we
learned that they knew nothing about the narwhal.
But one of them, the captain of the Monroe, knowing that Ned Land had
shipped on board the Abraham Lincoln, begged for his help in chasing a
whale they had in sight.
```

Предложения из файла *sentence.txt* обрабатываются в цикле поочередно. Метод `tagSentence()` возвращает список `List` объектов `TaggedWord`, как показано ниже. Класс `TaggedWord` реализует интерфейс `HasWord` и добавляет метод `tag()`, который возвращает тег, связанный с текущим словом. Вывод каждого предложения производится вызовом метода `toString()`:

```
List<TaggedWord> taggedSentence =
    tagger.tagSentence(sentence);
for(List<HasWord> sentence : sentences) {
    List<TaggedWord> taggedSentence =
        tagger.tagSentence(sentence);
    System.out.println(taggedSentence);
}
```

Выводится следующий результат:

```
[The/DT, voyage/NN, of/IN, the/DT, Abraham/NNP, Lincoln/NNP, was/VBD,
for/IN, a/DT, long/JJ, --- time/NN, marked/VBN, by/IN, no/DT, special/
JJ, incident/NN, ./.]
 [But/CC, one/CD, circumstance/NN, happened/VBD, which/WDT, showed/VBD,
the/DT, wonderful/JJ, dexterity/NN, of/IN, Ned/NNP, Land/NNP, ,/,, and/
CC, proved/VBD, what/WP, confidence/NN, we/PRP, might/MD, place/VB, in/
IN, him/PRP, ./.]
[The/DT, 30th/JJ, of/IN, June/NNP, ,/,, the/DT, frigate/NN, spoke/
VBD, some/DT, American/JJ, whalers/NNS, ,/,, from/IN, whom/WP, we/PRP,
learned/VBD, that/IN, they/PRP, knew/VBD, nothing/NN, about/IN, the/DT,
narwhal/NN, ./.]
[But/CC, one/CD, of/IN, them/PRP, ,/,, the/DT, captain/NN, of/IN, the/
DT, Monroe/NNP, ,/,, knowing/VBG, that/IN, Ned/NNP, Land/NNP, had/VBD,
shipped/VBN, on/IN, board/NN, the/DT, Abraham/NNP, Lincoln/NNP, ,/,,
begged/VBN, for/IN, his/PRP$, help/NN, in/IN, chasing/VBG, a/DT, whale/
NN, they/PRP, had/VBD, in/IN, sight/NN, ./.]
```

Можно также воспользоваться методом `listToString()` класса `Sentence` и преобразовать размеченное тегами предложение в простой объект типа `String`.

Второй параметр этого метода, имеющий значение `false`, используется методом `toString()` объекта `HasWord` для формирования строки результата:

```
List<TaggedWord> taggedSentence =
    tagger.tagSentence(sentence);
for(List<HasWord> sentence : sentences) {
    List<TaggedWord> taggedSentence = tagger.tagSentence(sentence);
    System.out.println(
        Sentence.listToString(taggedSentence, false));
}
```

Этот вариант производит более эстетичный, с точки зрения восприятия, результат:

```
The/DT voyage/NN of/IN the/DT Abraham/NNP Lincoln/NNP was/VBD for/IN a/
DT long/JJ time/NN marked/VBN by/IN no/DT special/JJ incident/NN ./.
But/CC one/CD circumstance/NN happened/VBD which/WDT showed/VBD the/DT
wonderful/JJ dexterity/NN of/IN Ned/NNP Land/NNP ,/, and/CC proved/VBD
what/WP confidence/NN we/PRP might/MD place/VB in/IN him/PRP ./.
The/DT 30th/JJ of/IN June/NNP ,/, the/DT frigate/NN spoke/VBD some/DT
American/JJ whalers/NNS ,/, from/IN whom/WP we/PRP learned/VBD that/IN
they/PRP knew/VBD nothing/NN about/IN the/DT narwhal/NN ./.
But/CC one/CD of/IN them/PRP ,/, the/DT captain/NN of/IN the/DT Monroe/
NNP ,/, knowing/VBG that/IN Ned/NNP Land/NNP had/VBD shipped/VBN on/IN
board/NN the/DT Abraham/NNP Lincoln/NNP ,/, begged/VBN for/IN his/PRP$
help/NN in/IN chasing/VBG a/DT whale/NN they/PRP had/VBD in/IN sight/NN
./.
```

Получить точно такой же результат позволяет следующий пример. Методы `word()` и `tag()` извлекают слова и соответствующие им теги:

```
List<TaggedWord> taggedSentence =
    tagger.tagSentence(sentence);
for(TaggedWord taggedWord : taggedSentence) {
    System.out.print(taggedWord.word() + "/" +
        taggedWord.tag() + " ");
}
System.out.println();
```

Чтобы найти все вхождения заданного тега, можно воспользоваться следующим кодом, который выводит только список существительных в единственном числе (`NN`):

```
List<TaggedWord> taggedSentence = tagger.tagSentence(sentence);
for(TaggedWord taggedWord : taggedSentence) {
    if(taggedWord.tag().startsWith("NN")) {
       System.out.print(taggedWord.word() + " ");
    }
}
System.out.println();
```

Из каждого предложения выводятся только имена существительные в единственном числе:

```
NN Tagged: voyage Abraham Lincoln time incident
NN Tagged: circumstance dexterity Ned Land confidence
NN Tagged: June frigate whalers nothing narwhal
NN Tagged: captain Monroe Ned Land board Abraham Lincoln help whale sight
```

Использование класса MaxentTagger для разметки текста на смс-языке

Для обработки текстов из Твиттера, в котором могут встречаться фрагменты на смс-языке, необходима специальная модель. Такая модель для работы с твит-текстами была разработана компанией GATE (https://gate.ac.uk/wiki/twitter-postagger.html). Приведем пример использования этой модели для обработки смс-языка (textese):

```
MaxentTagger tagger = new MaxentTagger(getModelDir()
    + "\\gate-EN-twitter.model");
```

Здесь применяется метод `tagString()` класса `MaxentTagger`, описанный в разделе «Трудности в идентификации частей речи» в начале текущей главы:

```
System.out.println(tagger.tagString("AFAIK she H8 cth!"));
System.out.println(tagger.tagString(
    "BTW had a GR8 tym at the party BBIAM."));
```

После обработки будет выведен следующий результат:

```
AFAIK_NNP she_PRP H8_VBP cth!_NN
BTW_UH had_VBD a_DT GR8_NNP tym_NNP at_IN the_DT party_NN BBIAM._NNP
```

Использование конвейера, поддерживаемого библиотекой Stanford, для POS-разметки

В некоторых примерах мы уже использовали конвейер, поддерживаемый библиотекой Stanford. В этом разделе тот же конвейер применяется для извлечения тегов, обозначающих части речи. Как и ранее, конвейер создается на основе набора аннотаторов: `tokenize`, `ssplit` и `pos`.

Аннотаторы обозначают соответствующие операции: токенизация, разделение текста на предложения, затем поиск тегов, обозначающих части речи:

```
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, pos");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
```

В качестве потока ввода для объекта `Annotator` используется переменная `theSentence`. Затем вызывается метод конвейера `annotate()`:

```
Annotation document = new Annotation(theSentence);
pipeline.annotate(document);
```

Поскольку конвейер может выполнять различные типы обработки, для доступа к словам и тегам формируется список объектов `CoreMap`. Метод `get()` класса `Annotation` возвращает список предложений, как показано ниже.

```
List<CoreMap> sentences =
    document.get(SentenceAnnotation.class);
```

Для доступа к содержимому объектов `CoreMap` можно воспользоваться методом `get()` этого класса, принимающим класс с требуемой информацией. Как показано в следующем примере, доступ к токенам осуществляется с помощью класса `TextAnnotation`, а теги частей речи извлекаются посредством класса `PartOfSpeechAnnotation`. Затем выводится каждое слово каждого предложения и соответствующий ему тег:

```
for(CoreMap sentence : sentences) {
    for(CoreLabel token : sentence.get(TokensAnnotation.class)) {
        String word = token.get(TextAnnotation.class);
        String pos = token.get(PartOfSpeechAnnotation.class);
        System.out.print(word + "/" + pos + " ");
    }
    System.out.println();
}
```

Выполнение этого кода дает следующий результат:

```
The/DT voyage/NN of/IN the/DT Abraham/NNP Lincoln/NNP was/VBD for/IN a/
DT long/JJ time/NN marked/VBN by/IN no/DT special/JJ incident/NN ./.
```

Конвейер может использовать дополнительные параметры, управляющие разметкой. Например, по умолчанию принимается модель разметки *english-left3words-distsim.tagger*. Мы можем выбрать другую модель с помощью свойства `pos.model`, как показано ниже. Свойство `pos.maxlen` позволяет определить максимальный размер предложения:

```
props.put("pos.model",
"C:/.../Models/english-caseless-left3words-distsim.tagger");
```

Иногда удобнее представить размеченный тегами документ в XML-формате. Метод `xmlPrint()` класса `StanfordCoreNLP` позволяет получить такой документ. В первом аргументе метод принимает аннотатор для отображения. Второй – объект потока вывода `OutputStream`, куда будет записываться результат. В следующем фрагменте результаты разметки, полученные ранее, записываются в стандартный поток вывода.

Код заключен в блок try-catch, чтобы получить возможность обработки исключений при вводе-выводе:

```
try {
    pipeline.xmlPrint(document, System.out);
} catch(IOException ex) {
    // Обработка исключения.
}
```

Ниже показана часть вывода. В целях экономии места приведены только два первых и последнее слово. Каждый xml-тег token содержит слово, его позицию в предложении и тег, определяющий часть речи:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="CoreNLP-to-HTML.xsl" type="text/xsl"?>
<root>
<document>
<sentences>
<sentence id="1">
<tokens>
<token id="1">
<word>The</word>
<CharacterOffsetBegin>0</CharacterOffsetBegin>
<CharacterOffsetEnd>3</CharacterOffsetEnd>
<POS>DT</POS>
</token>
<token id="2">
<word>voyage</word>
<CharacterOffsetBegin>4</CharacterOffsetBegin>
<CharacterOffsetEnd>10</CharacterOffsetEnd>
<POS>NN</POS>
</token>
...
<token id="17">
<word>.</word>
<CharacterOffsetBegin>83</CharacterOffsetBegin>
<CharacterOffsetEnd>84</CharacterOffsetEnd>
<POS>.</POS>
</token>
</tokens>
</sentence>
</sentences>
</document>
</root>
```

Метод prettyPrint() работает похожим образом:

```
pipeline.prettyPrint(document, System.out);
```

В действительности результаты этого метода выглядят не так хорошо, как показано ниже. Выводится исходное предложение, за ним по-

очередно следуют все слова, их позиции и теги. Для удобства чтения результат был дополнительно переформатирован:

```
The voyage of the Abraham Lincoln was for a long time marked by no
special incident.
[Text=The CharacterOffsetBegin=0 CharacterOffsetEnd=3 PartOfSpeech=DT]
[Text=voyage CharacterOffsetBegin=4 CharacterOffsetEnd=10
PartOfSpeech=NN]
[Text=of CharacterOffsetBegin=11 CharacterOffsetEnd=13 PartOfSpeech=IN]
[Text=the CharacterOffsetBegin=14 CharacterOffsetEnd=17 PartOfSpeech=DT]
[Text=Abraham CharacterOffsetBegin=18 CharacterOffsetEnd=25
PartOfSpeech=NNP]
 [Text=Lincoln CharacterOffsetBegin=26 CharacterOffsetEnd=33
PartOfSpeech=NNP]
 [Text=was CharacterOffsetBegin=34 CharacterOffsetEnd=37
PartOfSpeech=VBD]
 [Text=for CharacterOffsetBegin=38 CharacterOffsetEnd=41
PartOfSpeech=IN]
 [Text=a CharacterOffsetBegin=42 CharacterOffsetEnd=43 PartOfSpeech=DT]
 [Text=long CharacterOffsetBegin=44 CharacterOffsetEnd=48
PartOfSpeech=JJ]
 [Text=time CharacterOffsetBegin=49 CharacterOffsetEnd=53
PartOfSpeech=NN]
 [Text=marked CharacterOffsetBegin=54 CharacterOffsetEnd=60
PartOfSpeech=VBN]
 [Text=by CharacterOffsetBegin=61 CharacterOffsetEnd=63 PartOfSpeech=IN]
 [Text=no CharacterOffsetBegin=64 CharacterOffsetEnd=66 PartOfSpeech=DT]
 [Text=special CharacterOffsetBegin=67 CharacterOffsetEnd=74
PartOfSpeech=JJ]
 [Text=incident CharacterOffsetBegin=75 CharacterOffsetEnd=83
PartOfSpeech=NN]
 [Text=. CharacterOffsetBegin=83 CharacterOffsetEnd=84 PartOfSpeech=.]
```

Использование инструментов разметки по частям речи из библиотеки LingPipe

Библиотека LingPipe для поддержки процесса разметки по частям речи использует интерфейс `Tagger`, в котором определен единственный метод `tag()`, возвращающий экземпляр списка `List` объектов `Tagging`. Объекты содержат слова и соответствующие им теги. Этот интерфейс реализуют классы `ChainCrf` и `HmmDecoder`.

Класс `ChainCrf` использует декодирование и оценку определяемых тегов на основе метода статистического моделирования с помощью

линейно-последовательных марковских условно случайных полей (*conditional random fields, CRF*). Класс `HmmDecoder` применяет *скрытую марковскую модель* (*Hidden Markov Model, HMM*) для разметки. Ниже мы рассмотрим практическое применение обоих классов.

Класс `HmmDecoder` использует метод `tag()` для определения наиболее подходящих тегов (в первую очередь рассматриваются наилучшие варианты). Метод `tagNBest()` оценивает возможный вариант разметки и возвращает итератор, дающий возможность последовательного обхода оценок тегов. Библиотека LingPipe предлагает три модели разметки по частям речи, их можно скачать с сайта http://alias-i.com/lingpipe/web/models.html. Краткие характеристики моделей приведены в табл. 5.6. Ниже будет показано использование модели Brown Corpus.

Таблица 5.6. Модели разметки по частям речи из библиотеки LingPipe

Модель	Имя файла
Текст общей тематики на английском языке: Brown Corpus	`pos-en-general-brown.HiddenMarkovModel`
Текст биомедицинской тематики на английском языке: MedPost Corpus	`pos-en-bio-medpost. HiddenMarkovModel`
Текст биомедицинской тематики на английском языке: GENIA Corpus	`pos-en-bio-genia. HiddenMarkovModel`

Использование класса HmmDecoder с тегами Best_First

В данном примере необходим блок `try` попытки захвата ресурсов для обработки возможных исключений. Начнем с создания экземпляра класса `HmmDecoder`, как показано ниже.

Модель извлекается из файла и затем передается конструктору класса `HmmDecoder`:

```
try(FileInputStream inputStream =
        new FileInputStream(getModelDir()
        + "\\pos-en-general-brown.HiddenMarkovModel");
    ObjectInputStream objectStream =
        new ObjectInputStream(inputStream);) {
    HiddenMarkovModel hmm =
        (HiddenMarkovModel)objectStream.readObject();
    HmmDecoder decoder = new HmmDecoder(hmm);
    ...
} catch(IOException ex) {
    // Обработка исключения.
} catch(ClassNotFoundException ex) {
    // Обработка исключения.
};
```

Далее выполняется разметка текста в переменной theSentence, но сначала требуется токенизация этого текста. Для этой цели используется токенизатор Indo-European. Для работы метода tokenizer() необходимо преобразовать текстовую строку в массив символов. Затем метод tokenize() возвращает массив токенов, представленных в виде строк:

```
TokenizerFactory TOKENIZER_FACTORY =
    IndoEuropeanTokenizerFactory.INSTANCE;
char[] charArray = theSentence.toCharArray();
Tokenizer tokenizer =
    TOKENIZER_FACTORY.tokenizer(charArray, 0, charArray.length);
String[] tokens = tokenizer.tokenize();
```

Собственно разметка выполняется методом tag() класса HmmDecoder. Но этот метод требует экземпляра списка List, сформированного из токенов типа String. Список создается с помощью метода asList() класса Arrays. Экземпляр класса Tagging содержит последовательность токенов и тегов:

```
List<String> tokenList = Arrays.asList(tokens);
Tagging<String> tagString = decoder.tag(tokenList);
```

После этого можно вывести токены и связанные с ними теги. В следующем цикле вызываются методы token() и tag() для доступа к токенам и тегам соответственно, хранящимся в объекте Tagging:

```
for(int i=0; i < tagString.size(); ++i) {
    System.out.print(tagString.token(i) + "/"
    + tagString.tag(i) + " ");
}
```

Выполнив код примера, получим следующий результат:

```
The/at voyage/nn of/in the/at Abraham/np Lincoln/np was/bedz for/in a/at
lon/jj time/nn marked/vbn by/in no/at special/jj incident/nn ./.
```

Использование класса HmmDecoder с тегами NBest

В процессе разметки рассматривается несколько комбинаций тегов. Метод tagNBest() класса HmmDecoder возвращает итератор для последовательности объектов ScoredTagging, отражающих степень достоверности разных вариантов. Метод принимает список токенов и числовое значение, определяющее максимальное количество требуемых результатов.

Предложение, обработанное в примере из предыдущего раздела, недостаточно сложное для определения наиболее достоверной ком-

бинации тегов из нескольких возможных. Вместо него мы возьмем предложение из нескольких слов с неоднозначно определяемой принадлежностью к той или иной части речи:

```
String[] sentence = {"Bill", "used", "the", "force",
    "to", "force", "the", "manager", "to", "tear", "the",
    "bill", "in", "two."};
List<String> tokenList = Arrays.asList(sentence);
```

Следующий пример, демонстрирующий применение метода `tagNBest()`, начинается с объявления максимального количества требуемых результатов:

```
int maxResults = 5;
```

Воспользуемся объектом `decoder` из предыдущего раздела и применим к нему метод `tagNBest()`, как показано ниже:

```
Iterator<ScoredTagging<String>> iterator =
    decoder.tagNBest(tokenList, maxResults);
```

Итератор позволяет получить доступ к каждой из пяти различных оценок. Класс `ScoredTagging` предлагает метод `score()`, возвращающий числовое значение оценки степени достоверности текущего варианта тегов. В следующем фрагменте метод `printf()` выводит значения оценок. Далее в цикле выводятся токены и их теги.

Окончательный результат представлен строкой, состоящей из оценки и последовательности слов (токенов) с присвоенными тегами:

```
while(iterator.hasNext()) {
    ScoredTagging<String> scoredTagging = iterator.next();
    System.out.printf("Score: %7.3f  Sequence: ",
       scoredTagging.score());
    for(int i=0; i < tokenList.size(); ++i) {
       System.out.print(scoredTagging.token(i) + "/" +
           scoredTagging.tag(i) + " ");
    }
    System.out.println();
}
```

Результат работы примера приведен ниже. Обратите внимание, что слову `"force"` присваиваются три варианта тегов – `nn`, `jj` или `vb`:

```
Score: -148.796  Sequence: Bill/np used/vbd the/at force/nn to/to force/
vb the/at manager/nn to/to tear/vb the/at bill/nn in/in two./nn
Score: -154.434  Sequence: Bill/np used/vbn the/at force/nn to/to force/
vb the/at manager/nn to/to tear/vb the/at bill/nn in/in two./nn
Score: -154.781  Sequence: Bill/np used/vbd the/at force/nn to/in force/
nn the/at manager/nn to/to tear/vb the/at bill/nn in/in two./nn
```

```
Score: -157.126  Sequence: Bill/np used/vbd the/at force/nn to/to force/
vb the/at manager/jj to/to tear/vb the/at bill/nn in/in two./nn
Score: -157.340  Sequence: Bill/np used/vbd the/at force/jj to/to force/
vb the/at manager/nn to/to tear/vb the/at bill/nn in/in two./nn
```

Определение степени достоверности назначенного тега с помощью класса HmmDecoder

Статистический анализ можно выполнить с использованием алгебраической структуры, именуемой *решеткой* (*lattice*), удобной для анализа разных вариантов распределения тегов по словам. Решетка содержит верхние и нижние границы оценок. Метод `tagMarginal()` класса `HmmDecoder` возвращает экземпляр класса `TagLattice`, представляющий решетку.

Мы можем исследовать каждый токен решетки с помощью экземпляра класса `ConditionalClassification`. В следующем примере метод `tagMarginal()` возвращает экземпляр класса `TagLattice`. Цикл позволяет поочередно получить экземпляр класса `ConditionalClassification` для каждого токена в решетке.

Воспользуемся списком токенов `tokenList`, созданным в предыдущем разделе:

```
TagLattice<String> lattice = decoder.tagMarginal(tokenList);
for(int index = 0; index < tokenList.size(); index++) {
    ConditionalClassification classification =
        lattice.tokenClassification(index);
    ...
}
```

Класс `ConditionalClassification` содержит методы `score()` и `category()`. Метод `score()` возвращает сравнительную оценку для заданной категории. Метод `category()` возвращает саму категорию, то есть в нашем случае тег. После этого выводятся исследуемое слово, его оценка и категория (тег):

```
System.out.printf("%-8s", tokenList.get(index));
for(int i=0; i < 4; ++i) {
    double score = classification.score(i);
    String tag = classification.category(i);
    System.out.printf("%7.3f/%-3s ", score, tag);
}
System.out.println();
```

Результат выполнения примера показан ниже:

```
Bill     0.974/np   0.018/nn   0.006/rb   0.001/nps
used     0.935/vbd  0.065/vbn  0.000/jj   0.000/rb
```

```
the       1.000/at   0.000/jj   0.000/pps   0.000/pp$$
force     0.997/nn   0.016/jj   0.006/vb    0.001/rb
to        0.944/to   0.055/in   0.000/rb    0.000/nn
force     0.945/vb   0.053/nn   0.002/rb    0.001/jj
the       1.000/at   0.000/jj   0.000/vb    0.000/nn
manager   0.982/nn   0.018/jj   0.000/nn$   0.000/vb
to        0.988/to   0.012/in   0.000/rb    0.000/nn
tear      0.991/vb   0.007/nn   0.001/rb    0.001/jj
the       1.000/at   0.000/jj   0.000/vb    0.000/nn
bill      0.994/nn   0.003/jj   0.002/rb    0.001/nns
in        0.990/in   0.004/rp   0.002/nn    0.001/jj
two.      0.960/nn   0.013/np   0.011/nns   0.008/rb
```

Обучение модели POSModel из библиотеки OpenNLP

Обучение модели `POSModel` из библиотеки OpenNLP во многом похоже на ранее рассмотренные примеры обучения моделей. Необходим файл, содержащий достаточно большой текст, чтобы предоставить необходимый обучающий материал. Каждое предложение обучающего текста должно располагаться на отдельной строке и состоять из токенов с присоединенным к ним символом подчеркивания и тегом, соответствующим части речи для данного токена.

Обучающий текст сформирован из первых пяти предложений все той же пятой главы романа «20 000 лье под водой». Для реального обучения моделей он недостаточно велик, но для демонстрации основных принципов процесса обучения достаточно и такого объема текста.

Обучающий текст содержится в файле с именем *sample.train*:

```
The_DT voyage_NN of_IN the_DT Abraham_NNP Lincoln_NNP was_VBD for_IN a_
DT long_JJ time_NN marked_VBN by_IN no_DT special_JJ incident._NN

But_CC one_CD circumstance_NN happened_VBD which_WDT showed_VBD the_DT
wonderful_JJ dexterity_NN of_IN Ned_NNP Land,_NNP and_CC proved_VBD
what_WP confidence_NN we_PRP might_MD place_VB in_IN him._PRP$

The_DT 30th_JJ of_IN June,_NNP the_DT frigate_NN spoke_VBD some_DT
American_NNP whalers,_NNS from_IN whom_WP we_PRP learned_VBD that_IN
they_PRP knew_VBD nothing_NN about_IN the_DT narwhal._NN

But_CC one_CD of_IN them,_PRP$ the_DT captain_NN of_IN the_DT Monroe,_
NNP knowing_VBG that_IN Ned_NNP Land_NNP had_VBD shipped_VBN on_IN
board_NN the_DT Abraham_NNP Lincoln,_NNP begged_VBD for_IN his_PRP$
help_NN in_IN chasing_VBG a_DT whale_NN they_PRP had_VBD in_IN sight._NN
```

Ниже показаны создание модели с помощью метода `train()` класса `POSModel` и сохранение обученной модели в файле. Прежде всего необходимо объявить переменную-экземпляр класса `POSModel`:

```
POSModel model = null;
```

Открытие файла с обучающим текстом выполняется в блоке попытки захвата ресурса try:

```
try(InputStream dataIn = new FileInputStream("sample.train");) {
    ...
} catch(IOExcepotion ex) {
    // Обработка исключения.

}
```

Создается экземпляр класса PlainTextByLineStream для передачи в конструктор класса WordTagSampleStream, чтобы получить экземпляр класса ObjectStream<POSSample>. Это позволяет перевести входные данные в формат, требуемый для метода train():

```
ObjectStream<String> lineStream =
    new PlainTextByLineStream(dataIn, "UTF-8");
ObjectStream<POSSample> sampleStream =
    new WordTagSampleStream(lineStream);
```

Параметры метода train() определяют язык обучающего текста, поток ввода данных, дополнительный набор параметров для обучения и словари (в данном примере не используются), как показано ниже:

```
model = POSTaggerME.train("en", sampleStream,
    TrainingParameters.defaultParams(), null, null);
```

В процессе обучения выводится очень большой объем выходных данных. Для экономии места ниже приводится только наиболее важная часть результата:

```
Indexing events using cutoff of 5
  Computing event counts... done. 90 events
  Indexing... done.
Sorting and merging events... done. Reduced 90 events to 82.
Done indexing.
Incorporating indexed data for training... done.
  Number of Event Tokens: 82
      Number of Outcomes: 17
    Number of Predicates: 45
...done.
Computing model parameters...
Performing 100 iterations.
  1: ... loglikelihood=-254.98920096505964  0.14444444444444443
  2: ... loglikelihood=-201.19283975630537  0.6
  3: ... loglikelihood=-174.8849213436524  0.6111111111111112
  4: ... loglikelihood=-157.58164262220754  0.6333333333333333
  5: ... loglikelihood=-144.69272379986646  0.6555555555555556
...
```

```
 99: ... loglikelihood=-33.461128002846024  0.9333333333333333
100: ... loglikelihood=-33.29073273669207  0.9333333333333333
```

Сохранение обученной модели в файл выполняется в следующем фрагменте. Создается поток вывода, и метод `serialize()` класса `POSModel` записывает образ модели в файл *en_pos_verne.bin*:

```
try(OutputStream modelOut = new BufferedOutputStream(
      new FileOutputStream(new File("en-pos-verne.bin")));) {
    model.serialize(modelOut);
} catch(IOException ex) {
    // Обработка исключения.
}
```

Резюме

Разметка по частям речи – эффективный способ идентификации грамматических элементов предложения. Она обеспечивает удобное представление данных для последующих задач, таких как анализ вопросов/запросов и анализ эмоциональной окраски текста. К этой теме мы вернемся при рассмотрении синтаксического анализа в главе 7 «Использование синтаксического анализатора (парсера) для выделения взаимосвязей».

Разметка по частям речи достаточно сложна из-за неоднозначности, свойственной большинству языков. Частое употребление смс-языка еще более усложняет этот процесс. К счастью, уже созданы модели, успешно обрабатывающие и такой тип текста. Но подобные модели периодически требуют дополнительного обучения и усовершенствования, поскольку постоянно возникают новые термины и сленговые выражения.

Мы подробно рассмотрели поддержку разметки по частям речи, предоставляемую библиотеками OpenNLP, Stanford API и LingPipe. Эти библиотеки применяют принципиально различные подходы, основанные как на наборах правил, так и на использовании специализированных моделей. На практическом примере мы наблюдали, как словари могут улучшить процесс разметки.

Некоторое внимание было уделено процессу обучения модели. Предварительно размеченные образцы текста использовались как входные данные для этого процесса, а в результате мы получали заново обученную модель. Мы не рассматривали процедуру проверки новой модели, но методика проверки ничем не отличается от описанной в предыдущих главах.

Различные методики разметки по частям речи можно сравнивать по различным характеристикам, например по точности определения частей речи, по скорости их выполнения и т. д. Здесь мы не касались этого вопроса, но в Интернете существует множество сайтов, посвященных этой теме, например сравнение скорости работы различных методик можно найти на сайте http://mattwilkens.com/2008/11/08/evaluating-pos-taggers-speed/.

В следующей главе будут подробно рассматриваться методики классификации документов на основе их содержимого.

Глава 6

Классификация текстов и документов

В этой главе демонстрируется практическое применение различных библиотек NLP для задачи классификации текстов. Этот процесс не следует путать с кластеризацией текста. *Кластеризация текста* (*text clustering*) выполняет идентификацию текста без использования каких-либо предварительно используемых категорий. В отличие от кластеризации, *классификация текста* (*text classification*) работает с предварительно определенными категориями. Мы сосредоточимся на классификации текста, когда тексту присваиваются теги, определяющие его тип.

При обобщенном подходе к классификации текста самым первым шагом является обучение модели. Затем модель проверяется, и после этого ее можно применять для классификации документов. В данной главе главное внимание будет уделено этапам обучения и практического использования моделей.

Документы могут классифицироваться по любому количеству атрибутов, таких как тема документа, тип документа, дата публикации, автор, язык документа и удобочитаемость с точки зрения восприятия информации (уровень сложности текста). В некоторых методиках классификации требуется участие человека для разметки выборочных данных.

Анализ эмоциональной окраски текста (*sentiment analysis*) представляет собой один из типов классификации. Его главная цель – определить, что именно текст пытается донести до читателя, обычно в форме положительного или отрицательного отношения к обсуждаемой теме. Мы рассмотрим некоторые методики выполнения этого типа анализа.

Как используется классификация текста

Классификация текста может преследовать следующие цели:

- обнаружение спама;
- установление авторства;
- анализ эмоциональной окраски текста;
- определение возраста и пола автора;
- определение главной темы документа;
- идентификация языка.

К сожалению, рассылка спама в наше время стала одним из самых больших неудобств, причиняемых пользователям электронной почты. Если электронное сообщение имеет признаки спама, оно обычно перемещается в специальный каталог. Текстовое сообщение можно проанализировать и по определенным признакам классифицировать как спам. К таким признакам можно причислить неправильное написание слов (орфографические ошибки), отсутствие достоверного адреса электронной почты для ответа и нестандартный URL.

Классификация также используется для установления авторства документа. Это особенно важно для исторических документов, таких как «The Federalist Papers»[1] и «Primary Colors»[2], авторы которых были в конечном итоге определены.

Анализ эмоциональной окраски текста – это методика, позволяющая определять эмоциональное отношение автора текста к обсуждаемой теме. Наиболее часто применяется в обзорах и рецензиях кинопродукции, но может использоваться при составлении обзора практически любых товаров и продуктов. Такой подход помогает компаниям лучше оценить, как их продукция воспринимается покупателями и клиентами. Часто тексту присваивается характеристика «отрицательный» или «положительный». Анализ эмоциональной окраски текста также называют *определением мнения, отношения* (*opinion extraction/mining*) и *анализом субъективности текста* (*subjectivity analysis*). Степень доверия потребителей и ситуацию на фондовой

---

[1] «Записки федералиста» – сборник из 85 статей в поддержку ратификации Конституции США (1787–1788 гг.). Статьи были написаны под общим псевдонимом Публий в честь знаменитого римского консула. Под псевдонимом скрывались Александр Гамильтон, Джеймс Мэдисон и Джон Джей. – *Прим. перев.*

[2] «Primary Colors» – так называемый «роман с ключом», политическая сатира. Автор романа был указан как Anonimous, впоследствии выяснилось, что написал его Джо Клейн (Joe Klein). – *Прим. перев.*

бирже можно предсказывать по сообщениям в Твиттере и по другим источникам.

Классификация может использоваться для определения возраста и пола автора текста, а кроме того, способна предоставить некоторую дополнительную информацию об авторе. Достаточно часто количественный анализ местоимений, *определяющих слов* и *именных групп* позволяет определить пол пишущего. Женщины склонны употреблять больше местоимений, мужчины же чаще пользуются определяющими словами.

Определение главной темы текста весьма полезно в случаях, когда необходимо упорядочить большое количество документов. Механизмы поиска интенсивно используют эту методику, но в основном для простого распределения документов по различным категориям, что нагляднее всего проявляется при формировании облаков тегов. *Облако тегов* (*tag cloud*) – это группа слов, отображающая сравнительную частоту появления каждого слова в тексте.

На рис. 6.1 изображено облако тегов, сформированное программным генератором IBM Word Cloud Generator (http://www.softpedia.com/get/Office-tools/Other-Office-Tools/IBM-Word-Cloud-Generator.shtml). Это изображение также можно найти в Википедии: http://upload.wikimedia.org/wikipedia/commons/9/9e/Foundation-l_word_cloud_without_headers_and _quotes.png.

Методики классификации поддерживают определение языка, на котором написан документ. Этот тип анализа необходим для реше-

Рис. 6.1. Пример облака тегов, сформированного программным генератором IBM Word Cloud Generator

ния многих обобщенных задач обработки естественного языка, когда требуется выбрать и применить модель для конкретного языка.

Особенности анализа эмоциональной окраски текста

При анализе эмоциональной окраски текста все внимание сосредоточено на определении эмоционального отношения автора (или авторов) к обсуждаемому продукту, товару, услуге или любой другой теме. Мы можем узнать о положительном или отрицательном отношении жителей некоторого города к спортивной команде, представляющей этот город. Здесь можно разделить отношение к выступлениям самой команды и к ее руководству.

Немалую пользу может принести автоматическое определение отношения к характеристикам или свойствам некоторого продукта (товара) и оформление результатов этого процесса в удобочитаемой, легко воспринимаемой форме. Наглядный пример можно найти в обзоре 2014 года модели автомобиля Toyota Camry, подготовленном Kelly Blue Book (http://www.kbb.com/toyota/camry/2014-toyota-camry/?r=471659652516861060) и показанном на рис. 6.2.

Рис. 6.2. Обзорная статья, подготовленная Kelly Blue Book, о модели Toyota Camry

Все характеристики, такие как «Общая оценка» (Overall Rating), «Цена» (Value) и прочие, представлены в виде гистограмм и числовых значений. Вычисление этих значений можно выполнить автоматически с помощью анализа эмоциональной окраски текста.

Анализ эмоциональной окраски текста применим к предложению, к группе предложений или ко всему документу. При анализе эмоциональной окраски текста может просто определяться положительное или отрицательное отношение автора к обсуждаемой теме или же конкретная оценка отношения в числовом выражении по шкале значений от 1 до 10. Возможны и более сложные типы выражения отношения.

Усложняет процесс анализа как в случае одного предложения, так и в случае документа в целом возможность выражения разнообразных эмоций по отношению к различным темам обсуждения.

Как определить слова, соответствующие тому или иному типу эмоций? Ответ на этот вопрос помогают получить *лексические определители эмоций* (*sentiment lexicons*). В данном контексте лексические определители представляют собой словари, описывающие эмоциональную окраску различных слов. Одним из таких лексических определителей является General Inquirer (http://www.wjh.harvard.edu/~inquirer/), содержащий 1915 слов, эмоциональный смысл которых считается положительным. Кроме того, в нем имеется список слов, определяющих другие признаки: огорчение, удовольствие, твердую убежденность, поощрение и т. п. Существуют и другие лексические определители, например MPQA Subjectivity Cues Lexicon (http://mpqa.cs.pitt.edu/).

Иногда может потребоваться создание собственного лексического определителя. Для этого обычно применяется *методика полуавтоматического обучения* (*semisupervised learning*)[1], при котором используются несколько специально размеченных примеров или правил для так называемой «самораскрутки» (bootstrap) процесса построения лексического определителя. Такой подход применяется в случаях, когда предметная область лексического определителя не вполне соответствует предметной области решаемой задачи.

Наша задача состоит не только в простом определении положительного или отрицательного отношения автора к теме, гораздо

[1] Полуавтоматическое обучение – один из способов машинного обучения, занимающий промежуточное положение между *обучением с учителем* (*supervised learning*) и *обучением без учителя* (*unsupervised learning*). – *Прим. перев.*

чаще необходимо выявить его атрибуты, иногда называемые целью (target). Рассмотрим следующий пример:

«The ride was very rough but the attendants did an excellent job of making us comfortable».

(«Поездка была очень трудной, но обслуживающий персонал сделал все, чтобы обеспечить для нас все удобства».)

Данное предложение содержит два типа отношения автора: трудности и удобства. Первое выражает отрицательные чувства, второе – положительные. Цель, или атрибут, положительного отношения – выполненная работа, приложенные усилия (job), цель отрицательного отношения – поездка (ride).

Методики классификации текста

При классификации рассматривается отдельный документ и определяется его принадлежность к одной из нескольких групп документов. Существуют две основные методики классификации текста:

- на основе набора правил;
- машинное обучение с учителем.

Классификация на основе набора правил использует списки словосочетаний и прочих атрибутов, систематизированные в соответствии с правилами, тщательно проработанными экспертом. Такой подход может быть весьма эффективным, но формирование правил связано с большими трудозатратами.

Машинное обучение с учителем (Supervised Machine Learning, SML) предполагает обработку набора специально размеченных обучающих документов для создания модели, которую обычно называют *классификатором* (*classifier*). Среди множества методов машинного обучения можно выделить *наивный байесовский классификатор* (*Naive Bayes*), *метод опорных векторов* (*Support Vector Machine, SVM*) и *метод k ближайших соседей* (*k-nearest neighbor*).

В данной книге не рассматриваются внутренняя организация и функционирование этих методов, но интересующиеся читатели смогут без труда найти многочисленные источники, подробно описывающие эти и другие методы машинного обучения.

Использование библиотек NLP API для классификации текста

Для демонстрации практического применения различных методик классификации текстов мы воспользуемся библиотеками OpenNLP, Stanford API и LingPipe. При этом большее внимание будет уделено библиотеке LingPipe и предоставляемым ею методикам классификации.

Использование библиотеки OpenNLP

Интерфейс `DocumentCategorizer` определяет методы поддержки процесса классификации. Этот интерфейс реализует класс `DocumentCategorizerME`, который обеспечивает классификацию текста по предварительно определенным категориям, используя для этого программно реализованный принцип максимальной энтропии. В этом разделе мы рассмотрим:

- процесс обучения модели;
- практическое применение обученной модели.

Обучение классификационной модели из библиотеки OpenNLP

Прежде всего модель необходимо обучить, поскольку библиотека OpenNLP не предоставляет готовых к употреблению классификационных моделей. Процесс обучения состоит из создания файла с обучающими данными и последующей передачи этих данных в модель `DocumentCategorizerME` для выполнения собственно процедуры обучения. Созданная и обученная модель обычно сохраняется в файле для дальнейшего повторного использования.

Формат обучающего файла определен как последовательность строк, каждая из которых представляет документ. Первое слово в строке является названием категории. За категорией следует текст, разделяемый пробелами. Пример текста категории dog (собака):

```
dog The most interesting feature of a dog is its ...
(собака Наиболее характерным отличительным признаком собаки является ее ...)
```

Для демонстрации процесса обучения создан файл *en-animals.train*, в который включены две категории: собака (dog) и кошка (cat). В качестве обучающего текста использованы разделы Википедии. Для

категории dog (http://en.wikipedia.org/wiki/Dog) взят раздел As Pets (Домашние собаки). Для категории cat (http://en.wikipedia.org/wiki/Cats_and_humans) взяты раздел Pets (Домашние кошки) и первый абзац раздела Domesticated varieties (Разновидности домашних кошек). Из текста были удалены цифровые ссылки.

Для экономии места ниже приведены только начальные фрагменты каждой строки:

```
dog The most widespread form of interspecies bonding occurs ...
dog There have been two major trends in the changing status of ...
dog There are a vast range of commodity forms available to ...
dog An Australian Cattle Dog in reindeer antlers sits on Santa's lap ...
dog A pet dog taking part in Christmas traditions ...
dog The majority of contemporary people with dogs describe their ...
dog Another study of dogs' roles in families showed many dogs have ...
dog According to statistics published by the American Pet Products ...
dog The latest study using Magnetic resonance imaging (MRI) ...
cat Cats are common pets in Europe and North America, and their ...
cat Although cat ownership has commonly been associated ...
cat The concept of a cat breed appeared in Britain during ...
cat Cats come in a variety of colors and patterns. These are physical ...
cat A natural behavior in cats is to hook their front claws periodically ...
cat Although scratching can serve cats to keep their claws from growing ...
```

При создании обучающих данных важно использовать образец текста достаточно большого размера. Приведенный здесь фрагмент текста слишком мал для некоторых видов анализа. Но, как мы увидим чуть ниже, этого объема вполне достаточно для правильного определения категорий.

Класс `DoccatModel` поддерживает категоризацию и классификацию текста. Модель обучается с использованием метода `train()` и подготовленного размеченного текста. Метод `train()` принимает строку, определяющую язык текста, и экземпляр `ObjectStream<DocumentSample>` с обучающими данными. В экземпляре `DocumentSample` хранятся размеченный текст и соответствующие категории.

В следующем примере для обучения конкретной модели используется файл *en-animal.train*. На основе потока ввода создается экземпляр `PlainTextByLineStream`, который затем преобразуется в экземпляр `ObjectStream<DocumentSample>`. После этого применяется метод `train()`. Для обработки возможных исключений код заключен в блок попытки захвата ресурса `try`. Кроме того, создается поток вывода для сохранения обученной модели:

```
DoccatModel model = null;
try(InputStream dataIn = new FileInputStream("en-animal.train");
       OutputStream dataOut =
```

```
            new FileOutputStream("en-animal.model");) {
    ObjectStream<String> lineStream =
        new PlainTextByLineStream(dataIn, "UTF-8");
    ObjectStream<DocumentSample> sampleStream =
        new DocumentSampleStream(lineStream);
    model = DocumentCategorizerME.train("en", sampleStream);
    ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

Полученный результат, приведенный ниже, сокращен в целях экономии места:

```
Indexing events using cutoff of 5
  Computing event counts... done. 12 events
  Indexing... done.
Sorting and merging events... done. Reduced 12 events to 12.
Done indexing.
Incorporating indexed data for training... done.
 Number of Event Tokens: 12
     Number of Outcomes: 2
   Number of Predicates: 30
...done.
Computing model parameters ...
Performing 100 iterations.
  1: ... loglikelihood=-8.317766166719343  0.75
  2: ... loglikelihood=-7.1439957443937265  0.75
  3: ... loglikelihood=-6.560690872956419  0.75
  4: ... loglikelihood=-6.106743124066829  0.75
  5: ... loglikelihood=-5.721805583104927  0.8333333333333334
  6: ... loglikelihood=-5.3891508904777785  0.8333333333333334
  7: ... loglikelihood=-5.098768040466029  0.8333333333333334
...
 98: ... loglikelihood=-1.4117372921765519  1.0
 99: ... loglikelihood=-1.4052738190352423  1.0
100: ... loglikelihood=-1.398916120150312  1.0
```

Сохранение модели выполняется с помощью метода `serialize()`, как показано ниже. Модель записывается в файл *en-animal.model*, открытый в теле блока `try`:

```
OutputStream modelOut = null;
modelOut = new BufferedOutputStream(dataOut);
model.serialize(modelOut);
```

Использование класса DocumentCategorizerME для классификации текста

После создания и обучения модели можно воспользоваться классом `DocumentCategorizerME` для классификации текста. Необходимо

прочитать модель, создать экземпляр класса DocumentCategorizerME и затем вызвать метод categorize(), чтобы получить массив вероятностей принадлежности текста к соответствующей категории.

Поскольку в примере выполняется операция чтения из файла, следует предусмотреть обработку возможных исключений:

```
try(InputStream modelIn =
       new FileInputStream(new File("en-animal.model"));) {
    ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

Используя поток ввода, мы создаем экземпляры классов DoccatModel и DocumentCategorizerME:

```
DoccatModel model = new DoccatModel(modelIn);
DocumentCategorizerME categorizer =
    new DocumentCategorizerME(model);
```

Метод categorize() принимает строку и возвращает массив значений типа double, в котором каждый элемент соответствует вероятности принадлежности обрабатываемого текста к указанной категории. Метод getNumberOfCategories() класса DocumentCategorizerME возвращает количество категорий, обработанных моделью. Метод getCategory() того же класса возвращает категорию по индексу массива.

В следующем фрагменте кода используются все перечисленные выше методы для вывода названия каждой категории и соответствующего ей значения вероятности:

```
double[] outcomes = categorizer.categorize(inputText);
for(int i=0; i < categorizer.getNumberOfCategories(); i++) {
    String category = categorizer.getCategory(i);
    System.out.println(category + " - " + outcomes[i]);
}
```

Для тестирования модели используем фрагмент статьи в Википедии о персонаже Тото из сказки Ф. Баума «Удивительный волшебник из страны Оз» (http://en.wikipedia.org/wiki/Toto_%28Oz%29), собаке девочки Дороти[1]. Возьмем первое предложение из раздела «The classic books»:

```
String toto = "Toto belongs to Dorothy Gale, the heroine of "
        + "the first and many subsequent books. In the first "
```

[1] В России эта сказка известна в переводе Александра Волкова под названием «Волшебник Изумрудного города», в которой песика зовут Тотошка, а девочку Элли. – *Прим. перев.*

```
        + "book, he never spoke, although other animals, native "
        + "to Oz, did. In subsequent books, other animals "
        + "gained the ability to speak upon reaching Oz or "
        + "similar lands, but Toto remained speechless.";
```

Для тестирования по категории cat взято первое предложение из раздела «Tortoiseshell and Calico» (Кошки с пестрым «черепаховым» окрасом) в уже упоминавшейся статье Википедии о кошках (http://en.wikipedia.org/wiki/Cats_and_humans):

```
String calico = "This cat is also known as a calimanco cat or "
        + "clouded tiger cat, and by abbreviation 'tortie'. "
        + "In the cat fancy, a tortoiseshell cat is patched "
        + "over with red (or its dilute form, cream) and black "
        + "(or its dilute blue) motted throughout the coat."
```

После обработки текста в переменной toto получим следующий результат. По нему можно предположить, что текст следует считать принадлежащим к категории dog:

```
dog - 0.5870711529777994
cat - 0.41292884702220056
```

Для текста в переменной calico принадлежность к категории cat выглядит более убедительно:

```
dog - 0.28960436044424276
cat - 0.7103956395557574
```

Также можно воспользоваться методом getBestCategory(), чтобы получить только категорию с наивысшим показателем вероятности. Метод использует массив полученных значений вероятности и возвращает строку. Метод getAllResults() возвращает все результаты в форме строки. Применение этих двух методов показано ниже:

```
System.out.println(categorizer.getBestCategory(outcomes));
System.out.println(categorizer.getAllResults(outcomes));
```

Для переменной calico выводятся следующие результаты:

```
cat
dog[0.2896]  cat[0.7104]
```

Использование библиотеки Stanford API

Библиотека Stanford API поддерживает несколько классификаторов. Мы рассмотрим класс ColumnDataClassifier для общей классификации и конвейер StanfordCoreNLP для анализа эмоциональной окраски текста. Иногда практическое применение классификаторов из библиотеки Stanford API может вызывать некоторые затруднения. С помощью

класса `ColumnDataClassifier` будет продемонстрирована классификация размеров коробок. На примере конвейера будет показано определение положительного или отрицательного отношения, выражаемого короткими фразами. Классификатор можно скачать с сайта http://www-nlp.stanford.edu/wiki/Software/Classifier.

Использование класса ColumnDataClassifier для классификации текста

Этот классификатор использует тип с возможностью хранения нескольких значений для описания данных. В рассматриваемом примере для создания классификатора будет использоваться обучающий файл. Затем будет обработан тестовый файл для оценки эффективности классификатора. Кроме того, класс использует файл свойств для настройки процесса создания.

В данном случае мы создадим классификатор для классификации коробок по их размерам. Определены три возможные категории: маленькие (small), средние (medium) и большие (large). Высота (height), ширина (width) и длина (length) выражаются числами с плавающей точкой – это основные характеристики коробки.

В файле свойств определяются значения параметров и имена файлов с обучающими и тестовыми данными. Здесь можно описать множество разнообразных свойств, но в рассматриваемом примере используются лишь наиболее важные.

Свойства будут храниться в файле *box.prop*. Первая группа свойств определяет количество характеристик в обучающем и тестовом файлах. Выше мы приняли к рассмотрению три характеристики, поэтому здесь описываются три столбца `realValued`. Свойства `trainFile` и `testFile` определяют местоположение и имена соответствующих файлов:

```
useClassFeature=true
1.realValued=true
2.realValued=true
3.realValued=true
trainFile=.box.train
testFile=.box.test
```

Обучающий и тестовый файлы имеют одинаковый формат. Каждая строка включает категорию и три значения – высоту, ширину и длину, – разделенные символами табуляции. Обучающий файл *box.train* содержит 60 записей, тестовый файл *box.test* содержит 30 записей. Эти файлы можно скачать с сайта www.dmkpress.com или www.

дмк.рф в разделе «Читателям – Файлы к книгам». Для экономии места здесь приведена только первая строка файла *box.train*, в которой указаны категория, высота, ширина и длина:

```
small 2.34  1.60  1.50
```

Ниже показано создание классификатора. Экземпляр `ColumnDataClassifier` создается с использованием имени файла свойств как аргумента для конструктора класса. Экземпляр интерфейса `Classifier` возвращается методом `makeClassifier()`. Этот интерфейс поддерживает три метода, два из которых задействованы в данном примере. Метод `readTrainingExamples()` читает обучающие данные из заданного файла:

```
ColumnDataClassifier cdc =
    new ColumnDataClassifier("box.prop");
Classifier<String, String> classifier =
    cdc.makeClassifier(cdc.readTrainingExamples("box.train"));
```

При выполнении примера получим развернутый вывод результата. Ниже демонстрируются и обсуждаются только наиболее важные его части. Первый фрагмент вывода воспроизводит содержимое файла свойств:

```
3.realValued=true
testFile=.box.test
...
trainFile=.box.train
```

В следующем фрагменте сообщаются количество наборов данных и различная информация о характеристиках:

```
Reading dataset from box.train ... done [0.1s, 60 items].
numDatums: 60
numLabels: 3 [small, medium, large]
...
AVEIMPROVE    The average improvement / current value
EVALSCORE   The last available eval score
Iter ## evals ## <SCALING> [LINESEARCH] VALUE TIME |GNORM| {RELNORM}
AVEIMPROVE EVALSCORE
```

Затем следуют итерации по всем данным для создания классификатора:

```
Iter 1 evals 1 <D> [113M 3.107E-4] 5.985E1 0.00s |3.829E1| {1.959E-1}
0.000E0 -
Iter 2 evals 5 <D> [M 1.000E0] 5.949E1 0.01s |1.862E1| {9.525E-2}
3.058E-3 -
Iter 3 evals 6 <D> [M 1.000E0] 5.923E1 0.01s |1.741E1| {8.904E-2}
3.485E-3 -
```

```
...
Iter 21 evals 24 <D> [1M 2.850E-1] 3.306E1 0.02s |4.149E-1| {2.122E-3}
1.775E-4 -
Iter 22 evals 26 <D> [M 1.000E0] 3.306E1 0.02s
QNMinimizer terminated due to average improvement: | newest_val -
previous_val | / |newestVal| < TOL
Total time spent in optimization: 0.07s
```

После этого классификатор готов к работе. Для проверки классификатора используется тестовый файл *box.test*. Метод `getLineIterator()` класса `ObjectBank` позволяет последовательно читать строки из файла. Кроме того, этот класс поддерживает преобразование прочитанных данных в более стандартную форму. При каждом вызове метод `getLineIterator()` возвращает очередную строку данных в формате, пригодном для работы классификатора. Для организации процесса тестирования необходим цикл:

```
for(String line :
       ObjectBank.getLineIterator("box.test", "utf-8")) {
    ...
}
```

В теле цикла `for-each` создается экземпляр класса `Datum` из полученной строки, затем вызывается его метод `classOf()`, возвращающий прогнозируемую категорию, как показано ниже. Интерфейс `Datum` поддерживает объекты, содержащие свойства (характеристики). При передаче экземпляра `Datum` методу `classOf()` возвращается категория, определенная классификатором:

```
Datum<String, String> datum = cdc.makeDatumFromLine(line);
System.out.println("Datum: ["
    + line + "]\tPredicted Category: "
    + classifier.classOf(datum));
```

При выполнении этой части кода последовательно обрабатывается каждая строка тестового файла и выводится предположительная категория, как показано ниже. В целях экономии места здесь приведены результаты только для первых двух и последних двух строк. Классификатор правильно классифицировал почти все тестовые данные:

```
Datum: [small 1.33 3.50 5.43] Predicted Category: medium
Datum: [small 1.18 1.73 3.14] Predicted Category: small
...
Datum: [large 6.01 9.35 16.64]  Predicted Category: large
Datum: [large 6.76 9.66 15.44]  Predicted Category: large
```

Для тестирования отдельных записей из файла можно воспользоваться методом `makeDatumFromStrings()`, чтобы создать экземпляр `Datum`. В следующем фрагменте кода создается одномерный массив строк, в котором каждый элемент представляет отдельный компонент данных для коробки. Первый компонент, соответствующий категории, остается пустым (`null`, пустая строка). Затем экземпляр `Datum` передается в метод `classOf()` для определения категории:

```
String sample[] = {"", "6.90", "9.8", "15.69"};
Datum<String, String> datum = cdc.makeDatumFromStrings(sample);
System.out.println("Category: " + classifier.classOf(datum));
```

Выполнение этого кода дает результат, правильно классифицирующий коробку с заданными размерами:

```
Category: large
```

Использование конвейера, поддерживаемого библиотекой Stanford для анализа эмоциональной окраски текста

В этом разделе демонстрируется, как можно использовать библиотеку Stanford для анализа эмоциональной окраски текста. Для выполнения этой задачи будет использован конвейер `StanfordCoreNLP`.

Для демонстрации будут использоваться три фрагмента текста, определенных ниже. Они взяты из обзора кинофильма «Форрест Гамп» (Forrest Gump) с сайта, название которого недвусмысленно характеризует его направленность – Rotten Tomatoes (Гнилые помидоры) (http://www.rottentomatoes.com/m/forrest_gump/):

```
String review = "An overly sentimental film with a somewhat "
    + "problematic message, but its sweetness and charm "
    + "are occasionally enough to approximate true depth "
    + "and grace.";

String sam = "Sam was an odd sort of fellow. Not prone "
    + "to angry and not prone to merriment. Overall, "
    + "an odd fellow.";

String mary = "Mary thought that custard pie was the "
    + "best pie in the world. However, she loathed "
    + "chocolate pie.";
```

Для этого типа анализа требуется аннотатор эмоциональной окраски `sentiment`, а также привычные аннотаторы `tokenize`, `ssplit` и `parse`. Последний позволяет получить более струтурированную информацию о тексте, но более подробно этот вопрос мы обсудим в главе 7

«Использование синтаксического анализатора (парсера) для выделения взаимосвязей», а пока ограничимся его применением:

```
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, parse, sentiment");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
```

Текст используется для создания экземпляра класса `Annotation`, который в дальнейшем служит аргументом метода `annotate()`, выполняющего работу по аннотированию текста:

```
Annotation annotation = new Annotation(review);
pipeline.annotate(annotation);
```

Следующий массив содержит строки с разными характеристиками отношения («Резко отрицательное», «Отрицательное», «Нейтральное», «Положительное», «Весьма положительное»):

```
String[] sentimentText = {"Very Negative", "Negative",
    "Neutral", "Positive", "Very Positive"};
```

Метод `get()` класса `Annotation` возвращает объект, реализующий интерфейс `CoreMap`. В нашем случае эти объекты представляют результаты разделения исходного текста на предложения, как показано в следующем фрагменте. Для каждого предложения формируется экземпляр класса `Tree`, представляющий древовидную структуру синтаксического разбора текста для анализа его эмоциональной окраски. Метод `getPredictedClass()` возвращает индекс в массиве `sentimentText`, соответствующий строке отношения для текущего теста:

```
for(CoreMap sentence : annotation.get(
       CoreAnnotations.SentencesAnnotation.class)) {
    Tree tree = sentence.get(
       SentimentCoreAnnotations.AnnotatedTree.class);
    int score = RNNCoreAnnotations.getPredictedClass(tree);
    System.out.println(sentimentText[score]);
}
```

Если выполнить этот код со строкой `review`, будет выведен следующий результат:

```
Positive
```

Текст в строке `sam` состоит из трех предложений. Оценки эмоциональной окраски для каждого предложения выводятся на отдельных строках:

```
Neutral
Negative
Neutral
```

В строке mary два предложения. Вот результаты их анализа:

```
Positive
Neutral
```

Использование библиотеки LingPipe для классификации текста

В этом разделе библиотека LingPipe будет использоваться для решения нескольких задач классификации: общая классификация текста с помощью обученных моделей, анализ эмоциональной окраски текста и идентификация языка документа. Мы рассмотрим следующие темы:

- подготовка обучающего текста с помощью класса Classified;
- обучение моделей с использованием других обучающих категорий;
- классификация текста средствами библиотеки LingPipe;
- анализ эмоциональной окраски текста средствами библиотеки LingPipe;
- определение языка, на котором написан текст.

Некоторые примеры в данном разделе будут использовать общие предварительно определенные переменные, объявленные ниже. В библиотеке LingPipe имеются уже подготовленные обучающие данные для нескольких категорий. Массив categories содержит имена категорий, поддерживаемых библиотекой LingPipe:

```
String[] categories = {"soc.religion.christian",
    "talk.religion.misc", "alt.atheism", "misc.forsale"};
```

Класс DynamicLMClassifier предназначен для классификации. Его экземпляр создается с использованием массива categories для чтения названий категорий. Значение переменной nGramSize определяет количество элементов в последовательности, с которой работает модель в процессе классификации:

```
int nGramSize = 6;
DynamicLMClassifier<NGramProcessLM> classifier =
    DynamicLMClassifier.createNGramProcess(categories, NGramSize);
```

Подготовка обучающего текста с помощью класса Classified

Решение задачи общей классификации текста средствами LingPipe включает предварительный этап обучения экземпляра класса DynamicLMClassifier на обучающих файлах и последующее использование

обученного экземпляра для выполнения собственно классификации. Библиотека LingPipe предоставляет несколько готовых обучающих наборов данных, размещенных в подкаталоге *demos/data/fourNewsGroups/4news-train*. Этими наборами мы воспользуемся для демонстрации процесса обучения. Рассматриваемый ниже пример является упрощенной версией программы из руководства http://alias-i.com/lingpipe/demos/tutorial/classify/read-me.html.

Сначала определяется каталог, в котором размещены обучающие файлы:

```
String directory = ".../demos";
File trainingDirectory = new File(directory
    + "/data/fourNewsGroups/4news-train");
```

В этом каталоге находятся четыре подкаталога с именами в массиве `categories`. В каждом подкаталоге находится группа файлов с числовыми именами. Файлы содержат информацию по темам, соответствующим именам подкаталогов (http://qwone.com/~jason/20Newsgroups).

В процессе обучения метод `handle()` класса `DynamicLMClassifier` работает с каждым файлом и с каждой категорией, используя конкретный файл для создания обучающего экземпляра текущей категории, затем этот экземпляр добавляется к модели. В процессе применяются вложенные циклы `for`.

В теле внешнего цикла `for` создается объект `File`, использующий имя текущего каталога, затем к этому объекту применяется метод `list()`, возвращающий список файлов в данном каталоге. Имена файлов сохраняются в массиве `trainingFiles`, необходимом для обработки во внутреннем цикле:

```
for(int i=0; i < categories.length; ++i) {
    File classDir = new File(trainingDirectory, categories[i]);
    String[] trainingFiles = classDir.list();
    // Здесь размещается внутренний цикл for.
}
```

Ниже показано тело внутреннего цикла `for`, где открывается каждый файл и считывается текст из него. Экземпляр класса `Classification` представляет классификацию по заданной категории. Он используется вместе с текстом для создания экземпляра класса `Classified`. Метод `handle()` класса `DynamicLMClassifier` добавляет в модель новую информацию:

```
for(int j=0; j < trainingFiles.length; ++j) {
    try {
        File file = new File(classDir, trainingFiles[j]);
```

```
            String text = Files.readFromFile(file, "ISO-8859-1");
            Classification classification =
                new Classification(categories[i]);
            Classified<CharSequence> classified =
                new Classified<>(text, classification);
            classifier.handle(classified);
        } catch(IOException ex) {
            // Обработка исключения.
        }
    }
```

Чтобы стал доступным метод `readFromFile()`, используйте класс из пакета расширения `com.aliasi.util.Files` вместо стандартного `java.io.File`.

Полученный классификатор можно сериализовать и сохранить для повторного использования. Сериализацию объектов поддерживает вспомогательный класс `AbstractExternalizable`. Он предлагает статический метод `compileTo()`, принимающий экземпляр класса `Compilable` и объект типа `File`. Процедура записи объекта в файл показана ниже:

```
try {
    AbstractExternalizable.compileTo((Compilable)classifier,
        new File("classifier.model"));
} catch(IOException ex) {
    // Обработка исключения.
}
```

Процедура загрузки и использования классификатора будет рассматриваться в разделе «Классификация текста с помощью библиотеки LingPipe» ниже.

Использование других обучающих категорий

Данные из других групп новостей размещены на сайте http://qwone.com/~jason/20Newsgroups/. Эти наборы можно использовать для обучения других моделей по темам, перечисленным в табл. 6.1. Несмотря на то что предлагается всего лишь 20 категорий, их тематика позволяет обучать достаточно полезные модели. Кроме того, можно скачать три различных варианта, в которых данные отсортированы и из них удалены повторяющиеся фрагменты.

Таблица 6.1. Данные из групп новостей, собранных на сайте http://qwone.com/~jason/20Newsgroups/

Имена групп новостей	
comp.graphics	sci.crypt
comp.os.ms-windows.misc	sci.electronics
comp.sys.ibm.pc.hardware	sci.med

Окончание табл. 6.1

Имена групп новостей	
comp.sys.mac.hardware	sci.space
comp.windows.x	misc.forsale
rec.autos	talk.politics.misc
rec.motorcycles	talk.politics.guns
rec.sport.baseball	talk.politics.mideast
rec.sport.hockey	talk.religion.misc
alt.atheism	

Классификация текста с помощью библиотеки LingPipe

Для классификации текста воспользуемся методом `classify()` класса `DynamicLMClassifier` и продемонстрируем его применение на двух различных фрагментах текста:

- `forSale` – первый абзац с сайта http://www.homes.com/for-sale/;
- `martinLuther` – первое предложение из второго абзаца в статье о Мартине Лютере из Википедии http://en.wikipedia.org/wiki/Martin_Luther.

Фрагменты текста определяются как строки:

```
String forSale =
    "Finding a home for sale has never been "
    + "easier. With Homes.com you can search new "
    + "homes, foreclosures, multi-family homes, "
    + "as well as condos and townhouses for sale. "
    + "You can even search our real estate agent "
    + "directory to work with a professional "
    + "Realtor and find your perfect home.";
String martinLuther =
    "Luther taught that salvation and subsequently "
    + "eternity in heaven is not earned by good deeds "
    + "but is received only as a free gift of God's "
    + "grace through faith in Jesus Christ as redeemer "
    + "from sin and subsequently eternity in Hell.";
```

Для повторного использования классификатора, сериализованного и сохраненного в предыдущем разделе, воспользуемся методом `readObject()` класса `AbstractExternalizable`. Но применим класс `LMClassifier` вместо `DynamicLMClassifier`. Оба класса поддерживают метод `classify()`, но сериализация объекта типа `DynamicLMClassifier` более сложна:

```
LMClassifier classifier = null;
try {
    classifier =
```

```
        (LMClassifier)AbstractExternalizable.readObject(
            new File("classifier.model"));
} catch(IOException | ClassNotFoundException ex) {
    // Обработка исключения.
}
```

В следующем фрагменте применяется метод `classify()` класса `LMClassifier`. Метод возвращает экземпляр класса `JointClassification`, который используется для определения наиболее подходящей категории для обрабатываемого текста:

```
JointClassification classification =
    classifier.classify(text);
System.out.println("Text: " + text);
String bestCategory = classification.bestCategory();
System.out.println("Best Category: " + bestCategory);
```

При обработке текста `forSale` выводится следующий результат:

```
Text: Finding a home for sale has never been easier. With Homes.
com you can search new homes, foreclosures, multi-family homes, as
well as condos and townhouses for sale. You can even search our real
estate agent directory to work with a professional Realtor and find your
perfect home.
Best Category: misc.forsale
```

Для текста `martinLuther` результат будет таким:

```
Text: Luther taught that salvation and subsequently eternity in heaven
is not earned by good deeds but is received only as a free gift of
God's grace through faith in Jesus Christ as redeemer from sin and
subsequently eternity in Hell.
Best Category: soc.religion.christian
```

В обоих случаях текст классифицирован правильно.

Анализ эмоциональной окраски текста с помощью библиотеки LingPipe

Анализ эмоциональной окраски текста очень похож на общую классификацию текста с той лишь разницей, что использует только две категории: положительное и отрицательное отношения.

Для обучения модели нужны файлы с обучающими данными. Мы воспользуемся упрощенной версией примера анализа эмоциональной окраски из руководства http://alias-i.com/lingpipe/demos/tutorial/sentiment/read-me.html и данными, сформированными для оценки кинофильмов (http://www.cs.cornell.edu/people/pabo/movie-review-data/

review_polarity.tar.gz) на основе 1000 положительных и 1000 отрицательных рецензий, взятых из киноархива IMDb[1].

Архивный файл с обзорами необходимо скачать и распаковать. При этом будет создан каталог *txt_sentoken* с двумя подкаталогами – *neg* и *pos*, – содержащими соответствующие типы рецензий. Несмотря на то что некоторые из этих файлов зарезервированы для проверки и оценки эффективности созданной модели, мы будем использовать все файлы, чтобы упростить описание процесса.

Начнем с инициализации переменных, объявленных в разделе «Использование библиотеки LingPipe для классификации текста». Массив `categorize` теперь состоит из двух элементов, соответствующих вышеназванным категориям. Переменной `classifier` присваивается новый экземпляр класса `DynamicLMClassifier`, созданный с помощью нового массива категорий и переменной `nGramSize`, которой присвоено значение 8:

```
categories = new String[2];
categories[0] = "neg";
categories[1] = "pos";
nGramSize = 8;
classifier = DynamicLMClassifier.createNGramProcess(
    categories, nGramSize);
```

Как и ранее, будет создана последовательность экземпляров на основе содержимого обучающих файлов. Следующий фрагмент не требует подробных разъяснений, поскольку очень похож на код из раздела «Подготовка обучающего текста с помощью класса Classified». Основное различие лишь в том, что здесь используются только две категории:

```
String directory = "...";
File trainingDirectory = new File(directory, "txt_sentoken");
for(int i=0; i < categories.length; ++i) {
    Classification classification =
        new Classification(categories[i]);
    File file = new File(trainingDirectory, categories[i]);
    File[] trainingFiles = file.listFiles();
    for(int j=0; j < trainingFiles.length; ++j) {
        try {
            String review = Files.readFromFile(
                trainingFiles[j], "ISO-8859-1");
            Classified<CharSequence> classified =
```

[1] IMDb – Internet Movie Database, интернет-база кинофильмов. – *Прим. перев.*

```
                new Classified<>(review, classification);
            classifier.handle(classified);
        } catch(IOException ex) {
            // Обработка исключения.
        }
    }
}
```

Теперь модель готова к практическому использованию. Для обработки возьмем фразу из обзора фильма «Форрест Гамп»:

```
String review = "An overly sentimental film with a somewhat "
    + "problematic message, but its sweetness and charm "
    + "are occasionally enough to approximate true depth
    + "and grace.";
```

Метод `classify()` выполняет собственно анализ эмоциональной окраски текста. Он возвращает экземпляр класса `Classification`, метод `bestCategory()` которого позволяет получить наиболее подходящую категорию, как показано ниже:

```
Classification classification = classifier.classify(review);
String bestCategory = classification.bestCategory();
System.out.println("Best Category: " + bestCategory);
```

После выполнения получим следующий результат:

```
Best Category: pos
```

Эта методика превосходно работает и с другими категориями текста.

Определение языка документа с помощью библиотеки LingPipe

Библиотека LingPipe содержит модель *langid-leipzig.classifier*, специально обученную для идентификации нескольких языков. Модель находится в каталоге *demos/models*. Список поддерживаемых языков приведен в табл. 6.2. Эта модель разработана с использованием данных, основанных на корпусе Leipzig Corpora Collection (http://corpora.uni-leipzig.de/). Еще один неплохой инструмент можно найти на сайте http://code.google.com/language-detection/.

Таблица 6.2. Список языков, определение которых поддерживается моделью библиотеки LingPipe

Язык	Обозначение	Язык	Обозначение
Каталонский	cat	Итальянский	it
Датский	dk	Японский	jp
Английский	en	Корейский	kr

Окончание табл. 6.2

Язык	Обозначение	Язык	Обозначение
Эстонский	ee	Норвежский	no
Финский	fi	Диалект славянского языка в юго-восточной Германии (Sorbian)	sorb
Французский	fr	Шведский	se
Немецкий	de	Турецкий	tr

Для применения модели используется тот же код, который был приведен выше в разделе «Классификация текста с помощью библиотеки LingPipe». Начнем с хорошо знакомого фрагмента обзора фильма «Форрест Гамп»:

```
String text = "An overly sentimental film with a somewhat "
    + "problematic message, but its sweetness and charm "
    + "are occasionally enough to approximate true depth "
    + "and grace.";
System.out.println("Text: " + text);
```

Создается экземпляр класса `LMClassifier` с использованием файла *langid-leipzig.classifier*:

```
LMClassifier classifier = null;
try {
    classifier = (LMClassifier)AbstractExternalizable.readObject(
        new File(".../langid-leipzig.classifier"));
} catch(IOException | ClassNotFoundException ex) {
    // Обработка исключения.
}
```

Вызывается метод `classify()`, затем метод `bestCategory()` для выбора наиболее подходящего языка:

```
Classification classification = classifier.classify(text);
String bestCategory = classification.bestCategory();
System.out.println("Best Language: " + bestCategory);
```

Результат демонстрирует правильное определение текста на английском языке:

```
Text: An overly sentimental film with a somewhat problematic message,
but its sweetness and charm are occasionally enough to approximate true
depth and grace.
Best Language: en
```

Для следующего примера используется первое предложение из статьи Википедии на шведском языке (http://sv.wikipedia.org/wiki/Svenska):

```
text = "Svenska är ett östnordiskt språk som talas av cirka "
    + "tio miljoner personer[1], främst i Finland "
    + "och Sverige.";
```

Для этого образца текста правильно определяется шведский язык:

```
Text: Svenska är ett östnordiskt språk som talas av cirka tio miljoner
personer[1], främst i Finland och Sverige.
Best Language: se
```

Обучение данной модели производится точно так же, как и всех ранее рассмотренных моделей из библиотеки LingPipe, но с учетом одной особенности: обрабатываемый текст может быть написан на нескольких разных языках. Это может усложнить процесс определения языка.

Резюме

В этой главе обсуждались вопросы классификации текста и рассматривались методики выполнения этого процесса. Классификация текста необходима как основа для выполнения других задач, таких как обнаружение спама, рассылаемого по электронной почте, определение автора документа, идентификация пола автора и определения языка, на котором написан текст.

На отдельном примере был продемонстрирован процесс выполнения анализа эмоциональной окраски текста. Этот тип анализа позволяет определить положительное или отрицательное отношение автора к обсуждаемой теме. Кроме того, возможно получение и некоторых других эмоциональных характеристик.

Большинство рассмотренных методик требовало обучения модели на специально подготовленных данных. Как правило, обученная модель проверяется на отдельном наборе тестовых данных. После обучения и проверки модель готова к практическому использованию.

В следующей главе будут подробно рассматриваться процесс синтаксического анализа и его роль в определении взаимосвязей в исследуемом тексте.

Глава 7

Использование синтаксического анализатора (парсера) для выделения взаимосвязей

Синтаксический анализ, или *парсинг* (*parsing*) – это процесс создания дерева синтаксической структуры для текстового фрагмента, который может представлять собой строку исходного кода или предложение на естественном языке. Реализовать такой анализ для языков программирования достаточно просто, так как они формализованы в достаточной степени для выполнения этой задачи. С другой стороны, та же формализация иногда несколько затрудняет написание исходного кода. Реализовать синтаксический анализ для естественного языка значительно сложнее из-за неоднозначности, присущей практически всем естественным языкам. Синтаксическому анализу языков программирования посвящены многие замечательные книги, мы же обратим свое внимание на естественные языки.

Дерево синтаксического анализа (*parse tree*) – это иерархическая структура данных, представляющая синтаксическую структуру предложения. Часто она изображается в форме древовидного графа с единственным корнем, как мы увидим ниже. Дерево синтаксического анализа помогает определить взаимосвязи между элементами синтаксической структуры.

Синтаксический анализ необходим как основа для выполнения многих других задач:

- машинная трансляция (компиляция) исходных кодов на языках программирования;
- синтез речи по тексту;
- распознавание речи;
- грамматический контроль;
- извлечение информации.

Кореферентность, или *референциальное тождество* (*coreference resolution*), – это такое отношение между двумя или несколькими словами/словосочетаниями, при котором они ссылаются на один и тот же объект, ситуацию, свойство и т. п. Например, рассмотрим предложение:

«Ted went to the party where he made an utter fool of himself».

(«Тэд отправился на вечеринку и выглядел там как набитый дурак»[1].)

Здесь слова «Ted» (Тэд), «he» (он) и «himself» (себя) ссылаются на один и тот же объект «Ted». Это важно для правильной интерпретации текста и определения относительной смысловой значимости его частей. Ниже мы рассмотрим решение этой задачи средствами библиотеки Stanford API.

Извлечение взаимосвязей и информации из текста представляет собой важную задачу обработки естественного языка. Взаимосвязи могут существовать между объектами, например в предложении между подлежащим и его дополнениями, или другими объектами, и даже между подлежащим и образом его действий. Иногда требуется определение всех взаимосвязей и представление их в структурированной форме. Эту информацию можно использовать либо для выдачи результата для дальнейшего использования человеком, либо в качестве исходных данных для последующих задач обработки естественного языка.

В этой главе рассматриваются процесс синтаксического анализа (парсинга) и практическое применение дерева синтаксического анализа. Кроме того, демонстрируются процесс извлечения взаимосвязей и определение их типов, обсуждаются области приложения полученных взаимосвязей, а также использование библиотек NLP для решения всех перечисленных задач.

---

[1] Для проявления кореферентности в русском языке вариант перевода может быть более формализованным: «Тэд отправился на вечеринку, где он показал себя как набитый дурак». – *Прим. перев.*

Типы взаимосвязей

Существует множество возможных типов взаимосвязей. Несколько категорий и примеров представлено в табл. 7.1. Огромное количество типов категорий можно найти на сайте Freebase (https://www.freebase.com/). Это база данных людей, мест и прочих вещей, упорядоченная по категориям. Словарь-тезаурус WordNet (http://wordnet.princeton.edu/) содержит достаточно большое количество типов взаимосвязей.

Таблица 7.1. Некоторые типы взаимосвязей и примеры для них

Взаимосвязь	Пример
Личная	father-of, sister-of, girlfriend-of отец (кого-л.), сестра (кого-л.), подруга (кого-л.)
Организационная	subsidiary-of, subcommittee-of подконтрольный (кому-л.), подчиненный (кому-л.), подкомиссия (чего-л.)
Пространственная	near-to, northeast-of, under около (чего-л.), к северо-востоку от (чего-л.), под (чем-л.)
Физическая	part-of, composed-of часть (чего-л.), состоящий из (чего-л.)
Взаимодействие	bonds-with, associates-with, reacts-with связан с (чем-л.), объединен с (чем-л.), воздействует на (что-л.)

В главе 4 «Поиск людей и именованных объектов» рассматривалась задача распознавания и идентификации именованных объектов, представляющая собой низкоуровневый тип NLP-классификации. Но многие приложения требуют гораздо большего, поэтому обращаются к методикам определения различных типов взаимосвязей. Например, после выполнения идентификации человека по имени ставится задача уточнения всех существующих взаимосвязей этого человека с другими объектами.

После определения таких объектов создаются ссылки на документы, содержащие эти объекты, или сами объекты используются для индексации. В приложениях, предназначенных для ответов на запросы, именованные объекты часто используются в ответах. При определении эмоциональной окраски текста необходимо связать эту окраску (отношение) с некоторым объектом.

Например, рассмотрим следующее исходное предложение:

He was the last person to see Fred.
(Он был последним, кто видел Фреда.)

В главе 4 использовалась методика распознавания и идентификации именованных объектов, поддерживаемая библиотекой OpenNLP, и для приведенного выше предложения был получен следующий результат:

```
Span: [7..9) person
Entity: Fred
```

С помощью синтаксического анализатора (парсера) из библиотеки OpenNLP можно получить гораздо больше информации о том же предложении:

```
(TOP (S (NP (PRP He)) (VP (VBD was) (NP (NP (DT the) (JJ last)
(NN person)) (SBAR (S (VP (TO to) (VP (VB see))))))) (. Fred.)))
```

Рассмотрим еще одно предложение:

The cow jumped over the moon.
(Корова перепрыгнула через луну.)

Для этого предложения результат синтаксического анализа будет таким:

```
(TOP (S (NP (DT The) (NN cow)) (VP (VBD jumped) (PP (IN over)
(NP (DT the) (NN moon))))))
```

Существуют два типа синтаксического анализа:

- *определение зависимости (dependency)*: здесь все внимание сосредоточено на взаимосвязях между словами;
- *определение структуры выражения (phrase structure)*: построение рекурсивной структуры выражения (предложения, словосочетания).

При определении зависимостей могут использоваться метки, такие как подлежащее (subject), определяющее слово (determiner) и предлоги (prepositions), – упрощающие поиск взаимосвязей. К числу методик синтаксического анализа относятся *перенос-свертка (shift-reduce)*, *остовное дерево (spanning tree)* и *нисходящий (каскадный) поверхностный синтаксический анализ (cascaded chunking)*. Отличительные особенности этих методик здесь не обсуждаются, мы сосредоточимся на использовании нескольких реализаций синтаксических анализаторов и получении конкретных результатов от них.

Деревья синтаксического анализа

Дерево синтаксического анализа (parsing tree) представляет иерархическую структуру взаимосвязей между элементами текста. Напри-

мер, дерево зависимостей показывает взаимосвязи между грамматическими элементами предложения. Снова обратимся к предложению:

The cow jumped over the moon.
(Корова перепрыгнула через луну.)

Дерево синтаксического анализа для этого предложения показано ниже. Оно сгенерировано с помощью методики, рассматриваемой ниже, в разделе «Использование класса LexicalizedParser»:

```
(ROOT
  (S
    (NP (DT The) (NN cow))
    (VP (VBD jumped)
      (PP (IN over)
            (NP (DT the) (NN moon))))
      (. .)))
```

Графическое представление дерева синтаксического анализа для рассматриваемого предложения показано на рис. 7.1. Изображение сформировано с помощью приложения с сайта http://nlpviz.bpodgursky.com/home. Представить текст в виде графической структуры позволяет редактор GrammarScope (http://grammarscope.sourceforge.net/).

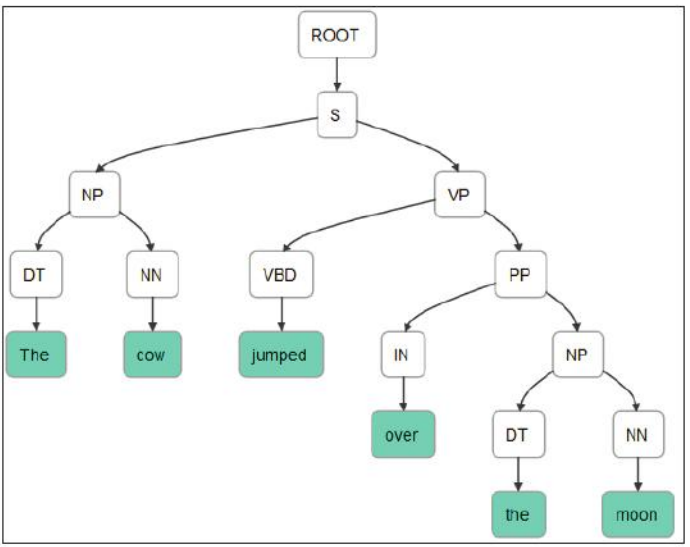


Рис. 7.1. Дерево синтаксического анализа для простого предложения

Этот инструмент использует библиотеку Stanford и графический пользовательский интерфейс на основе библиотеки Swing для генерации дерева синтаксического анализа, грамматической структуры, зависимостей, классифицированных по типам, и семантической схемы (семантического графа) текста.

Синтаксический анализ предложения можно выполнить несколькими способами. Синтаксический анализ труден из-за необходимости одновременно обрабатывать большой объем текста, который может содержать множество неоднозначно трактуемых элементов. Ниже приведены результаты, показывающие другие возможные варианты дерева зависимостей для рассматриваемого выше примера предложения. Деревья сгенерированы с помощью библиотеки OpenNLP, а сам процесс получения этих результатов будет продемонстрирован ниже, в разделе «Использование библиотеки OpenNLP»:

```
(TOP (S (NP (DT The) (NN cow)) (VP (VBD jumped) (PP (IN over)
(NP (DT the) (NN moon))))))
(TOP (S (NP (DT The) (NN cow)) (VP (VP (VBD jumped) (PRT (RP over)))
(NP (DT the) (NN moon)))))
(TOP (S (NP (DT The) (NNS cow)) (VP (VBD jumped) (PP (IN over)
(NP (DT the) (NN moon))))))
```

Каждый из приведенных вариантов представляет незначительно отличающийся от других результат синтаксического анализа одного и того же предложения. Наиболее достоверный вариант синтаксического анализа выведен первым.

Использование полученных взаимосвязей

Выделенные из текста взаимосвязи можно использовать для различных целей:

- создание баз знаний (knowledge bases);
- создание справочников, указателей, каталогов;
- выполнение операций поиска;
- патентный анализ;
- анализ фондового рынка;
- анализ разведданных.

Примером представления взаимосвязей может служить информационный блок из статьи в Википедии, показанный на рис. 7.2. В данном случае в блоке представлена информация о штате Оклахома, содержащая такие типы взаимосвязей, как официальный язык, столица, сведения о площади, размерах и т. п.

В Интернете существует множество баз данных, сформированных на основе Википедии, которые выполняют операции извлечения взаимосвязей и информации, например:

- *Resource Description Framework (RDF)* – использует выражения-триады, такие как Yosemite-location-California (Йосемити-местонахождение-Калифорния), в которых второе слово – location – обозначает конкретную взаимосвязь (местонахождение). С этой базой данных можно познакомиться на сайте http://www.w3.org/RDF/;
- *DBPedia* – содержит более миллиарда выражений-триад и представляет собой яркий пример базы знаний, созданной на основе Википедии. Более подробное описание этой базы данных можно найти по адресу http://dbpedia.org/About.

Рис. 7.2. Представление взаимосвязей на примере информационного блока из статьи в Википедии

Другим простым, но весьма показательным примером является информационный блок, выводимый на поисковом сайте Google при некоторых вариантах определения объекта поиска, например в рассматриваемом случае задан образец «planet mercury». Как показано на рис. 7.3, по этому запросу выводится не только обычный список ссылок, но также список обобщенных свойств (категорий взаимосвязей) и ряд изображений планеты Меркурий в правой части веб-страницы.

Процесс извлечения информации также используется при создании поисковых веб-индексов, разрабатываемых для конкретных

сайтов. Такие индексы облегчают навигацию по сайту. Пример веб-индекса для Бюро переписи населения США (U.S. Census Bureau – http://www.census.gov/main/www/a2z) показан на рис. 7.4.

Рис. 7.3. Представление взаимосвязей (общих свойств) на примере информационного блока на странице результатов поиска сайта Google

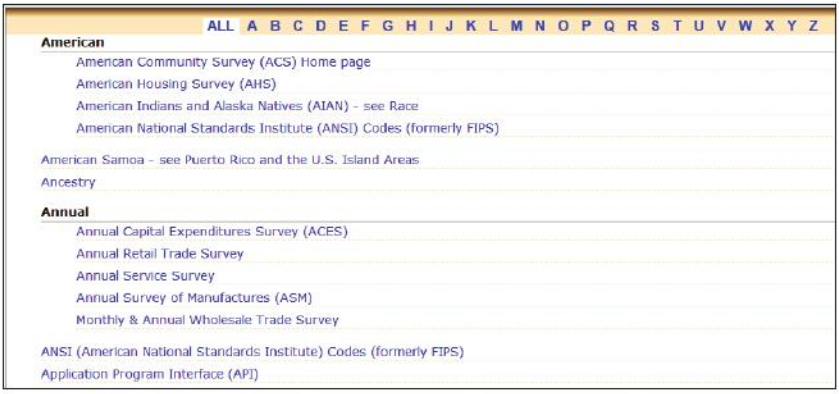


Рис. 7.4. Пример веб-индекса для Бюро переписи населения США

Извлечение взаимосвязей из текста

Существует множество методик извлечения взаимосвязей из текста. Их можно классифицировать по следующим группам:

- шаблоны (образцы), создаваемые вручную;
- методы с полным контролем;
- методы с неполным контролем или методы без контроля:
  - методы самораскрутки (bootstrapping);
  - методы с дистанционным контролем;
  - методы без контроля.

Модели, создаваемые вручную, используются при отсутствии обучающих данных. Такая ситуация может возникать при работе в новой сфере бизнеса или с абсолютно новым типом проекта. Зачастую при этом требуются специализированные правила. Правило может быть следующим:

«Если используются слова “actor” (актер) или “actress” (актриса), но отсутствует слово “movie” (фильм) или слово “commercial” (коммерческий), текст следует классифицировать как play (спектакль, театральная постановка)».

Следует отметить, что такой подход требует больших трудозатрат и дополнительной подготовки обрабатываемого текста.

Если обучающих данных не очень много, наилучшим выбором будет *наивный байесовский классификатор* (*Naive Bayes classifier*). Если объем обучающих данных достаточно велик, можно воспользоваться *методом опорных векторов* (*Support Vector Machine, SVM*), *методом логистической регрессии с регуляризацией* (*Regularized Logistic Regression*) и алгоритмом обучения под названием *«случайный лес»* (*Random Forest*).

Несмотря на то что понимание сущности работы этих методик весьма полезно, здесь мы не будем рассматривать внутренних функциональных особенностей, а сосредоточимся на их практическом применении.

Использование библиотек NLP API

Для демонстрации синтаксического анализа и извлечения взаимосвязанной информации из текста будут использоваться библиотеки OpenNLP и Stanford API. Библиотека LingPipe также вполне пригодна для решения названных задач, но в данном разделе не рассматри-

вается. Пример применения LingPipe для синтаксического анализа биомедицинской литературы можно найти по адресу: http://alias-i.com/lingpipe-3.9.3/demos/tutorial/medline/read-me.html.

Использование библиотеки OpenNLP

Класс `ParserTool` обеспечивает полную поддержку синтаксического анализа текста. Его статический метод `parseLine()` возвращает экземпляр класса `Parser` и принимает три аргумента:

- строку с текстом, подлежащим анализу;
- экземпляр класса `Parser`;
- целочисленное значение, определяющее, сколько вариантов синтаксического анализа будет возвращено.

Экземпляр класса `Parser` содержит элементы синтаксического анализа. Варианты синтаксического анализа возвращаются в порядке их предполагаемой достоверности. Для создания экземпляра класса `Parser` используется метод `create()` класса `ParserFactory`. Этот метод принимает экземпляр класса `ParserModel`, созданный из содержимого файла *en-parser-chunking.bin*.

Весь процесс показан ниже, при этом поток ввода для чтения файла модели создается в блоке попытки захвата ресурсов `try`. Сначала создается экземпляр `ParserModel`, затем экземпляр `Parser`:

```
String fileLocation = getModelDir() + "/en-parser-chunking.bin";
try(InputStream modelInputStream =
           new FileInputStream(fileLocation);) {
     ParserModel model = new ParserModel(modelInputStream);
    Parser parser = ParserFactory.create(model);
    ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

Для демонстрации процесса синтаксического анализа будет использоваться простое предложение, с которым мы уже встречались в начале главы. В следующем фрагменте вызывается метод `parseLine()` с числом 3 в третьем аргументе, чтобы получить три варианта синтаксического анализа:

```
String sentence = "The cow jumped over the moon.";
Parse parses[] = ParserTool.parseLine(sentence, parser, 3);
```

Далее выводятся варианты синтаксического анализа с указанием вероятностей их достоверности, как показано ниже:

```
for(Parse parse : parses) {
    parse.show();
    System.out.println("Probability: " + parse.getProb());
}
```

Результат выглядит следующим образом:

```
(TOP (S (NP (DT The) (NN cow)) (VP (VBD jumped) (PP (IN over)
(NP (DT the) (NN moon))))))
Probability: -1.043506016751117
(TOP (S (NP (DT The) (NN cow)) (VP (VP (VBD jumped) (PRT (RP over)))
(NP (DT the) (NN moon)))))
Probability: -4.248553665013661
(TOP (S (NP (DT The) (NNS cow)) (VP (VBD jumped) (PP (IN over)
(NP (DT the) (NN moon))))))
Probability: -4.761071294573854
```

Обратите внимание, что варианты синтаксического анализа незначительно отличаются порядком и типами присвоенных тегов. Ниже показан другой формат вывода первого варианта, более удобный для чтения:

```
(TOP
      (S
          (NP
               (DT The)
               (NN cow)
          )
          (VP
               (VBD jumped)
               (PP
                    (IN over)
                    (NP
                         (DT the)
                         (NN moon)
                    )
               )
          )
     )
)
```

Для вывода результата также можно использовать метод `showCodeTree()`, показывающий взаимосвязи типа «предок–потомок» (parent–child):

```
parse.showCodeTree();
```

Результат для первого варианта синтаксического анализа показан ниже. В первой части каждой строки выводятся уровни элемента

в квадратных скобках. Далее указывается тег, за которым следуют два хэш-значения, разделенные комбинацией символов ->. Первое хэш-значение соответствует текущему элементу, второе – его предку. Например, в третьей строке для определяющего слова (DT) `The` в качестве предка указана именная группа (noun phrase – NP) `The cow`:

```
[0] S -929208263 -> -929208263 TOP The cow jumped over the moon
[0.0] NP -929237012 -> -929208263 S The cow
[0.0.0] DT -929242488 -> -929237012 NP The
[0.0.0.0] TK -929242488 -> -929242488 DT The
[0.0.1] NN -929034400 -> -929237012 NP cow
[0.0.1.0] TK -929034400 -> -929034400 NN cow
[0.1] VP -928803039 -> -929208263 S jumped over the moon
[0.1.0] VBD -928822205 -> -928803039 VP jumped
[0.1.0.0] TK -928822205 -> -928822205 VBD jumped
[0.1.1] PP -928448468 -> -928803039 VP over the moon
[0.1.1.0] IN -928460789 -> -928448468 PP over
[0.1.1.0.0] TK -928460789 -> -928460789 IN over
[0.1.1.1] NP -928195203 -> -928448468 PP the moon
[0.1.1.1.0] DT -928202048 -> -928195203 NP the
[0.1.1.1.0.0] TK -928202048 -> -928202048 DT the
[0.1.1.1.1] NN -927992591 -> -928195203 NP moon
[0.1.1.1.1.0] TK -927992591 -> -927992591 NN moon
```

Другой способ доступа к элементам синтаксического анализа предоставляет метод `getChildren()`, возвращающий массив объектов `Parse`, каждый из которых соответствует отдельному элементу анализа. Используя различные методы класса `Parse`, можно получить текст, тег и метки каждого элемента, как показано ниже:

```
Parse children[] = parse.getChildren();
for(Parse parseElement : children) {
    System.out.println(parseElement.getText());
    System.out.println(parseElement.getType());
    Parse tags[] = parseElement.getTagNodes();
    System.out.println("Tags");
    for(Parse tag : tags) {
        System.out.println("[" + tag + "]"
            + " type: " + tag.getType()
            + "  Probability: " + tag.getProb()
            + "  Label: " + tag.getLabel());
    }
}
```

Выполнение этого кода дает следующий результат:

```
The cow jumped over the moon
S
Tags
```

```
[The] type: DT  Probability: 0.9380626549164167  Label: null
[cow] type: NN  Probability: 0.9574993337971017  Label: null
[jumped] type: VBD  Probability: 0.9652983971550483  Label: S-VP
[over] type: IN  Probability: 0.7990638213315913  Label: S-PP
[the] type: DT  Probability: 0.9848023215770413  Label: null
[moon] type: NN  Probability: 0.9942338356992393  Label: null
```

Использование библиотеки Stanford API

Библиотека Stanford API поддерживает несколько методик синтаксического анализа. Сначала рассмотрим синтаксический анализатор общего назначения, представленный классом `LexicalizedParser`. Затем один из способов вывода результата синтаксического анализа с помощью класса `TreePrint`. Далее будет показан процесс определения зависимостей между словами, использующий класс `GrammaticalStructure`.

Использование класса LexicalizedParser

Класс `LexicalizedParser` – это лексикализированный синтаксический анализатор вероятностной контекстно-свободной грамматики (ВКС – Probablistic Context-Free Grammar, PCFG). Для синтаксического анализа он может использовать различные модели. Метод `apply()` в сочетании с экземпляром списка `List`, состоящего из объектов `CoreLabel`, позволяет создать дерево синтаксического анализа.

В следующем фрагменте экземпляр синтаксического анализатора создается на основе модели *englishPCFG.ser.gz*:

```
String parserModel = ".../models/lexparser/englishPCFG.ser.gz";
LexicalizedParser lexicalizedParser =
    LexicalizedParser.loadModel(parserModel);
```

Экземпляр списка объектов `CoreLabel` создается с помощью метода `toCoreLabelList()` класса `Sentence`. Объект `CoreLabel` содержит обрабатываемое слово и прочую информацию, которая тем не менее не содержит тегов и меток для данного слова. Предложение уже разделено на слова-токены:

```
String[] sentenceArray = {"The", "cow", "jumped", "over",
    "the", "moon", "."};
List<CoreLabel> words = Sentence.toCoreLabelList(sentenceArray);
```

Теперь можно вызвать метод `apply()`:

```
Tree parseTree = lexicalizedParser.apply(words);
```

Вывести результаты синтаксического анализа легко можно с помощью метода `pennPrint()`, который выводит дерево синтаксического

анализа точно так же, как Penn TreeBank (http://www.sfs.uni-tuebingen.de/~dm/07/autumn/795.10/ptb-annotation-guide/root.html):

```
parseTree.pennPrint();
```

Результат выглядит так:

```
(ROOT
  (S
    (NP (DT The) (NN cow))
    (VP (VBD jumped)
      (PP (IN over)
        (NP (DT the) (NN moon))))
    (. .)))
```

Класс `Tree` предоставляет множество методов для работы с деревьями синтаксического анализа.

Использование класса TreePrint

Класс `TreePrint` предлагает простой способ вывода дерева. Экземпляр этого класса создается с использованием строки, описывающей требуемый формат вывода. Массив допустимых форматов вывода можно получить с помощью статической переменной `outputTreeFormat` (см. табл. 7.2).

Таблица 7.2. Список возможных форматов вывода дерева синтаксического анализа

Строки, обозначающие формат вывода дерева синтаксического анализа		
penn	dependencies	collocations
oneline	typedDependencies	semanticGraph
rootSymbolOnly	typedDependenciesCollapsed	conllStyleDependencies
words	latexTree	conll2007
wordsAndTags	xmlTree	

Для обозначения грамматических взаимосвязей в предложении библиотека Stanford использует тип «зависимости» (dependencies). Этот тип подробно описан в специальном справочном руководстве «Stanford Typed Dependencies Manual» (http://nlp.stanford.edu/software/dependencies_manual.pdf).

В следующем примере показано, как можно использовать класс `TreePrint`. Саму операцию вывода выполняет метод `printTree()`.

В данном случае создается объект типа `TreePrint`, выводящий тип зависимостей «typeDependenciesCollapsed».

```
TreePrint treePrint = new TreePrint("typedDependenciesCollapsed");
treePrint.printTree(parseTree);
```

Результат работы этого примера показан ниже, здесь числовые значения отражают позицию слова в предложении:

```
det(cow-2, The-1)
nsubj(jumped-3, cow-2)
root(ROOT-0, jumped-3)
det(moon-6, the-5)
prep_over(jumped-3, moon-6)
```

Если использовать формат вывода «penn», получится следующий результат:

```
(ROOT
  (S
    (NP (DT The) (NN cow))
    (VP (VBD jumped)
      (PP (IN over)
        (NP (DT the) (NN moon))))
    (. .)))
```

Строка «dependencies» позволяет получить простой список зависимостей:

```
dep(cow-2,The-1)
dep(jumped-3,cow-2)
dep(null-0,jumped-3,root)
dep(jumped-3,over-4)
dep(moon-6,the-5)
dep(over-4,moon-6)
```

Можно указывать сразу несколько форматов, перечисляя их через запятую. Следующая строка объединяет форматы «penn» и «typedDependenciesCollapsed»:

```
"penn,typedDependenciesCollapsed"
```

Поиск зависимостей между словами с помощью класса GrammaticalStructure

Другая методика синтаксического анализа текста предполагает использование объекта класса `LexicalizedParser`, созданного в предыдущем разделе, в сочетании с интерфейсом `TreebankLanguagePack`. *Treebank* – это корпус текстов, содержащий разметку по синтаксической или семантической информации, что позволяет получить сведения о внутренней структуре предложения. Самым первым основным корпусом текстов Treebank был Penn TreeBank (http://www.cis.upenn.

edu/~treebank/). Корпусы текстов Treebank могут создаваться вручную или в полуавтоматическом режиме.

В следующем примере показано, как отформатировать простую строку с помощью синтаксического анализатора. Экземпляр токенизатора создается с помощью фабрики токенизаторов.

Здесь повторно используется класс `CoreLabel`, рассматривавшийся в разделе «Использование класса LexicalizedParser»:

```
String sentence = "The cow jumped over the moon.";
TokenizerFactory<CoreLabel> tokenizerFactory =
    PTBTokenizer.factory(new CoreLabelTokenFactory(), "");
Tokenizer<CoreLabel> tokenizer =
    tokenizerFactory.getTokenizer(new StringReader(sentence));
List<CoreLabel> wordList = tokenizer.tokenize();
parseTree = lexicalizedParser.apply(wordList);
```

Интерфейс `TreebankLanguagePack` определяет методы для работы с корпусом текстов Treebank. В следующем фрагменте создается последовательность объектов и затем – экземпляр `TypedDependency`, который нужен для получения информации об элементах предложения. Объект класса `GrammaticalStructureFactory` используется для создания экземпляра класса `GrammaticalStructure`.

По имени этого класса легко понять, что его экземпляр хранит грамматическую информацию о взаимосвязях между элементами, представленную в форме дерева:

```
TreebankLanguagePack tlp = lexicalizedParser.treebankLanguagePack;
GrammaticalStructureFactory gsf = tlp.grammaticalStructureFactory();
GrammaticalStructure gs = gsf.newGrammaticalStructure(parseTree);
List<TypedDependency> tdl = gs.typedDependenciesCCprocessed();
```

Теперь можно просто вывести полученный список, как показано ниже:

```
System.out.println(tdl);
```

Результат выглядит следующим образом:

```
[det(cow-2, The-1), nsubj(jumped-3, cow-2), root(ROOT-0, jumped-3),
det(moon-6, the-5), prep_over(jumped-3, moon-6)]
```

Эту информацию также можно извлечь с помощью методов `gov()`, `reln()` и `dep()`, которые возвращают соответственно управляющее слово, взаимосвязь (отношение) и зависимый элемент, как показано ниже:

```
for(TypedDependency dependency : tdl) {
    System.out.println("Governor word: [" + dependency.gov()
        + "] Relation: [" + dependency.reln().getLongName()
        + "] Dependent Word: [" + dependency.dep() + "]");
}
```

В этом случае результат будет выглядеть так:

```
Governor Word: [cow/NN] Relation: [determiner] Dependent Word: [The/DT]
Governor Word: [jumped/VBD] Relation: [nominal subject] Dependent Word:
[cow/NN]
Governor Word: [ROOT] Relation: [root] Dependent Word: [jumped/VBD]
Governor Word: [moon/NN] Relation: [determiner] Dependent Word: [the/DT]
Governor Word: [jumped/VBD] Relation: [prep_collapsed] Dependent Word:
[moon/NN]
```

Здесь можно наблюдать взаимосвязи в предложении и отдельные взаимосвязанные элементы.

Поиск референциального тождества между объектами

Кореференция, или *референциальное тождество (coreference resolution),* обозначает наличие в тексте двух и более выражений, указывающих на одно и то же имя, объект или ситуацию. Рассмотрим следующее предложение:

«He took his cash and she took her change and together they bought their lunch».

(«Он взял свои (его) деньги, она взяла свою (ее) сдачу, и они вместе заплатили за свой (их) обед».)

В этом предложении встречается несколько кореференций. Слово «his» (его) указывает на местоимение «He» (он), а слово «her» (ее) указывает на местоимение «she» (она). Кроме того, слово «they» (они) указывает на оба местоимения «He» и «she».

Эндофора (endophora) – это выражение, которое ссылается на что-либо в одном и том же тексте, встречающееся раньше или позже самого этого выражения. Эндофора делится на подкатегории: *анафора (anaphora)* и *катафора (cataphora)* (иногда рассматривается и третья подкатегория – *ссылка на себя (self-reference)*). В следующей группе предложений слово «It» (Он) является анафорой, которая ссылается на свой антецедент (предшествующее имя объекта – antecedent) «the earthquake» (подземный толчок):

«Mary felt the earthquake. It shook the entire building».

(«Мэри почувствовала подземный толчок. Он потряс все здание».)

В следующем предложении слово «she» (она) является катафорой, указывающей на последующее имя объекта (*постцедент – postcedent*) «Mary»:

«As she sat there, Mary felt the earthquake».

(«Как только она села на место, Мэри почувствовала подземный толчок» – нелитературный дословный перевод. Обычно в литературном русском языке подобные катафоры заменяются причастными или деепричастными оборотами: «Присев, Мэри почувствовала подземный толчок».)

Библиотека Stanford API поддерживает определение кореференции при помощи класса `StanfordCoreNLP`, использующего аннотацию `dcoref`. Применение этого класса будет продемонстрировано на примере предложения, приведенного в начале раздела.

Сначала создается конвейер и используется метод `annotate()`, как показано ниже:

```
String sentence = "He took his cash and she took her change "
    + "and together they bought their lunch.";
Properties props = new Properties();
props.put("annotators",
    "tokenize, ssplit, pos, lemma, ner, parse, dcoref");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
Annotation annotation = new Annotation(sentence);
pipeline.annotate(annotation);
```

Метод `get()` класса `Annotation`, вызванный с аргументом `CorefChainAnnotation.class`, возвращает экземпляр ассоциативного массива `Map` объектов `CorefChain`, как показано ниже. Эти объекты хранят информацию о кореференциях, найденных в предложении:

```
Map<Integer, CorefChain> corefChainMap =
    annotation.get(CorefChainAnnotation.class);
```

Набор объектов `CorefChain` индексируется целыми числами (ключами). Итерация по объектам демонстрируется в следующем фрагменте. По очередному ключу выводится соответствующий объект `CorefChain`:

```
Set<Integer> set = corefChainMap.keySet();
Iterator<Itegrator> setIterator = set.iterator();
while(setIterator.hasNext()) {
    CorefChain corefChain = corefChainMap.get(setIterator.next());
    System.out.println("CorefChain: " + corefChain);
}
```

После выполнения кода примера получим следующий результат:

```
CorefChain: CHAIN1-["He" in sentence 1, "his" in sentence 1]
CorefChain: CHAIN2-["his cash" in sentence 1]
```

```
CorefChain: CHAIN4-["she" in sentence 1, "her" in sentence 1]
CorefChain: CHAIN5-["her change" in sentence 1]
CorefChain: CHAIN7-["they" in sentence 1, "their" in sentence 1]
CorefChain: CHAIN8-["their lunch" in sentence 1]
```

При помощи методов классов `CorefChain` и `CorefMention` мы получаем более подробную информацию. Второй класс хранит информацию о конкретной кореференции, найденной в предложении.

Для получения и вывода этой информации необходимо добавить в тело цикла `while` следующий фрагмент. Поля `startIndex` и `endIndex` класса `CorefMention` соответствуют позициям слов в обрабатываемом предложении:

```
System.out.print("ClusterId: " + corefChain.getChainID());
CorefMention mention = corefChain.getRepresentativeMention();
System.out.println(" CorefMention: " + mention
    + " Span: [" + mentionSpan + "]");

List<CorefMention> mentionList =
    corefChain.getMentionsInTextualOrder();
Iterator<CorefMention> mentionIterator = mentionList.iterator();
while(mentionIterator.hasNext()) {
    CorefMention cfm = mentionIterator.next();
    System.out.println("\tMention: " + cfm
        + " Span: [" + mention.mentionSpan + "]");
    System.out.print("\tMention Type: " + cfm.mentionType
        + " Gender: " + cfm.gender);
    System.out.println(" Start: " + cfm.startIndex + " End: "
        + cfm.endIndex);
}
System.out.println();
```

В целях экономии места ниже приводятся результаты только для первой и последней кореференций:

```
CorefChain: CHAIN1-["He" in sentence 1, "his" in sentence 1]
ClusterId: 1 CorefMention: "He" in sentence 1 Span: [He]
  Mention: "He" in sentence 1 Span: [He]
  Mention Type: PRONOMINAL Gender: MALE Start: 1 End: 2
  Mention: "his" in sentence 1 Span: [He]
  Mention Type: PRONOMINAL Gender: MALE Start: 3 End: 4
...
CorefChain: CHAIN8-["their lunch" in sentence 1]
ClusterId: 8 CorefMention: "their lunch" in sentence 1 Span: [their
lunch]
  Mention: "their lunch" in sentence 1 Span: [their lunch]
  Mention Type: NOMINAL Gender: UNKNOWN Start: 14 End: 16
```

Извлечение взаимосвязей для системы «вопрос–ответ»

В этом разделе рассматривается методика извлечения взаимосвязей, которая может оказаться полезной для ответов на запросы, сформулированные в вопросительной форме. Примеры таких запросов приведены ниже:

Who is/was the 14th president of the United States?
(Кто является/был 14-м президентом США?)

What is the 1st president's home town?
(В каком городе родился 1-й президент?)

When was Herbert Hoover president?
(Когда Херберт Гувер был президентом?)

Процесс получения ответов на подобные типы вопросов не так уж прост. Мы рассмотрим одну из методик нахождения ответов на вопросы определенного типа, но при этом многие аспекты процесса будут упрощены. Однако даже при всех допущенных ограничениях и упрощениях мы убедимся, что система правильно отвечает на запросы.

Процесс включает несколько этапов, или шагов:

1) поиск взаимосвязей (зависимостей) между словами;
2) определение типа вопросов;
3) извлечение взаимосвязанных компонентов вопроса;
4) поиск ответа;
5) представление ответа.

Ниже демонстрируется обобщенная рабочая среда (основа) для определения принадлежности вопросов к типам who (кто), what (что), when (когда) или where (где). Далее мы рассмотрим некоторые особенности, связанные с поиском ответов на вопросы типа who (кто).

Чтобы не усложнять пример, ограничим предметную область вопросов тематикой, связанной с президентами США. Для поиска ответов на вопросы будет использоваться простейшая база данных, в которой хранится информация о президентах.

Поиск взаимосвязей (зависимостей) между словами

Вопрос, на который необходимо найти ответ, сохраняется в обычной строке:

```
String question = "Who is the 32nd president of the United States?";
```

Для дальнейшей работы потребуется экземпляр класса LexicalizedParser, использовавшийся в разделе «Поиск зависимостей между словами с помощью класса GrammaticalStructure». Для удобства мы повторим соответствующий код:

```
String parserModel = ".../englishPCFG.ser.gz";
LexicalizedParser lexicalizedParser =
    LexicalizedParser.loadModel(parserModel);

TokenizerFactory<CoreLabel> tokenizerFactory =
    PTBTokenizer.factory(new CoreLabelTokenFactory(), "");
Tokenizer<CoreLabel> tokenizer =
    tokenizerFactory.getTokenizer(new StringReader(question));
List<CoreLabel> wordList = tokenizer.tokenize();
Tree parseTree = lexicalizedParser.apply(wordList);

TreeBankLanguagePack tlp = lexicalizedParser.treebankLanguagePack();
GrammaticalStructureFactory gsf = tlp.grammaticalStructureFactory();
GrammaticalStructure gs = gsf.newGrammaticalStructure(parseTree);
List<TypedDependency> tdl = gs.typedDependenciesCCprocessed();
System.out.println(tdl);
for(TypedDependency dependency : tdl) {
    System.out.println("Governor Word: [" + dependency.gov()
        + "] Relation: [" + dependency.reln().getLongName()
        + "] Dependent Word: [" + dependency.dep() + "]");
}
```

При выполнении этого кода получим следующий ответ на вопрос выше:

```
[root(ROOT-0, Who-1), cop(Who-1, is-2), det(president-5, the-3),
amod(president-5, 32nd-4), nsubj(Who-1, president-5), det(States-9,
the-7), nn(States-9, United-8), prep_of(president-5, States-9)]
Governor Word: [ROOT] Relation: [root] Dependent Word: [Who/WP]
Governor Word: [Who/WP] Relation: [copula] Dependent Word: [is/VBZ]
Governor Word: [president/NN] Relation: [determiner] Dependent Word:
[the/DT]
Governor Word: [president/NN] Relation: [adjectival modifier] Dependent
Word: [32nd/JJ]
Governor Word: [Who/WP] Relation: [nominal subject] Dependent Word:
[president/NN]
Governor Word: [States/NNPS] Relation: [determiner] Dependent Word:
[the/DT]
Governor Word: [States/NNPS] Relation: [nn modifier] Dependent Word:
[United/NNP]
Governor Word: [president/NN] Relation: [prep_collapsed] Dependent Word:
[States/NNPS]
```

Эта информация послужит основой для определения типа вопроса.

Определение типа вопроса

Обнаруженные взаимосвязи позволяют перейти к предполагаемым способам определения различных типов вопросов. Например, чтобы определить, что вопрос относится к типу who (кто), можно проверить взаимосвязь `nominal subject` (формальное подлежащее) и обнаружить, что управляющим словом здесь является `who` (кто).

Следующий фрагмент выполняет итерации по типам зависимостей в вопросе, чтобы найти совпадение со словосочетанием `nominal subject`. Если совпадение найдено, вызывается метод `processWhoQuestion()` для обработки этого типа вопроса:

```
for(TypedDependency dependency : tdl) {
    if("nominal subject".equals(dependency.reln().getLongName())
       && "who".equalsIgnoreCase(dependency.gov().originalText())) {
       processWhoQuestion(tdl);
    }
}
```

Приведенная выше простая конструкция распознавания типа вопроса работает достаточно хорошо. Она правильно определяет все варианты одного и того же вопроса:

```
Who is the 32nd president of the United States?
Who was the 32nd president of the United States?
The 32nd president of the United States was who?
The 32nd president is who of the United States?
```

Также можно определять другие типы вопросов, используя соответствующие условия выбора. Ниже приводятся примеры различных типов вопросов:

```
What was the 3rd President's party?
(Какую партию представлял 3-й президент?)

When was the 12th president inaugurated?
(Когда происходила инаугурация 12-го президента?)

Where is th 30th president's home town?
(Где находится родной город 30-го президента?)
```

Определить тип вопроса можно с помощью взаимосвязей (отношений), перечисленных в табл. 7.3.

Следует отметить, что такой подход требует непосредственного включения строк, описывающих отношения, в код программы.

Таблица 7.3. Взаимосвязи между словами, позволяющие определить тип вопроса

Тип вопроса	Отношение	Управляющее слово	Зависимое слово
What (что)	nominal subject (формальное подлежащее)	what (что)	отсутствует
When (когда)	adverbial modifier (определение-наречие)	отсутствует	when (когда)
Where (где)	adverbial modifier (определение-наречие)	отсутствует	where (где)

Поиск ответа на вопрос

После определения типа вопроса найденные в тексте взаимосвязи можно использовать для поиска ответа на заданный вопрос. Для демонстрации этого процесса мы реализуем метод `processWhoQuestion()`. Он использует список `TypedDependency` для сбора информации, необходимой для ответов на вопросы типа who из предметной области «президенты США». В частности, по порядковому номеру определяет, о каком именно президенте идет речь.

Для поиска конкретной информации также будет нужен список президентов. Для решения этой задачи предназначен метод `createPresidentList()`, который читает файл *PresidentList* с полным именем президента, год его инаугурации и последний год президентства. Файл можно скачать с сайта www.packtpub.com, а также с сайта www.dmkpress.com или www.дмк.рф – в разделе «Читателям – Файлы к книгам». Формат файла показан на примере одной записи из него:

```
George Washington (1789-1797)
```

Метод `createPresidentList()` использует класс `SimpleTokenizer` из библиотеки OpenNLP для токенизации каждой строки. Имя президента может быть сформировано из различного количества токенов. После определения имени годы президентства определяются без затруднений:

```
public List<President> createPresidentList() {
    ArrayList<President> list = new ArrayList<>();
    String line = null;
    try(FileReader reader = new FileReader("PresidentList");
           BufferedReader br = new BufferedReader(reader)) {
       while((line = br.readLine()) != null) {
           SimpleTokenizer simpleTokenizer = SimpleTokenizer.INSTANCE;
           String tokens[] = simpleTokenizer.tokenize(line);
           String name = "";
```

```
            String start = "";
            String end = "";
            int i = 0;
            while(!"(".equals(tokens[i])) {
                name += tokens[i] + " ";
                i++;
            }
            start = tokens[i+1];
            end = tokens[i+3];
            if(end.equalsIgnoreCase("present")) {
                end = start;
            }
            list.add(new President(name, Integer.parseInt(start),
                Integer.parseInt(end)));
        }
    } catch(IOException ex) {
        // Обработка исключения.
    }
    return list;
}
```

Класс `President`, определяемый ниже, позволяет хранить информацию о президентах. Методы извлечения информации здесь не приводятся ввиду очевидности их реализации, а также в целях экономии места:

```
public class President {
    private String name;
    private int start;
    private int end;

    public President(String name, int start, int end) {
        this.name = name;
        this.start = start;
        this.end = end;
    }
    ...
}
```

Далее приводится определение метода `processWhoQuestion()`. Здесь тоже используются типы зависимостей для извлечения порядковых числительных из текста вопроса. Если управляющим словом является `president`, а отношение обозначено как `adjectival modifier` (определение в форме наречия), зависимым словом является порядковое числительное. Найденная строка передается в метод `getOrder()`, возвращающий целое число, соответствующее числительному. Полученное значение необходимо скорректировать на 1, так как список имен президентов нумеруется не с нуля, а с единицы:

```
public void processWhoQuestion(List<TypeDependency> tdl) {
    List<President> list = createPresidentList();
    for(TypedDependency dependency : tdl) {
        if("president".equalsIgnoreCase(
                dependency.gov().originalText())
                && "adjectival modifier".equals(
                  dependency.reln().getLongName())) {
            String positionText = dependency.dep().originalText();
            int position = getOrder(positionText) - 1;
            System.out.println("The president is "
                + list.get(position).getName());
        }
    }
}
```

Метод `getOrder()` принимает первые цифровые символы и преобразует их в целое число. Это упрощенная версия, так как для реальных операций преобразования требуется распознавание различных форм числительных, в том числе таких, как «first» (первый) и «sixteenth» (шестнадцатый):

```
private static int getOrder(String position) {
    String tmp = "";
    int i = 0;
    while(Character.isDigit(position.charAt(i))) {
        tmp += position.charAt(i++);
    }
    return Integer.parseInt(tmp);
}
```

Если выполнить этот пример, он выведет следующий результат:

```
The president is Franklin D . Roosevelt
```

Приведенная реализация представляет простой пример извлечения информации из заданного предложения и последующего ее использования для поиска ответов на вопросы. Обработка других типов вопросов осуществляется аналогично, и их реализацию читатель может выполнить в качестве практического упражнения.

Резюме

Мы подробно рассмотрели процесс синтаксического анализа и возможности использования его результатов для определения взаимосвязей в тексте. Синтаксический анализ применяется также для многих других целей, например для проверки грамматической корректности текста и для автоматического перевода. В любом тексте

возможно наличие ряда взаимосвязей между словами, таких как «родитель (кого-либо)», «около (чего-либо)», «под (чем-либо)» и многих других. Такие отношения определяют связи элементов текста друг с другом.

Синтаксический анализ позволяет определить взаимосвязи в исследуемом тексте. Выявленные взаимосвязи можно использовать для извлечения требуемой информации. В этой главе были продемонстрированы методики синтаксического анализа текста с применением библиотек OpenNLP и Stanford API.

Кроме того, было показано применение библиотеки Stanford API для поиска кореференций в тексте. Кореференции (референциальные тождества) возникают в случаях, когда два и более выражения, таких как «he» (он) или «they» (они), ссылаются на одно и то же имя или объект.

В конце главы был приведен пример применения синтаксического анализатора для определения взаимосвязей в предложении. Найденные взаимосвязи использовались для извлечения информации при поиске ответа на простой запрос типа «who» (кто) о президентах США.

В следующей главе будет рассматриваться объединение методик, разработанных в этой и в предыдущих главах, для решения более сложных комплексных задач.

Глава 8

Комплексные методики

В этой главе будут рассматриваться вопросы использования сочетаний нескольких простых методик для решения более крупных задач обработки естественного языка. Начнем мы с краткого введения в процесс подготовки данных. Затем будут описаны конвейеры и способы их формирования. Конвейер – это всего лишь последовательность нескольких элементарных задач, объединенных для решения более сложных задач. Основное преимущество конвейера заключается в возможности вставки и удаления различных его элементов для решения общей задачи несколькими способами, слегка отличающимися друг от друга.

Библиотека Stanford API поддерживает хорошо спроектированную архитектуру конвейера, которая неоднократно использовалась в данной книге. Здесь мы уделим гораздо больше внимания подробностям этой методики, затем покажем, как создавать конвейеры с помощью библиотеки OpenNLP.

Подготовка данных является важным начальным этапом при решении многих задач обработки естественного языка. Некоторые этапы процесса подготовки данных были показаны в главе 1 «Основы обработки естественного языка», затем процесс нормализации исходных данных обсуждался в главе 2 «Поиск фрагментов текста». В этой главе мы сосредоточимся на извлечении текста из разнообразных источников данных, таких как HTML-страницы, документы в формате Word и PDF.

Класс `StanfordCoreNLP` из библиотеки Stanford представляет собой удачный пример конвейера, практическое применение которого не вызывает затруднений. В некотором смысле он уже предварительно подготовлен для работы. Последовательность реально выполняемых задач-компонентов зависит от добавляемых аннотаций. Такая схема успешно работает при решении многих типов задач.

Несмотря на то что, в отличие от библиотеки Stanford API, другие библиотеки не поддерживают непосредственно конвейерную архитектуру и создание конвейера связано с определенными трудностями, конвейерные методики могут оказаться более гибкими для многих приложений. Мы покажем процесс формирования такого конвейера на примере использования средств библиотеки OpenNLP.

Подготовка данных

При решении большинства задач обработки естественного языка самым первым этапом является извлечение текста из какого-либо источника. Здесь мы кратко рассмотрим процедуры извлечения текста из документов в форматах HTML, Word и PDF. Существует несколько специализированных библиотек, поддерживающих извлечение текста, но мы воспользуемся следующими:

- для HTML – Boilerpipe (https://code.google.com/p/boilerpipe/);
- для Word – POI (http://poi.apache.org/index.html);
- для PDF – PDFBox (http://pdfbox.apache.org/).

Некоторые библиотеки поддерживают формат XML для ввода и вывода. Например, класс `XMLUtils` из библиотеки Stanford обеспечивает поддержку чтения XML-файлов и обработки XML-данных. Класс `XMLParser` из библиотеки LingPipe позволяет выполнить синтаксический анализ текста в формате XML.

Организации хранят свои данные и документы в разнообразных форматах, но при этом крайне редко используются простые текстовые файлы. Презентации хранятся в виде слайдов PowerPoint, спецификации создаются как документы Word, а маркетинговые и прочие материалы для широкой публики выпускаются в формате PDF. Практически все организации и компании представлены в Интернете, следовательно, много полезной информации можно найти в HTML-документах. Перечисленные выше источники данных весьма широко распространены, поэтому необходимы инструментальные средства для извлечения из них текстов для обработки.

Использование библиотеки Boilerpipe для извлечения текста из HTML-документов

Существует несколько библиотек, поддерживающих извлечение текста из документов в формате HTML. В этом разделе демонстрируется использование библиотеки Boilerpipe (https://code.google.com/p/boilerpipe/) для выполнения этой операции. Библиотека предоставляет

гибкий прикладной программный интерфейс не только для извлечения полного текста из HTML-документа, но и для получения выбранных фрагментов документа, например заголовка и отдельных блоков текста.

В качестве обрабатываемого документа взята HTML-страница из Википедии со статьей о Берлине http://en.wikipedia/wiki/Berlin. Часть этой страницы показана на рис. 8.1. Чтобы воспользоваться библиотекой Boilerpipe, необходимо скачать компонент для Xerces Parser с сайта http://xerces.apache.org/index.html.

Рис. 8.1. Фрагмент HTML-страницы, представляющей статью Википедии Berlin

Сначала необходимо создать объект `URL`, представляющий исходную HTML-страницу. Блок `try-catch` позволяет обрабатывать исключения:

```
try {
    URL url = new URL("http://en.wikipedia.org/wiki/Berlin");
    ...
    } catch(MalformedURLException ex) {
```

```
        // Обработка исключения.
    } catch(BoilerpipeProcessingException | SAXException
          | IOException ex) {
        // Обработка исключений.
}
```

Для извлечения текста будут использоваться два класса. Первый класс `HTMLDocument` представляет исходный документ в формате HTML. Класс `TextDocument` представляет текст в HTML-документе. Этот класс включает один или несколько объектов типа `TextBlock`, причем к каждому из этих объектов можно получить независимый доступ.

В следующем фрагменте кода создается экземпляр класса `HTMLDocument` для страницы Berlin. Класс `BoilerpipeSAXInput` использует этот источник ввода данных для создания экземпляра `TextDocument`. Затем метод `getText()` класса `TextDocument` извлекает текст. Он принимает два аргумента: первый определяет необходимость включения в текст экземпляров `TextBlock`, отмеченных как контент (содержимое), а второй определяет необходимость включения в текст экземпляров `TextBlock`, имеющих метку «не контент» (noncontent). В рассматриваемом примере в текст включаются оба типа блоков:

```
HTMLDocument htmlDoc = HTMLFetcher.fetch(url);
InputSource is = htmlDoc.toInputSource();
TextDocument document =
    new BoilerpipeSAXInput(is).getTextDocument();
System.out.println(document.getText(true, true));
```

Исходная страница весьма велика, поэтому при выводе получаем извлеченный текст достаточно большого объема. В целях экономии места ниже приведена лишь часть результата:

```
Berlin
From Wikipedia, the free encyclopedia
Jump to: navigation , search
This article is about the capital of Germany. For other uses, see Berlin
(disambiguation) .
...
Privacy policy
About Wikipedia
Disclaimers
Contact Wikipedia
Developers
Mobile view
```

Метод `getTextBlocks()` возвращает список объектов типа `TextBlock` для обрабатываемого документа. Прочие методы позволяют полу-

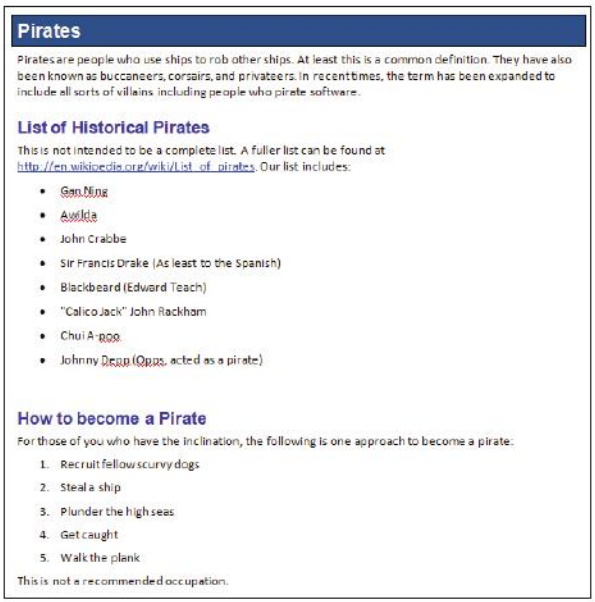
чить доступ к тексту в блоках и к информации о тексте, например можно узнать количество слов в блоке.

Использование библиотеки POI для извлечения текста из документов в формате Word

Apache POI Project (http://poi.apache.org/index.html) – библиотека, используемая для извлечения текста из документов в форматах Microsoft Office. Эта обширная библиотека поддерживает извлечение информации из документов Word и из документов в других форматах, таких как Excel и Outlook.

Помимо библиотеки POI, потребуется также библиотека XMLBeans (http://xmlbeans.apache.org/), обеспечивающая поддержку POI. Скомпилированные компоненты XMLBeans можно скачать с сайта http://www.java2s.com/Code/Jar/x/Downloadxmlbeans230jar.htm.

Основная цель данного раздела – продемонстрировать практическое применение библиотеки POI для извлечения текста из документов в формате Word. В примере используется файл *TestDocument.docx*, содержимое которого показано на рис. 8.2.



Pirates

Pirates are people who use ships to rob other ships. At least this is a common definition. They have also been known as buccaneers, corsairs, and privateers. In recent times, the term has been expanded to include all sorts of villains including people who pirate software.

List of Historical Pirates

This is not intended to be a complete list. A fuller list can be found at http://en.wikipedia.org/wiki/List_of_pirates. Our list includes:

- Gan Ning
- Awilda
- John Crabbe
- Sir Francis Drake (At least to the Spanish)
- Blackbeard (Edward Teach)
- "Calico Jack" John Rackham
- Chui A-poo
- Johnny Depp (Opps, acted as a pirate)

How to become a Pirate

For those of you who have the inclination, the following is one approach to become a pirate:

1. Recruit fellow scurvy dogs
2. Steal a ship
3. Plunder the high seas
4. Get caught
5. Walk the plank

This is not a recommended occupation.

Рис. 8.2. Содержимое обрабатываемого файла TestDocument.docx

Разные версии текстового процессора Word используют различные форматы файлов. Чтобы упростить выбор класса для извлечения текста, соответствующего формату, за основу взят класс-фабрика `ExtractorFactory`.

Впечатляюще широкие возможности библиотеки POI нисколько не усложняют процесс извлечения текста. Как показано в следующем фрагменте кода, объект типа `FileInputStream`, представляющий файл *TestDocument.docx*, используется методом `createExtractor()` класса `ExtractorFactory` для выбора требуемого экземпляра `POITextExtractor`. Это базовый класс для нескольких классов, поддерживающих операции извлечения текста. Метод `getText()` применяется к выбранному экстрактору для получения текста:

```
try {
    FileInputStream fis = new FileInputStream("TestDocument.docx");
    POITextExtractor textExtractor =
        ExtractorFactory.createExtractor(fis);
    System.out.println(textExtractor.getText());
} catch(IOException ex) {
    // Обработка исключения.
} catch(OpenXML4JException | XmlException ex) {
    // Обработка исключений.
}
```

В целях экономии места ниже приводится только часть извлеченного текста:

```
Pirates
Pirates are people who use ships to rob other ships. At least this is a
common definition. They have also been known as buccaneers, corsairs, and
privateers. In
...
Our list includes:
Gan Ning
Awilda
...
Get caught
Walk the plank
This is not a recommended occupation.
```

Иногда необходимо получить немного больше информации об исходном Word-документе. В библиотеке POI имеется класс `POIXMLPropertiesTextExtractor`, который предоставляет доступ к основным, дополнительным расширенным и настраиваемым свойствам документа. Существуют два способа получения строки, содержащей многие из этих свойств:

- использование метода getMetadataTextExtractor() с последующим вызовом метода getText(), как показано ниже:

```
POITextExtractor metaExtractor =
    textExtractor.getMetadataTextExtractor();
System.out.println(metaExtractor.getText());
```

- создание экземпляра класса POIXMLPropertiesTextExtractor с использованием объекта XWPFDocument, представляющего Word-документ, как показано ниже:

```
fis = new FileInputStream("TestDocument.docx");
POIXMLPropertiesTextExtractor properties =
    new POIXMLPropertiesTextExtractor(new XWPFDocument(fis));
System.out.println(properties.getText());
```

Оба описанных выше способа дают одинаковый результат:

```
Created = Sat Jan 03 18:27:00 CST 2015
CreatedString = 2015-01-04T00:27:00Z
Creator = Richard
LastModifiedBy = Richard
LastPrinted = Sat Jan 03 18:27:00 CST 2015
LastPrintedString = 2015-01-04T00:27:00Z
Modified = Mon Jan 05 14:01:00 CST 2015
ModifiedString = 2015-01-05T20:01:00Z
Revision = 3
Application = Microsoft Office Word
AppVersion = 12.0000
Characters = 762
CharactersWithSpaces = 894
Company =
HyperlinksChanged = false
Lines = 6
LinksUpToDate = false
Pages = 1
Paragraphs = 1
Template = Normal.dotm
TotalTime = 20
```

Класс CoreProperties предназначен для хранения набора основных свойств документа. Его метод getCoreProperties() предоставляет доступ к основным свойствам:

```
CoreProperties coreProperties = properties.getCoreProperties();
System.out.println(properties.getCorePropertiesText());
```

Здесь перечислены основные свойства:

```
Created = Sat Jan 03 18:27:00 CST 2015
CreatedString = 2015-01-04T00:27:00Z
```

```
Creator = Richard
LastModifiedBy = Richard
LastPrinted = Sat Jan 03 18:27:00 CST 2015
LastPrintedString = 2015-01-04T00:27:00Z
Modified = Mon Jan 05 14:01:00 CST 2015
ModifiedString = 2015-01-05T20:01:00Z
Revision = 3
```

Для доступа к отдельным свойствам существуют специализированные методы, такие как, например, `getCreator()`, `getCreated()`, `getModified()` и др. Дополнительные расширенные свойства представлены классом `ExtendedProperties` и доступны через метод `getExtendedProperties()`, как показано ниже:

```
ExtendedProperties extendedProperties =
    properties.getExtendedProperties();
System.out.println(properties.getExtendedPropertiesText());
```

Этот фрагмент выведет следующий результат:

```
Application = Microsoft Office Word
AppVersion = 12.0000
Characters = 762
CharactersWithSpaces = 894
Company =
HyperlinksChanged = false
Lines = 6
LinksUpToDate = false
Pages = 1
Paragraphs = 1
Template = Normal.dotm
TotalTime = 20
```

Такие методы, как `getApplication()`, `getAppVersion()`, `getPages()` и прочие, позволяют получить доступ к отдельным дополнительным свойствам.

Использование библиотеки PDFBox для извлечения текста из документов в формате PDF

Проект *Apache PDFBox* (http://pdfbox.apache.org/) – это библиотека для обработки документов в формате PDF. Она поддерживает извлечение текста и многие другие операции, такие как объединение документов, заполнение форм и создание PDF-файлов. В этом разделе будет продемонстрирован только процесс извлечения текста.

В рассматриваемом примере будет использоваться файл *TestDocument.pdf*, полученный путем сохранения в формате PDF файла

TestDocument.docx, с которым мы работали в предыдущем разделе «Использование библиотеки POI для извлечения текста из документов в формате Word».

Процесс извлечения текста прост и понятен. Сначала создается объект `File` для доступа к PDF-документу. Класс `PDDocument` представляет собственно документ, а класс `PDFTextStripper` позволяет выполнить операцию извлечения текста с помощью метода `getText()`:

```
try {
    File file = new File("TestDocument.pdf");
    PDDocument pdDocument = PDDocument.load(file);
    PDFTextStripper stripper = new PDFTextStripper();
    String text = stripper.getText(pdDocument);
    System.out.println(text);
    pdDocument.close();
} catch(IOException ex) {
    // Обработка исключения.
}
```

Извлеченный текст очень длинный, поэтому ниже приведена только часть результата:

```
Pirates
Pirates are people who use ships to rob other ships. At least this is a
common definition. They have also been known as buccaneers, corsairs, and
privateers. In
...
Our list includes:
Gan Ning
Awilda
...
4. Get caught
5. Walk the plank
This is not a recommended occupation.
```

Следует отметить, что в извлеченный текст включены некоторые специальные символы, обозначающие элементы нумерованных и маркированных списков.

Конвейеры

Конвейер (pipeline) – это последовательность операций, где результат одной операции используется в качестве исходных данных для следующей. Мы уже видели работу конвейеров в нескольких примерах из предыдущих глав, но подробности функционирования конвейеров не рассматривались. В частности, мы убедились, что класс

StanfordCoreNLP (библиотека Stanford) обеспечивает полноценную поддержку концепции конвейеров, используя для этого объекты-аннотаторы. Более подробно мы обсудим этот подход в следующем разделе.

Одним из преимуществ правильно организованного конвейера является простота добавления и удаления его элементов, то есть изменения его конфигурации. Например, если одним из компонентов конвейера является операция преобразования символов в нижний регистр, можно просто удалить этот компонент без какого-либо ущерба для прочих компонентов.

Но некоторые типы конвейеров не обладают такой гибкостью. Иногда для правильного выполнения некоторого шага требуется обязательный подготовительный этап. Например, в одном из конвейеров, поддерживаемых классом StanfordCoreNLP, для выполнения разметки по частям речи необходим следующий набор аннотаторов:

```
props.put("annotators", "tokenize, ssplit, pos");
```

Если удалить аннотатор ssplit, при выполнении кода будет сгенерировано исключение:

```
java.lang.IllegalArgumentException: annotator "pos" requires annotator
"ssplit"
(java.lang.IllegalArgumentException: для аннотатора "pos" требуется ан-
нотатор "ssplit")
```

Формирование и настройка конвейера средствами библиотеки Stanford осуществляются без особых трудозатрат, но организация конвейеров другими средствами может оказаться более сложной. Эта проблема подробно рассматривается в разделе «Создание конвейера для поиска в тексте».

Использование конвейера, поддерживаемого библиотекой Stanford

В этом разделе подробно рассматривается конвейер, поддерживаемый библиотекой Stanford. Этот тип конвейера уже использовался в некоторых примерах данной книги, но при этом не уделялось внимания всем его функциональным возможностям. Тем не менее знание основ применения конвейера поможет лучше понять более эффективные методы его использования. Данный раздел позволит вам глубже изучить свойства конвейера, весьма полезные в практической деятельности.

Пакет `edu.stanford.nlp.pipeline` содержит основной класс `StanfordCoreNLP` поддержки конвейера и классы аннотаторов. Обобщенный подход предполагает использование кода, подобного приведенному ниже, где обрабатывается строка `text`. Экземпляр класса `Properties` содержит имена аннотаторов:

```
String text = "The robber took the cash and ran.";
Properties props = new Properties();
props.put("annotators",
    "tokenize, ssplit, pos, lemma, ner, parse, dcoref");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
```

Класс `Annotation` представляет обрабатываемый текст. Конструктор в следующем фрагменте принимает строку и добавляет экземпляр `CoreAnnotations.TextAnnotation` в объект `Annotation`. Метод `annotate()` класса `StanfordCoreNLP` применяет аннотаторы, перечисленные в списке свойств, к объекту `Annotation`:

```
Annotation annotation = new Annotation(text);
pipeline.annotate(annotation);
```

Интерфейс `CoreMap` – это базовый интерфейс для всех аннотируемых объектов. Он использует имена типов (классов) объектов в качестве значений ключей. Тип аннотации `TextAnnotation` является ключом для текста в ассоциативном массиве `CoreMap`. Ключи `CoreMap` предназначены для использования различных типов аннотаций, перечисляемых в списке свойств. Значение зависит от типа ключа.

На рис. 8.3 показана графическая схема иерархии классов и интерфейсов. Это упрощенный вариант схемы взаимосвязей между классами и интерфейсами, используемыми в рассматриваемом примере конвейера. Горизонтальные стрелки представляют реализации интерфейсов, а вертикальные обозначают наследование классов.

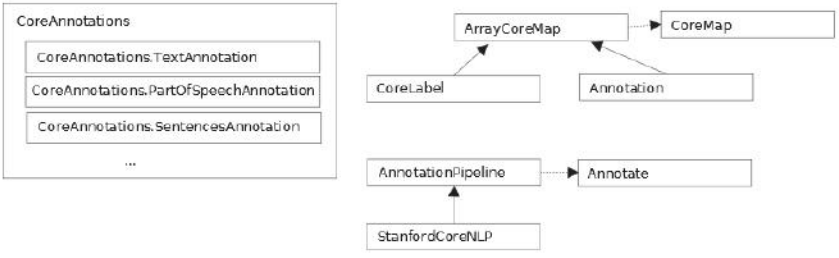


Рис. 8.3. Иерархия классов и интерфейсов, используемых в примере работы конвейера

Проверка результатов метода annotate() производится с помощью следующего фрагмента. Метод keyset() возвращает последовательность всех ключей аннотаций, хранящихся в текущий момент в объекте Annotation. Ключи выводятся до и после вызова метода annotate():

```
System.out.println("Before annotate method executed ");
Set<Class<?>> annotationSet = annotation.keySet();
for(Class c : annotationSet) {
    System.out.println("\tClass: " + c.getName());
}

pipeline.annotate(annotation);

System.out.println("After annotate method executed ");
annotationSet = annotation.keySet();
for(Class c : annotationSet) {
    System.out.println("\tClass: " + c.getName());
}
```

Из результата видно, что при создании объекта Annotation в него добавляется расширение TextAnnotation. После вызова метода annotate() добавляется еще несколько типов аннотаций:

```
Before annotate method executed
    Class: edu.stanford.nlp.ling.CoreAnnotations.TextAnnotation
After annotate method executed
    Class: edu.stanford.nlp.ling.CoreAnnotations.TextAnnotation
    Class: edu.stanford.nlp.ling.CoreAnnotations.TokensAnnotation
    Class: edu.stanford.nlp.ling.CoreAnnotations.SentencesAnnotation
    Class: edu.stanford.nlp.dcoref.CorefCoreAnnotations.
CorefChainAnnotation
```

Класс CoreLabel реализует интерфейс CoreMap. Он представляет одно слово и связанную с ним информацию об аннотации. Эта информация зависит от свойств, установленных при создании конвейера. При этом всегда остается доступной информация о местоположении элемента, то есть его начальная и конечная позиции или сведения о предшествующих и последующих пробелах.

Метод get() обоих классов – CoreMap и CoreLabel – возвращает информацию, связанную с переданным аргументом. Это перегруженный метод, и он возвращает значение, зависящее от типа аргумента. В качестве примера ниже приводится объявление класса SentencesAnnotation, реализующего интерфейс CoreAnnotation<List<>>:

```
public static class CoreAnnotations.SentencesAnnotation
    extends Object
    implements CoreAnnotation<List<CoreMap>>
```

При использовании в следующей инструкции класс SentencesAnnotation возвращает экземпляр списка List<CoreMap>:

```
List<CoreMap> sentences =
    annotation.get(SentencesAnnotation.class);
```

Аналогично класс TokenizerAnnotation реализует CoreAnnotation<List<CoreLabel>>, как показано ниже:

```
public static class CoreAnnotation.TokensAnnotation
    extends Object
    implements CoreAnnotation<List<CoreLabel>>
```

Метод get() этого класса возвращает список объектов CoreLabel, который используется в цикле for-each:

```
for(CoreLabel token : sentence.get(TokensAnnotation.class)) {
```

В предыдущих главах для доступа к предложениям в текущей аннотации использовался класс SentenceAnnotation:

```
List<CoreMap> sentences = annotation.get(SentencesAnnotation.class);
```

Класс CoreLabel применялся для доступа к отдельным словам в текущем предложении:

```
for(CoreMap sentence : sentences) {
    for(CoreLabel token : sentence.get(TokensAnnotation.class)) {
        String word = token.get(TextAnnotation.class);
        String pos = token.get(PartOfSpeechAnnotation.class);
    }
}
```

Возможные варианты аннотаторов можно найти на сайте библиотеки Stanford: http://nlp.stanford.edu/software/corenlp.shtml. В следующем примере показано использование аннотатора для определения модели разметки по частям речи. Свойство pos.model для выбранной модели устанавливается с помощью метода put() класса Property:

```
props.put("pos.model",
  "C:/.../Models/english-caseless-left3words-distsim.tagger");
```

Список возможных аннотаторов приведен в табл. 8.1. В первом столбце указана строка, применяемая в списке свойств. Второй столбец содержит имя только базового класса аннотации, третий – краткое описание.

Таблица 8.1. Возможные варианты аннотаторов для списка свойств

Имя свойства	Имя базового класса аннотации	Описание
tokenize	TokensAnnotation	Токенизация
cleanxml	XmlContextAnnotation	Удаление XML-токенов
ssplit	SentencesAnnotation	Разделение предложения на токены
pos	PartOfSpeechAnnotation	Создание тегов разметки по частям речи
lemma	LemmaAnnotation	Генерация лемм
ner	NamedEntityTagAnnotation	Создание тегов при распознавании именованных объектов
regexner	NamedEntityTagAnnotation	Создание тегов при распознавании именованных объектов с использованием регулярных выражений
sentiment	SentimentCoreAnnotations	Анализ тональности текста
truecase	TrueCaseAnnotation	Анализ правильности падежей
parse	TreeAnnotation	Генерация дерева синтаксического анализа
depparse	BasicDependenciesAnnotation	Анализатор синтаксических зависимостей
dcoref	CorefChainAnnotation	Определение кореференций
relation	MachineReadingAnnotations	Извлечение взаимосвязей и отношений

Рассмотрим следующий фрагмент кода, в котором создается конвейер:

```
String text = "The robber took the cash and ran.";
Properties props = new Properties();
props.put("annotators",
    "tokenize, ssplit, pos, lemma, ner, parse, dcoref");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
```

При выполнении аннотирования будет получен следующий результат. Здесь можно проследить все случаи применения каждого аннотатора:

```
Adding annotator tokenize
TokenizerAnnotator: No tokenizer type provided. Defaulting to
PTBTokenizer.
Adding annotator ssplit
edu.stanford.nlp.pipeline.AnnotatorImplementations:
Adding annotator pos
Reading POS tagger model from edu/stanford/nlp/models/pos-tagger/
english-left3words/english-left3words-distsim.tagger ... done [2.5 sec].
Adding annotator lemma
```

```
Adding annotator ner
Loading classifier from edu/stanford/nlp/models/ner/english.all.3class.
distsim.crf.ser.gz ... done [6.7 sec].
Loading classifier from edu/stanford/nlp/models/ner/english.muc.7class.
distsim.crf.ser.gz ... done [5.0 sec].
Loading classifier from edu/stanford/nlp/models/ner/english.conll.4class.
distsim.crf.ser.gz ... done [5.5 sec].
Adding annotator parse
Loading parser from serialized file edu/stanford/nlp/models/lexparser/
englishPCFG.ser.gz ... done [0.9 sec].
Adding annotator dcoref
```

При вызове метода `annotate()` можно воспользоваться методом `timingInformation()` для измерения времени выполнения каждого этапа конвейерного процесса:

```
System.out.println("Total time: " + pipeline.timingInformation());
```

Для конвейера, определенного выше, мы получили следующий результат:

```
Total time: Annotation pipeline timing information:
TokenizerAnnotator: 0.0 sec.
WordsToSentencesAnnotator: 0.0 sec.
POSTaggerAnnotator: 0.0 sec.
MorphaAnnotator: 0.1 sec.
NERCombinerAnnotator: 0.0 sec.
ParserAnnotator: 2.5 sec.
DeterministicCoreAnnotator: 0.1 sec.
TOTAL: 2.8 sec. for 8 tokens at 2.9 tokens/sec.
```

Использование нескольких ядер процессора для конвейера библиотеки Stanford

У метода `annotate()` имеется еще одно важное достоинство: возможность распараллеливания работы на нескольких ядрах процессора. Это перегруженный метод, и каждая его версия принимает экземпляр `Iterable<Annotation>`, что позволяет обрабатывать каждый экземпляр `Annotation`, включая в работу все доступные процессоры или ядра.

Для демонстрации описанной версии метода `annotate()` будет использоваться объект `pipeline`, определенный в предыдущем разделе.

Сначала создаются четыре объекта `Annotation` на основе четырех коротких предложений, как показано ниже. Но в полной мере преимущества методики использования многоядерности процессора проявляются при обработке больших объемов реальных данных.

```
Annotation annotation1 =
    new Annotation("The robber took the cash and ran.");
```

```
Annotation annotation2 =
    new Annotation("The policeman chased him down the street.");
Annotation annotation3 = new Annotation(
    "A passerby, watching the action, tripped the thief "
    + "as he passed by.");
Annotation annotation4 = new Annotation(
    "They all lived happily ever after, except for the thief "
    + "of course.");
```

Класс `ArrayList` реализует интерфейс `Iterable`. Создадим экземпляр этого класса, затем добавим в него четыре объекта `Annotation`. После этого созданный список присвоим переменной типа `Iterable`:

```
ArrayList<Annotation> list = new ArrayList();
list.add(annotation1);
list.add(annotation2);
list.add(annotation3);
list.add(annotation4);
Iterable<Annotation> iterable = list;
```

Выполним метод `annotate()`:

```
pipeline.annotate(iterable);
```

Теперь воспользуемся экземпляром `annotation2` для демонстрации результата – вывода списка слов из предложения и соответствующих им тегов частей речи:

```
List<CoreMap> sentences = annotation2.get(SentencesAnnotation.class);
for(CoreMap sentence : sentences) {
    for(CoreLabel token : sentence.get(TokenAnnotation.class)) {
        String word = token.get(TextAnnotation.class);
        String pos = token.get(PartOfSpeechAnnotation.class);
        System.out.println("Word: " + word + " POS Tag: " + pos);
    }
}
```

Результат приводится ниже:

```
Word: The POS Tag: DT
Word: policeman POS Tag: NN
Word: chased POS Tag: VBD
Word: him POS Tag: PRP
Word: down POS Tag: RP
Word: the POS Tag: DT
Word: street POS Tag: NN
Word: . POS Tag: .
```

Пример выше показывает, насколько просто организовать параллельное выполнение при использовании конвейера, поддерживаемого библиотекой Stanford.

Создание конвейера для текстового поиска

Текстовый поиск – это отдельная большая и сложная тема. Существует множество разнообразных видов поиска и большое количество разных методик поиска в тексте. Основной задачей данного раздела является демонстрация применения методик обработки естественного языка для решения этой задачи.

На большинстве компьютеров обработка одного текстового документа может быть выполнена за один проход и займет относительно короткое время. Но гораздо чаще необходимо выполнять поиск во множестве документов с большим объемом текста, и тогда следует применять обобщенную методику – создание индекса документов. Этот подход позволяет завершать операцию поиска за приемлемое время.

Здесь будет продемонстрирована одна из методик создания индекса (указателя) и выполнения процедуры поиска с его использованием. Текст, с которым мы будем работать, имеет небольшой объем, но его вполне достаточно, чтобы показать все нюансы процесса.

Нам потребуется:

1) прочитать текст из файла;
2) выполнить токенизацию и определить границы предложений;
3) удалить шумовые слова (стоп-слова);
4) собрать статистические данные для индекса;
5) записать сформированный индекс в файл.

На содержимое индексного файла могут влиять следующие факторы:

- устранение шумовых слов;
- необходимость в поиске с учетом регистра символов;
- определение синонимов;
- использование стемминга и лемматизации;
- допустимость поиска с пересечением границ предложений.

Для демонстрации процесса используется библиотека OpenNLP. Основная цель примера – показать возможность объединения нескольких методик обработки естественного языка в конвейер для решения задачи определенного типа, в нашем случае для выполнения поиска в тексте. Предлагаемое решение не является полным и не предназначено для практического применения, так как в нем не предусмотрено использование некоторых методик, например стемминга. Кроме того, в примере не представлено действительное созда-

ние файла индекса, но читатель может самостоятельно реализовать эту процедуру в качестве упражнения. Здесь мы полностью сосредоточимся на совместном применении нескольких методик обработки естественного языка в конвейере.

В рассматриваемом примере будут выполнены следующие операции:

- разделение текста на предложения;
- преобразование символов предложений в нижний регистр;
- удаление шумовых слов;
- создание внутренней структуры данных для формирования индекса.

Будут определены два класса поддержки структуры индекса: `Word` и `Positions`. Кроме того, в класс `StopWords`, разработанный в главе 2 «Поиск фрагментов текста», будет добавлена перегруженная версия метода `removeStopWords()`, чтобы процедура удаления шумовых слов в большей степени соответствовала текущей задаче.

Разумеется, в данном случае необходим блок попытки захвата ресурсов `try`, чтобы успешно создать потоки ввода для чтения файла модели предложения *en-sent.bin* и файла с текстом из романа Жюля Верна «20 000 лье под водой» в английском переводе (Jules Verne, «Twenty Thousands Leagues Under the Sea»). Книгу можно скачать на сайте проекта Гутенберг (http://www.gutenberg.org/ebooks/164), ее текст немного отредактирован – удалены колонтитулы проекта Гутенберг для упрощения обработки:

```
try(InputStream is = new FileInputStream(
    new File("C:/Current Books/NLP and Java/Models/en-sent.bin"));
    FileReader fr = new FileReader("Twenty Thousands.txt");
    BufferedReader br = new BufferedReader(fr)) {
        ...
} catch(IOException ex) {
    // Обработка исключения.
}
```

Для создания экземпляра класса `SentenceDetectorME` необходима модель предложения:

```
SentenceModel model = new SentenceModel(is);
SentenceDetectorME detector = new SentenceDetectorME(model);
```

Далее с помощью экземпляра `StringBuilder` создается строка для поддержки определения границ предложений. Файл с текстом книги читается и добавляется в экземпляр `StringBuilder`. Затем вызывается метод `sentDetect()` для создания массива предложений:

```
String line;
StringBuilder sb = new StringBuilder();
while((line = br.readLine()) != null) {
    sb.append(line + " ");
}
String sentences[] = detector.sentDetect(sb.toString());
```

Для отредактированной версии текста книги из обрабатываемого файла этот метод создал массив из 14 859 предложений.

Далее используется метод `toLowerCase()` для преобразования символов текста в нижний регистр. Это делается, чтобы обеспечить удаление всех шумовых слов.

```
for(int i=0; i < sentences.length; i++) {
    sentences[i] = sentences[i].toLowerCase();
}
```

Преобразование символов в нижний регистр и удаление шумовых слов сокращают область поиска. Но это следует считать особенностью только данной реализации, так как в других реализациях условия могут быть совершенно другими.

На следующем шаге удаляются шумовые слова. Ранее уже отмечалось, что в данном примере добавлена перегруженная версия метода `removeStopWords()`, чтобы упростить его применение для нашего случая. Новая версия метода приведена ниже:

```
public String removeStopWords(String words) {
    String arr[] = WhitespaceTokenizer.INSTANCE.tokenize(words);
    StringBuilder sb = new StringBuilder();
    for(int i=0; i < arr.length; i++) {
        if(stopWords.contains(arr[i])) {
            // Ничего не делать.
        } else {
            sb.append(arr[i] + " ");
        }
    }
    return sb.toString();
}
```

Экземпляр класса `StopWords` создается с использованием файла *stop-words_english_2_en.txt*, как показано в следующем фрагменте. Указанный файл предоставляет один из нескольких списков шумовых слов, которые можно скачать с сайта https://code.google.com/p/stop-words/. Именно этот файл выбран потому, что содержащийся в нем набор шумовых слов лучше всего подходит для обработки текста книги Ж. Верна.

```
StopWords stopWords = new StopWords("stop-words_english_2_en.txt");
for(int i=0; i < sentences.length; i++) {
    sentences[i] = stopWords.removeStopWords(sentences[i]);
}
```

Предварительная обработка текста завершена. Следующий шаг – создание структуры данных для формирования индекса на основе подготовленного текста. В этой структуре используются классы `Word` и `Positions`. Класс `Word` состоит из полей для самого слова и для списка `ArrayList` с объектами типа `Positions`. Поскольку слово может встречаться в документе неоднократно, в списке хранятся все его позиции. Основной класс определяется, как показано ниже:

```
public class Word {
    private String word;
    private final ArrayList<Positions> positions;

    public Word() {
        this.positions = new ArrayList();
    }

    public void addWord(String word, int sentence, int position) {
        this.word = word;
        Positions counts = new Positions(sentence, position);
        position.add(counts);
    }

    public ArrayList<Positions> getPositions() {
        return positions;
    }

    public String getWord() {
        return word;
    }
}
```

Класс `Positions` содержит поле `sentence` номера предложения и поле `position` позиции слова в предложении. Определение класса приведено ниже:

```
class Positions {
    int sentence;
    int position;

    Positions(int sentence, int position) {
        this.sentence = sentence;
        this.position = position;
    }
}
```

```
reef is found at line 1885, word 8
reef is found at line 2062, word 12
```

Предложенная выше реализация относительно проста, но позволяет наглядно показать, как объединить разные методики обработки естественного языка для создания и использования структуры данных индекса текста, который может быть сохранен в специальном файле. Функциональные возможности данной реализации можно расширить, добавив:

- дополнительные операции фильтрации текста;
- сохранение информации о документе в классе `Positions`;
- сохранение информации о главе в классе `Positions`;
- дополнительные функции поиска:
  - поиск с учетом регистра символов;
  - поиск текста, точно совпадающего с заданным образцом;
- улучшенную обработку исключений.

Все эти задачи предлагается решить читателю в качестве дополнительных упражнений.

Резюме

В этой главе мы обсудили процесс подготовки данных и подробно рассмотрели организацию конвейеров. Были продемонстрированы некоторые методики извлечения текста из документов в форматах HTML, Word и PDF.

Мы выяснили, что конвейер представляет собой последовательность простых задач, объединенных для решения некоторой более сложной задачи. При необходимости можно добавлять элементы в конвейер и удалять их. Архитектура конвейера из библиотеки Stanford была рассмотрена во всех подробностях, описаны все возможные аннотаторы. На практическом примере показана возможность использования конвейером преимуществ многоядерных процессоров или многопроцессорных систем.

Далее было показано формирование конвейера, создающего и использующего индекс для текстового поиска, на основе средств библиотеки OpenNLP. Этот способ предоставляет большую гибкость при организации конвейера, в отличие от предопределенного механизма конвейера из библиотеки Stanford.

Автор надеется, что эта книга стала полезным введением в технологию обработки естественного языка с применением языка программирования Java. Мы рассмотрели все важные базовые задачи

обработки естественного языка и разобрали на примерах различные методики решения этих задач с использованием различных библиотек NLP API. Обработка естественного языка – это большая и сложная область информатики, и автору остается лишь пожелать читателям успехов в разработке приложений для этой области.

Предметный указатель

N

O

P

R

S

T

U

W

А

Б

В

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ч

Ш

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Ричард Риз

Обработка естественного языка на Java

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Киселев А. Н.*
Перевод *Снастин А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 16,5. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф