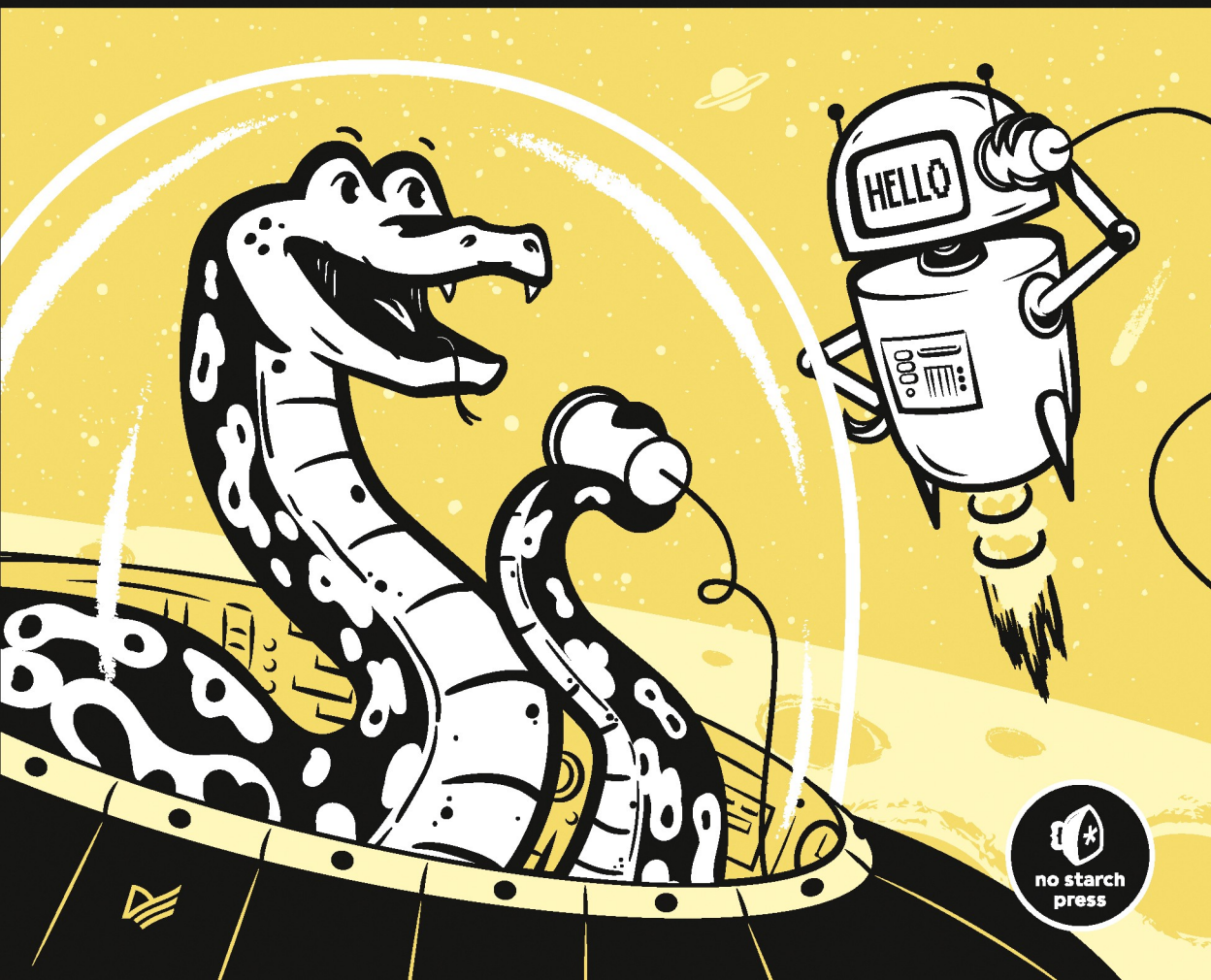
ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА PYTHON И SPACY

НА ПРАКТИКЕ

ЮЛИЙ ВАСИЛЬЕВ



NATURAL LANGUAGE PROCESSING WITH PYTHON AND SPACY

A Practical Introduction

by Yuli Vasiliev

no starch
press

San Francisco

ЮЛИЙ ВАСИЛЬЕВ

ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА PYTHON И SPACY

НА ПРАКТИКЕ

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

ББК 81.111
УДК 81.322.2+004.43
В19

Васильев Юлий

В19 Обработка естественного языка. Python и spaCy на практике. — СПб.: Питер, 2021. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1506-8

Python и spaCy помогут вам быстро и легко создавать NLP-приложения: чат-боты, сценарии для сокращения текста или инструменты принятия заказов. Вы научитесь использовать spaCy для интеллектуального анализа текста, определять синтаксические связи между словами, идентифицировать части речи, а также определять категории для имен собственных. Ваши приложения даже смогут поддерживать беседу, создавая собственные вопросы на основе разговора.

Прочитав эту книгу, вы можете сами расширить приведенные в книге сценарии, чтобы обрабатывать разнообразные варианты ввода и создавать приложения профессионального уровня.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 81.111
УДК 81.322.2+004.43

ISBN 978-1718500525 англ.

ISBN 978-5-4461-1506-8

Краткое содержание

Оглавление

Об авторе

Юлий Васильев — программист, писатель и консультант, специализирующийся в сфере разработки приложений с открытым исходным кодом, технологий баз данных Oracle и обработки текстов на естественном языке (NLP). В настоящее время работает консультантом проекта чат-бота Porphyry, способного за счет реализации технологий NLP выдавать осмысленные ответы на вопросы пользователей. Больше узнать о боте можно в Telegram @Porphyry_bot.

О научном редакторе

Иван Бригида родился и вырос в Краснодаре. Получил высшее образование в области компьютерных наук в Московском государственном университете и диплом магистра экономики в Российской экономической школе (РЭШ). Несколько лет трудился финансовым аналитиком. Позднее перешел на работу в Google, став аналитиком в сфере интернет-рекламы. В настоящее время занимается BI-аналитикой и разрабатывает модели машинного обучения для Online Partnerships Group в компании Google, специализируясь на монетизации мобильных приложений.

Введение

Все чаще, позвонив в банк или поставщику услуг Интернета, мы можем услышать нечто вроде: «Здравствуйте, я ваш виртуальный помощник. Слушаю вас!» С каждым днем роботы становятся умнее и уже могут общаться с людьми на естественном языке. Но как работают такие технологии и как их использовать в своих проектах?

Обработка естественного языка (NLP) — одно из направлений искусственного интеллекта (ИИ), ставящее перед собой цель научить машины понимать человеческие языки и реагировать на высказывания. Без этой базовой технологии невозможен ни один виртуальный помощник. Книга поможет вам обрести навыки, необходимые для создания собственного чат-бота и NLP-приложений, способных решать реальные задачи, такие как анализ предложений, улавливание смысла текста, составление текстов и т. д.

Применение языка Python для обработки естественного языка

Существует широкий спектр утилит и технологий, которые можно использовать при создании NLP-приложения. Все примеры в книге реализованы на языке Python с использованием библиотеки spaCy NLP. Приведу несколько убедительных аргументов в пользу выбора Python и spaCy для создания приложений обработки естественного языка.

Python — высокоуровневый язык программирования со следующими особенностями.

- *Простота*. Если вы новичок в программировании, Python — отличный язык для начала знакомства с этим увлекательным миром. Python исключительно прост в изучении, на нем можно

писать понятный код. Простота Python помогает разработчикам чат-ботов сотрудничать с лингвистами, не имеющими опыта программирования.

- *Широкая распространенность*. Python — один из самых популярных языков. У абсолютного большинства широко используемых API есть обертки для языка Python, легко устанавливаемые с помощью утилиты pip. Возможность установки оберток Python посредством pip упрощает процесс получения сторонних утилит, которые могут понадобиться в NLP-приложениях.
- *Присутствие в экосистеме ИИ в значительных объемах*. В экосистеме ИИ существует множество библиотек Python, что позволяет выбрать наиболее подходящую из них для решения конкретной задачи. Это существенно упрощает разработку NLP-приложений.

Библиотека spaCy

В книге используется spaCy — популярная библиотека Python, содержащая все лингвистические данные и алгоритмы, необходимые для обработки текстов на естественном языке. По ходу чтения вы убедитесь, что spaCy крайне проста в работе благодаря объектам-контейнерам, которые соответствуют элементам текста на естественном языке, например предложениям и словам. У этих объектов, в свою очередь, есть атрибуты, соответствующие лингвистическим признакам, — например, принадлежность к той или иной части речи. На момент написания книги в spaCy были включены предобученные модели для английского, немецкого, греческого, испанского, французского, итальянского, литовского, норвежского стандарта букмол[1], нидерландского и португальского языков, а также многоязыковая модель[2]. Кроме того, в spaCy есть встроенные средства визуализации, позволяющие генерировать

[1] Один из двух основных стандартов (форм) норвежского языка. — *Здесь и далее примеч. пер.*

[2] На момент выпуска русского издания книги к ним добавились модели для китайского, японского, польского и румынского языков. На GitHub можно найти неофициальные модели и для русского языка (например, по адресам https://github.com/buriy/spacy-ru и https://github.com/aatimofeev/spacy_russian_tokenizer).

наглядное представление о синтаксической структуре предложений или об именованных сущностях документа.

Библиотека spaCy предлагает нативную поддержку продвинутых возможностей NLP, отсутствующую в других популярных библиотеках NLP для языка Python. Например, spaCy, в отличие от пакета Natural Language Toolkit (NLTK), может похвастаться нативной поддержкой векторов слов (подробнее о них см. в главе 5). При использовании NLTK пришлось бы обратиться к сторонней утилите наподобие Gensim — реализации алгоритма word2vec для языка Python.

При работе со spaCy можно настроить под себя уже существующие модели или отдельные компоненты моделей, обучить собственные модели с нуля в соответствии с потребностями своих приложений (как это сделать, описано в главе 10), а также подключить статистические модели, обученные с помощью других популярных библиотек машинного обучения: TensorFlow, Keras, scikit-learn и PyTorch. Кроме того, spaCy без проблем может взаимодействовать с другими библиотеками экосистемы ИИ языка Python, позволяя, например, использовать для чат-бота возможности машинного зрения (более подробно читайте в главе 12).

Для кого предназначена книга

Эта книга для всех, кто интересуется практическим применением NLP. В частности, она может заинтересовать желающих разрабатывать чат-боты как для коммерческого применения, так и просто для развлечения. Вне зависимости от вашего образования, от опыта работы с NLP и в программировании в целом у вас не должно возникнуть проблем с разбором примеров кода, приведенных в книге, поскольку все они сопровождаются подробными пояснениями.

Однако хотя бы небольшой опыт работы с языком Python не помешает, так как основы синтаксиса Python в этой книге описываться не будут. Кроме того, примеры предполагают знание грамматики и синтаксиса английского языка как минимум на уровне средней школы. Сведения о некоторых менее известных лингвистических понятиях можно найти в приложении. Следить за кодом примеров будет проще при хорошем понимании основных концепций NLP и программирования.

Что вы найдете в издании

«Обработка естественного языка. Python и spaCy на практике» начинается с краткого введения, которое знакомит с основными элементами и методами технологий NLP, предназначенными для обработки и анализа данных на естественных языках. Далее рассматриваются более сложные методики, использующиеся для решения комплексных задач по компьютерной обработке и анализу, с которыми приходится сталкиваться исследователям естественного языка. Для закрепления пройденного материала почти в каждой главе есть раздел «Попробуйте сами».

Краткое содержание глав.

Глава 1 «Как происходит обработка текстов на естественном языке». Введение в основные элементы технологий NLP. Описание методики машинного обучения для генерации данных, используемых библиотеками NLP (например, spaCy), включая статистическое моделирование языка и статистические сетевые модели, предназначенные для решения задач NLP. Задачи и проблемы, с которыми сталкиваются разработчики приложений NLP.

Глава 2 «Конвейер обработки текста». Что такое библиотека spaCy, для чего она предназначена и как быстро начать с ней работать. Настройка рабочей среды и написание кода с помощью конвейера обработки текста — набора базовых операций NLP, применяемых для определения смысла и подтекста дискурса.

Глава 3 «Работа с объектами-контейнерами и настройка spaCy под свои нужды». В этой главе речь идет об архитектуре spaCy, причем основное внимание уделено центральным структурам данных библиотеки. Следуя приведенным примерам, вы получите реальный опыт работы с ключевыми объектами spaCy. Кроме того, научитесь настраивать компоненты конвейера под нужды своего приложения.

Глава 4 «Выделение и использование лингвистических признаков». Демонстрируется возможность выделения лингвистических признаков: меток зависимости, тегов частей речи и именованных сущностей. Вы

научитесь генерировать дерево зависимостей предложения и обходить его, исследуя синтаксические отношения, чтобы иметь возможность поддерживать разговор с пользователем чат-бота программными средствами, сокращать длинные тексты и выполнять другие полезные задачи.

Глава 5 «Работа с векторами слов». В этой главе объясняется, как модели spaCy ставят в соответствие словам естественного языка векторы вещественных чисел, позволяя выполнять над словами математические действия. Вы научитесь использовать метод `similarity` объектов spaCy, предназначенный для сравнения векторов слов объектов-контейнеров с целью выяснения близости их смыслов.

Глава 6 «Поиск паттернов и обход деревьев зависимостей». Эта глава углубляется в вопросы извлечения смысла, синтаксического разбора зависимостей, разбивки существительных на части и распознавания сущностей. Вы пройдете все этапы выделения смысла исходного текста, используя шаблоны последовательностей слов и обхода деревьев зависимостей. Кроме того, познакомитесь с утилитой Matcher библиотеки spaCy, предназначенной для поиска паттернов, а также узнаете, в каких случаях для определения нужного подхода к обработке может понадобиться анализ контекста.

Глава 7 «Визуализация». Здесь рассматриваются вопросы использования встроенного средства визуализации displaCy библиотеки spaCy, с помощью которого можно наглядно отображать синтаксические зависимости и именованные сущности в браузере. Визуализация дает возможность сразу же выявить паттерны, содержащиеся в данных.

Глава 8 «Распознавание намерений». Демонстрация распознавания намерений — распространенной задачи при разработке чат-ботов: вы научитесь выделять смысл из исходного текста. Для решения этой непростой в большинстве случаев задачи достаточно всего нескольких строк кода на языке Python.

Глава 9 «Сохранение данных, введенных пользователем, в базе данных». Вы научитесь автоматически выделять ключевые слова в данных, введенных пользователем, и сохранять их в реляционной базе данных,

чтобы в дальнейшем использовать для заполнения форм заказов или других коммерческих документов.

Глава 10 «Обучение моделей». Вы узнаете, как для удовлетворения потребностей тех приложений, которые по умолчанию не охватываются моделями библиотеки spaCy, обучить средство распознавания именованных сущностей библиотеки spaCy и средство анализа зависимостей. Вам предлагается подробное описание процесса обучения уже существующей (предобученной) модели на новых примерах данных и «чистой» модели с нуля.

Глава 11 «Развертывание собственного чат-бота». В этой главе вам предлагается подробное руководство по процессу развертывания чат-бота в популярном сервисе мгновенных сообщений Telegram для взаимодействия с другими пользователями.

Глава 12 «Реализация веб-данных и обработка изображений». Демонстрация того, как с помощью spaCy, а также других библиотек из экосистемы ИИ языка Python чат-бот может извлекать из «Википедии» ответы на вопросы и реагировать на отправляемые пользователем изображения.

Приложение «Начальное руководство по лингвистике». Глава содержит краткое руководство по грамматическим и синтаксическим элементам, наиболее часто обсуждаемым в книге. Читатели, не имеющие лингвистического образования, могут использовать его в качестве справочника.

Скачивание примеров кода

Файлы с примерами кода из этой книги вы можете скачать на сайте GitHub по адресу https://github.com/nlptechbook/examples.

От издательства

Ваши замечания, предложения, вопросы отправляйте по электронному адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Поскольку в книге рассматриваются примеры на английском языке, мы не стали переводить теги частей речи. Список основных тегов выглядит следующим образом:

ADJ — прилагательное;
ADP — предлог;
ADV — наречие;
AUX — вспомогательный глагол;
CONJ — союз;
DET — определитель;
INTJ — междометие;
NOUN — существительное;
NUM — числительное;
PART — частица;
PRON — местоимение;
PROPN — собственное существительное;
PUNCT — знак препинания;
VERB — глагол.

1 Как происходит обработка текстов на естественном языке

В XIX веке на острове Рапа Нуи (более известном как остров Пасхи) исследователи обнаружили деревянные дощечки с письменами *ронго-ронго* — системой загадочных иероглифов. Ученым пока не удалось их расшифровать и определить, что это — письмо или протописьменность (пиктографические символы, несущие информацию, но не связанные с языком). Известно, что создатели надписей воздвигли моаи — гигантские каменные человекоподобные фигуры, которыми знаменит остров Пасхи: однако предназначение моаи до сих пор неясно, о нем можно лишь догадываться.

Если письменность, то есть способ описания людьми различных вещей, непонятна, то, скорее всего, непонятными останутся и прочие аспекты их жизни, в том числе то, что они делали и почему.

Обработка текстов на естественном языке (Natural Language Processing, NLP) — направление искусственного интеллекта, нацеленное на обработку и анализ данных на естественном языке и обучение машин взаимодействию с людьми на естественном языке (языке, сформировавшемся естественным путем на протяжении истории). Через создание алгоритмов машинного обучения, предназначенных для работы с более объемными в сравнении с двумя десятками табличек, найденных на Рапа Нуи, наборами неизвестных данных, исследователи изучают, как люди используют язык. Таким образом появляется возможность добиться гораздо большего, чем просто расшифровка древних надписей.

Сегодня с помощью алгоритмов можно вести научные наблюдения за языками, семантика и грамматические правила которых хорошо известны (в отличие от надписей на ронго-ронго), создавая затем приложения для программного «понимания» высказываний на этих языках. С помощью таких приложений коммерческие компании могут освободить людей от утомительных и однообразных задач. Например, приложение может принимать заказы на доставку еды или отвечать на постоянно повторяющиеся вопросы пользователей в рамках технической поддержки.

Неудивительно, что генерация и понимание естественного языка — наиболее многообещающие, но в то же время сложные задачи NLP. В этой книге для создания обработчика текстов на естественном языке используется язык программирования Python. Задача решается с помощью библиотеки spaCy — ведущей библиотеки Python для обработки естественного языка с открытым исходным кодом. Но для начала вы узнаете, что происходит «за кулисами» в процессе создания средства обработки естественного языка.

Как компьютеры понимают естественный язык

Как можно научить компьютеры — бесчувственные машины — понимать человеческий язык и должным образом реагировать на высказывания? Конечно, машины не способны понимать естественный язык так, как человек. Чтобы компьютер мог производить вычислительные операции над языковыми данными, необходима система преобразования слов естественного языка в числовую форму.

Задание соответствий слов и чисел с помощью вложений слов

Вложение слов (word embedding) — методика задания соответствий слов числам. Словам при вложении слов в соответствие ставятся векторы вещественных чисел, а значение слова распределяется по координатам соответствующего вектора слов. Слова со схожим значением близки в векторном пространстве, поэтому можно определять значение слова по его соседям.

Вот фрагмент подобного отображения:

```
the 0.0897 0.0160 -0.0571 0.0405 -0.0696  ...
and -0.0314 0.0149 -0.0205 0.0557 0.0205  ...
of -0.0063 -0.0253 -0.0338 0.0178 -0.0966 ...
to 0.0495 0.0411 0.0041 0.0309 -0.0044    ...
in -0.0234 -0.0268 -0.0838 0.0386 -0.0321 ...
```

В этом фрагменте словам *the*, *and*, *of*, *to* и *in* соответствуют следующие за ними координаты. Если отобразить эти координаты визуально, то близкие по значению слова окажутся соседями на графике. (Но это не значит, что близкие по значению слова будут сгруппированы и в текстовом представлении так же, как в данном фрагменте. Текстовое представление пространства векторов слов обычно начинается с наиболее распространенных слов, например с *the*, *and* и т. д. Именно таким образом генераторы пространства векторов слов располагают слова.)

ПРИМЕЧАНИЕ

Визуальное представление многомерного векторного пространства можно реализовать в виде двухмерной или трехмерной проекции. Для ее создания используются соответственно первые две или три главные компоненты (координаты) вектора. К этому вопросу мы вернемся в главе 5.

При наличии матрицы соответствий слов числовым векторам над этими векторами можно производить арифметические действия. Среди прочего, можно определять *семантическое подобие* (синонимию) слов, предложений и даже целых документов, а затем использовать эту информацию, чтобы выяснить, например, какой теме посвящен текст.

Математически определение семантического подобия между двумя словами сводится к вычислению косинусного коэффициента между соответствующими векторами, то есть вычислению косинуса угла между ними. И хотя подробное описание вычисления семантического подобия выходит за рамки данной книги, некоторые детали работы с векторами слов описаны в главе 5.

Применение машинного обучения для обработки естественного языка

Вычислить числовые компоненты векторов можно с помощью алгоритмов машинного обучения. *Машинное обучение* (machine learning) — подобласть искусственного интеллекта по созданию компьютерных систем, способных автоматически обучаться на основе передаваемых им данных без необходимости программировать те явным образом. Алгоритмы машинного обучения позволяют предсказывать характеристики новых, не встречавшихся им ранее данных, распознавать изображения и речь, классифицировать фотографии и текстовые документы, автоматизировать управление. Также помогают они и при разработке игр.

Благодаря машинному обучению компьютеры могут решать задачи, которые ранее были им не под силу. Представьте, сколько условных операторов `if...else` пришлось бы написать, чтобы научить машину играть в шахматы, используя традиционный подход — с явным указанием на то, что алгоритм должен делать в той или иной ситуации. Пользователи подобного приложения быстро обнаружили бы слабые места в вашей логике и смогли бы выигрывать проще — пока вы не исправили бы код.

Напротив, приложения, в основе которых лежат алгоритмы машинного обучения, не зависят от заранее описываемой логики, а учатся на своих ошибках. Таким образом, основанная на машинном обучении шахматная программа ищет позиции, встречавшиеся ей в предыдущих партиях, и делает ходы, ведущие к оптимальной позиции. Накопленный опыт она хранит в статистической модели, которую мы рассмотрим в разделе «Что такое статистическая модель в NLP» на с. 31.

Помимо генерации векторов слов, машинное обучение в библиотеке spaCy позволяет выполнить три задачи: *разбор синтаксических зависимостей* (определение взаимосвязи между словами в предложении), *частеречную разметку* (выявление существительных, глаголов и иных частей речи) и *распознавание именованных сущностей* (разбиение имен собственных по категориям — люди, организации, местоположения и т. д.). В следующих главах мы поговорим об этом более подробно.

Жизненный цикл обычной системы машинного обучения включает три этапа: обучение модели, контроль и выполнение предсказаний.

Обучение модели

На первом этапе модель обучается путем подачи на вход алгоритма большого массива данных. Чтобы получить результат, заслуживающий доверия, необходимо обеспечить значительный объем входных данных — намного больший, чем количество надписей на табличках ронго-ронго, например. Что касается NLP, такие платформы, как «Википедия» и Google News, содержат достаточно текста для практически любого алгоритма машинного обучения. Но модель, специально предназначенная для вашего конкретного сценария использования, должна обучаться в том числе на действиях пользователей вашего сайта.

На рис. 1.1 приведена общая картина этапа обучения модели.

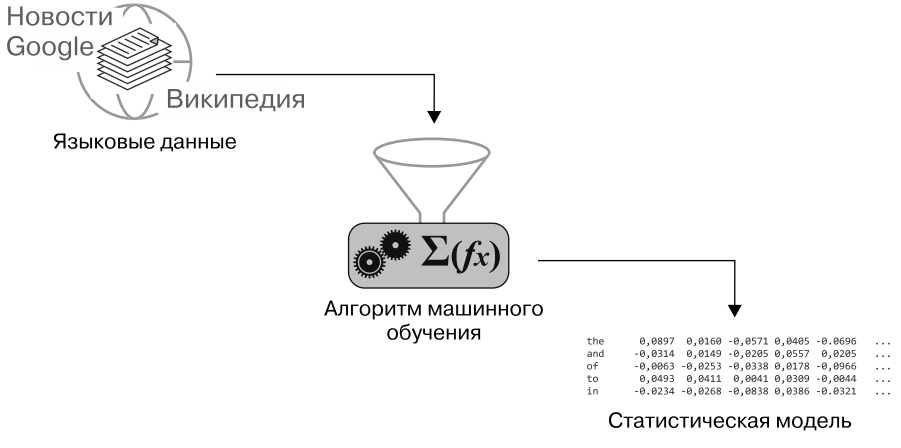


Рис. 1.1. Генерация статистической модели с помощью алгоритма машинного обучения, на основе большого массива текста в качестве входных данных

Чтобы найти слова со схожими характеристиками, модель обрабатывает огромные массивы текстовых данных, а затем создает для этих слов векторы, отражающие общие характеристики.

Из раздела «Что такое статистическая модель в NLP» на с. 31 вы узнаете, что **подобное** пространство векторов слов не единственный компонент статистической модели, предназначенной для NLP. На самом деле ее структура сложнее, что позволяет выделять лингвистические признаки для каждого из слов в зависимости от контекста.

В главе 10 вы узнаете, как обучить и уже существующую (предобученную) модель на новых примерах, и «чистую» модель с самого начала.

Контроль

При желании после обучения модели можно проверить, насколько хорошо она работает. Для контроля работы модели необходимо подать ей на вход текст, который она пока еще не «видела», и проверить, сможет ли она правильно идентифицировать семантические подобия и другие признаки, усвоенные во время обучения.

Выполнение предсказаний

Если все работает как часы, в приложении NLP можно выполнять предсказания на основе обученной модели. Например, предсказать с ее помощью структуру дерева зависимостей для вводимого текста, как показано на рис. 1.2. *Структура дерева зависимостей* отражает взаимосвязи между словами в предложении.

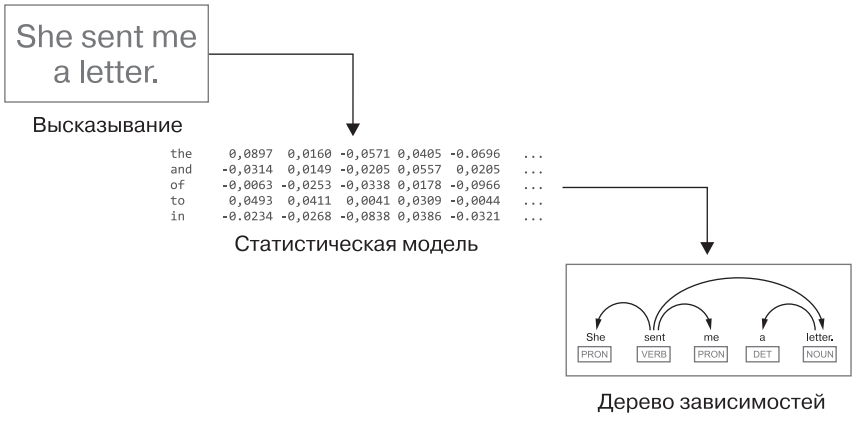


Рис. 1.2. Предсказание структуры дерева зависимостей для высказывания с помощью статистической модели

Наглядно дерево зависимостей можно представить с помощью дуг различной длины, соединяющих синтаксически связанные пары слов. Из приведенного здесь дерева зависимостей видно, что глагол *sent* согласуется с местоимением *she*.

Зачем использовать машинное обучение для обработки естественного языка

Выполняемые алгоритмом предсказания не обычная констатация фактов: предсказания вычисляются с некоторой долей вероятности. Для достижения точности приходится реализовывать все более хитрые алгоритмы, не такие эффективные и менее осуществимые на практике. Обычно стараются достичь разумного баланса безошибочности и быстродействия.

Поскольку идеальное предсказание с помощью моделей машинного обучения недостижимо, может возникнуть вопрос: является ли машинное обучение наилучшим подходом к созданию используемых в приложениях NLP моделей. Другими словами, не существует ли более надежного подхода, в основе которого лежали бы четко заданные правила, аналогичные используемым в компиляторах и интерпретаторах для обработки кода на различных языках программирования? Если коротко: нет. Поясню почему.

Прежде всего, количество слов в языке программирования относительно невелико. Например, в языке Java всего 61 зарезервированное слово, причем смысл каждого из них определен заранее.

Напротив, Оксфордский словарь английского языка, выпущенный в 1989 году, содержит 171 476 словарных статей для используемых в наши дни слов. В 2010 году команда исследователей из Гарвардского университета и Google насчитала около 1 022 000 слов в корпусе оцифрованных текстов, содержащем лишь 4 % когда-либо опубликованных книг. Согласно этому исследованию, словарный состав английского языка ежегодно увеличивается на несколько тысяч слов. Чтобы поставить в соответствие каждому слову число, потребовалось бы очень много времени.

Но даже если попытаться это сделать, существует несколько причин, по которым невозможно определить количество слов естественного языка. Во-первых, непонятно, что именно считать отдельным словом. Например, слово *count* следует считать одним словом, двумя или вообще тремя? В одном контексте оно может обозначать «иметь значение, быть важным», в другом — «произносить числа одно за другим», а в третьем *count* — это существительное.

Следует ли считать различные словоформы — множественное число существительных, времена глаголов и т. д. — отдельными сущностями? Можно ли считать частью языка слова, *заимствованные* из иностранных языков, научные термины, сленг и сокращения? Очевидно, что строгого определения словарного запаса языка не существует, поскольку не так-то просто выяснить, какие группы слов в него нужно включать. В языке программирования наподобие Java попытка включить в код неизвестное слово приведет к ошибке компилятора.

Аналогичная ситуация и с формальными правилами. Как и словарный запас, многие правила любого естественного языка сформулированы достаточно расплывчато и нередко приводят к возникновению неоднозначности. Например, *инфинитивы с отделенной частицей* — грамматическая конструкция, в которой наречие ставится между глаголом в неопределенной форме и его предлогом:

```
spaCy allows you to programmatically extract the meaning of an utterance.
```

В этом примере наречие *programmatically* разделяет предлог и глагол в неопределенной форме *to extract*. Если вы настроены против инфинитивов с отделенной частицей, то можете переписать приведенное предложение следующим образом:

```
spaCy allows you to extract the meaning of an utterance programmatically.
```

Вне зависимости от того, что вы думаете относительно инфинитивов с отделенной частицей, ваше NLP-приложение должно одинаково хорошо понимать оба эти предложения.

Программа же, предназначенная для обработки кода, написанного на каком-либо языке программирования, на решение подобных проблем не рассчитана. Дело в строгой формулировке правил языка программирования, не оставляющей возможности для разночтений. В качестве еще одного примера рассмотрим оператор, написанный на языке программирования SQL, с помощью которого можно вставить данные в таблицу БД:

```
INSERT INTO table1 VALUES(1, 'Maya', 'Silver')
```

Этот оператор говорит сам за себя. Даже если вы не знаете язык SQL, вы легко догадаетесь, что оператор вставляет три значения в таблицу `table1`.

Теперь представьте, что вы изменили его следующим образом:

```
INSERT VALUES(1, 'Maya', 'Silver') INTO table1
```

С точки зрения англоязычного читателя смысл второго оператора не отличается от смысла первого: в конце концов, его можно прочитать как фразу на английском языке и смысл останется прежним. Если же попытаться выполнить второй оператор в SQL-утилите, будет возвращена ошибка `missing INTO keyword` (не найдено ключевое слово `INTO`). Дело в том, что синтаксический анализатор SQL, —как и синтаксический анализатор любого другого языка программирования, —основывается на жестко заданных правилах. Поэтому, чтобы получить желаемое, программист обязан четко описать свои требования. В данном случае синтаксический анализатор SQL ожидает, что ключевое слово `INTO` будет следовать сразу за ключевым словом `INSERT`.

Само собой, в естественном языке подобные ограничения немыслимы. Таким образом, с учетом всех различий совершенно очевидно, что задавать вычислительную модель для естественного языка с помощью набора формальных правил (как это делается с языками программирования) неэффективно или вовсе нереально.

Поэтому наш подход будет основан не на правилах, а на наблюдениях. Вместо кодирования языка путем назначения для каждого слова заранее заданного числа алгоритмы машинного обучения генерируют статистические модели для выявления закономерностей в больших массивах языковых данных и последующего выполнения предсказаний относительно синтаксической структуры новых, еще незнакомых модели текстовых данных.

Рисунок 1.3 резюмирует процессы обработки текстов на естественных языках и языках программирования соответственно.

Система обработки текстов на естественном языке сначала на основе статистической модели выполняет предсказание относительно смысла входного текста, после чего реагирует соответствующим образом. Компилятор же, обрабатывающий код программы, применяет к этому коду набор строго заданных правил.

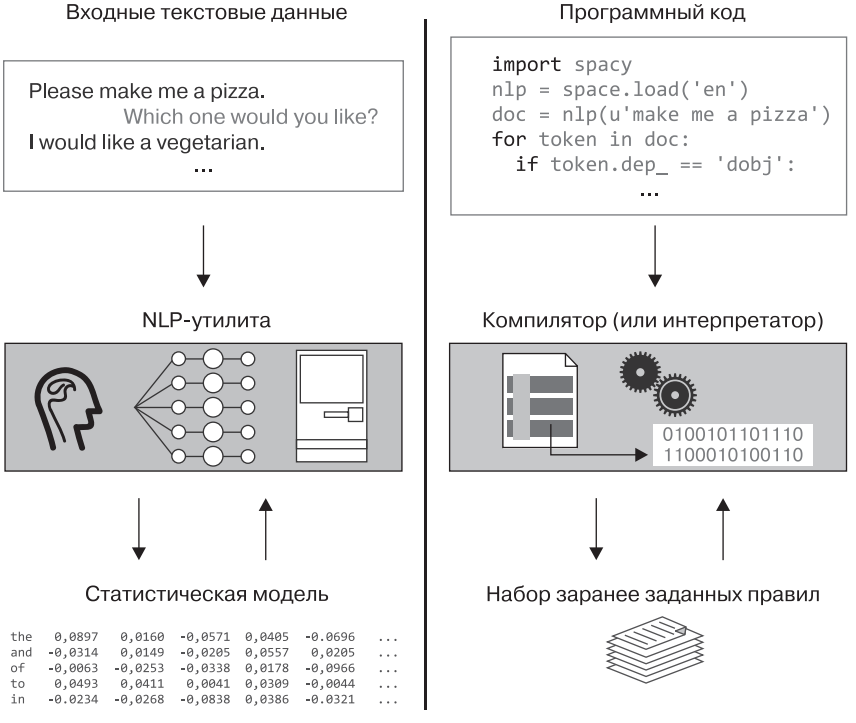


Рис. 1.3. Слева приведен упрощенный технологический процесс обработки текстов на естественном языке, справа — упрощенный технологический процесс обработки кода на языке программирования

Что такое статистическая модель в NLP

В NLP *статистическая модель* (statistical model) содержит оценки распределения вероятностей языковых единиц, например слов или фраз, что позволяет ставить им в соответствие лингвистические признаки. В теории вероятностей и статистике *распределение вероятностей* (probability distribution) для конкретной случайной величины представляет собой таблицу соответствий значений этой величины вероятностям их выпадения (в эксперименте). Таблица 1.1 иллюстрирует пример распределения вероятностей тегов частей речи слова *count* для заданного предложения. (Напомню, что в зависимости от контекста отдельное слово в английском языке может относиться к разным частям речи.)

Таблица 1.1. Пример распределения вероятностей для языковой единицы в конкретном контексте заданного предложения

Глагол	Существительное
78 %	22 %

Конечно, в другом контексте эти числа для слова *count* будут отличаться.

Статистическое моделирование языка играет особо важную роль в решении многих задач обработки естественного языка, в его генерации и понимании. Именно поэтому статистические модели являются краеугольным камнем практически любого NLP-приложения.

На рис. 1.4 в общих чертах показано, как NLP-приложение использует статистическую модель.

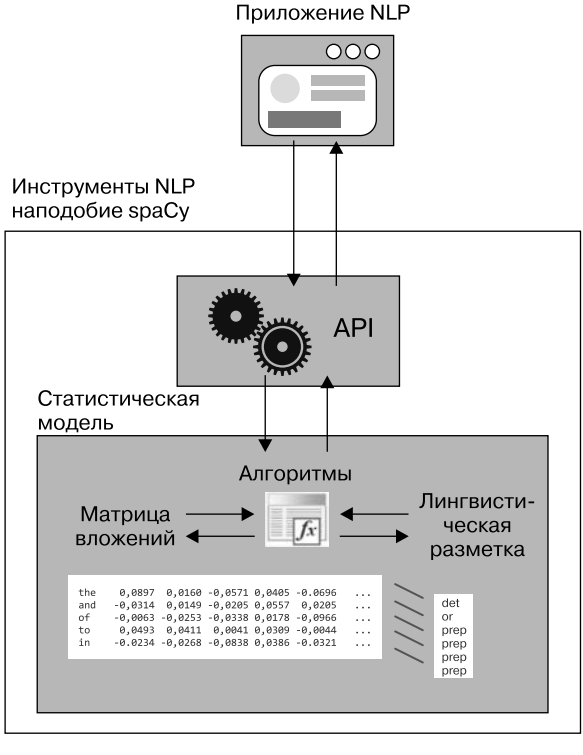


Рис. 1.4. Укрупненный вид архитектуры приложений NLP

Приложение взаимодействует с API spaCy, который абстрагирует статистическую модель, лежащую в его основе. Модель содержит такую информацию, как векторы слов, лингвистическая разметка и т. д. Лингвистическая разметка может иметь различные признаки, например теги частей речи и синтаксическую разметку. В статистической модели также есть набор алгоритмов машинного обучения для извлечения необходимой информации из данных.

На практике данные модели обычно хранятся в двоичном формате. Двоичные данные не рассчитаны на людей, но отлично подходят для машин, так как их можно легко хранить и быстро загружать.

Нейросетевые модели

В таких инструментах NLP, как spaCy, для разбора синтаксических зависимостей, частеречной разметки и распознавания именованных сущностей используются нейросетевые модели. *Нейронная сеть* (neural network) представляет собой набор алгоритмов предсказания. Она состоит из большого числа простых обрабатывающих элементов, подобных нейронам в мозге человека, которые взаимодействуют между собой путем отправки сигналов в соседние узлы и получения встречных сигналов.

Обычно узлы в нейронной сети сгруппированы по слоям: имеются входной и выходной слои, а между ними — один скрытый слой или более. Каждый узел в слое (за исключением выходного слоя) соединяется с каждым узлом из следующего слоя, и каждому соединению соответствует весовой коэффициент. Во время процесса обучения алгоритм подбирает веса таким образом, чтобы минимизировать ошибку предсказаний. Благодаря подобной архитектуре нейронная сеть способна выявлять паттерны даже в сложных входных данных.

По сути, нейронную сеть можно представить так, как на рис. 1.5.

Поступающий сигнал умножается на весовой коэффициент, который представляет собой вещественное число. Источниками передаваемых нейронной сети входных значений и весовых коэффициентов обычно служат векторы слов, сгенерированных во время обучения сети.

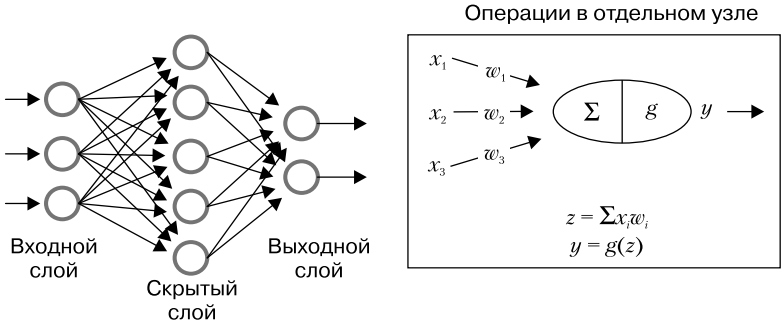


Рис. 1.5. Концептуальная схема нейронной сети и операций, происходящих в отдельном узле

Нейронная сеть складывает результаты этих умножений для всех узлов и передает вычисленную сумму функции активации. Та, в свою очередь, выдает результат (обычно в диапазоне от 0 до 1), генерируя новый сигнал, который затем передается в каждый из узлов последующего слоя, или, в случае выходного слоя, возвращая выходной сигнал. Обычно число узлов выходного слоя равно числу различных возможных исходов для данного алгоритма. Например, количество узлов в нейронной сети, предназначенной для частеречной разметки, должно совпадать с числом поддерживаемых системой тегов частей речи, как показано на рис. 1.6.

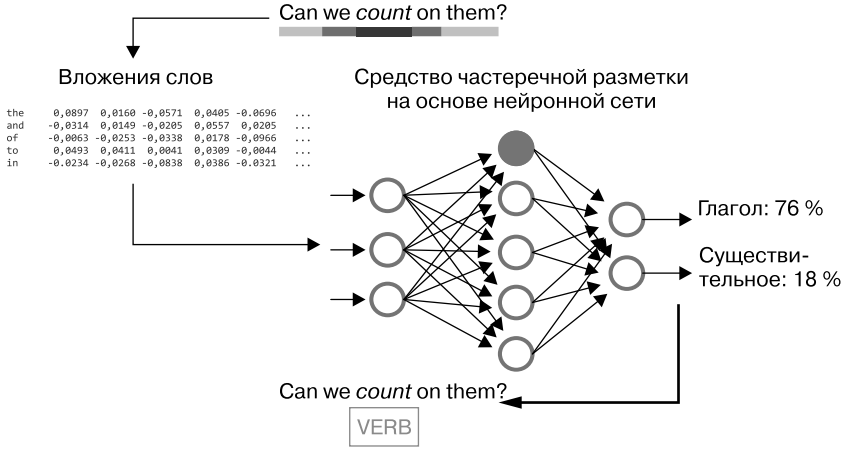


Рис. 1.6. Упрощенная схема процесса частеречной разметки

На выходе частеречной разметки выдается распределение вероятностей по всем возможным частям речи для заданного слова в конкретном контексте.

Использование сверточных нейронных сетей для NLP

Архитектура настоящей нейросетевой модели может быть очень сложной и состоять из большого числа различных слоев. Например, нейросетевая модель, применяемая в spaCy, представляет собой так называемую *сверточную нейронную сеть* (convolutional neural network, CNN), имеющую сверточный слой, который используется и при частеречной разметке, и в синтаксическом анализаторе, и при распознавании именованных сущностей. Сверточный слой применяет к отдельным областям входных данных набор фильтров обнаружения, проверяя, присутствуют ли в этих данных определенные признаки.

Посмотрим, как могла бы работать CNN при задаче частеречной разметки предложения из предыдущего примера:

```
Can we count on them?
```

Вместо того чтобы анализировать каждое слово по отдельности, сверточный слой сначала разбивает предложение на части. Предложение в NLP можно считать матрицей, каждая строка которой соответствует слову, представленному в виде вектора. Таким образом, если размерность каждого из векторов слов равна 300, а длина предложения — пять слов, получится матрица размером 5×300. Если размер фильтра обнаружения в сверточном слое равен 3 (то есть он применяется к трем последовательным словам), размер покрывающих входные данные областей составит 3×300. Такого контекста будет достаточно, чтобы соотнести каждое слово с тегом части речи.

Операция частеречной разметки с помощью сверточного подхода показана на рис. 1.7.

В предыдущем примере самым сложным для средства разметки было определить, к какой части речи относится слово *count*. Проблема в том, что в зависимости от контекста это слово может быть и глаголом, и существительным. Но задача резко упрощается, когда процедура

разметки видит фрагмент с сочетанием слов *we count on*, из которого ясно, что *count* может быть только глаголом.

Рис. 1.7. Упрощенная схема применения сверточного подхода к задаче NLP

Подробный рассказ обо всем происходящем «под капотом» сверточной архитектуры выходит за рамки данной книги. Узнать больше об архитектуре нейросетевых статистических моделей spaCy можно из раздела Neural Network Model Architecture («Архитектура нейросетевых моделей») документации API spaCy.

Какие задачи остаются за вами?

В предыдущем разделе вы узнали, что библиотека spaCy использует нейросетевые модели для разбора синтаксических зависимостей, для частеречной разметки и для распознавания именованных сущностей. Поскольку spaCy решает эти задачи сама, что же остается вам как разработчику NLP-приложения?

Одна из вещей, которую библиотека spaCy не может сделать за вас, — распознать намерения пользователя. Допустим, вы занимаетесь продажей одежды и онлайн-приложение для приема заказов получило следующий запрос от клиента:

```
I want to order a pair of jeans.
```

Ваше приложение должно распознать, что клиент намерен разместить заказ на пару джинсов.

Если воспользоваться spaCy для разбора синтаксических зависимостей приведенной фразы, получится результат, показанный на рис. 1.8.

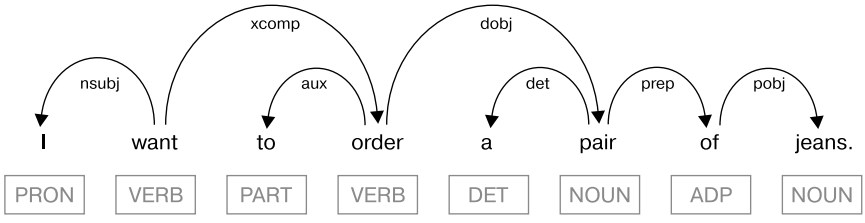


Рис. 1.8. Дерево зависимостей для нашего примера высказывания

Обратите внимание, что spaCy не отмечает элементы сгенерированного дерева как отражающие намерения пользователя. Иное было бы даже странно. Дело в том, что spaCy не знает, как именно вы реализовали логику приложения и какую разновидность намерений хотели бы видеть. Определять ключевые слова для решения задачи по распознаванию намерений можете только вы.

Для распознавания смысла высказывания или текста важны следующие основные аспекты: ключевые слова, контекст и переход значения.

Ключевые слова

Чтобы выбрать наиболее важные слова для распознавания смысла, можно воспользоваться результатами разбора синтаксических зависимостей. В примере с фразой *I want to order a pair of jeans* ключевыми словами, очевидно, являются *order* и *jeans*.

Обычно для формирования намерения вполне достаточно переходного глагола и его прямого дополнения. Но в данном примере ситуация более запутанная. Необходимо пройтись по дереву зависимостей и извлечь *order* (переходный глагол) и *jeans* (предложное дополнение, относящееся к прямому дополнению *pair*).

Контекст

Контекст может играть важную роль при выборе ключевых слов, поскольку смысл одной и той же фразы может быть разным в разных контекстах. Допустим, нужно обработать следующее высказывание:

```
I want the newspaper delivered to my door.
```

В зависимости от контекста это утверждение может быть либо запросом подписаться на газету, либо требованием доставить ее до двери. В первом случае ключевыми словами будут *want* и *newspaper*. Во втором — *delivered* и *door*.

Переход значения

Зачастую даже самые простые намерения люди выражают с помощью не одного, а нескольких предложений. Обратимся к следующему тексту:

```
I already have a relaxed pair of jeans. Now I want a skinny pair.
```

Слова, отражающие намерение, здесь встречаются в двух разных предложениях. Это показано на рис. 1.9.

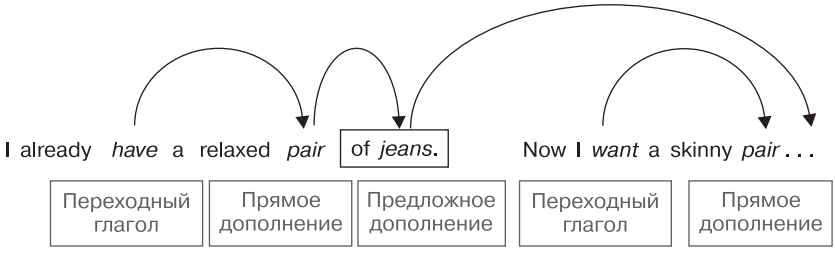


Рис. 1.9. Распознавание намерения текста

Как вы, вероятно, поняли, лучше всего намерение этого текста описывают слова *want* и *jeans*. Общие шаги поиска ключевых слов в данном конкретном примере следующие.

1. Найти в тексте переходный глагол в настоящем времени.
2. Найти прямое дополнение найденного на шаге 1 переходного глагола.
3. В случае местоименной формы найденного на предыдущем шаге прямого дополнения найти его антецедент в предыдущем предложении.

С помощью spaCy эти шаги можно легко реализовать программным образом. Процесс описан в главе 8.

Резюме

В этой главе вы познакомились с основами обработки естественного языка. Теперь вы знаете, что в отличие от людей машины работают с векторными представлениями слов, благодаря чему можно производить арифметические действия над элементами естественного языка, в том числе над словами, предложениями и документами.

Вы узнали, что векторы слов реализуются в статистических моделях, в основе которых лежит архитектура нейронных сетей. Кроме того, вы поняли, какие задачи остаются за вами как за разработчиком NLP-приложений.

2 Конвейер обработки текста

Теперь, разобравшись со структурой NLP-приложения, перейдем к практическим действиям. В этой главе вам предстоит установить spaCy и настроить рабочую среду. Затем вы узнаете о *конвейере обработки текста* — наборе базовых операций NLP, используемых для выяснения смысла и намерения связного текста. В числе операций NLP — токенизация, лемматизация, частеречная разметка, разбор синтаксических зависимостей и распознавание именованных сущностей.

Настройка рабочей среды

Прежде чем приступить к использованию spaCy, необходимо настроить рабочую среду, установив на машине следующие программные компоненты:

- Python 2.7 (или более позднюю версию) либо Python 3.4 (или более позднюю версию)[1];
- библиотеку spaCy;
- статистическую модель для spaCy.

Для использования библиотеки spaCy v2.0.x вам понадобится Python версии 2.7 либо 3.4 (или более поздние версии). Скачать можно по

[1] Существует две ветки Python, развивающиеся практически независимо и зачастую несовместимые, — 2.x и 3.x.

ссылке https://www.python.org/downloads/ и далее выполнить инструкции по настройке среды Python. Установите spaCy в среде Python с помощью pip, введя команду:

```
$ pip install spacy
```

Если в вашей системе установлено несколько версий Python, выберите pip, связанный с той версией, с которой вы хотели бы работать. Например, для использования spaCy с Python 3.5 выполните команду:

```
$ pip3.5 install spacy
```

Если библиотека spaCy уже установлена в вашей системе, можете обновить ее до текущей версии. Примеры, приведенные в книге, предполагают использование spaCy v2.0.x. Проверить, какая версия spaCy установлена у вас, можно с помощью команды:

```
$ python -m spacy info
```

Опять же необходимо заменить команду `python` на команду исполняемого файла той версии Python, которая используется в вашей конкретной среде, — скажем, `python3.5`. Здесь и далее я буду указывать в примерах команд просто `python` и `pip`.

Для обновления до последней версии установленного в системе пакета spaCy можно воспользоваться следующей командой `pip`:

```
$ pip install -U spacy
```

Установка статистических моделей для библиотеки spaCy

В установочные пакеты spaCy не включены статистические модели, необходимые при использовании библиотеки. Статистические модели содержат знания о конкретном языке, собранные из множества источников. Нужную вам модель придется отдельно скачать и установить.

Доступно несколько предобученных статистических моделей для различных языков. С сайта spaCy вы можете скачать, например, такие модели для английского языка: `en_core_web_sm`, `en_core_web_md`,

`en_core_web_lg` и `en_vectors_web_lg`. Названия моделей создаются по следующему принципу: *lang_type_genre_size*. *Lang* обозначает язык. *Type* указывает на возможности модели (например, `core` означает, что речь идет об универсальной модели, имеющей словарь, синтаксис, сущности и векторы). *Genre* указывает на тип текстов, которые лежали в основе обучения данной модели: `web` («Википедия» или подобные ресурсы) или `news` (новостные статьи). *Size* обозначает размер модели: `lg` — большая, `md` — средняя и `sm` — маленькая. Чем больше модель, тем больше дискового пространства ей нужно. Например, модель `en_vectors_web_lg-2.1.0` занимает 631 Мбайт, в то время как `en_core_web_sm-2.1.0` — лишь 10 Мбайт.

Чтобы следить за ходом примеров, достаточно модели `en_core_web_sm`, занимающей минимум места. Она выбирается библиотекой spaCy автоматически при установке с помощью команды:

```
$ python -m spacy download en
```

Сокращение `en` указывает spaCy, что нужно скачать и установить по умолчанию самую подходящую модель для английского языка. «Самая подходящая» в этом случае — сгенерированная для указанного языка (в данном примере — английского), универсальная и требующая минимум дискового пространства.

Для скачивания конкретной модели необходимо указать ее название следующим образом:

```
$python -m spacy download en_core_web_md
```

После установки модель можно загрузить с помощью того же самого сокращенного названия, что и во время установки:

```
nlp = spacy.load('en')
```

Базовые операции NLP в библиотеке spaCy

Начнем с цепочки базовых операций NLP — конвейера обработки. Библиотека SpaCy выполняет все эти операции неявно, поэтому мы можем сосредоточиться на логике приложения. На рис. 2.1 приведена упрощенная схема процесса.

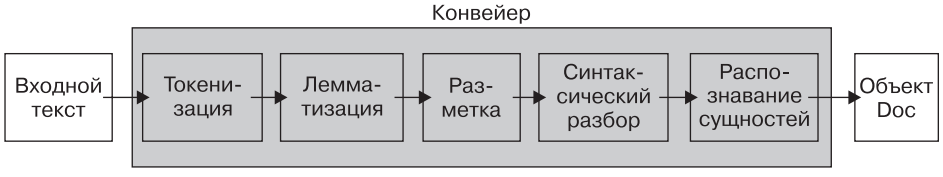


Рис. 2.1. Упрощенная схема конвейера обработки

Конвейер обработки обычно включает операции токенизации, лемматизации, частеречной разметки, разбора синтаксических зависимостей и распознавания именованных сущностей.

Токенизация

Самое первое действие любого NLP-приложения — разбор текста на *токены*, которые могут быть словами, числами или знаками препинания. Токенизация — самая первая операция в конвейере, поскольку для всех остальных операций необходимо, чтобы текст был разобран на токены.

Следующий код демонстрирует процесс токенизации:

```
❶ import spacy
❷ nlp = spacy.load('en')
❸ doc = nlp(u'I am flying to Frisco')
❹ print([w.text for w in doc])
```

Начинаем с импорта библиотеки spaCy ❶, чтобы получить доступ к ее функциональности, после чего загружаем пакет модели с помощью сокращения `en` ❷, чтобы создать экземпляр класса `Language` библиотеки spaCy. Объект `Language` содержит словарь языка и другие данные статистической модели. Назовем его `Language nlp`.

Далее применяем ❸ только что созданный объект к нашему примеру предложения, создавая объект `Doc` — контейнер для последовательности объектов `Token`. Библиотека spaCy генерирует его неявным образом на основе переданного ей текста.

К этому моменту с помощью всего лишь трех строк кода нам удалось добиться от spaCy генерации грамматической структуры для нашего

примера предложения. Как ее использовать — решаете вы. В данном очень простом примере нам удалось только вывести *текстовое содержимое* (text content) всех токенов предложения ❹.

Наш код выводит токены предложения в виде списка:

```
['I', 'am', 'flying', 'to', 'Frisco']
```

Текстовое содержимое — это группа символов, составляющих токен. Например, буквы *a* и *m* в токене *am* — лишь одно из многих свойств объекта `Token`. В следующих примерах вы увидите, что можно извлекать разнообразные лингвистические признаки, присвоенные токену.

Лемматизация

Лемма — это базовая, фактически словарная форма токена. Например, лемма токена *flying* — *fly*. *Лемматизация* — процесс сведения словоформ к соответствующим леммам. В следующем коде приведен простой пример лемматизации с помощью библиотеки spaCy:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'this product integrates both libraries for downloading
and applying patches')
for token in doc:
  print(❶token.text, ❷token.lemma_)
```

Первые три строки этого сценария такие же, как и в предыдущем сценарии. Напомню, что они импортируют библиотеку spaCy, загружают модель английского языка с помощью сокращения `en`, создают конвейер обработки текста и применяют этот конвейер к нашему примеру предложения. Иначе говоря, создают объект `Doc`, через который можно получить доступ к грамматической структуре предложения.

ПРИМЕЧАНИЕ

В грамматике структурой предложения называется расстановка отдельных слов, а также фраз и частей сложного предложения. Грамматический смысл предложения зависит от его структуры.

Получив объект `Doc` с токенами, проходим в цикле по этим токенам, после чего выводим текстовое содержимое каждого из них ❶ вместе с соответствующей леммой ❷. Результат, который выводит этот сценарий, выглядит следующим образом:

```
this        this
product     product
integrates  integrate
both        both
libraries   library
for         for
downloading download
and         and
applying    apply
patches     patch
```

Столбец слева содержит токены, столбец справа — соответствующие леммы.

Использование лемматизации для распознавания смысла

При решении задачи распознавания смысла особо важное значение имеет этап лемматизации. Чтобы понять почему, вернемся к предложению-примеру из предыдущего раздела:

```
I am flying to Frisco.
```

Пусть это предложение было подано на вход NLP-приложения, взаимодействующего с онлайн-системой, которая предоставляет API для бронирования билетов на различный транспорт. Приложение обрабатывает запрос пользователя, выделяет из него необходимую информацию и передает ее нижележащему API. Схема работы выглядит примерно так, как показано на рис. 2.2.

NLP-приложение пытается извлечь из запроса пользователя следующую информацию: вид путешествия (авиа, поезд, автобус и т. д.) и пункт назначения. Сначала приложению необходимо выяснить, что хочет приобрести пользователь: авиабилет, билет на поезд или билет на автобус. Для этого оно ищет слово, совпадающее с одним из ключевых слов заранее заданного списка. Простейший способ

облегчить поиск — преобразовать все слова в предложении в соответствующие леммы, и тогда список ключевых слов может быть более коротким и понятным. Например, нет необходимости включать в него все словоформы глагола *fly* (такие как *fly*, *flying*, *flew* и *flown*), чтобы приложение могло понять желание пользователя заказать именно авиабилет. Достаточно свести все возможные варианты к базовой форме слова — *fly*.

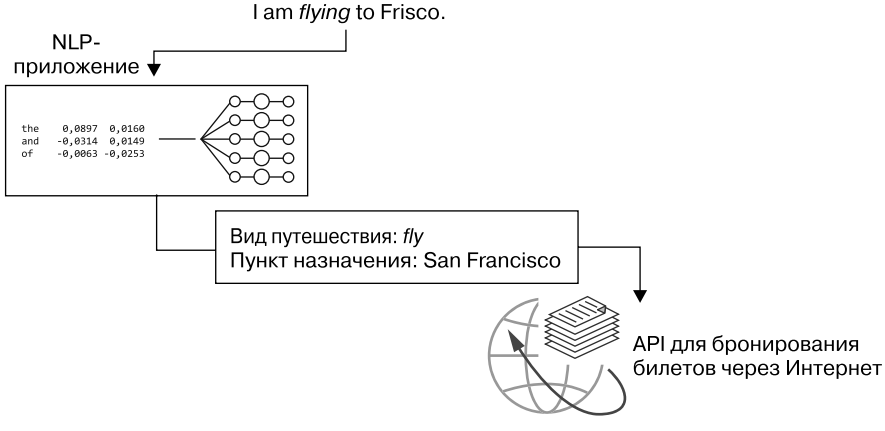


Рис. 2.2. Использование лемматизации при извлечении необходимой информации из запросов пользователей

Лемматизация также удобна при определении пункта назначения на основе запроса пользователя. У многих городов мира есть прозвища, но для системы бронирования билетов необходимы официальные названия. Конечно, объекту `Tokenizer`, который выполняет лемматизацию по умолчанию, неизвестна разница между разговорными и официальными названиями городов, стран и т. д. Для решения этой проблемы добавим в имеющийся экземпляр `Tokenizer` правила для подобных исключений.

Следующий сценарий иллюстрирует возможную реализацию лемматизации для нескольких городов назначения. В нем выводятся леммы составляющих предложение слов.

```
import spacy
from spacy.symbols import ORTH, LEMMA
```

```
nlp = spacy.load('en')
doc = nlp(u'I am flying to Frisco')
print([w.text for w in doc])
❶ special_case = [{ORTH: u'Frisco', LEMMA: u'San Francisco'}]
❷ nlp.tokenizer.add_special_case(u'Frisco', special_case)
❸ print([w.lemma_ for w in nlp(u'I am flying to Frisco')])
```

Описываем *исключение* (special case) для слова *Frisco* ❶, заменив его лемму по умолчанию на *San Francisco*. Далее добавляем это исключение в экземпляр `Tokenizer` ❷: теперь `Tokenizer` всякий раз будет использовать данное исключение при запросе леммы для *Frisco*. Для проверки выводим леммы всех слов в предложении ❸.

В результате выполнения сценария получится следующее:

```
['I', 'am', 'flying', 'to', 'Frisco']
['-PRON-', 'be', 'fly', 'to', 'San Francisco']
```

Как видим, выведены леммы всех слов заданного предложения, за исключением `Frisco`, вместо которого выводится `San Francisco`.

Частеречная разметка

Тег части речи (part-of-speech tag) указывает, к какой части речи в этом конкретном предложении относится конкретное слово (оно может быть существительным, глаголом, наречием и т. д.). В главе 1 я уже отмечал тот факт, что в английском языке слова в зависимости от контекста выступают в роли различных частей речи.

В библиотеке spaCy теги частей речи нередко содержат и подробную информацию о токене. Например, в информации о глаголе могут быть указаны следующие признаки: время — прошедшее, настоящее или будущее; вид (аспект) — простой, длительный или совершенный; лицо — 1-е, 2-е или 3-е; число — единственное или множественное.

Извлечение тегов частей речи для глаголов может помочь с определением намерения пользователя, когда токенизации и лемматизации недостаточно. Скажем, сценарий лемматизации для приложения бронирования билетов из предыдущего раздела не поможет NLP-приложению выбрать слова в предложении для составления запроса

к нижележащему API: на практике подобная задача бывает достаточно сложной. Например, если запрос пользователя состоит более чем из одного предложения:

```
I have flown to LA. Now I am flying to Frisco.
```

Результат лемматизации этих предложений выглядит следующим образом:

```
['-PRON-', 'have', 'fly', 'to', 'LA', '.', 'now', '-PRON-', 'be', 'fly',
'to', 'San Francisco', '.']
```

Одной лемматизации здесь недостаточно, ведь приложение может счесть ключевыми словами леммы *fly* и *LA* из первого предложения, указывающие на желание пользователя полететь в Лос-Анджелес, хотя на самом деле тот намерен отправиться в Сан-Франциско. Проблему усложняет тот факт, что при лемматизации глаголы сводятся к формам инфинитива, из-за чего сложно определить их роль в предложении.

В такой ситуации на помощь приходят теги частей речи. В английском языке в число основных частей речи входят существительное, местоимение, определитель, прилагательное, глагол, наречие, числительное, служебные части речи: предлог, союз, артикль и междометие (больше информации вы найдете в кратком грамматическом справочнике в приложении). В библиотеке spaCy представлены те же категории, а также еще несколько для обозначения *укрупненных частей речи* (coarse-grained parts of speech) — символов, знаков препинания и пр. Все категории доступны в виде фиксированного набора тегов через атрибуты `Token.pos` (тип int) и `Token.pos_` (тип unicode).

Кроме того, spaCy предоставляет теги для *уточненных частей речи* (fine-grained parts of speech) с более подробной информацией о токене, где указаны морфологические признаки, времена глаголов и типы (разряды) местоимений. Естественно, в списке уточненных частей речи тегов гораздо больше, чем в списке укрупненных. Теги уточненных частей речи доступны в атрибутах `Token.tag` (тип int) и `Token.tag_` (тип unicode).

В табл. 2.1 перечислены некоторые распространенные теги частей речи, используемые в spaCy для моделей английского языка.

Таблица 2.1. Некоторые распространенные теги частей речи библиотеки spaCy

TAG (укрупненные части речи)	POS (уточненные части речи)	Морфология	Описание
NN	NOUN	Number=sing	Существительное, единственное число
NNS	NOUN	Number=plur	Существительное, множественное число
PRP	PRON	PronType=prs	Местоимение, личное
PRP$	PRON	PronType=prs Poss=yes	Местоимение, притяжательное
VB	VERB	VerbForm=inf	Глагол, инфинитив
VBD	VERB	VerbForm=fin Tense=past	Глагол, прошедшее время
VBG	VERB	VerbForm=part Tense=pres Aspect=prog	Глагол, герундий или причастие настоящего времени
JJ	ADJ	Degree=pos	Прилагательное

ПРИМЕЧАНИЕ

Полный список тегов уточненных частей речи библиотеки spaCy можно найти в разделе Part-of-Speech Tagging («Частеречная разметка») в руководстве Annotation Specifications («Спецификации маркирования») по ссылке https://spacy.io/api/annotation#pos-tagging.

Вероятно, самые интересные для NLP-приложений свойства глаголов — время и вид. Взятые вместе, они указывают на то, как определенное глаголом действие протекает во времени. Например, форма *длительного вида настоящего времени* (present tense progressive aspect) описывает происходящее прямо сейчас или ожидаемое в самом ближайшем будущем. Форма длительного вида настоящего времени формируется путем добавления формы настоящего времени глагола *to be* перед требуемым глаголом с окончанием *-ing*. Например, в предложении *I am looking into it* перед глаголом *looking* стоит *am* — глагол *to be* в форме первого лица настоящего времени. В этом примере *am* указывает на настоящее время, а *looking* — на длительный вид.

Поиск соответствующих глаголов с помощью тегов частей речи

Приложение по бронированию билетов может использовать теги частей речи из библиотеки spaCy, чтобы отфильтровать глаголы в тексте и оставить лишь те, которые нужны для определения намерений пользователя.

Прежде чем перейти к реализации кода этого процесса, давайте подумаем, какими фразами пользователи могут выражать намерение забронировать билет, скажем, в Лос-Анджелес. Начнем с нескольких фраз, содержащих следующее сочетание лемм: *fly*, *to* и *LA*. Вот несколько простейших вариантов:

```
I flew to LA.
I have flown to LA.
I need to fly to LA.
I am flying to LA.
I will fly to LA.
```

Обратите внимание: несмотря на то что после сведения до лемм все эти предложения будут включать сочетание *fly to LA*, лишь часть из них выражает реальное намерение пользователя забронировать билет в Лос-Анджелес. Первые два явно не подходят.

Проанализировав ситуацию, мы поймем, что формы глагола *fly* простого прошедшего времени и прошедшего совершенного времени, использованные в первых двух предложениях, не подразумевают искомое намерение. Нам подходят только формы инфинитива и настоящего длительного времени. Следующий сценарий иллюстрирует, как можно найти эти формы в нашем примере текста:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
print([w.text for w in doc if ❶w.tag_== ❷'VBG' or w.tag_== ❸'VB'])
```

Свойство `tag_` ❶ объекта `Token` содержит атрибут уточненной части речи этого объекта. Чтобы проверить, был ли токену присвоен атрибут `VB` (глагол в неопределенной, то есть инфинитивной, форме) ❸ или `VBG` (глагол в форме настоящего длительного времени) ❷, проходим в цикле по всем токенам, составляющим текст.

В нашем примере текста указанным условиям удовлетворяет лишь глагол *flying* из второй фразы, поэтому результат будет следующим:

```
['flying']
```

Конечно, теги уточненных частей речи присваиваются не только глаголам. Например, spaCy распознает, что *LA* и *Frisco* — имена собственные (то есть имена людей или названия мест, объектов и организаций), и присваивает им теги `PROPN`. При желании в предыдущий сценарий можно было бы добавить следующую строку кода:

```
print([w.text for w in doc if w.pos_ == 'PROPN'])
```

В результате ее добавления будет выведен еще и такой список:

```
['LA', 'Frisco']
```

Здесь перечислены имена собственные из обоих предложений нашего примера текста.

Важность контекста

Тегов уточненных частей речи не всегда достаточно для определения смысла высказывания — иногда необходимо учитывать и контекст. В качестве примера рассмотрим следующее высказывание: *I am flying to LA*. Здесь *flying* — глагол в форме настоящего длительного времени, и средство частеречной разметки соотнесет его с тегом `VBG`. Но, поскольку данная форма глагола может описывать как то, что происходит прямо сейчас, так и то, что произойдет в ближайшем будущем, высказывание может означать и *I'm already in the sky, flying to LA* («Я уже в небе, лечу в Лос-Анджелес»), и *I'm going to fly to LA* («Я собираюсь лететь в Лос-Анджелес»). Но приложение бронирования билетов должно интерпретировать лишь одно из этих предложений как *I need an air ticket to LA* («Я хочу приобрести билет в Лос-Анджелес»). Таким же образом рассмотрим следующий связный текст: *I am flying to LA. In the evening, I have to be back in Frisco.* Вероятнее всего, говорящий хочет купить авиабилет из Лос-Анджелеса в Сан-Франциско на вечерний рейс. Больше примеров распознавания смысла по контексту вы найдете в подразделе «Усовершенствование чат-бота для бронирования билетов путем учета контекста» на с. 137.

Синтаксические отношения

Теперь соединим имена собственные с глаголом, который был выбран ранее средством частеречной разметки. Напомню, что список глаголов, доступных к использованию для идентификации намерения связного текста, включает лишь глагол *flying* из второго предложения. Как же теперь получить пару «глагол/имя собственное», лучше всего описывающую намерение текста? Человек, конечно, с легкостью составит пары «глагол/имя собственное» из слов одного и того же предложения. А поскольку глагол *flown* из первого предложения не удовлетворяет заданному условию (ему удовлетворяют только неопределенная форма и форма настоящего длительного времени), подобную пару можно составить лишь для второго предложения: *flying*, *Frisco*.

Для обработки подобных ситуаций программным образом в spaCy есть средство разбора синтаксических зависимостей, которое выявляет синтаксические отношения между отдельными токенами в предложении и соединяет дугами синтаксически связанные пары слов.

Метки синтаксической зависимости (syntactic dependency labels), равно как и леммы и теги частей речи, — это лингвистические признаки, присваиваемые библиотекой spaCy объектам `Token`, которые образуют содержащийся в объекте `Doc` текст. Например, метка зависимости `dobj` соответствует прямому дополнению (direct object). Соответствующее синтаксическое отношение можно проиллюстрировать направленной дугой, как показано на рис. 2.3.

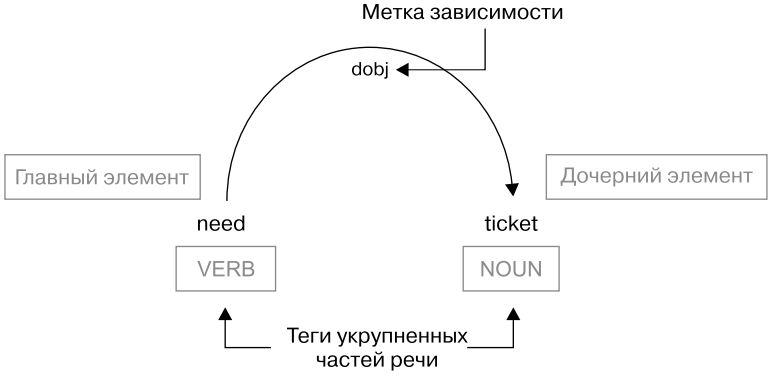


Рис. 2.3. Графическое представление дуги синтаксической зависимости

ГЛАВНЫЙ И ДОЧЕРНИЙ ЭЛЕМЕНТЫ

Метка синтаксической зависимости описывает вид синтаксического отношения между двумя словами в предложении. В подобной паре одно слово является синтаксически управляющим элементом (его иногда называют главным или родительским элементом), а второе зависит от него (его называют дочерним элементом). spaCy присваивает метку синтаксической зависимости дочернему элементу пары. Например, в извлеченной из предложения *I need a plane ticket* паре *need*, *ticket* слово *ticket* — дочернее, а слово *need* — главное, поскольку *need* — это глагол в так называемой глагольной группе. В том же предложении *a plane ticket* — именная группа: существительное *ticket* является главным, а *a* и *plane* — его дочерними элементами. Более подробную информацию вы найдете в разделе «Грамматика зависимостей и грамматика с фразовой структурой» на с. 249.

У каждого слова в предложении есть только один главный элемент. Соответственно, любое слово может быть дочерним по отношению лишь к одному главному элементу. Обратное утверждение справедливо не всегда: одно и то же слово может вообще не выступать в роли главного элемента или играть эту роль как в одной, так и в нескольких парах. Последнее означает, что у главного элемента есть несколько дочерних. Таким образом, метка зависимости всегда присваивается дочернему элементу пары.

Метка `dobj` присваивается слову *ticket*, поскольку в данном отношении оно играет роль дочернего элемента. В коде можно определить главный элемент отношения с помощью атрибута `Token.head`.

Другие отношения главного и дочернего элементов в нашем предложении показаны на рис. 2.4.

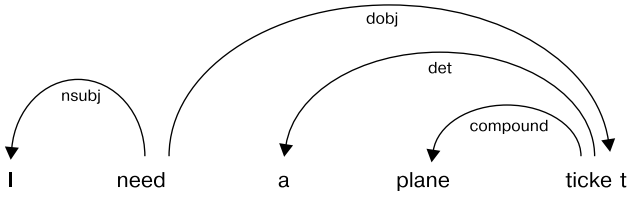


Рис. 2.4. Отношения главного и дочернего элементов во всем предложении

Как видите, одно и то же слово в предложении может участвовать в нескольких синтаксических отношениях. В табл. 2.2 перечислены

некоторые метки зависимостей английского языка, используемые чаще всего.

Таблица 2.2. Некоторые часто встречающиеся метки зависимостей

Метка зависимости	Описание
acomp	Прилагательное-комплемент
amod	Определение-прилагательное
aux	Вспомогательный глагол
compound	Составное слово
dative	Дательный падеж
det	Определитель
dobj	Прямое дополнение
nsubj	Субстантивированное подлежащее
pobj	Предложное дополнение
ROOT	Корневой элемент

Метка `ROOT` обозначает токен, главным элементом для которого является он сам. Обычно такую метку spaCy присваивает смысловому глаголу предложения (то есть глаголу, составляющему основу сказуемого). В каждом полном предложении должны быть глагол с тегом `ROOT` и подлежащее с тегом `nsubj`. Остальные элементы не обязательны.

ПРИМЕЧАНИЕ

В большинстве примеров, приведенных в книге, предполагается, что подаваемый на вход текст — законченное предложение, поэтому для поиска смыслового глагола предложения используется тег ROOT. Учтите, что этому условию удовлетворяют не все входные данные.

Следующий сценарий иллюстрирует обращение к меткам синтаксической зависимости токенов для текста примера из подраздела «Частеречная разметка» на с. 47:

```
import spacy
nlp = spacy.load('en_core_web_md')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
```

```
for token in doc:
  print(token.text, ❶token.pos_, ❷token.dep_)
```

Сценарий выводит теги укрупненных частей речи ❶ (см. табл. 2.1) и метки зависимостей, присвоенные составляющим этот текст токенам ❷:

```
I      PRON  nsubj
have   VERB  aux
flown  VERB  ROOT
to     ADP   prep
LA     PROPN pobj
.      PUNCT punct
Now    ADV   advmod
I      PRON  nsubj
am     VERB  aux
flying VERB  ROOT
to     ADP   prep
Frisco PROPN pobj
.      PUNCT punct
```

Однако выведенный список не демонстрирует связь слов в предложении с помощью упомянутых в начале данного раздела *дуг зависимостей*. Чтобы посмотреть дуги зависимостей для нашего примера текста, замените цикл из предыдущего сценария таким образом:

```
for token in doc:
  print(❶token.head.text, token.dep_, token.text)
```

Свойство `head` объекта `Token` ❶ ссылается на синтаксический главный элемент данного токена. Вывод этой информации позволяет увидеть, какими синтаксическими зависимостями связаны между собой слова в предложениях текста. Если изобразить зависимости графически, каждой строке следующего вывода будет соответствовать дуга. Исключение составляет отношение `ROOT` (поскольку слово с этой меткой — единственное в предложении, у которого нет главного элемента[1]):

```
flown   nsubj  I
flown   aux    have
flown   ROOT   flown
```

[1] Точнее, главным элементом для этого слова является оно само.

```
flown   prep   to
to      pobj   LA
flown   punct  .
flying  advmod Now
flying  nsubj  I
flying  aux    am
flying  ROOT   flying
flying  prep   to
to      pobj   Frisco
flying  punct  .
```

Учитывая приведенный выше список синтаксических зависимостей, попробуем разобраться, какие метки указывают на токены, лучше всего описывающие намерение пользователя. Другими словами, найдем пару, которая способна более или менее точно передать основной посыл текста.

В этом смысле многообещающе выглядят токены, маркированные метками зависимостей `ROOT` и `pobj` (в данном примере они играют ключевую роль в распознавании намерения). Как упоминалось ранее, метка `ROOT` обозначает смысловой глагол предложения, а `pobj` в этом примере отмечает сущность, которая в сочетании с глаголом резюмирует смысл всего высказывания.

В следующем сценарии найдем слова, соответствующие этим двум меткам зависимостей:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
❶ for sent in doc.sents:
  ❷ print([w.text for w in sent ❸if w.dep_ == 'ROOT' or w.dep_ == 'pobj'])
```

Здесь мы *разрезаем связный текст* ❶, разбивая его на предложения с помощью свойства `doc.sents`, предназначенного для прохода в цикле по отдельным фразам документа. Разрезание текста на отдельные предложения удобно, например, для поиска определенных частей речи в каждом из предложений текста. (О свойстве `doc.sents` поговорим в следующей главе, когда будем рассматривать пример того, как ссылаться на токены документа с помощью индексов уровня предложения.) Благодаря этому на основе конкретных меток зависимости, присвоенных токенам ❷, для каждого предложения можно

создать список потенциальных ключевых слов. Условия фильтрации в примере выбирались исходя из анализа синтаксически связанных пар, сгенерированных предыдущим сценарием. В частности, выбирались токены с метками зависимости `ROOT` и `pobj` ❸, поскольку именно они формируют интересующие нас пары.

Результаты выполнения сценария должны выглядеть так:

```
['flown', 'LA']
['flying', 'Frisco']
```

В обеих парах предложений выходные существительные были маркированы как `pobj`, что в приложении бронирования билетов поможет в выборе существительного, лучше всего подходящего к глаголу. В данном случае это глагол *flying* и существительное *Frisco*.

Приведенный пример — упрощенный вариант выделения информации с помощью меток зависимостей. В следующих главах мы рассмотрим более сложные примеры обхода деревьев зависимостей отдельных предложений или даже целого текста с извлечением необходимой информации.

Попробуйте сами

Теперь, когда вы уже знаете, как использовать возможности лемматизации, частеречной разметки и меток синтаксических зависимостей, попробуйте сделать что-нибудь самостоятельно. Например, объедините примеры из предыдущих разделов в единый сценарий, который смог бы верно определить намерение говорящего лететь в Сан-Франциско.

В результате выполнения вашего сценария должно выводиться следующее:

```
['fly', 'San Francisco']
```

Начните с последнего в этом разделе сценария и расширьте условное выражение в цикле, добавив в него условия для учета уточненных частей речи, как обсуждалось в подразделе «Частеречная разметка» на с. 47.

Распознавание именованных сущностей

Именованная сущность (named entity) — объект, на который можно ссылаться по его собственному наименованию. Именованной сущностью может быть человек, организация, место или другая сущность. Именованные сущности играют важную роль в NLP, поскольку позволяют выяснить, о каком месте или организации говорит пользователь. Следующий сценарий ищет именованные сущности в тексте, который мы использовали и в предыдущих примерах:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
for token in doc:
❶ if token.ent_type != 0:
    print(token.text, ❷token.ent_type_)
```

Если атрибуту `ent_type` токена не присвоено значение `0` ❶, значит, этот токен — именованная сущность. В таком случае выводим атрибут `ent_type_` токена ❷, содержащий тип именованной сущности в поле типа unicode. В результате выполнения нашего сценария должно выводиться следующее:

```
LA      GPE
Frisco  GPE
```

И `LA`, и `Frisco` маркированы меткой `GPE`, которая расшифровывается как *geopolitical entity* («геополитическая сущность») и обозначает страны, города, штаты и другие географические объекты.

Резюме

Вы научились настраивать рабочую среду для использования библиотеки spaCy. Кроме того, вы изучили простые сценарии, иллюстрирующие применение spaCy для выполнения простейших операций NLP (токенизации, лемматизации и выяснения синтаксических отношений между отдельными токенами в предложении) и извлечения важной информации. Приведенные в этой главе примеры сильно упрощены и не отражают всех реальных ситуаций, которые могут встретиться на практике. Для написания с помощью spaCy более сложных сценариев

нужно реализовать алгоритм получения необходимых токенов из дерева зависимостей на основе присвоенных токенам лингвистических признаков. Вопрос выделения и использования лингвистических признаков более подробно будет изучен в главе 4, а деревья зависимостей мы рассмотрим в главе 6.

Следующая глава концентрирует свое внимание на важнейших объектах API spaCy, в том числе на контейнерах и компонентах конвейера обработки. Вы научитесь использовать структуры данных и интерфейсы уровня языка C библиотеки spaCy для создания модулей Python, пригодных для обработки больших объемов текста.

3 Работа с объектами-контейнерами и настройка spaCy под свои нужды

Основные объекты, из которых состоит API spaCy, можно разделить на две категории: контейнеры (например, объекты `Token` и `Doc`) и компоненты конвейеров обработки (например, средства частеречной разметки и распознавания именованных сущностей). В этой главе продолжим изучение объектов-контейнеров: благодаря им и их методам можно получать доступ к лингвистическим меткам, которые библиотека spaCy присваивает всем токенам в тексте.

Кроме того, вы узнаете, как настроить компоненты конвейера под свои конкретные задачи и как использовать код Cython для ускорения задач NLP, требующих много времени.

Объекты-контейнеры библиотеки spaCy

Объекты-контейнеры (container object) служат для группировки нескольких элементов в одно целое. Эти элементы могут быть коллекцией объектов — токенов или предложений или набором меток, относящихся к одному объекту. Скажем, объект `Token` библиотеки spaCy представляет собой контейнер для набора меток (например, указывающих часть речи), относящихся к одному токену текста. Объекты-контейнеры в spaCy повторяют структуру текстов на естественном языке: текст состоит из предложений, каждое из которых содержит токены.

Используемые чаще всего (с точки зрения пользователя) объекты-контейнеры библиотеки spaCy — `Token`, `Span` и `Doc` — служат для представления токена, фразы/предложения и текста целиком соответственно. Контейнер может включать в себя другие контейнеры: `Doc`, например, содержит объекты `Token`. Данный раздел посвящен работе с такими объектами-контейнерами.

Получение индекса токена в объекте Doc

Объект `Doc` содержит коллекцию объектов `Token`, сгенерированных в результате токенизации входного текста. К этим токенам можно обращаться по индексам, соответствующим их позиции в тексте (рис. 3.1).

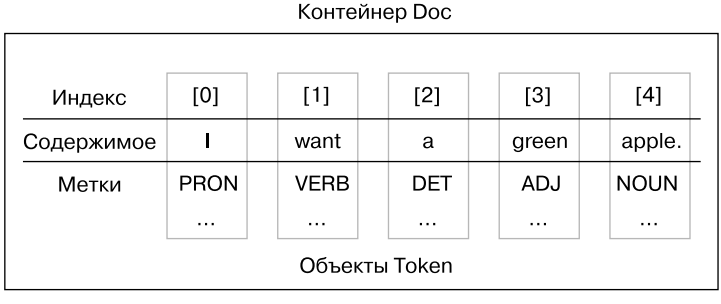


Рис. 3.1. Токены в объекте Doc

Токены индексируются с 0, так что индекс последней позиции равен длине документа минус 1. Для разрезания экземпляра `Doc` на токены их необходимо поместить в список Python, пройдя по объекту `Doc` от первого до последнего токена:

```
>>> [doc[i] for i in range(len(doc))]
[A, severe, storm, hit, the, beach, .]
```

Стоит отметить, что можно создать объект `Doc` явным образом и воспользоваться конструктором, как показано в следующем примере:

```
>>> from spacy.tokens.doc import Doc
>>> from spacy.vocab import Vocab
>>> doc = Doc(❶Vocab(), ❷words=[u'Hi', u'there'])
>>> doc
Hi there
```

Здесь вызываем конструктор класса `Doc` и передаем ему два параметра: объект *vocab* ❶ — контейнер хранилища со словарными данными, например типами лексем (прилагательное, глагол, существительное и т. д.) — и список токенов для добавления в создаваемый объект `Doc` ❷.

Обход в цикле синтаксических дочерних элементов токена

Пусть нам нужно найти левосторонний дочерний элемент токена в дереве синтаксических зависимостей предложения. Данная операция позволяет найти прилагательные (при их наличии) для заданного существительного. Это может понадобиться, если нужно узнать, какие прилагательные могут модифицировать заданное существительное. В качестве примера рассмотрим следующее предложение:

```
I want a green apple.
```

На рис. 3.2 выделены интересующие нас синтаксические зависимости.

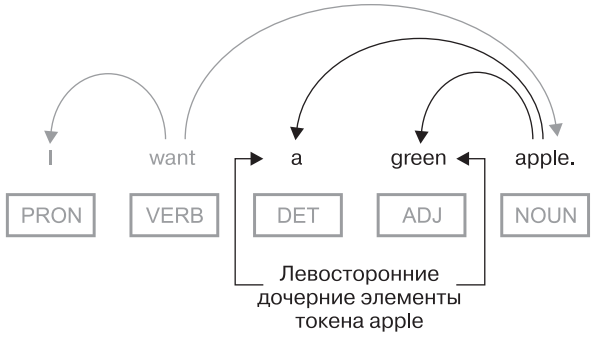


Рис. 3.2. Пример левосторонних синтаксических зависимостей

Для получения программным образом левосторонних дочерних элементов токена *apple* в данном предложении можно воспользоваться следующим кодом:

```
>>> doc = nlp(u'I want a green apple.')
>>> [w for w in doc[4].lefts]
[a, green]
```

В этом сценарии просто проходим в цикле по дочерним элементам токена *apple*, выводя их в список.

Интересно отметить, что здесь у слова *apple* есть только левосторонние синтаксические дочерние элементы. На практике это означает, что можно заменить атрибут `Token.lefts` на `Token.children`, служащий для поиска всех дочерних элементов токена:

```
>>> [w for w in doc[4].children]
```

В результате будет получен тот же список.

Можно также использовать атрибут `Token.rights` для получения правосторонних синтаксических дочерних элементов: в этом примере слово *apple* является правосторонним дочерним элементом слова *want*, как показано на рис. 3.1.

Контейнер doc.sents

Обычно смысл синтаксических меток, присваиваемых токенам, проявляется лишь в контексте предложения, где встречается данный токен. Например, информация о том, чем является слово — существительным или глаголом, может оказаться применимой лишь к содержащему это слово конкретному предложению (как в случае со словом *count*, которое мы обсуждали в предыдущих главах). В подобных ситуациях желательно иметь возможность обращаться к токенам документа по индексам уровня предложения.

С помощью свойства `doc.sents` объекта `Doc` текст можно разделить на отдельные предложения, как показано в следующем примере:

```
>>> doc = nlp(u'A severe storm hit the beach. It started to rain.')
❶ >>> for sent in doc.sents:
❷ ...   [sent[i] for i in range(len(sent))]
...
[A, severe, storm, hit, the, beach, .]
[It, started, to, rain, .]
>>>
```

Здесь проходим по предложениям из объекта `doc` ❶, создавая отдельный список токенов для каждого предложения ❷.

В то же время можно ссылаться на токены в состоящем из множества предложений тексте с помощью глобальных индексов уровня документа, как показано вот здесь:

```
>>> [doc[i] for i in range(len(doc))]
[A, severe, storm, hit, the, beach, ., It, started, to, rain, .]
```

Возможность ссылаться на объекты `Token` в документе по их индексам уровня предложения удобна, когда нужно, например, проверить, является ли первое слово во втором предложении обрабатываемого текста местоимением (если необходимо, скажем, выяснить связь между двумя предложениями, первое из которых содержит существительное, а второе — ссылающееся на это существительное местоимение):

```
>>> for i,sent in enumerate(doc.sents):
...   if i==1 and sent[0].pos_== 'PRON':
...     print('The second sentence begins with a pronoun.')
The second sentence begins with a pronoun.
```

В этом примере выбираем предложения по индексу с помощью перечислителя в цикле `for`: отфильтровав таким образом не интересующие нас предложения, проверяем только второе.

Выбор первого слова в предложении — элементарная задача, поскольку его индекс всегда равен 0. А как насчет последнего? Например, что делать, если необходимо определить, сколько предложений в тексте оканчивается глаголом (не считая точек и прочих знаков препинания)?

```
>>> counter = 0
>>> for sent in doc.sents:
...   if sent[len(sent)-2].pos_ == 'VERB':
...     counter+=1
>>> print(counter)
1
```

Хотя длины предложений различны, их можно легко вычислить с помощью функции `len()`. Вычитаем 2 из значения `len(sent)` по следующим причинам: во-первых, индексы всегда начинаются с 0 и заканчиваются на `size-1`, во-вторых, последний токен в обоих предложениях нашего примера текста — точка, которую не нужно учитывать.

Контейнер doc.noun_chunks

С помощью свойства `doc.noun_chunks` объекта `Doc` можно пройти по именным фрагментам. *Именной фрагмент* (noun chunk) — это фраза, главным элементом которой является существительное. Предложение *A noun chunk is a phrase that has a noun as its head* включает следующие именные фрагменты:

```
A noun chunk
a phrase
a noun
its head
```

Благодаря свойству `doc.noun_chunks` их можно извлечь следующим образом:

```
>>> doc = nlp(u'A noun chunk is a phrase that has a noun as its head.')
>>> for chunk in doc.noun_chunks:
...   print(chunk)
```

Или же для извлечения именных фрагментов можно обойти в цикле существительные в предложении и найти синтаксические дочерние элементы каждого существительного, чтобы из них образовать именные фрагменты. В подразделе «Обход в цикле синтаксических дочерних элементов токена» на с. 62 приводился пример извлечения фразы с помощью разбора синтаксических зависимостей. Попробуем применить эту методику к нашему тексту в данном примере, чтобы сформировать именные фрагменты вручную:

```
  for token in doc:
❶ if token.pos_=='NOUN':
    chunk = ''
   ❷ for w in token.children:
     ❸ if w.pos_ == 'DET' or w.pos_ == 'ADJ':
        chunk = chunk + w.text + ' '
   ❹ chunk = chunk + token.text
    print(chunk)
```

Проходим по токенам в коде и выбираем только существительные ❶. Далее, во внутреннем цикле, проходим по дочерним элементам существительных ❷, выбирая лишь токены — определяющие слова или

прилагательные для именного фрагмента (именные фрагменты могут включать и некоторые другие части речи, например наречия) ❸. Затем присоединяем существительное к полученному фрагменту ❹. В результате при выполнении этого сценария должно получиться то же, что и в предыдущем примере.

Попробуйте сами

Обратите внимание: используемые для модификации существительных слова (определители и прилагательные) всегда являются левосторонними дочерними элементами для существительного. Поэтому в предыдущем коде можно заменить `Token.children` на `Token.lefts` и убрать проверку, являются ли дочерние элементы определителями или прилагательными.

Перепишите предыдущий фрагмент кода, внеся в него вышеупомянутые изменения. Полученный в результате выполнения вашего кода набор именных фрагментов должен остаться прежним.

Объект Span

Объект `Span` (от англ. span — «интервал») представляет собой часть объекта `Doc`. В предыдущих разделах вы видели, как `Span` можно использовать в качестве контейнера для предложения и именного фрагмента, полученных из `doc.sents` и `doc.noun_chunks` соответственно.

Но применение объекта `Span` не ограничивается только исполнением роли контейнера для предложений и именных фрагментов. Этот объект можно использовать для хранения произвольного множества смежных токенов документа, указав диапазон индексов так, как показано в следующем примере:

```
>>> doc=nlp('I want a green apple.')
>>> doc[2:5]
a green apple
```

Объект `Span` включает несколько методов, самый интересный из которых — `span.merge()`. С его помощью интервал можно объединять

в единый токен, производя повторную токенизацию документа[1]. Это удобно, когда текст содержит названия из нескольких слов.

Предложение в следующем примере содержит два географических названия из нескольких слов, которые нам необходимо сгруппировать, — *Golden Gate Bridge* и *San Francisco*. При токенизации по умолчанию эти названия, состоящие из нескольких слов, не воспринимаются как единые токены.

Посмотрим, что выводится в списке токенов текста:

```
>>> doc = nlp(u'The Golden Gate Bridge is an iconic landmark in San
Francisco.')
>>> [doc[i] for i in range(len(doc))]
[The, Golden, Gate, Bridge, is, an, iconic, landmark, in, San, Francisco, .]
```

Каждому слову и знаку препинания соответствует отдельный токен.

С помощью метода `span.merge()` можно изменить поведение по умолчанию:

```
>>> span = doc[1:4]
>>> lem_id = doc.vocab.strings[span.text]
>>> span.merge(lemma = lem_id)
Golden Gate Bridge
```

В этом примере была создана лемма для интервала *Golden Gate Bridge*, после чего ее в качестве параметра передали в метод `span.merge()`. (Точнее говоря, передали идентификатор этой леммы, полученный с помощью атрибута `doc.vocab.strings`.)

Обратите внимание, что метод `span.merge()` не склеивает соответствующие леммы по умолчанию. Если вызвать его без параметров, лемме объединенного токена будет присвоено значение леммы первого токена объединяемого интервала. Для указания леммы, которую нужно присвоить объединенному токену, необходимо передать ей в метод `span.merge()` в качестве параметра `lemma`, как показано выше.

---

[1] Стоит отметить, впрочем, что этот метод считается устаревшим, начиная с версии 2.1.0 библиотеки spaCy. Вместо него рекомендуется использовать более эффективный и менее подверженный ошибкам метод Doc.retokenize.

Посмотрим, насколько корректно смогут обработать эту новую лемму лемматизатор и средства частеречной разметки и разбора зависимостей:

```
>>> for token in doc:
      print(token.text, token.lemma_, token.pos_, token.dep_)
```

Результат выполнения выглядит следующим образом:

```
The                the                DET   det
Golden Gate Bridge Golden Gate Bridge PROPN nsubj
is                 be                 VERB  ROOT
an                 an                 DET   det
iconic             iconic             ADJ   amod
landmark           landmark           NOUN  attr
in                 in                 ADP   prep
San                san                PROPN compound
Francisco          francisco          PROPN pobj
.                  .                  PUNCT punct
```

Все приведенные в листинге атрибуты были корректно присвоены токену *Golden Gate Bridge*.

Попробуйте сами

Предложение в предыдущем примере включает также *San Francisco* — еще одно географическое название из нескольких слов, которые желательно объединить в один токен. Для этого произведите операции, аналогичные проделанным в предыдущем фрагменте кода для интервала *Golden Gate Bridge*.

При указании начальной и конечной позиций интервала *San Francisco* не забудьте, что индексы токенов, расположенных справа от только что созданного токена *Golden Gate Bridge*, уменьшились на 2.

Настройка конвейера обработки текста под свои нужды

Из предыдущих разделов вы узнали, что объекты-контейнеры spaCy соответствуют различным лингвистическим единицам (например, целому тексту или отдельному токену), что позволяет извлекать линг-

вистические признаки этих единиц. Теперь же рассмотрим объекты API spaCy, предназначенные для создания контейнеров и заполнения их подходящими данными.

Эти объекты называют компонентами конвейера обработки. Как вы уже знаете, по умолчанию набор операций конвейера включает в себя процедуры частеречной разметки, разбора зависимостей и распознавания именованных сущностей. Посмотреть доступные для объекта `nlp` компоненты конвейера можно с помощью команды:

```
>>> nlp.pipe_names
['tagger', 'parser', 'ner']
```

В следующих разделах демонстрируется, каким образом spaCy позволяет пользователям настраивать компоненты конвейера под себя.

Отключение компонентов конвейера

spaCy позволяет выбирать загружаемые компоненты конвейера и отключать те, которые не нужны. Это можно сделать при создании объекта `nlp`, задав параметр `disable`:

```
nlp = spacy.load('en', disable=['parser'])
```

В данном случае мы создадим конвейер обработки без утилиты разбора зависимостей. При вызове такого экземпляра `nlp` для конкретного текста токены в этом тексте не получат метки зависимостей. Вышесказанное наглядно иллюстрирует следующий пример:

```
>>> doc = nlp(u'I want a green apple.')
>>> for token in doc:
...   print(❶token.text, ❷token.pos_, ❸token.dep_)
I     PRON
want  VERB
a     DET
green ADJ
apple NOUN
.     PUNCT
```

Вы попытались вывести для всех токенов предложения следующую информацию: текстовое содержимое ❶, тег части речи ❷ и метку зависимостей ❸. Однако метки зависимостей выведены не были.

Пошаговая загрузка модели

С помощью метода `spacy.load()`, загружающего модель, за один раз можно произвести несколько операций. Например, при выполнении оператора:

```
nlp = spacy.load('en')
```

библиотека spaCy осуществляет такие действия.

1. По названию загружаемой модели определяет, какой экземпляр класса `Language` следует инициализировать. В данном примере spaCy создает экземпляр подкласса `English`, включающий общий словарь и прочие языковые данные.
2. Проходит в цикле по названиям компонентов конвейера обработки, создает соответствующие компоненты и добавляет их в конвейер обработки.
3. Загружает данные модели с диска, делая их доступными для экземпляра класса `Language`.

Метод `spacy.load()` скрывает эти нюансы реализации, в большинстве случаев экономя ваши силы и время. Но иногда для более тонкой настройки процесса эти шаги приходится реализовывать явным образом: подобное, например, может понадобиться для подключения к конвейеру обработки пользовательского компонента для вывода какой-либо информации об объекте `Doc` (числа токенов, наличия/отсутствия определенных частей речи и т. д.).

Чем шире возможности настройки, тем больше информации требуется указать. Чтобы узнать путь к пакету модели, лучше не использовать сокращенное наименование, а получить настоящее название модели.

Узнать полное название можно следующим образом:

```
>>> print(nlp.meta['lang'] + '_' + nlp.meta['name'])
en_core_web_sm
```

Используемый в этом коде атрибут `nlp.meta` представляет собой ассоциативный массив с метаданными загруженной модели, из которых в данном случае нам нужны лишь язык модели и ее название.

Узнав название модели, вы можете выяснить, где именно она находится в вашей системе. Поможет вспомогательная функция `get_package_path`:

```
>>> from spacy import util
>>> util.get_package_path('en_core_web_sm')
PosixPath('/usr/local/lib/python3.5/site-packages/en_core_web_sm')
```

В зависимости от каталога установки Python путь может отличаться от приведенного в данном примере. Но еще в путь необходимо добавить один каталог, название которого состоит из названия модели и присоединенной в его конец версии модели (именно там располагается пакет модели). Определить название можно следующим образом:

```
>>> print(nlp.meta['lang'] + '_' + nlp.meta['name'] + '-' + nlp.
meta['version'])
en_core_web_sm-2.0.0
```

Иногда полезно взглянуть на список компонентов конвейера, используемых с моделью (важно знать, какие компоненты поддерживаются в контексте данной модели и могут быть загружены в конвейер). Список можно получить из поля `'pipeline'` атрибута `nlp.meta`, как показано ниже, или через атрибут `nlp.pipe_names`, о котором шла речь в начале раздела «Настройка конвейера обработки текста под свои нужды» на с. 68:

```
>>> nlp.meta['pipeline']
['tagger', 'parser', 'ner']
```

При наличии этой информации можно написать сценарий, реализующий шаги, описанные в начале данного раздела:

```
>>> lang = 'en'
>>> pipeline = ['tagger', 'parser', 'ner']
>>> model_data_path = '/usr/local/lib/python3.5/site-packages/en_core_web_sm/
en_core_web_sm-2.0.0'
❶ >>> lang_cls = spacy.util.get_lang_class(lang)
>>> nlp = lang_cls()
❷ >>> for name in pipeline:
❸ ...     component = nlp.create_pipe(name)
❹ ...     nlp.add_pipe(component)
❺ >>> nlp.from_disk(model_data_path)
```

В этом сценарии для загрузки класса `Language` используется метод `spacy.util.get_lang_class()` ❶. Какой именно подкласс загружается, зависит от двухбуквенного кода языка, указываемого в качестве

параметра. В данном случае мы загружаем подкласс для английского языка. Далее в цикле ❷ создаем ❸ и добавляем ❹ компоненты конвейера в конвейер обработки, после чего загружаем с диска модель, указывая путь к ней на вашей машине ❺.

По коду в сценарии может показаться, что компоненты конвейера становятся работоспособными сразу же после добавления их в конвейер обработки. На самом деле их нельзя использовать до тех пор, пока не будут загружены данные модели, поэтому без последней строки кода вы не сможете создать объект `Doc` с помощью этого экземпляра `nlp`.

Настройка компонентов конвейера под свои нужды

Настройка компонентов конвейера позволяет лучше решать задачи приложения. Допустим, система распознавания именованных сущностей вашей модели должна определять, что слово *Festy* означает один из районов города, а по умолчанию это слово считается названием организации, как показано в следующем примере:

```
>>> doc = nlp(u'I need a taxi to Festy.')
>>> for ent in doc.ents:
...  print(ent.text, ent.label_)
Festy ORG
```

Метка `ORG` обозначает различные компании, государственные бюро и прочие учреждения. Но нам нужно, чтобы средство распознавания сущностей классифицировало его как сущность типа `DISTRICT`.

Компонент для распознавания сущностей реализован в API библиотеки spaCy в виде класса `EntityRecognizer`. С помощью методов этого класса можно инициализировать экземпляр `ner` и применить его к тексту. В большинстве случаев описывать операции явным образом не нужно: spaCy делает это автоматически при создании объекта `nlp`, а затем и объекта `Doc` соответственно.

Но для внесения собственных примеров в систему распознавания сущностей уже существующей модели придется поработать с методами объекта `ner` явным образом.

В следующем примере сначала добавим новую метку DISTRICT в список поддерживаемых типов сущностей. Затем создадим обучающий пример данных и подадим его на вход средства распознавания сущностей, чтобы оно поняло, к чему должна относиться метка DISTRICT. Вот простейшая реализация этих подготовительных шагов:

```
LABEL = 'DISTRICT'
TRAIN_DATA = [
   ❶ ('We need to deliver it to Festy.', {
       ❷ 'entities': [(25, 30, 'DISTRICT')]
    }),
   ❸ ('I like red oranges', {
        'entities': []
    })
]
```

Для простоты обучающий набор состоит лишь из двух примеров данных (обычно их требуется гораздо больше). Каждый из обучающих примеров включает в себя предложение, содержащее или не содержащее интересующую нас сущность (сущности), которой должна присваиваться эта новая метка сущности ❶. Если в примере данных имеется нужная сущность, указываем ее начальную и конечную позиции ❷. Второе предложение в обучающем наборе данных вообще не содержит слова *Festy* ❸. Причина этого в том, как организован процесс обучения. В главе 10 этот процесс рассмотрен во всех подробностях.

Следующий этап — добавление новой метки сущности DISTRICT в компонент распознавания сущностей. Но сначала необходимо получить экземпляр компонента конвейера ner:

```
ner = nlp.get_pipe('ner')
```

Выполнив этот шаг, в полученный объект ner можно добавить новую метку с помощью метода ner.add_label():

```
ner.add_label(LABEL)
```

Прежде чем приступить к обучению средства распознавания сущностей, необходимо отключить остальные конвейеры, чтобы во время обучения обновлялся только компонент распознавания сущностей:

```
nlp.disable_pipes('tagger')
nlp.disable_pipes('parser')
```

Теперь можно начинать обучение компонента распознаванию сущностей на примерах данных из списка `TRAIN_DATA`, который был создан ранее в этом разделе:

```
optimizer = nlp.entity.create_optimizer()
import random

for i in range(25):
    random.shuffle(TRAIN_DATA)
    for text, annotations in TRAIN_DATA:
        nlp.update([text], [annotations], sgd=optimizer)
```

Примеры данных во время обучения демонстрируются модели в цикле в случайном порядке, чтобы как можно эффективнее обновлять данные модели и избегать каких-либо обобщающих выводов из очередности обучающих примеров. Выполнение этого кода займет некоторое время.

По завершении выполнения можно проверить, как обновленный оптимизатор распознает токен *Festy*:

```
>>> doc = nlp(u'I need a taxi to Festy.')
>>> for ent in doc.ents:
...  print(ent.text, ent.label_)
...
Festy DISTRICT
```

Как видим из выведенных результатов, все работает отлично.

Учтите: внесенные только что обновления будут утрачены сразу после закрытия текущего сеанса интерпретатора Python. Для решения этой проблемы в классе `Pipe`, родительском для класса `EntityRecognizer` и других классов компонентов конвейера, предусмотрен метод `to_disk()`, предназначенный для сериализации конвейера на диск:

```
>>> ner.to_disk('/usr/to/ner')
```

Теперь при необходимости можно загрузить обновленный компонент в новом сеансе, используя метод `from_disk()`. Чтобы убедиться в этом, закройте текущий сеанс интерпретатора, откройте новый и выполните следующий код:

```
>>> import spacy
>>> from spacy.pipeline import EntityRecognizer
❶ >>> nlp = spacy.load('en', disable=['ner'])
```

```
❷ >>> ner = EntityRecognizer(nlp.vocab)
❸ >>> ner.from_disk('/usr/to/ner')
❹ >>> nlp.add_pipe(ner)
```

Вы загрузили модель, отключив ее компонент `ner` по умолчанию ❶. Затем создали новый экземпляр `ner` ❷, после чего загрузили данные с диска ❸. Добавили компонент `ner` в конвейер обработки ❹.

Теперь можно проверить, что все работает:

```
>>> doc = nlp(u'We need to deliver it to Festy.')
>>> for ent in doc.ents:
...  print(ent.text, ent.label_)

Festy DISTRICT
```

Как видите, компонент распознавания сущностей маркирует название *Festy* правильно.

Я показал вам, как настраивать под свои нужды средство распознавания сущностей, но аналогичным образом можно настроить и прочие компоненты конвейера.

Использование структур данных уровня языка C библиотеки spaCy

Операции NLP, связанные с обработкой больших объемов текста, даже при использовании spaCy могут занимать существенное время. Например, чтобы составить список прилагательных, наиболее подходящих для конкретного существительного, придется просмотреть много текстовой информации. Если скорость обработки играет для приложения критическую роль, spaCy позволяет воспользоваться структурами данных и интерфейсами Cython уровня языка C. Cython — один из двух языков программирования, на которых написана библиотека spaCy (второй язык — Python). Поскольку Cython представляет собой расширение Python, то почти весь код на Python считается допустимым и на Cython. Помимо функциональности Python, Cython позволяет вызывать функции языка C нативным образом и объявлять типы языка C, благодаря чему компилятор может генерировать высокоэффективный код. Cython стоит использовать для ускорения длительных операций обработки.

Основные структуры данных библиотеки spaCy реализованы в виде объектов Cython, к которым можно обращаться через открытый API spaCy. Подробности вы найдете в документации на странице Cython Architecture («Архитектура Cython») по ссылке https://spacy.io/api/cython/.

Принципы работы

Для использования кода Cython с помощью spaCy необходимо преобразовать его в модуль расширения Python и импортировать в свою программу, как показано на рис. 3.3.

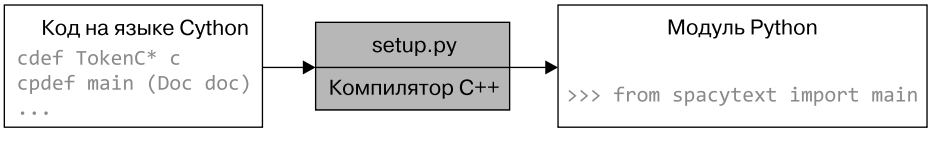


Рис. 3.3. Создание модуля расширения Python из сценария на языке Cython

Чтобы сделать это, сохраните код Cython в файле с расширением `.pyx` и запустите сценарий Python `setup.py`, который сначала преобразует код Cython в соответствующий код C или C++, а затем вызовет компилятор C или C++. Результатом работы этого сценария будет модуль расширения Python.

Подготовка рабочей среды и получение текстовых файлов

Прежде чем приступить к созданию кода Cython, нужно установить Cython на своей машине и найти для работы большой текстовый файл.

Установите Cython с помощью `pip`:

```
pip install Cython
```

Для имитации задачи, требующей больших временных затрат, в качестве объемного текстового файла подойдет, например, файл дампа «Википедии», содержащий набор страниц в XML. Файлы дампов «Википедии» можно скачать по ссылке https://dumps.wikimedia.org/enwiki/latest/. Прокрутите до файлов `enwiki-latest-pages-articles*.xml-*.bz2` и выберите из них подходящий по размеру для наших тестов. Однако

не выбирайте слишком большой файл, чтобы не пришлось ждать часами, пока машина выполнит код теста: файла размером 10–100 Мбайт будет вполне достаточно.

После скачивания файла извлеките из него необработанный текст с помощью соответствующей утилиты, например `gensim.corpora.wikicorpus` (https://radimrehurek.com/gensim/corpora/wikicorpus.html), которая специально предназначена для создания корпусов текста из файлов дампов «Википедии».

Сценарий Cython

Теперь напишем сценарий на Cython для анализа текстового файла. Для простоты допустим, что нам нужно подсчитать число личных местоимений в тексте, то есть нужно узнать число токенов, которым присвоена метка части речи `PRP`.

ВНИМАНИЕ

Как указано в документации, предназначенные для использования из кода на Cython методы уровня языка C ориентированы на скорость, а не на безопасность. Ошибки в их коде могут привести к аварийному прерыванию выполнения.

Создаем в одном из каталогов локальной файловой системы файл `spacytext.pyx` и вставляем в него следующий код:

```
from cymem.cymem cimport Pool
from spacy.tokens.doc cimport Doc
from spacy.structs cimport TokenC
from spacy.typedefs cimport hash_t

❶ cdef struct DocStruct:
      TokenC* c
      int length

❷ cdef int counter(DocStruct* doc, hash_t tag):
      cdef int cnt = 0
      for c in doc.c[:doc.length]:
         if c.tag == tag:
            cnt += 1
      return cnt
```

```
❸ cpdef main(Doc mydoc):
      cdef int cnt
      cdef Pool mem = Pool()
      cdef DocStruct* doc_ptr = <DocStruct*>mem.alloc(1, sizeof(DocStruct))
      doc_ptr.c = mydoc.c
      doc_ptr.length = mydoc.length
      tag = mydoc.vocab.strings.add('PRP')
      cnt = counter(doc_ptr, tag)
      print(doc_ptr.length)
      print(cnt)
```

Код начинается с нескольких операторов `cimport`, необходимых для импорта моделей Cython (в основном из библиотеки spaCy).

Далее описываем Cython-структуру `DocStruct` языка — контейнер для обрабатываемого текста и переменную `TokenC*` ❶ — указатель на структуру `TokenC`, которая используется в spaCy в качестве контейнера данных для объекта `Token`.

Затем описываем Cython-функцию `counter` ❷: она подсчитывает количество личных местоимений в тексте.

ПРИМЕЧАНИЕ

> В коде на языке Python, импортирующем модуль, функции cpdef недоступны. Чтобы функция была видима в Python и при этом использовала возможности структур данных и интерфейсов уровня языка C, необходимо объявить ее как cpdef.

Наконец, функцию `main` Cython/Python объявляем как `cpdef` ❸ и теперь можем ее использовать в коде на языке Python.

Сборка модуля Cython

Код на языке Cython, в отличие от написанного на Python, необходимо компилировать. Сделать это можно несколькими способами, но лучше всего написать сценарий Python `setup.py` для distutils/setuptools. Создаем файл `setup.py` в каталоге, где располагается наш сценарий Cython. Файл должен содержать следующий код:

```
from distutils.core import setup
from Cython.Build import cythonize
```

```
❶ import numpy

setup(name='spacy text app',
  ❷ ext_modules=cythonize("spacytext.pyx", language="c++"),
   ❸ include_dirs=[numpy.get_include()]
     )
```

Это обычный сценарий `setup.py` для distutils/setuptools, за исключением двух дополнений, связанных с нашим примером. Во-первых, мы импортировали `numpy` ❶, после чего явным образом указали, где искать заголовочные (`.h`) файлы библиотеки ❸ во избежание ошибки компиляции `numpy/arrayobject.h`, возникающей на некоторых системах. Во-вторых, мы воспользовались еще одной опцией — `language="c++"` ❷, указав тем самым на необходимость применить компилятор C++, а не C, как это происходит по умолчанию.

После подготовки установочного сценария компилируем код Cython. Сделать это можно из системного терминала:

```
python setup.py build_ext --inplace
```

Во время компиляции вы увидите ряд сообщений, часть из которых может быть предупреждениями, однако, как правило, некритичными. Возможно, вы увидите такое сообщение:

```
# warning "Using deprecated NumPy API ...
```

Тестирование модуля

После успешного завершения процесса компиляции модуль `spacytext` будет добавлен в среду Python. Для его тестирования откройте сеанс Python и выполните команду:

```
>>> from spacytext import main
```

Если при этом не будет возвращено никаких ошибок, можете ввести следующие команды (они предполагают, что ваши текстовые данные располагаются в файле `test.txt`):

```
  >>> import spacy
  >>> nlp = spacy.load('en')
❶ >>> f= open("test.txt","rb")
  >>> contents =f.read()
❷ >>> doc = nlp(contents[:100000].decode('utf8'))
```

```
❸ >>> main(doc)
  21498
  216
```

Открываем файл с текстовыми данными в двоичном режиме для получения байтового объекта ❶. В случае если файл слишком большой, при создании объекта `Doc` ❷ получится прочитать лишь часть его содержимого. После создания объекта `Doc` можно проверить работу недавно созданного с помощью Cython модуля `spacytext`, вызвав его функцию `main()` ❸.

Первое выведенное в результате выполнения функции `spacytext.main()` число означает общее количество токенов, найденных в тексте. Второе число представляет собой количество обнаруженных там же личных местоимений.

Резюме

В этой главе мы рассмотрели важнейшие объекты-контейнеры библиотеки spaCy. Кроме того, вы научились настраивать компоненты конвейера под свои нужды, а также использовать структуры данных и интерфейсы уровня языка C из Cython.

4 Выделение и использование лингвистических признаков

В предыдущих главах вы узнали, как в конвейере обработки получить доступ к лингвистическим признакам — тегам частей речи, синтаксическим зависимостям и именованным сущностям. В этой главе вы научитесь выделять и генерировать текст на основе тегов частей речи и синтаксических зависимостей, что позволит создавать отвечающие на вопросы чат-боты, находить в тексте конкретные фразы и многое другое.

Практически всем NLP-приложениям приходится выделять из текста определенную информацию и генерировать новый текст под конкретную ситуацию. Например, чат-бот должен поддерживать разговор с пользователем, а значит, должен уметь распознавать конкретные части вводимого пользователем текста и реагировать на них должным образом. Давайте взглянем, как все это можно осуществить с помощью лингвистических признаков.

Выделение и генерация текста с помощью тегов частей речи

Теги частей речи полезны для извлечения конкретного вида информации из текста, а также для генерации новых предложений на основе предложения, введенного пользователем. В этом разделе вы познакомитесь с несколькими новыми тегами частей речи и напишете сценарий

для поиска фраз, описывающих денежные суммы, и преобразования утвердительных высказываний в вопросительные. Список тегов частей речи, которые часто используются для моделей английского языка в spaCy, можно найти в табл. 2.1.

Теги для чисел, символов и знаков препинания

Помимо тегов частей речи для существительных, глаголов и прочих слов, в библиотеке spaCy есть теги для символов, чисел и знаков препинания. Посмотрим на них на примере обработки следующего предложения:

```
The firm earned $1.5 million in 2017.
```

Для начала выделим из токенов признаки общих частей речи и увидим, как spaCy распознает различные части речи:

```
>>> import spacy
>>> nlp = spacy.load('en')
>>> doc = nlp(u"The firm earned $1.5 million in 2017.")
>>> for token in doc:
...   print(token.text, ❶token.pos_, ❷spacy.explain(token.pos_))
...
```

Мы создали для входного предложения объект `Doc` и вывели теги общих частей речи ❶, а также воспользовались функцией `spacy.explain()`, которая возвращает описание для заданного лингвистического признака ❷.

Результаты выполнения сценария выглядят следующим образом:

```
The     DET   determiner
firm    NOUN  noun
earned  VERB  verb
$       SYM   symbol
1.5     NUM   numeral
million NUM   numeral
in      ADP   adposition
2017    NUM   numeral
.       PUNCT punctuation
```

Обратите внимание на отдельные теги общих частей речи для числительных, символов и знаков препинания. Как видите, было распознано даже числительное «миллион» в буквенном виде.

Теперь сравним теги общих и уточненных частей речи для того же предложения, выведя в отдельном столбце описание для тегов уточненных частей речи:

```
>>> for token in doc:
...   print(token.text, token.pos_, token.tag_, spacy.explain(token.tag_))
```

Результаты должны выглядеть так:

```
The     DET   DT   determiner
firm    NOUN  NN   noun, singular
earned  VERB  VBD  verb, past tense
$       SYM   $    symbol, currency
1.5     NUM   CD   cardinal number
million NUM   CD   cardinal number
in      ADP   IN   conjunction, subordinating or preposition
2017    NUM   CD   cardinal number
.       PUNCT .    punctuation mark, sentence closer
```

Второй и третий столбцы содержат теги общих и уточненных частей речи соответственно. В четвертом столбце приведено описание тегов уточненных частей речи из третьего столбца.

С помощью тегов уточненных частей речи каждая категория разбивается на подкатегории. Например, у общей категории `SYM` (символы) есть три уточненные подкатегории: `$` для обозначения валют, `#` — для символа «решетка» и `SYM` — для всех прочих символов, таких как `+`, `–`, `×`, `÷`, `=`. Подобное разделение удобно, когда нужно различать отдельные типы символов. Например, при обработке математических статей сценарий должен распознавать символы, часто встречающиеся в математических формулах, а сценарий для обработки финансовых отчетов должен распознавать обозначения валют.

ПРИМЕЧАНИЕ

Поскольку средство частеречной разметки библиотеки spaCy генерирует метки на основе контекста токенов, в различных контекстах токены могут получать различные метки.

Теперь посмотрим, как эти теги частей речи используются для выделения и генерации текста.

Выделение описаний денежных сумм

Предположим, нам нужно создать приложение для обработки финансовых отчетов, которое должно из длинных скучных текстов выделять фрагменты с необходимой информацией. На практике финансовые отчеты бывают достаточно большими, но нас в них интересуют только цифры. В частности, нам нужны фразы, описывающие количество денег и начинающиеся с символа валюты. Например, наш сценарий должен извлечь из предыдущего примера только фразу `"$1.5 million"`, а не `"2017"`.

Следующий сценарий иллюстрирует, как фразу `"$1.5 million"` можно выделить из предложения на основе одних лишь тегов частей речи токенов. Можете сохранить этот сценарий в файле и затем выполнить код из сеанса Python:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"The firm earned $1.5 million in 2017.")
phrase = ''
❶ for token in doc:
 ❷  if token.tag_ == '$':
        phrase = token.text
        i = token.i+1
    ❸  while doc[i].tag_ == 'CD':
           phrase += doc[i].text + ' '
           i += 1
    ❹  break
phrase = phrase[:-1]
print(phrase)
```

В этом коде проходим по токенам предложения в цикле ❶ в поисках токена с тегом `$` уточненной части речи ❷. Данный тег обозначает символ валюты, с которого обычно начинается фраза, описывающая некоторую сумму денег. Найдя символ валюты, начинаем составлять фразу, но при этом проверяем, являются ли числами токены, следующие за символом: для этого реализуем цикл `while`, в котором по очереди проверяем теги токенов, расположенных справа от символа валюты, на равенство тегу `CD` — тегу уточненной части речи для кардинального числа ❸. Достигнув первого нечислового токена, выходим из цикла `while`, а также завершаем внешний цикл `for` ❹, в котором проходим по токенам предложения.

Результат выполнения данного сценария выглядит следующим образом:

```
$1.5 million
```

Именно его и нужно было получить.

Учтите, что символом валюты, которому присвоен тег уточненной части речи `$`, не обязательно окажется $. Этот тег части речи может маркировать и другие символы валют, например £ и €. Предыдущий сценарий распознал бы и фразу £1 500 000.

Попробуйте сами

Приведенный выше сценарий выделяет из предложения одну фразу, которая описывает денежную сумму. После обнаружения фразы выполнение сценария прекращается. Но на практике предложения могут содержать несколько таких фраз, как показано в следующем примере: *The firm earned $1.5 million in 2017, in comparison with $1.2 million in 2016*.

Модифицируйте этот сценарий так, чтобы он выделял из предложения все фразы, описывающие денежные суммы. Для этого уберите оператор `break` — тогда цикл не будет заканчиваться после обнаружения первой такой фразы. Далее перенесите внутрь цикла код, отвечающий за подготовку и вывод найденной фразы (последние две строки сценария), чтобы вызывать эти две строки для каждой из найденных в предложении фраз.

Преобразование утвердительных высказываний в вопросительные

Допустим, наше NLP-приложение должно генерировать вопросы на основе полученных утвердительных высказываний. Чат-боты могут поддерживать разговор с пользователем, задавая ему уточняющие вопросы. Скажем, чат-бот может спросить у пользователя: «Ты действительно уверен?» — в ответ на пользовательское: «Я уверен». Для этого чат-бот должен уметь формировать соответствующие вопросы.

Пусть пользователь написал следующее:

```
I can promise it is worth your time.
```

Предложение содержит несколько глаголов и местоимений, причем с различной морфологией. Чтобы в этом убедиться, взглянем на теги частей речи, которые spaCy присваивает токенам предложения:

```
>>> doc = nlp(u"I can promise it is worth your time.")
>>> for token in doc:
...   print(token.text, token.pos_, token.tag_)
...
```

Выведем токены, а также их теги общих и уточненных частей речи, в результате чего получим следующее:

```
I       PRON   PRP
can     VERB   MD
promise VERB   VB
it      PRON   PRP
is      VERB   VBZ
worth   ADJ    JJ
your    ADJ    PRP$
time    NOUN   NN
.       PUNCT  .
```

По тегам уточненных частей речи можно различить морфологические категории глаголов и местоимений, присутствующих в предложении. Например, тег уточненной части речи `PRP` обозначает личные местоимения, а тег `PRP$` — притяжательные, благодаря чему эти два типа местоимений можно различить программным образом. Данная информация пригодится во время работы над примером.

Подтверждающий вопрос для вышеупомянутого предложения может выглядеть так (разумеется, для другого утверждения понадобится другой вопрос):

```
Can you really promise it is worth my time?
```

С точки зрения человека, формирование вопроса по приведенному утверждению — элементарная задача: нужно поменять порядок некоторых слов, соответствующим образом изменить местоимения и добавить к главному глаголу наречие-модификатор *really* (следующий сразу за подлежащим). Но как это реализовать программным способом?

Давайте взглянем на некоторые теги частей речи. В нашем предложении в формировании вопроса участвуют глаголы *can* и *promise*. Первый

из них помечен тегом уточненной части речи MD, который означает вспомогательный модальный глагол, а второй — тегом для глагола в неопределенной форме. Обратите внимание, что в предыдущем подтверждающем вопросе вспомогательный модальный глагол и местоимение поменялись местами — произошла так называемая *инверсия*. Нам придется реализовать это в сценарии.

В отношении местоимений чат-бот должен следовать шаблону, применяемому в обычном разговоре. В табл. 4.1 кратко описываются способы использования местоимений в подобном приложении.

Таблица 4.1. Использование местоимений в чат-боте

	Личные местоимения	**Притяжательные местоимения**
Чат-бот	I, me	my, mine
Пользователь	you	your, yours

Другими словами, чат-бот говорит о себе словами *I* или *me*, а о пользователе — *you*.

Ниже приведены основные шаги генерации вопроса из исходного утверждения.

1. Поменять порядок слов в исходном предложении с «подлежащее + + вспомогательный модальный глагол + глагол в неопределенной форме» на «модальный вспомогательный глагол + глагол в неопределенной форме + подлежащее».
2. Заменить личное местоимение *I* (подлежащее в предложении) на *you*.
3. Заменить притяжательное местоимение *your* на *my*.
4. Вставить наречие-модификатор *really* перед словом *promise* для усиления последнего.
5. Заменить знак препинания . на ? в конце предложения.

Эти шаги реализованы в следующем сценарии:

```
import spacy
nlp = spacy.load('en')
```

```
doc = nlp(u"I can promise it is worth your time.")
sent = ''
for i,token in enumerate(doc):
❶ if token.tag_ == 'PRP' and doc[i+1].tag_ == 'MD' and doc[i+2].tag_ == 'VB':
    ❷ sent =  doc[i+1].text.capitalize() + ' ' + doc[i].text
       sent = sent + ' ' + ❸doc[i+2:].text
    ❹ break
# К этому моменту предложение должно выглядеть так:
# 'Can I promise it is worth your time.'
# Повторная токенизация
❺ doc=nlp(sent)
for i,token in enumerate(doc):
 ❻ if token.tag_ == 'PRP' and token.text == 'I':
       sent = doc[:i].text + ' you ' +  doc[i+1:].text
       break
# К этому моменту предложение должно выглядеть так:
# 'Can you promise it is worth your time.'
doc=nlp(sent)
for i,token in enumerate(doc):
 ❼ if token.tag_ == 'PRP$' and token.text == 'your':
       sent = doc[:i].text + ' my ' +  doc[i+1:].text
       break
# К этому моменту предложение должно выглядеть так:
# 'Can you promise it is worth my time.'
doc=nlp(sent)
for i,token in enumerate(doc):
  if token.tag_ == 'VB':
    ❽ sent = doc[:i].text + ' really ' +  doc[i:].text
      break
# К этому моменту предложение должно выглядеть так:
# 'Can you really promise it is worth my time.'
doc=nlp(sent)
❾ sent = doc[:len(doc)-1].text + '?'
# Наконец, мы получаем: 'Can you really promise it is worth my time?'
print(sent)
```

Первые четыре шага реализованы в отдельных циклах `for`. Прежде всего проходим в цикле по токенам предложения и меняем местами существительное и глагол, чтобы предложение стало вопросительным. В данном примере находим следующий за личным местоимением вспомогательный модальный глагол (с тегом `MD`), за которым, в свою очередь, следует глагол в неопределенной форме ❶. Найдя эту последовательность слов, ставим вспомогательный модальный глагол перед личным местоимением, помещая его, таким образом, в начало предложения ❷.

Для формирования нового предложения применим методику, которую в языке Python называют *созданием срезов* (slicing). С ее помощью из объекта последовательности можно извлечь подпоследовательность, например строку или список, просто указав начальный и конечный индексы. В данном случае методом срезов извлекаем нужные нам подпоследовательности токенов из объекта `Doc`. Например, срез `doc[2:]` содержит токены документа, начиная с токена с индексом 2 и до конца объекта `doc`, то есть *promise it is worth your time* ❸. После перемещения модального глагола на новое место выходим из цикла `for` ❹.

Возможно, вы недоумеваете: почему бы просто не воспользоваться индексами личного местоимения и вспомогательного модального глагола? Раз нам известно, что индекс личного местоимения — 0, а модального глагола — 1, зачем проходить в цикле по всему набору токенов в поисках позиции модального глагола? Разве сказуемое не всегда следует за подлежащим, то есть является вторым словом в предложении?

Дело в том, что предложение не обязательно начинается с подлежащего. Например, в таком случае:

Sure enough, I can promise it is worth your time.

сценарий пропустит первые два слова и начнет обработку с подлежащего.

В результате инверсии получаем новое предложение в виде строки. Для дальнейшей обработки ее необходимо преобразовать в объект `Doc` ❺.

Далее создаем новый цикл `for`, который заменит личное местоимение *I* личным местоимением *you*. Для этого ищем личные местоимения (помеченные тегами `PRP`). Если личное местоимение — *I*, меняем его на *you* ❻ и выходим из цикла `for`.

Повторяем этот процесс. Ищем тег `PRP$` ❼ и меняем притяжательное местоимение *your* на *my*. В новом цикле `for` находим глагол в неопределенной форме и вставляем перед ним наречие-модификатор *really* ❽.

Наконец, заменяем точку в конце предложения вопросительным знаком: это единственный шаг, для которого не нужен цикл. Дело в том, что во всех возможных предложениях точка и вопросительный знак

располагаются в конце предложения, так что их можно с уверенностью найти по индексу `len(doc)-1` ❾.

При выполнении кода будет выведен следующий результат:

```
Can you really promise it is worth my time?
```

Для начала этот сценарий неплох, но подходит он не для каждого утвердительного высказывания. Кроме *I* утверждение может содержать и другие личные местоимения — но наш сценарий это не проверит. А некоторые предложения не содержат вспомогательных глаголов, например такое: *I love eating ice cream*. Для формирования вопроса в подобных случаях вместо слов *can* или *should* необходимо использовать слово *do*: *Do you really love eating ice cream?*. Но в предложениях с глаголом *to be* (как *I am sleepy*) его придется переставить в начало: *Are you sleepy?*.

Настоящая реализация такого чат-бота должна уметь выбирать подходящий вариант для введенного предложения. Пример с *do* вы увидите в подразделе «Выясняем, какой вопрос должен задать чат-бот» на с. 93.

Попробуйте сами

Возможно, изучая сценарий из подраздела «Преобразование утвердительных высказываний в вопросительные», вы обратили внимание, что некоторые блоки кода очень похожи, а производимые в них операции повторяются. На всех шагах в предложении производится замена и последующая повторная токенизация. Это значит, что можно попытаться обобщить этот код и вынести повторяющиеся операции в отдельную функцию.

Прежде чем ее писать, подумайте, какие параметры она должна иметь, чтобы выполнять показанные в сценарии выше операции над текстом. В частности, необходимо явным образом указать, какой токен нам нужен и какую операцию мы хотим с ним выполнить (заменить его другим токеном или добавить перед ним другой токен).

После описания этой функции напишите основной код, который бы ее вызывал, реализуя в итоге ту же функциональность, что и исходный сценарий.

Использование меток синтаксических зависимостей при обработке текста

Как вы узнали из раздела «Выделение и генерация текста с помощью тегов частей речи» на с. 81, теги частей речи — весьма мощный инструмент для интеллектуальной обработки текста. Но на практике для этого может понадобиться больше информации о токенах предложения.

Например, часто необходимо знать, чем является личное местоимение в предложении: подлежащим или дополнением. Иногда это несложно определить. Личные местоимения *I*, *he*, *she*, *they* и *we* практически всегда выступают в роли подлежащего. При использовании в качестве дополнения *I* превращается в *me*, как в предложении *A postman brought me a letter*.

Но с некоторыми другими личными местоимениями, например *you* или *it*, которые выглядят одинаково и в роли подлежащего и в роли дополнения, не всегда все очевидно. Рассмотрим два предложения: *I know you*. *You know me*. В первом предложении *you* является прямым дополнением глагола *know*. Во втором же *you* является подлежащим.

Попробуем решить эту задачу с помощью меток синтаксических зависимостей и тегов частей речи, после чего, опять же с помощью меток синтаксических зависимостей, создадим усовершенствованную версию нашего чат-бота, отвечающего на вопросы.

Различаем подлежащие и дополнения

Чтобы определить программным образом, чем в заданном предложении являются такие местоимения, как *you* или *it*, необходимо посмотреть на присвоенную им метку зависимости. Теги частей речи в сочетании с метками зависимостей позволяют получить гораздо больше информации о роли токена в предложении.

Вернемся к предложению из предыдущего примера и взглянем на результаты разбора зависимостей в нем:

```
>>> doc = nlp(u"I can promise it is worth your time.")
>>> for token in doc:
...   print(token.text, token.pos_, token.tag_, token.dep_, spacy.
explain(token.dep_))
```

Для токенов предложения были извлечены теги частей речи, метки зависимостей и их описание:

```
I       PRON  PRP  nsubj     nominal subject
can     VERB  MD   aux       auxiliary
promise VERB  VB   ROOT      None
it      PRON  PRP  nsubj     nominal subject
is      VERB  VBZ  ccomp     clausal complement
worth   ADJ   JJ   acomp     adjectival complement
your    ADJ   PRP$ poss      possession modifier
time    NOUN  NN   npadvmod  noun phrase as adverbial modifier
.       PUNCT .    punct     punctuation
```

Второй и третий столбцы содержат теги общих и уточненных частей речи соответственно. Четвертый столбец содержит метки зависимостей, а пятый — описания этих меток.

Сочетание тегов частей речи с метками зависимостей демонстрирует более ясную, чем теги частей речи или метки зависимости по отдельности, картину грамматической роли каждого из токенов в предложении. В данном примере тег части речи `VBZ`, присвоенный токену *is*, означает глагол третьего лица единственного числа настоящего времени, в то время как присвоенная тому же токену метка зависимости `ccomp` указывает, что *is* — это клаузальное дополнение (зависимое придаточное предложение с внутренним подлежащим). Здесь *is* представляет собой клаузальное дополнение глагола *promise* с внутренним подлежащим *it*.

Чтобы определить роль *you* в *I know you. You know me*, взглянем на следующий список тегов частей речи и меток зависимостей, присвоенных токенам:

```
I     PRON  PRP  nsubj  nominal subject
know  VERB  VBP  ROOT   None
you   PRON  PRP  dobj   direct object
.     PUNCT .    punct  punctuation
You   PRON  PRP  nsubj  nominal subject
know  VERB  VBP  ROOT   None
me    PRON  PRP  dobj   direct object
.     PUNCT .    Punct  punctuation
```

В обоих случаях токену *you* присвоены одни и те же теги частей речи: `PRON` и `PRP` (общий и уточненный соответственно). Но метки зависимости в этих двух случаях различны: `dobj` в первом предложении и `nsubj` — во втором.

Выясняем, какой вопрос должен задать чат-бот

Иногда для извлечения необходимой информации приходится обходить дерево зависимостей предложения. Рассмотрим следующий диалог между чат-ботом и пользователем:

```
User: I want an apple.
Bot: Do you want a red apple?
User: I want a green apple.
Bot: Why do you want a green one?
```

Чат-бот способен продолжать разговор, просто задавая вопросы. Но обратите внимание, что в выяснении того, какой вопрос ему следует задать, ключевую роль играет наличие/отсутствие прилагательного-модификатора.

В английском языке существует два основных типа вопросов: вопросы типа «да/нет» и информационные вопросы. Возможных ответов на вопросы типа «да/нет» (наподобие сгенерированного в примере из подраздела «Преобразование утвердительных высказываний в вопросительные» на с. 85) может быть только два: да или нет. Чтобы сформулировать подобный вопрос, необходимо поставить вспомогательный модальный глагол перед подлежащим, а смысловой глагол — после подлежащего. Например: *Could you modify it?*

Информационные вопросы предполагают развернутый ответ, а не только да/нет. Они начинаются с вопросительного слова, например с *what*, *where*, *when*, *why* или *how*. Далее процесс формирования информационного вопроса не отличается от процесса с вопросом типа «да/нет». Например: *What do you think about it?*

В первом случае в предыдущем примере с *apple* чат-бот задает вопрос типа «да/нет». Во втором случае, когда пользователь добавляет к слову *apple* модификатор *green*, чат-бот формулирует информационный вопрос.

Краткая сводка этого подхода приведена на рис. 4.1.

Следующий сценарий анализирует введенное предложение, выбирая, какой вид вопроса задать, после чего формирует соответствующий вопрос. Код этого сценария мы рассмотрим по частям в различных

разделах, но программу целиком я рекомендую сохранить в одном файле с названием `question.py`.

Рис. 4.1. Наличие/отсутствие модификатора во входном предложении определяет, какой вопрос задает чат-бот

Начнем с импорта модуля `sys`, который позволяет получить предложение в виде аргумента для дальнейшей обработки:

```
import spacy
import sys
```

Это шаг вперед по сравнению с предыдущими сценариями, в которых мы жестко «зашивали» анализируемое предложение. Теперь пользователи могут подавать на вход собственные предложения.

Далее опишем функцию для распознавания и извлечения произвольного именного фрагмента — прямого дополнения из входного документа. Например, если вы ввели документ, содержащий предложение *I want a green apple*., то будет возвращен фрагмент *a green apple*:

```
def find_chunk(doc):
  chunk = ''
❶ for i,token in enumerate(doc):
  ❷ if token.dep_ == 'dobj':
    ❸ shift = len([w for w in token.children])
    ❹ #print([w for w in token.children])
    ❺ chunk = doc[i-shift:i+1]
      break
  return chunk
```

Проходим в цикле по токенам введенного предложения ❶ и, проверяя теги зависимостей на равенство `dobj` ❷, ищем такой токен, который выступал бы в роли прямого дополнения. В предложении *I want a green*

apple. прямым дополнением является существительное *apple*. После обнаружения прямого дополнения необходимо определить элементы, являющиеся для него синтаксически дочерними ❸, поскольку именно из них состоит фрагмент, на основе которого будет определяться тип задаваемого вопроса. В целях отладки полезно вывести на экран дочерние элементы этого прямого дополнения ❹.

Для выделения нужного фрагмента производим срез объекта `Doc`, вычисляя начальный и конечный индексы. Начальный индекс равен индексу найденного прямого дополнения минус число его синтаксических дочерних элементов: как вы, возможно, догадались, он представляет собой индекс крайнего слева дочернего элемента. Конечный индекс равен индексу прямого дополнения плюс один, так что последним включаемым в искомый фрагмент токеном и является это прямое дополнение ❺.

Проще говоря, реализованный в сценарии алгоритм предполагает, что у прямого дополнения есть только левосторонние дочерние элементы. В действительности это не всегда так. Например, в предложении *I want to touch a wall painted green*. необходимо проверять и левосторонние, и правосторонние дочерние элементы прямого дополнения *wall*. Кроме того, поскольку *green* не является прямым дочерним элементом *wall*, необходимо обойти дерево зависимостей, чтобы определить, является ли *green* модификатором *wall*.

Следующая функция просматривает фрагмент и определяет, какой тип вопроса должен задать чат-бот:

```
def determine_question_type(chunk):
❶ question_type = 'yesno'
  for token in chunk:
 ❷ if token.dep_ == 'amod':
   ❸ question_type = 'info'
  return question_type
```

Сначала задаем начальное значение переменной `question_type` равным `'yesno'`, что соответствует вопросу типа «да/нет» ❶. Далее в переданном в функцию `chunk` ищем токен с тегом `amod`, который означает прилагательное-модификатор ❷. Если таковое находится, меняем значение переменной `question_type` на `'info'`, соответствующее информационному типу вопроса ❸.

Определив, какой тип вопроса нам нужен, генерируем в следующей функции вопрос на основе входного предложения:

```
def generate_question(doc, question_type):
  sent = ''
  for i,token in enumerate(doc):
    if token.tag_ == 'PRP' and doc[i+1].tag_ == 'VBP':
      sent =  'do ' + doc[i].text
      sent = sent + ' ' + doc[i+1:].text
      break
  doc=nlp(sent)
  for i,token in enumerate(doc):
    if token.tag_ == 'PRP' and token.text == 'I':
      sent = doc[:i].text + ' you ' +  doc[i+1:].text
      break
  doc=nlp(sent)
❶ if question_type == 'info':
    for i,token in enumerate(doc):
      if token.dep_ == 'dobj':
        sent = 'why ' + doc[:i].text + ' one ' +  doc[i+1:].text
        break
❷ if question_type == 'yesno':
    for i,token in enumerate(doc):
      if token.dep_ == 'dobj':
     ❸ sent = doc[:i-1].text + ' a red ' +  doc[i:].text
        break
  doc=nlp(sent)
  sent = doc[0].text.capitalize() +' ' + doc[1:len(doc)-1].text + '?'
  return sent
```

В серии циклов `for` превращаем входное утверждение в вопрос, производя инверсию и замену личных местоимений. Для формирования вопроса перед личным местоимением добавляем глагол *do*, поскольку в утверждении отсутствует вспомогательный модальный глагол. (Напомню, что такой алгоритм годится лишь для определенных предложений; в более полной реализации необходимо программным образом определять, какой подход к обработке использовать.)

Если значение переменной `question_type` равно `'info'`, добавляем слово *why* в начало вопроса ❶. Если значение переменной `question_type` равно `'yesno'` ❷, вставляем прилагательное для модификации прямого дополнения в вопросе. В данном примере ради простоты мы жестко «зашили» прилагательное в код, выбрав для этого прилагательное *red* ❸, которое в некоторых предложениях будет выглядеть

странно. Например, можно сказать *Do you want a red orange?*, но никак не *Do you want a red idea?*. В более совершенной реализации такого чат-бота необходимо определить программным образом подходящее прилагательное для модификации прямого дополнения. Этот вопрос будет рассмотрен в главе 6.

Обратите внимание: используемый алгоритм предполагает, что входное предложение оканчивается знаком препинания, например . или !.

После описания всех функций посмотрим на основной блок сценария:

```
❶ if len(sys.argv) > 1:
    sent = sys.argv[1]
    nlp = spacy.load('en')
  ❷ doc = nlp(sent)
  ❸ chunk = find_chunk(doc)
  ❹ if str(chunk) == '':
      print('The sentence does not contain a direct object.')
      sys.exit()
  ❺ question_type = determine_question_type(chunk)
  ❻ question = generate_question(doc, question_type)
    print(question)
  else:
    print('You did not submit a sentence!')
```

Прежде всего проверяем, передал ли пользователь предложение в виде аргумента командной строки ❶. Если да, то применяем к аргументу конвейер spaCy, создавая экземпляр объекта `Doc` ❷.

Далее передаем этот `doc` в функцию `find_chunk`, которая должна вернуть содержащий прямое дополнение именной фрагмент, например *a green apple*, для дальнейшей обработки ❸. Если же во входном предложении такого именного фрагмента нет ❹, мы получим сообщение *The sentence does not contain a direct object*.

Затем передаем только что выделенный фрагмент в функцию `determine_question_type`, которая, проанализировав его структуру, определяет, какой вопрос задать ❺.

Наконец, передаем входное предложение и определенный нами тип вопроса в функцию `generate_question`, генерирующую соответствующий вопрос и возвращающую его в виде строки ❻.

Выводимый сценарием результат зависит от введенного предложения. Вот несколько возможных вариантов:

```
❶ $ python question.py 'I want a green apple.'
  Why do you want a green one?
❷ $ python question.py 'I want an apple.'
  Do you want a red apple?
❸ $ python question.py 'I want...'
  The sentence does not contain a direct object.
❹ $ python question.py
  You did not submit a sentence!
```

Если в качестве аргумента командной строки передать предложение, содержащее прилагательное-модификатор (например, *green* для прямого дополнения наподобие *apple*), сценарий должен сгенерировать информационный вопрос ❶.

Если предложение содержит прямое дополнение без прилагательного-модификатора, сценарий должен ответить вопросом типа «да/нет» ❷.

Если же ввести предложение без прямого дополнения, сценарий должен сразу это заметить и предложить повторный ввод ❸.

Наконец, если вы забыли передать предложение, сценарий также должен вернуть соответствующее сообщение ❹.

Попробуйте сами

Как отмечалось ранее, сценарий, о котором шла речь в предыдущем разделе, годится не для всех предложений. Для формирования вопроса он добавляет глагол *do*, что подходит только для предложений без вспомогательного модального глагола.

Попробуйте расширить функциональность этого сценария для работы с утверждениями, содержащими вспомогательные модальные глаголы. Например, получив на входе утверждение *I might want a green apple.*, сценарий должен вернуть *Why might you want a green one?*. Подробнее о том, как преобразовать в вопрос утвердительное высказывание, содержащее вспомогательный модальный глагол, вы прочитали в разделе «Преобразование утвердительных высказываний в вопросительные» на с. 85.

Резюме

Лингвистические признаки — краеугольный камень всех задач NLP. В этой главе у вас была возможность изучить некоторые методики интеллектуальной обработки и генерации текста на основе лингвистических признаков. Вы узнали, как выделять фразы определенного типа (например, описывающие конкретные суммы денег) и написали сценарий с использованием меток зависимости и тегов частей речи для генерации осмысленных ответов на вводимые пользователем предложения.

Разговор о лингвистических признаках продолжится в главе 6, где нам предстоит работать с ними в более сложных сценариях.

5 Работа с векторами слов

Наборы вещественных чисел, отражающие смысл слов естественного языка, называются векторами слов: благодаря им, как вы знаете из главы 1, машины понимают естественные языки. В этой главе векторы слов будут использоваться для вычисления семантического подобия различных текстов, что позволит категоризировать тексты в соответствии с их тематикой.

Начнем с общего представления о векторах слов и математическом вычислении семантического подобия между словами, представленными в форме векторов. Далее рассмотрим применение алгоритмов машинного обучения для генерации векторов слов, реализованных в моделях spaCy. Воспользуемся *методом подобия* (similarity method) библиотеки spaCy, который определяет близость смыслов объектов-контейнеров путем сравнения соответствующих векторов слов. Кроме того, вы научитесь использовать векторы слов на практике и узнаете, как повысить эффективность операций, в частности выбор ключевых слов, с помощью предварительной обработки.

Смысл векторов слов

При создании статистических моделей словам в соответствие ставятся вещественные векторы, отражающие семантическое подобие этих слов. Пространство векторов слов можно представить в виде облака, где векторы располагаются тем ближе, чем ближе значения

слов. Например, вектор, соответствующий слову «картофель» будет ближе к вектору слова «морковь», чем к вектору слова «плакать». Для генерации таких векторов необходим способ кодирования смысла слов. Существует несколько подходов к кодированию смысла слов, которые будут рассмотрены в этом разделе.

Задание смысла с помощью координат

Один из способов генерации осмысленных векторов слов — сопоставление каждой координаты вектора с каким-либо объектом или категорией из реального мира. Например, представьте, что вы генерируете векторы слов для следующих понятий: Рим, Италия, Афины и Греция. Векторы слов должны математически отражать тот факт, что Рим — столица Италии и связан с Италией, а Афины — нет. В то же время они должны отражать и тот момент, что Афины и Рим — столицы, а Греция и Италия — страны. Таблица 5.1 демонстрирует, как могло бы выглядеть подобное векторное пространство.

Таблица 5.1. Упрощенное пространство векторов слов

	Страна	**Столица**	**Греческий**	**Итальянский**
Италия	1	0	0	1
Рим	0	1	0	1
Греция	1	0	1	0
Афины	0	1	1	0

Мы распределили значения слов по координатам в четырехмерном пространстве с категориями «Страна», «Столица», «Греческий» и «Итальянский». В этом упрощенном примере координаты могут принимать значения 0 и 1, указывая таким образом, относится ли слово к данной категории.

Если числовые векторы в векторном пространстве захватывают смысл слов, полезную информацию об их значениях можно узнать путем применения векторной арифметики. Чтобы определить, столицей какой страны являются Афины, воспользуемся следующим

уравнением, где каждый токен представляет соответствующий вектор, а X — неизвестный вектор:

```
Италия – Рим = X – Афины
```

Это уравнение выражает аналогию, в которой X соответствует вектору слов, относящемуся к вектору слов Афины так же, как вектор Италия относится к вектору Рим. Решаем это уравнение относительно X:

```
X = Италия – Рим + Афины
```

Сначала вычитаем вектор Рим из вектора Италия, вычитая соответствующие элементы векторов. Затем складываем получившийся вектор с вектором Афины. В табл. 5.2 показана схема вычислений.

Таблица 5.2. Векторная арифметика на пространстве векторов слов

		Страна	Столица	Греческий	Итальянский
–/+	Италия	1	0	0	1
	Рим	0	1	0	1
	Афины	0	1	1	0
	Греция	1	0	1	0

Вычитая вектор слов для токена Рим из вектора слов для токена Италия и прибавляя к результату вектор слов для токена Афины, получим вектор слов, равный вектору Греция.

Задание смысла по измерениям

В только что созданном векторном пространстве всего четыре категории, в то время как встречающиеся на практике векторные пространства могут содержать десятки тысяч категорий. Для большинства приложений использовать векторное пространство подобного размера нерационально, поскольку для него нужна гигантская матрица вложений векторов слов. Например, для 10 000 категорий и 1 000 000 кодируемых сущностей потребуется матрица вложений размером 10 000 × 1 000 000, операции над которой будут занимать слишком много времени. Очевидный способ уменьшения размера матрицы вложений — сокращение количества категорий в векторном пространстве.

Вместо определения для каждой категории своей координаты на практике при реализации пространства векторов слов семантические подобия выражаются количественно и распределяются по категориям на основе расстояния между векторами. Отдельные измерения обычно не несут никакого специфического смысла и всего лишь отражают положение в векторном пространстве, а на сходство смыслов соответствующих слов указывает расстояние между векторами.

Ниже приведен фрагмент 300-мерного пространства векторов слов, взятый из *fastText* — библиотеки векторов слов, которую можно скачать по ссылке https://fasttext.cc/docs/en/english-vectors.html:

```
compete   -0.0535 -0.0207  0.0574  0.0562 ... -0.0389 -0.0389
equations -0.0337  0.2013 -0.1587  0.1499 ...  0.1504  0.1151
Upper     -0.1132 -0.0927  0.1991 -0.0302 ... -0.1209  0.2132
mentor     0.0397  0.1639  0.1005 -0.1420 ... -0.2076 -0.0238
reviewer  -0.0424 -0.0304 -0.0031  0.0874 ...  0.1403 -0.0258
```

Каждая строка содержит слово в виде вещественного вектора в многомерном пространстве. Для наглядного представления подобного 300-мерного векторного пространства подойдет двухмерная или трехмерная проекция. Для ее формирования можно использовать первые две или три главные координаты вектора соответственно. На рис. 5.1 показаны векторы из 300-мерного векторного пространства в двухмерной проекции.

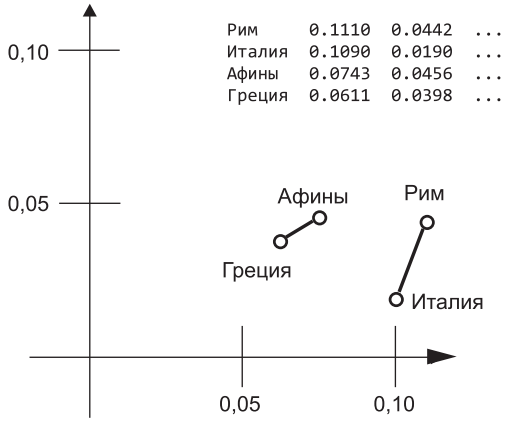


Рис. 5.1. Фрагмент двухмерной проекции многомерного векторного пространства

Возможно, вы обратили внимание на любопытный нюанс на этом рисунке: отрезки, соединяющие Грецию с Афинами и Италию с Римом, почти параллельны. Их длины также одного порядка. На практике это означает, что по трем векторам можно вычислить приближенное местонахождение оставшегося вектора, ведь известно, откуда и насколько необходимо сдвинуть вектор.

Векторы на графике иллюстрируют связь между странами и их столицами, но они с легкостью могут отражать и другие виды отношений, например «мужчина — женщина», времена глаголов и т. д.

Метод similarity

У всех типов объектов-контейнеров в библиотеке spaCy есть *метод* `similarity`, который используется для вычисления оценки семантического подобия двух объектов-контейнеров произвольного типа посредством сравнения их векторов слов. Для вычисления подобия интервалов и документов, не обладающих векторами слов, spaCy сравнивает средние значения векторов слов содержащихся в них токенов.

ПРИМЕЧАНИЕ

Малые модели spaCy (их размер обозначен как sm) не включают векторы слов, но и в этом случае можно использовать метод similarity для сравнения токенов, интервалов и документов. Результаты, однако, будут менее точными.

Можно вычислить семантическое подобие двух объектов-контейнеров даже для различных объектов. Например, можно сравнить объект `Token` с объектом `Span`, объект `Span` с объектом `Doc` и т. д.

В следующем примере вычисляется подобие объектов `Span` и `Doc`:

```
>>> doc=nlp('I want a green apple.')
>>> doc.similarity(doc[2:5])
0.7305813588233471
```

Этот код вычисляет оценку семантического подобия между предложением *I want a green apple.* и вырезанной из него фразой *a green apple*. Как видите, вычисленная оценка подобия достаточно высока, чтобы считать содержимое обоих объектов схожим (диапазон значений степени подобия от 0 до 1).

Неудивительно, что, если сравнить объект сам с собой, метод `similarity()` вернет `1`:

```
>>> doc.similarity(doc)
1.0
>>> doc[2:5].similarity(doc[2:5])
1.0
```

Можно также сравнить объект `Doc` со срезом другого объекта `Doc`:

```
>>> doc2=nlp('I like red oranges.')
>>> doc2.similarity(doc[2:5])
0.28546574467463354
```

Здесь сравнивались хранящееся в `doc2` предложение *I like red oranges.* и интервал *a green apple*, извлеченный из `doc`. В данном случае степень подобия оказалась не столь высокой. Да, и апельсин, и яблоко — фрукты (метод `similarity` это учитывает), но глаголы *want* и *like* выражают разные состояния.

Можно также сравнить два токена. В следующем примере сравниваем объект `Token` *oranges* с объектом `Span`, содержащим один токен *apple*:

```
>>> token = doc2[3:4][0]
>>> token
oranges
>>> token.similarity(doc[4:5])
0.3707084280155993
```

Прежде всего, с помощью ссылки на первый элемент интервала мы с вами преобразовали явным образом объект `Span` с одним токеном *oranges* в объект `Token`, а затем вычислили степень его подобия интервалу *apple*.

Метод `similarity()` распознает слова, которые относятся к одной или нескольким близким категориям и часто встречаются во взаимосвязанных контекстах, а также возвращает для подобных слов высокие значения уровня подобия.

Выбор ключевых слов для вычисления семантического подобия

Метод `similarity` берет на себя вычисление семантического подобия, но для извлечения пользы из этих расчетов необходимо подобрать правильные ключевые слова для сравнения. Для того чтобы убедиться в важности такого подбора, взглянем на следующий отрывок текста:

```
Redwoods are the tallest trees in the world. They are most common
in the coastal forests of California.
```

В зависимости от используемого набора категорий этот текст можно категоризировать по-разному. Например, если нас интересуют тексты о самых высоких растениях на планете, ключевыми окажутся фразы *tallest trees* и *in the world*. Сравнение их с поисковыми фразами *highest plants* и *on the planet* должно вернуть высокий уровень семантического подобия. Сделать это можно путем извлечения именных фрагментов с помощью свойства `doc.noun_chunk` объекта `Doc` и последующего вычисления подобия этих именных фрагментов с поисковыми фразами с помощью метода `similarity()`.

Если же нас интересуют тексты о различных местах в мире, ключевым словом будет *California*. Конечно, заранее не известно, какое географическое название встретится в тексте: им может оказаться *California* или, допустим, *Amazonia*. Но в любом случае оно будет семантически подобно слову *geography*, которое можно сравнить с прочими существительными в тексте (или, что лучше, только с его именованными сущностями).

При обнаружении высокого уровня подобия делается вывод, что интересующая нас именованная сущность представляет собой географическое название.

Можно также извлечь значение атрибута `token.ent_type` объекта `Token`, как описывалось в главе 2. Но использовать распознавание именованных сущностей для проверки подобия слов, которые не являются именованными сущностями, например названий фруктов, не получится.

Установка пакетов векторов слов

Если в среде Python установлена модель spaCy, векторы слов можно использовать сразу. Но не забывайте, что есть возможность установить сторонний пакет векторов слов. Разные статистические модели используют различные векторы слов, поэтому результаты операций могут незначительно отличаться в зависимости от используемой модели. Испытайте несколько моделей, чтобы выяснить, какая из них лучше подходит для вашего конкретного приложения.

Пользуемся векторами слов, прилагаемыми к моделям spaCy

В состав многих моделей библиотеки spaCy включены векторы слов. Например, `en_vectors_web_lg` содержит более миллиона уникальных векторов слов, определенных на 300-мерном векторном пространстве. Подробную информацию о конкретных моделях можно найти по адресу https://github.com/explosion/spacy-models/releases/.

Обычно маленькие модели (их названия оканчиваются на `sm`) содержат не векторы слов, а лишь контекстно зависимые *тензоры*, но это все равно позволяет использовать методы подобия для сравнения токенов, интервалов и документов, хотя и с меньшей точностью.

Чтобы следить за ходом приведенных в этой главе примеров, вам подойдет любая модель spaCy, даже маленькая. Но чем больше модель, тем выше точность результатов. Подробное описание установки моделей spaCy есть в разделе «Установка статистических моделей для библиотеки spaCy» на с. 41. Обратите внимание, что в одной среде может быть установлено несколько моделей.

Использование сторонних пакетов векторов слов

При желании вместе с библиотекой spaCy можно использовать сторонние пакеты векторов слов. Не исключено, что для вашего приложения они подойдут лучше, чем «родные» векторы слов. Например, можно воспользоваться предобученной моделью fastText (по ссылке

https://fasttext.cc/docs/en/english-vectors.html), включающей векторы слов для английского языка. Название пакета указывает на размер пакета и векторов слов, а также на вид данных, которые использовались для обучения этих векторов. Например, пакет `wiki-news-300d-1M.vec.zip` содержит один миллион 300-мерных векторов слов, обученных на наборах данных «Википедии» и statmt.org.

После скачивания пакета разархивируйте его и создайте новую модель из содержащихся в нем векторов, которую можно будет использовать с библиотекой spaCy. Для этого перейдите в каталог, где вы сохранили пакет, и запустите утилиту командной строки `init-model`:

```
$ python -m spacy init-model en /tmp/en_vectors_wiki_lg --vectors-loc
wiki-news-300d-1M.vec
```

Эта команда преобразует векторы слов из файла `wiki-news-300d-1M.vec` в формат библиотеки spaCy и создаст новый каталог модели для них — `/tmp/en_vectors_wiki_lg`. В случае успешного выполнения вы увидите следующие сообщения:

```
Reading vectors from wiki-news-300d-1M.vec
Open loc
999994it [02:05, 7968.84it/s]
Creating model...
0it [00:00, ?it/s]
    Successfully compiled vocab
    999731 entries, 999994 vectors
```

После создания модели ее можно загрузить как обычную модель spaCy:

```
nlp = spacy.load('/tmp/en_vectors_wiki_lg')
```

Далее создается объект `Doc`:

```
doc = nlp(u'Hi there!')
```

В отличие от обычных моделей spaCy сторонние модели, преобразованные для использования в spaCy, могут не поддерживать часть операций над содержащимся в объекте `Doc` текстом. Например, если попытаться разрезать документ на предложения с помощью `doc.sents`, вернется следующая ошибка: `ValueError: [E030] Sentence boundaries unset...`.

Сравнение объектов spaCy

Самая распространенная задача, для решения которой применяют векторы слов, — вычисление подобия объектов-контейнеров. В оставшейся части главы рассмотрим несколько сценариев, при которых может понадобиться определение семантического подобия различных лингвистических единиц.

Применение семантического подобия для задач категоризации

Вычисление семантического подобия двух объектов позволяет распределять тексты по категориям, а также выбирать из корпуса лишь те из них, которые относятся к нужной теме. Допустим, что из всех комментариев пользователей на сайте нам нужно выбрать комментарии, связанные со словом *fruits*. Пусть они включают следующие высказывания:

```
I want to buy this beautiful book at the end of the week.
Sales of citrus have increased over the last year.
How much do you know about this type of tree?
```

Человеку сразу ясно, что только второе предложение напрямую связано с фруктами, поскольку содержит слово *citrus*. Чтобы выбрать это предложение программным образом, необходимо сравнить вектор слов для *fruits* с векторами слов во всех приведенных предложениях.

Начнем с простейшего, но наименее результативного способа: сравним *fruits* с каждым из предложений. Как упоминалось ранее, spaCy определяет подобие объектов-контейнеров путем сравнения их векторов слов. Для сравнения отдельного токена с целым предложением spaCy усредняет векторы слов предложения, генерируя совершенно новый вектор. Следующий сценарий сравнивает каждое из предыдущих предложений со словом *fruits*:

```
import spacy
nlp = spacy.load('en')
❶ token = nlp(u'fruits')[0]
❷ doc = nlp(u'I want to buy this beautiful book at the end of the week.
Sales of citrus have increased over the last year. How much do you know
about this type of tree?')
❸ for sent in doc.sents:
```

```
    print(sent.text)
❹ print('similarity to', token.text, 'is', token.similarity(sent),'\n')
```

Сначала создаем объект `Token` для слова *fruits* ❶ и применяем конвейер к категоризируемым предложениям, создавая для них общий объект `Doc` ❷. Далее разрезаем его на предложения ❸, а затем выводим в консоль каждое из них и его семантическое подобие к токену *fruits*, которое вычисляется с помощью метода `similarity` объекта этого токена ❹.

Результаты работы должны выглядеть примерно так (хотя сами числовые значения зависят от используемой модели):

```
I want to buy this beautiful book at the end of the week.
similarity to fruits is 0.06307832979619851
Sales of citrus have increased over the last year.
similarity to fruits is 0.2712141843864381
How much do you know about this type of tree?
similarity to fruits is 0.24646341651210604
```

Степень подобия слова *fruits* и первого предложения крайне невелика, следовательно, предложение практически никак не связано с фруктами. Второе предложение, содержащее слово *citrus*, оказывается наиболее подобным слову *fruits*, так что сценарий правильно определил предложение, относящееся к данной теме.

Но обратите внимание: сценарий также определил, что с фруктами в какой-то степени связано и третье предложение. Вероятно, это произошло из-за слова *tree* — ведь многие фрукты растут на деревьях. Наивно полагать, что алгоритм вычисления подобия «знает», что апельсин и цитрус — фрукты. Ему известно только то, что соответствующие слова (*orange* и *citrus*) часто встречаются в одном контексте со словом *fruit*, а значит, они должны быть близки в пространстве векторов слов. Но слово *tree* также нередко встречается в одном контексте со словом *fruit*. Например, в текстах вполне можно встретить фразу *fruit tree*. Поэтому степень подобия слова *fruits* (и его леммы — слова *fruit*) и слова *tree* близка к результату, полученному для слов *citrus* и *fruits*.

С подобным подходом к категоризации текстов связана еще одна проблема. На практике, безусловно, иногда придется сталкиваться с текстами гораздо большего размера, чем приведенные в этом разделе примеры. Если усредняемый текст огромен, влияние самых важных слов на показатель семантического подобия может оказаться ничтожным.

Для получения от метода `similarity` более точных результатов необходимо провести с текстом определенную подготовительную работу. Давайте взглянем, как можно улучшить наш сценарий.

Выделение существительных как шаг предварительной обработки

Более удачный подход к категоризации — выделять только самые значимые слова и сравнивать только их. Подобная подготовка текста к обработке, которая называется *предварительной обработкой* (preprocessing), позволяет существенно повысить успешность операций NLP. Вместо сравнения векторов слов для целого объекта можно сравнивать векторы слов только для определенных частей речи. В большинстве случаев для распознавания смысла текста анализируются существительные, независимо от того, выступают они как подлежащие или прямые или непрямые дополнения. Например, в предложении *Nearly all wild lions live in Africa* нас будут интересовать, вероятнее всего, *lions*, *Africa* и *lions in Africa*. Аналогично в предложении о фруктах необходимо выделить существительное *citrus*. В прочих случаях, чтобы выяснить, чему посвящен текст, необходимо проанализировать другие слова, например глаголы. Представьте, что вы занимаетесь бизнесом, связанным с сельским хозяйством, и вам необходимо классифицировать коммерческие предложения, поступающие от производителей, переработчиков и продавцов сельскохозяйственной продукции. При этом вам часто будут встречаться фразы наподобие *We grow vegetables* и *We take tomatoes for processing*. В данных примерах глаголы не менее важны, чем существительные в высказываниях из предыдущего примера.

Модифицируем сценарий со с. 109 и сравним слово *fruits* не с целыми предложениями, а только с их существительными:

```
import spacy
nlp = spacy.load('en')
❶ token = nlp(u'fruits')[0]
doc = nlp(u'I want to buy this beautiful book at the end of the week.
Sales of citrus have increased over the last year. How much do you know
about this type of tree?')
similarity = {}
❷ for i, sent in enumerate(doc.sents):
```

```
❸ noun_span_list = [sent[j].text for j in range(len(sent)) if
   sent[j].pos_ == 'NOUN']
❹ noun_span_str = ' '.join(noun_span_list)
❺ noun_span_doc = nlp(noun_span_str)
❻ similarity.update({i:token.similarity(noun_span_doc)})
print(similarity)
```

Начнем с описания токена *fruits*, который будем использовать для ряда сравнений ❶. Проходя в цикле по токенам каждого предложения ❷, выделяем существительные и сохраняем их в списке Python ❸. Затем склеиваем существительные из этого списка в простую строку ❹, которую позже преобразуем в объект `Doc` ❺. Далее сравниваем объект `Doc` с токеном *fruits*, чтобы определить степень их семантического подобия. Значение семантического подобия сохраняется в ассоциативном массиве Python ❻, который затем выводится в консоль.

Результат работы сценария выглядит примерно так:

```
{0: 0.17012682516221458, 1: 0.5063824302533686, 2: 0.6277196645922878}
```

Если сравнить эти значения с результатами предыдущего сценария, можно заметить, что степень семантического подобия со словом *fruits* выше для всех предложений. В целом же результаты почти не изменились: степень подобия первого предложения самая низкая, а остальных двух — намного выше.

Попробуйте сами

В предыдущем примере, когда мы выделили только самые важные слова — существительные — и сравнили *fruits* только с ними, нам удалось улучшить результаты вычисления. Мы сравнивали слово *fruits* со всеми существительными, выделенными из совокупности предложений. Можно сделать шаг вперед и выяснить, насколько каждое из этих существительных семантически связано со словом *fruits*, и найти то из них, степень подобия которого максимальна: это может быть полезным для оценки общего подобия документа слову *fruits*. Потребуется модифицировать предыдущий сценарий так, чтобы он определял подобие токена *fruits* и каждого из существительных в предложении и находил то из них, которое демонстрирует наибольший уровень подобия.

Выделение и сравнение именованных сущностей

В некоторых случаях вместо выделения всех существительных из сравниваемых текстов имеет смысл выделять лишь определенный вид, например именованные сущности. Представьте, что нужно сравнить следующие тексты.

Google Search, often referred to as simply Google, is the most used search engine nowadays. It handles a huge number of searches each day.

Microsoft Windows is a family of proprietary operating systems developed and sold by Microsoft. The company also produces a wide range of other software for desktops and servers.

Titicaca is a large, deep, mountain lake in the Andes. It is known as the highest navigable lake in the world.

В идеале сценарий должен распознать, что первые два текста посвящены крупным высокотехнологичным компаниям, а третий — нет. Но сравнение всех существительных в тексте не поможет в достижении этой цели, поскольку многие из них, например *number* в первом предложении, не имеют отношения к контексту. Различия между предложениями заключены в следующих словах: *Google*, *Search*, *Microsoft*, *Windows*, *Titicaca* и *Andes*. Библиотека spaCy понимает, что это все — именованные сущности, что значительно упрощает поиск и выделение их из текста, как показано в следующем сценарии:

```
import spacy
nlp = spacy.load('en')
# Первый пример текста
doc1 = nlp(u'Google Search, often referred to as simply Google, is the
most
used search engine nowadays. It handles a huge number of searches each
day.')
# Второй пример текста
doc2 = nlp(u'Microsoft Windows is a family of proprietary operating
systems
developed and sold by Microsoft. The company also produces a wide range of
other software for desktops and servers.')
# Третий пример текста
doc3 = nlp(u"Titicaca is a large, deep, mountain lake in the Andes. It is
known as the highest navigable lake in the world.")
❶ docs = [doc1,doc2,doc3]
```

```
❷ spans = {}
❸ for j,doc in enumerate(docs):
  ❹ named_entity_span = [doc[i].text for i in range(len(doc)) if
    doc[i].ent_type != 0]
  ❺ print(named_entity_span)
  ❻ named_entity_span = ' '.join(named_entity_span)
  ❼ named_entity_span = nlp(named_entity_span)
  ❽ spans.update({j:named_entity_span})
```

Здесь мы группируем объекты `Doc` с примерами текстов в список, чтобы по ним можно было пройти в цикле ❶. Объявляем ассоциативный массив Python для хранения ключевых слов всех текстов ❷. Проходим в цикле по объектам `Doc` ❸, выделяя ключевые слова в отдельные списки для каждого из текстов. При этом выбор падает лишь на слова, маркированные как именованные сущности ❹. Далее выводим список в консоль, чтобы посмотреть его содержимое ❺. Трансформируем этот список в простую строку ❻, к которой применяем конвейер, а затем преобразуем ее в объект `Doc` ❼. Наконец, добавляем объект `Doc` к объявленному ранее ассоциативному массиву `spans` ❽.

В результате выполнения сценария должно быть выведено примерно следующее:

```
['Google', 'Search', 'Google']
['Microsoft', 'Windows', 'Microsoft']
['Titicaca', 'Andes']
```

Это слова из текстов, векторы которых нужно сравнить.

Теперь вызываем метод `similarity()` для каждого из интервалов и выводим результаты:

```
print('doc1 is similar to doc2:',spans[0].similarity(spans[1]))
print('doc1 is similar to doc3:',spans[0].similarity(spans[2]))
print('doc2 is similar to doc3:',spans[1].similarity(spans[2]))
```

На этот раз вывод в консоли будет выглядеть так:

```
doc1 is similar to doc2: 0.7864886939527678
doc1 is similar to doc3: 0.6797676349647936
doc2 is similar to doc3: 0.6621659567003596
```

Полученные значения указывают, что наиболее подобны первый и второй тексты, посвященные американским IT-компаниям. Как век-

торы слов это узнали? Вероятно, убедившись, что вместе слова *Google* и *Microsoft* чаще встречаются в одних и тех же текстах обучающего корпуса, а не в компании слов *Titicaca* и *Andes*.

Резюме

В этой главе вам довелось поработать с векторами слов — вещественными векторами, отражающими значение слов. Благодаря подобному представлению можно математически определять семантическое подобие языковых единиц, что очень важно при категоризации текстов.

Но для успешной работы этого математического подхода необходима предварительная обработка текстов: с ее помощью можно свести текст до наиболее важных для его смысла слов. Если текст очень большой, стоит выбрать из него именованные сущности, поскольку они лучше всего описывают категорию, к которой этот текст следует отнести.

6 Поиск паттернов и обход деревьев зависимостей

Для категоризации текста, выделения из него определенных фраз или вычисления семантического подобия другому тексту приложению необходимо «понимать» высказывания пользователей и генерировать осмысленные ответы на них.

Вы уже изучили несколько методик решения подобных задач. В этой главе рассматриваются еще два подхода: использование паттернов (закономерностей) последовательностей слов для классификации и генерации текста, а также выделение из высказывания необходимых элементов информации путем обхода его дерева синтаксических зависимостей. Вы познакомитесь с утилитой Matcher библиотеки spaCy, предназначенной для поиска закономерностей. А также узнаете, в каких случаях для определения нужного подхода к обработке все равно приходится учитывать контекст.

Паттерны последовательностей слов

Паттерн последовательности слов (word sequence pattern) состоит из признаков слов, накладывающих на каждое слово последовательности определенные требования. Например, фраза *I can* соответствует следующему паттерну последовательности слов: «местоимение + вспомогательный модальный глагол». Нахождение паттернов последовательностей слов позволяет распознавать последовательности слов со схожими языковыми признаками, а значит, и распределять входные данные по категориям и должным образом их обрабатывать.

Например, при получении вопроса, начинающегося с последовательности слов, которая соответствует паттерну «вспомогательный модальный глагол + имя собственное» (как фраза *Can George*), ясно, что он относится либо к способности, возможности или разрешению что-то сделать, либо к обязанности, налагаемой на кого-то или что-то, описываемое именем собственным.

В следующих разделах вы научитесь классифицировать предложения с помощью распознавания распространенных закономерностей лингвистических признаков.

Поиск паттернов лингвистических признаков

Поиск паттернов в текстах необходим по той причине, что в большинстве случаев в тексте не бывает двух совершенно одинаковых предложений: обычно текст состоит из разных предложений, содержащих разные слова. Писать отдельный код для обработки каждого предложения текста нерационально.

К счастью, некоторые выглядящие совершенно разными предложения имеют одинаковые паттерны последовательности слов. Возьмем, к примеру, такие два предложения: *We can overtake them. You must specify it.* У них нет ни одного общего слова. Но, если взглянуть на метки синтаксической зависимости слов этих предложений, станет ясна общая закономерность, как показано в следующем сценарии:

```
import spacy
nlp = spacy.load('en')
doc1 = nlp(u'We can overtake them.')
doc2 = nlp(u'You must specify it.')
❶ for i in range(len(doc1)-1):
  ❷ if doc1[i].dep_ == doc2[i].dep_:
    ❸ print(doc1[i].text, doc2[i].text, doc1[i].dep_,
     spacy.explain(doc1[i].dep_))
```

Поскольку число слов в предложениях одинаково, можно пройти по словам обоих предложений в одном цикле ❶. Если метка зависимости одинакова для слов с идентичными индексами ❷, то выводим эти слова, присвоенную им метку и ее описание ❸.

В результате должно быть выведено:

```
We       You     nsubj  nominal subject
can      must    aux    auxiliary
overtake specify ROOT   None
them     it      dobj   direct object
```

Как видите, список меток зависимости одинаков для обоих предложений. Это значит, что предложения строятся по одному паттерну последовательности слов, который описывается метками синтаксической зависимости таким образом: «подлежащее + вспомогательный глагол + глагол + прямое дополнение».

Обратите внимание, что списки тегов частей речи (общих и уточненных) для этих примеров предложений также совпадают. Если заменить все ссылки на атрибуты `.dep_` на `.pos_` в предыдущем сценарии, получим следующие результаты:

```
We       You     PRON  pronoun
can      must    VERB  verb
overtake specify VERB  verb
them     it      PRON  pronoun
```

Эти предложения соответствуют не только паттерну меток синтаксической зависимости, но и паттерну тегов частей речи.

Попробуйте сами

В предыдущем примере мы создали два объекта `Doc` — по одному для каждого примера предложения. На практике же текст обычно состоит из множества предложений, вследствие чего создавать объект `Doc` для каждого предложения нерационально. Перепишите сценарий таким образом, чтобы создавался один общий объект `Doc`. Затем воспользуйтесь свойством `doc.sents`, с которым вы познакомились в главе 2, для манипуляций с обоими предложениями.

Впрочем, учтите, что `doc.sents` — объект-генератор — не индексируется и, значит, к его элементам нельзя обращаться по индексам. Для решения этой проблемы преобразуйте `doc.sents` в список:

```
sents = list(doc.sents)
```

И конечно, для получения `sents` в требуемом для нашего цикла порядке можно пройти по `doc.sents` в цикле `for`.

Проверка высказывания на соответствие паттерну

В предыдущем примере мы сравнили два предложения в поисках паттерна, основанного на их общих лингвистических признаках. На практике же обычно нет смысла сравнивать предложения друг с другом для выяснения, объединяет ли их один паттерн. Вместо этого стоит проверить полученное предложение на соответствие нужному нам паттерну.

Например, во введенном пользователем тексте нас интересуют высказывания, выражающие способность, возможность, разрешение или обязанность сделать что-либо, а не высказывания, описывающие действия, которые действительно имели место быть, происходят сейчас или случаются регулярно. Допустим, нас интересует высказывание *I can do it*, но не высказывание *I've done it*.

Для разделения различных типов высказываний можно проверить, соответствуют ли высказывания следующему паттерну: «подлежащее + + вспомогательный глагол + глагол + ... + прямое дополнение...». Пропуски указывают, что прямое дополнение не обязано располагаться сразу за глаголом, служа тем самым отличием данного паттерна от паттерна из предыдущего примера.

Указанному паттерну удовлетворяет следующее предложение: *I might send them a card as a reminder*. В нем существительное *card* играет роль прямого дополнения, а местоимение *them* — непрямого дополнения, отделяющего *card* от глагола *send*. Паттерн не задает жестко, на каком месте должно располагаться прямое дополнение в предложении, а просто требует его наличия.

На рис. 6.1 это изображено схематически.

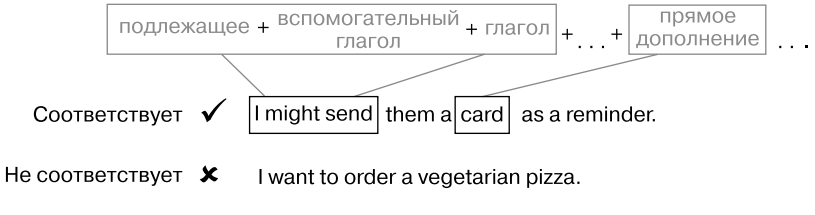


Рис. 6.1. Проверка вводимых пользователем высказываний на соответствие паттерну последовательности слов, основанному на лингвистических признаках

В следующем сценарии описываем функцию, реализующую указанный паттерн, а затем тестируем ее на примере предложения:

```
import spacy
nlp = spacy.load('en')
❶ def dep_pattern(doc):
  ❷ for i in range(len(doc)-1):
    ❸ if doc[i].dep_ == 'nsubj' and doc[i+1].dep_ == 'aux' and
       doc[i+2].dep_ == 'ROOT':
      ❹ for tok in doc[i+2].children:
          if tok.dep_ == 'dobj':
            ❺ return True
  ❻ return False
❼ doc = nlp(u'We can overtake them.')
  if ❽dep_pattern(doc):
    print('Found')
  else:
    print('Not found')
```

В этом сценарии описана функция `dep_pattern`, принимающая в качестве параметра объект `Doc` ❶. В ней проходим в цикле по токенам объекта `Doc` ❷ и ищем паттерн «подлежащее + вспомогательный глагол + глагол» ❸. Найдя его, проверяем, есть ли среди непосредственных дочерних элементов глагола прямое дополнение ❹. В случае обнаружения этого прямого дополнения функция вернет `True` ❺, в противном случае — `False` ❻.

В основном коде применяем к нашему примеру предложения конвейер обработки текста ❼ и передаем объект `Doc` в функцию `dep_pattern` ❽. Если пример удовлетворяет реализованному в функции паттерну, выводим `Found`, в противоположном случае — `Not found`.

Поскольку образец предложения из данного примера соответствует указанному паттерну, сценарий выдаст в консоль:

```
Found
```

В следующих разделах рассмотрим еще несколько примеров использования функции `dep_pattern`.

Использование утилиты Matcher библиотеки spaCy для поиска паттернов последовательностей слов

В предыдущем разделе вы увидели, как можно найти паттерн последовательности слов в объекте `Doc` путем обхода в цикле его токенов и проверки их лингвистических признаков. На самом деле в библиоте-

ке spaCy есть встроенная утилита для решения этой задачи — *Matcher*, специально разработанная для поиска последовательностей токенов, соответствующих заданному паттерну. Например, паттерн «подлежащее + вспомогательный глагол + глагол» с помощью Matcher можно реализовать так:

```
import spacy
from spacy.matcher import Matcher
nlp = spacy.load("en")
❶ matcher = Matcher(nlp.vocab)
❷ pattern = [{"DEP": "nsubj"}, {"DEP": "aux"}, {"DEP": "ROOT"}]
❸ matcher.add("NsubjAuxRoot", None, pattern)
doc = nlp(u"We can overtake them.")
❹ matches = matcher(doc)
❺ for match_id, start, end in matches:
    span = doc[start:end]
  ❻ print("Span: ", span.text)
    print("The positions in the doc are: ", start, "-", end)
```

Создаем экземпляр Matcher, передавая конструктору объект со словарным запасом, соответствующим документам, которые этот Matcher будет обрабатывать ❶. Далее описываем паттерн, указывая метки зависимости, которым должна удовлетворять последовательность слов ❷. Добавляем только что созданный паттерн в Matcher ❸.

Применяем Matcher к нашему примеру текста и получаем список соответствующих паттерну токенов ❹, а затем проходим по этому списку в цикле ❺, выводя начальную и конечную позиции токенов паттерна в тексте ❻.

Результаты работы сценария должны выглядеть следующим образом:

```
Span: We can overtake
The positions in the doc are: 0 - 3
```

Matcher дает возможность искать паттерны в тексте без явного обхода в цикле токенов текста, скрывая, таким образом, от разработчика подробности реализации. В результате можно легко получить начальную и конечную позиции слов, составляющих последовательность, которая соответствует заданному паттерну. Такой подход очень удобен для последовательностей слов, которые расположены одно за другим.

Но часто интересующие нас паттерны включают слова, разбросанные по тексту. Например, может понадобиться реализовать паттерн наподобие

«подлежащее + вспомогательный глагол + глагол + ... + прямое дополнение...», о котором шла речь в подразделе «Проверка высказывания на соответствие паттерну» на с. 119. Проблема в том, что заранее не известно, сколько слов будет разделять последовательность «подлежащее + + вспомогательный глагол + глагол» и прямое дополнение. Matcher не позволяет описывать подобные паттерны, поэтому оставшаяся часть данной главы будет посвящена ручному описанию паттернов.

Применение нескольких паттернов

Одно высказывание можно сопоставить с несколькими паттернами, дабы убедиться, что оно удовлетворяет всем нужным условиям. Например, с двумя паттернами: с реализующим последовательность меток зависимости (как обсуждалось в подразделе «Проверка высказывания на соответствие паттерну» на с. 119) и с проверяющим на соответствие последовательности тегов частей речи. Это может пригодиться, скажем, если будет нужно убедиться, что в роли прямого дополнения в высказывании выступает личное местоимение. В таком случае можно начать поиск соответствующего этому местоимению существительного, упомянутого где-то в другом месте текста.

Схематически это можно изобразить так, как на рис. 6.2.

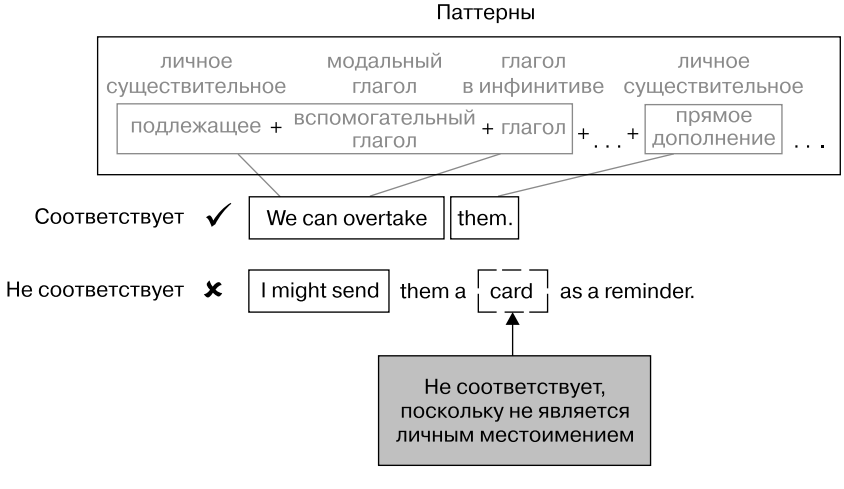


Рис. 6.2. Применение нескольких паттернов к введенным пользователем данным

Помимо последовательности меток зависимостей, описанной в подразделе «Проверка высказывания на соответствие паттерну» на с. 119, можно описать новую функцию, реализующую паттерн, основанный на тегах частей речи. Например, описываемый тегами частей речи паттерн будет содержать условие, что подлежащее и прямое дополнение должны быть личными местоимениями. Эта новая функция может среди прочего реализовывать паттерн «личное местоимение + вспомогательный модальный глагол + глагол в инфинитиве + ... + личное местоимение...».

Вот соответствующий код:

```
import spacy
nlp = spacy.load('en')
# Вставьте сюда функцию dep_pattern из предыдущего листинга
# ...
❶ def pos_pattern (doc):
  ❷ for token in doc:
      if token.dep_ == 'nsubj' and token.tag_ != 'PRP':
        return False
      if token.dep_ == 'aux' and token.tag_ != 'MD':
        return False
      if token.dep_ == 'ROOT' and token.tag_ != 'VB':
        return False
      if token.dep_ == 'dobj' and token.tag_ != 'PRP':
        return False
  ❸ return True
# Тестирование кода
doc = nlp(u'We can overtake them.')
❹ if dep_pattern(doc) and pos_pattern(doc):
    print('Found')
else:
    print('Not found')
```

Мы начали со вставки кода для функции `dep_pattern`, описанной в предыдущем сценарии. Для создания второго паттерна описали функцию `pos_pattern` ❶, включающую цикл `for` с рядом операторов `if` ❷. Операторы `if` проверяют, соответствует ли конкретная часть предложения определенному тегу части речи. При обнаружении несоответствия функция возвращает `False`. В противном случае по завершении всех проверок и при отсутствии каких-либо несоответствий функция возвращает `True` ❸.

Для проверки на паттерны применяем к предложению конвейер, после чего проверяем, удовлетворяет ли оно обоим паттернам ❹. Поскольку

используемый в этом примере образец предложения удовлетворяет обоим паттернам, должно появиться следующее:

```
Found
```

Если же заменить образец предложения на такой: *I might send them a card as a reminder*., будет выведено:

```
Not found
```

Дело в том, что предложение не удовлетворяет паттерну тегов частей речи, поскольку прямое дополнение *card* не является личным местоимением, хотя полностью удовлетворяет условиям первого паттерна.

Создание паттернов на основе пользовательских признаков

При создании паттерна последовательности слов может возникнуть необходимость расширить функциональность лингвистических признаков, предоставляемых spaCy, подогнав их под свои задачи. Например, может понадобиться, чтобы предыдущий сценарий распознавал паттерн, в котором учитывается число местоимений (единственное или множественное). Это полезно в том случае, если нужно найти существительное, к которому относится местоимение.

SpaCy же различает число существительных, но не местоимений. При этом возможность распознавания числа местоимений может пригодиться в процессе решения задачи распознавания смысла или выделения информации. Для примера рассмотрим следующий текст:

```
The trucks are traveling slowly. We can overtake them.
```

Если установим, что прямое дополнение *them* во втором предложении представляет собой местоимение во множественном числе, то сможем сделать вывод, что оно относится к существительному множественного числа *trucks* из первого предложения. Этот способ часто применяется для распознавания смысла существительных по контексту.

В следующем сценарии описывается функция `pron_pattern`, которая находит все прямые дополнения в передаваемом ей предложении, определяет, является ли прямое дополнение личным местоимением, а затем

выясняет, в каком оно числе: единственном или множественном. Далее функция применяется к примеру предложения после проверки его на соответствие двум паттернам, описанным в подразделах «Проверка высказывания на соответствие паттерну» на с. 119 и «Применение нескольких паттернов» на с. 122.

```
import spacy
nlp = spacy.load('en')
# Вставьте сюда функции dep_pattern и pos_pattern из предыдущих листингов
# ...
❶ def pron_pattern(doc):
  ❷ plural = ['we','us','they','them']
    for token in doc:
    ❸ if token.dep_ == 'dobj' and token.tag_ == 'PRP':
      ❹ if token.text in plural:
        ❺ return 'plural'
         else:
        ❻ return 'singular'
  ❼ return 'not found'
doc = nlp(u'We can overtake them.')
if dep_pattern(doc) and pos_pattern(doc):
    print('Found:', 'the pronoun in position of direct object is',
    pron_pattern(doc))
else:
    print('Not found')
```

Начинаем с добавления в сценарий функций `dep_pattern` и `pos_pattern`, описанных в подразделах «Проверка высказывания на соответствие паттерну» и «Применение нескольких паттернов». В функции `pron_pattern` ❶ описываем список Python, включающий все возможные личные местоимения во множественном числе ❷. Далее описываем цикл, в котором проходим по токенам полученного предложения в поисках прямого дополнения — личного местоимения ❸. Найдя, проверяем, встречается ли оно в списке личных местоимений во множественном числе ❹. Если да, наша функция возвращает `plural` ❺. В противном случае она возвращает `singular` ❻. Если функции не удалось обнаружить прямое дополнение или оно не является личным местоимением, возвращается `Not found` ❼.

Для предложения *We can overtake them.* должно быть выведено:

```
Found: the pronoun in position of direct object is plural
```

С помощью этой информации в предыдущем предложении можно найти соответствующее местоимению существительное.

Выбор применяемых паттернов

После описания паттернов можно выбрать, какие из них применять и в какой ситуации. Обратите внимание, что даже предложение, не соответствующее полностью паттернам функций `dep_pattern` и `pos_pattern`, может удовлетворять паттерну функции `pron_pattern`. Например, предложение *I know it.* не удовлетворяет паттерну ни функции `dep_pattern`, ни функции `pos_pattern`, поскольку не содержит вспомогательного модального глагола, но удовлетворяет `pos_pattern`, поскольку имеет личное местоимение, выступающее в роли прямого дополнения.

Столь слабое сцепление между паттернами позволяет использовать их совместно или по отдельности. Например, функцию `dep_pattern`, проверяющую предложение на соответствие паттерну «подлежащее + вспомогательный глагол + глагол + ... + прямое дополнение...», можно использовать вместе с функцией для паттерна «существительное + вспомогательный модальный глагол + глагол в неопределенной форме + ... + существительное...», дабы убедиться, что и подлежащее, и прямое дополнение в предложении — существительные. Этим двум паттернам будет удовлетворять следующий пример предложения:

```
Developers might follow this rule.
```

Как вы понимаете, возможность комбинирования паттернов позволяет обрабатывать большее число случаев с помощью меньшего объема кода.

Применение паттернов последовательностей слов в чат-ботах для генерации высказываний

Как упоминалось ранее, самые трудные задачи NLP — понимание и генерация текста на естественных языках. Чат-бот должен понимать вводимый пользователем текст и генерировать на него должный ответ. Паттерны последовательностей слов, основанные на лингвистических признаках, могут быть полезны при реализации этих функций.

В главе 4 вы узнали, как для поддержания разговора с пользователем утвердительное высказывание превратить в соответствующий вопрос.

С помощью паттернов последовательностей слов можно реагировать на высказывания пользователя и другим образом — например, генерируя подходящие утвердительные высказывания.

Пусть наш чат-бот получил от пользователя такой текст:

```
The symbols are clearly distinguishable. I can recognize them promptly.
```

Чат-бот мог бы отреагировать на это следующим образом:

```
I can recognize symbols promptly too.
```

Для подобной генерации текста можно воспользоваться паттернами, реализованными в предыдущих разделах. Перечень шагов подобной генерации выглядит примерно так.

1. Проверить вводимые в диалоговом режиме данные с помощью описанных выше функций `dep_pattern` и `pos_pattern` и найти высказывание, которое удовлетворяет паттернам «подлежащее + вспомогательный глагол + глагол + ... + прямое дополнение...» и «местоимение + вспомогательный модальный глагол + глагол в неопределенной форме + ... + + местоимение...» соответственно.
2. Проверить найденное при выполнении шага 1 высказывание на соответствие паттерну функции `pron_pattern` и выяснить, какое число имеет личное местоимение, выступающее в роли прямого дополнения: единственное или множественное.
3. Найти обозначаемое этим местоимением существительное путем поиска существительного, стоящего в том же числе, что и указанное личное местоимение.
4. Заменить в найденном при выполнении шага 1 предложении местоимение, играющее роль прямого дополнения, существительным, найденным на шаге 3.
5. Добавить в конец сгенерированного высказывания слово *too*.

Данные шаги реализованы в приведенном ниже сценарии. В нем используются описанные ранее в этой главе функции `dep_pattern`, `pos_pattern` и `pron_pattern` (их код опустим для экономии места). Кроме того, в сценарии появляются две новые функции: `find_noun`

и gen_utterance. Для удобства разобьем обсуждение кода на три части: начальные операции и функция find_noun, предназначенная для поиска существительного, соответствующего личному местоимению; функция gen_utterance, генерирующая код для анализа высказывания. Вот первая часть:

```
import spacy
nlp = spacy.load('en')
# Вставьте сюда функции dep_pattern, pos_pattern и pron_pattern
# из предыдущих листингов
# ...
❶ def find_noun(❷sents, ❸num):
    if num == 'plural':
    ❹ taglist = ['NNS','NNPS']
    if num == 'singular':
    ❺ taglist = ['NN','NNP']
  ❻ for sent in reversed(sents):
    ❼ for token in sent:
      ❽ if token.tag_ in taglist:
            return token.text
    return 'Noun not found'
```

После вставки кода функций dep_pattern, pos_pattern и pron_pattern была описана функция find_noun, принимающая два параметра ❶. Первый из них содержит список предложений от начала текста до предложения, удовлетворяющего всем паттернам. В данном примере этот список будет включать все предложения текста, поскольку только последнее из них удовлетворяет всем паттернам ❷. Однако существительное, обозначаемое местоимением, находится в одном из предыдущих предложений.

Второй из передаваемых функции find_noun параметров — число местоимения, выступающего в роли прямого дополнения, которое удовлетворяет всем паттернам ❸. Его определяет функция pron_pattern. Если значение этого аргумента 'plural', описываем список Python, содержащий теги уточненных частей речи, которые в spaCy отмечают существительные во множественном числе ❹. Если же 'singular', создаем список тегов общих частей речи, обозначающих существительные в единственном числе ❺.

Проходим в цикле for по предложениям в обратном порядке, начиная с предложения, ближайшего к тому, которое содержит заменяемое местоимение ❻. Начинать с ближайшего предложения стоит по причине

того, что именно там с наибольшей долей вероятности содержится искомое существительное. В данном случае используется функция `reversed` языка Python, возвращающая обратный итератор для обхода списка. Во внутреннем цикле проходим по токенам в каждом из предложений ❼ в поисках токена, тег уточненной части речи которого содержался бы в описанном ранее списке ❽.

Затем описываем функцию `gen_utterance`, предназначенную для генерации нового утвердительного высказывания:

```
def gen_utterance(doc, noun):
  sent = ''
❶ for i,token in enumerate(doc):
  ❷ if token.dep_ == 'dobj' and token.tag_ == 'PRP':
    ❸ sent = doc[:i].text + ' ' + noun + ' ' +
      doc[i+1:len(doc)-2].text + 'too.'
    ❹ return sent
❺ return 'Failed to generate an utterance'
```

Проходим в цикле `for` по токенам предложения ❶ и ищем личное местоимение, которое выступает в роли прямого дополнения ❷. Найдя его, генерируем новое высказывание. Меняем исходное предложение, заменяя личное местоимение соответствующим существительным и добавляя в конец *too* ❸. После этого функция `gen_utterance` возвращает только что сгенерированное утвердительное высказывание ❹. Если же найти прямое дополнение в форме личного местоимения не удалось, функция возвращает сообщение об ошибке ❺.

Теперь, подготовив все необходимые функции, можем проверить их в работе на примере утвердительного высказывания. В этом поможет следующий код:

```
❶ doc = nlp(u'The symbols are clearly distinguishable. I can recognize
  them promptly.')
❷ sents = list(doc.sents)
  response = ''
  noun = ''
❸ for i, sent in enumerate(sents):
    if dep_pattern(sent) and pos_pattern(sent):
   ❹ noun = find_noun(sents[:i], pron_pattern(sent))
      if noun != 'Noun not found':
     ❺ response = gen_utterance(sents[i],noun)
      break
  print(response)
```

После применения конвейера к нашему образцу текста ❶ преобразуем текст в список предложений ❷. Проходим в цикле по этому списку ❸ и ищем предложение, которое удовлетворяет описанным в функциях `dep_pattern` и `pos_pattern` паттернам. Затем с помощью функции `find_noun` определяем существительное, соответствующее местоимению из предложения, найденного на предыдущем шаге ❹. Наконец, вызываем функцию `gen_utterance` для генерации соответствующего высказывания ❺.

Результат выполнения предыдущего кода должен выглядеть следующим образом:

```
I can recognize symbols too.
```

Попробуйте сами

Обратите внимание, что предыдущий код можно улучшить, ведь в исходном высказывании перед существительным *symbols* стоит артикль *the*. Усовершенствованная версия должна выводить перед существительным тот же артикль. Модифицируйте сценарий для генерации наиболее уместного в данном контексте высказывания так, чтобы он вставлял перед существительным артикль *the* и чтобы в итоге получалось *I can recognize the symbols too*. Для этого вам придется проверять, предшествует ли существительному артикль, а затем его добавлять.

Выделение ключевых слов из деревьев синтаксических зависимостей

Благодаря поиску в предложении последовательности слов, которая удовлетворяет определенному паттерну, на основе входного текста можно построить грамматически корректный ответ — либо утвердительное высказывание, либо вопрос. Но с задачей выделения смысла текста эти паттерны помогут не всегда.

Допустим, пользователь вводит следующее предложение в приложении для бронирования билетов из главы 2:

```
I need an air ticket to Berlin.
```

Можно легко определить предполагаемый пункт назначения пользователя путем поиска паттерна *to* + `GPE`, где `GPE` — именованная сущность для стран, городов и штатов. Этому паттерну удовлетворяют, в частности, фразы *to London*, *to California* и т. д.

Но представьте, что вместо этого пользователь ввел одно из следующих высказываний:

```
I am going to the conference in Berlin. I need an air ticket.
I am going to the conference, which will be held in Berlin. I would like
to book an air ticket.
```

Как видите, ни в одном из этих примеров паттерн *to* + `GPE` не поможет определить пункт назначения. В обоих случаях *to* непосредственно относится к *the conference*, а не к *Berlin*. Вместо него необходимо что-то наподобие *to* + ... + `GPE`. Но как нам узнать, что требуется (или что может оказаться) между *to* и `GPE`? Например, следующее предложение содержит паттерн *to* + ... + `GPE`, но никак не связано с бронированием билета в Берлин:

```
I want to book a ticket on a direct flight without landing in Berlin.
```

Для получения необходимой информации часто приходится исследовать отношения слов в предложении. Именно для этого может пригодиться обход дерева зависимостей предложения.

Обход дерева зависимостей означает проход по нему в произвольном порядке — не обязательно от первого токена до последнего. Например, можно остановить обход дерева зависимостей сразу же после обнаружения необходимого компонента. Помните, что дерево зависимостей предложения описывает синтаксические отношения пар слов. Их часто изображают в виде стрелок, соединяющих главный элемент отношения с дочерним. Каждое слово в предложении участвует как минимум в одном отношении. Это значит, что, обойдя все сгенерированное для данного предложения дерево зависимостей, начиная с `ROOT`, вы гарантированно пройдете по всем словам в предложении.

В этом разделе мы будем исследовать структуру предложения для определения смысла, который вкладывает в него пользователь.

Выделение информации путем обхода дерева зависимостей

Вернемся к примеру с приложением для бронирования билетов. Для поиска желаемого пункта назначения придется пройти в цикле по дереву зависимостей предложения и выяснить, связано ли семантически *to* с *Berlin*. Это легко сделать, если вспомнить отношения между главным и дочерним элементами, из которых состоит дерево зависимостей, что описывалось во врезке «Главный и дочерний элементы» на с. 53.

На рис. 6.3 приведено дерево зависимостей для предложения *I am going to the conference in Berlin*.

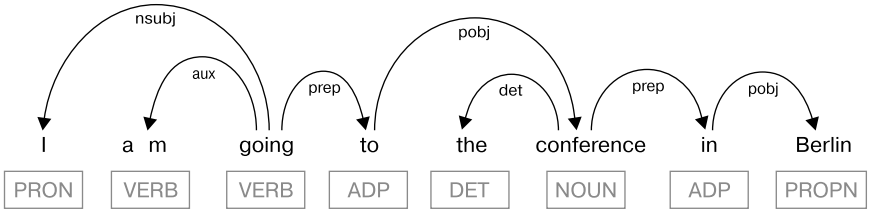


Рис. 6.3. Дерево синтаксических зависимостей высказывания

Глагол *going* — корневой элемент предложения, то есть он не является дочерним ни для какого другого слова. Его непосредственный дочерний элемент справа — *to*. Пройдя по дереву зависимостей по непосредственным правым дочерним элементам каждого слова, достигнем токена *Berlin*. Значит, между токенами *to* и *Berlin* в этом предложении существует семантическая связь.

Проход в цикле по главным элементам токенов

Теперь разберемся, как выразить отношение между словами *to* и *Berlin* программным образом. Один из способов — обход дерева зависимостей слева направо, начиная с *to*, с проходом только по правосторонним дочерним элементам всех токенов. Если подобным образом можно пройти от *to* до *Berlin*, значит, можно обоснованно считать, что между этими двумя словами существует семантическая связь.

Но у такого подхода есть недостаток. В некоторых случаях у слова может быть больше одного правостороннего дочернего элемента. Например, в предложении *I am going to the conference on spaCy, which will be held in Berlin*. у слова *conference* есть два непосредственных правых дочерних элемента: *on* и *held*. Поэтому приходится просматривать несколько веток дерева, что усложняет код.

С другой стороны, хотя у главного элемента может быть несколько дочерних, у каждого слова в предложении ровно один главный элемент. Это значит, что можно двигаться наоборот, справа налево, начав со слова *Berlin*, и пытаться достичь слова *to*. Этот процесс реализован в функции `det_destination` в следующем сценарии:

```
import spacy
nlp = spacy.load('en')
# Функция для определения пункта назначения
❶ def det_destination(doc):
    for i, token in enumerate(doc):
   ❷ if token.ent_type != 0 and token.ent_type_ == 'GPE':
     ❸ while True:
       ❹ token = token.head
          if token.text == 'to':
        ❺ return doc[i].text
       ❻ if token.head == token:
            return 'Failed to determine'
    return 'Failed to determine'
# Тестирование функции det_destination
doc = nlp(u'I am going to the conference in Berlin.')
❼ dest = det_destination(doc)
print('It seems the user wants a ticket to ' + dest)
```

В функции `det_destination` ❶ проходим в цикле по токенам полученного предложения в поисках сущности типа `GPE` ❷. В случае ее обнаружения начинаем цикл `while` ❸, в котором проходим по главным элементам всех токенов, начиная с токена, содержащего сущность типа `GPE` ❹. Цикл завершается по достижении либо токена, содержащего *to* ❺, либо корня предложения. Найти корень предложения можно путем сравнения каждого токена с его главным элементом ❻, поскольку главный элемент корневого токена всегда ссылается на него. Или же можно проверять тег `ROOT`.

Для тестирования функции применяем конвейер к нашему образцу предложения, после чего вызываем для него функцию `det_destination` ❼.

Результаты работы сценария должны выглядеть следующим образом:

```
It seems the user wants a ticket to Berlin
```

Если поменять образец предложения так, чтобы он не содержал *to* или именованной сущности типа `GPE`, будет выведено следующее:

```
It seems the user wants a ticket to Failed to determine
```

При необходимости можно усовершенствовать этот сценарий так, чтобы для случаев, когда не удается определить пункт назначения пользователя, выдавалось другое сообщение.

Краткое изложение текста с помощью деревьев зависимостей

Конечно, сфера применения подхода с деревьями синтаксических зависимостей не ограничивается чат-ботами. Этот подход можно использовать, например, в приложениях, предназначенных для обработки отчетов. Представьте, что вам нужно разработать приложение для краткого изложения отчетов о розничных продажах, чтобы в них была лишь самая важная информация.

Например, можно извлечь только предложения, содержащие числа. Таким образом получится краткий отчет об объемах продаж, доходах и затратах. (Извлечение чисел из текста обсуждалось в главе 4.) Для большей лаконичности отчета эти избранные предложения можно сократить.

В качестве быстрого примера рассмотрим следующее предложение:

```
The product sales hit a new record in the first quarter, with 18.6 million
units sold.
```

После обработки оно должно выглядеть следующим образом:

```
The product sales hit 18.6 million units sold.
```

Можно, например, проанализировать деревья зависимостей предложений по следующим шагам.

1. Выделить целиком фразу, в которой есть число (в данном примере `18.6`), пройдя по главным элементам токенов. Нужно начинать с токена, содержащего число, и двигаться слева направо.

2. Обойти дерево зависимостей от основного (смыслового) слова выделенной фразы (того, главный элемент которого находится за ее пределами) и до основного глагола предложения, проходя по главным элементам и выбирая их для использования в новом предложении.
3. Извлечь подлежащее, которое соответствует смысловому глаголу и включает определяющее слово и некоторые другие модификаторы, вместе с его левосторонними дочерними элементами.

Рис. 6.4. Сжатие предложения за счет включения лишь самых важных элементов

Для начала в этом примере необходимо выделить фразу *18.6 million units sold*. Следующий фрагмент кода демонстрирует реализацию первого шага программным образом:

```
doc = nlp(u"The product sales hit a new record in the first quarter,
   with 18.6 million units sold.")
phrase = ''
for token in doc:
❶ if token.pos_ == 'NUM':
      while True:
        phrase = phrase + ' ' + token.text
      ❷ token = token.head
      ❸ if token not in list(token.head.lefts):
          phrase = phrase + ' ' + token.text
        if list(token.rights):
          phrase = phrase + ' ' + doc[token.i+1:].text
       ❹ break
    ❺ break
print(phrase.strip())
```

Проходим в цикле по токенам предложения в поисках токена, представляющего число ❶. Когда таковой обнаруживаем, начинаем выполнение цикла `while`, в котором проходим по правосторонним главным элементам ❷, стартуя с токена числа. Далее формируем новую фразу, присоединяя текст всех главных элементов в конец переменной `phrase`. С целью убедиться, что главный элемент следующего токена располагается справа от него, проверяем, включен ли токен в список левосторонних дочерних элементов его главного элемента ❸. Если это условие ложно, прерываем выполнение цикла `while` ❹, а затем и внешнего цикла `for` ❺.

Далее обходим главные элементы токенов, начиная со смыслового слова фразы, содержащей число (в данном примере это слово *sold*), и до тех пор, пока не будет достигнут смысловой глагол предложения (в данном примере *hit*) за исключением предлога (в данном случае *with*). Реализовать это можно так, как показано в листинге:

```
while True:
❶ token = doc[token.i].head
  if token.pos_ != 'ADP':
  ❷ phrase = token.text + phrase
❸ if token.dep_ == 'ROOT':
  ❹ break
```

Обходим главные элементы токенов ❶ в цикле `while`, добавляя в конец формируемой фразы текст из этих элементов ❷. По достижении смыслового глагола (с меткой `ROOT`) ❸ прерываем выполнение цикла ❹.

Наконец, выбираем подлежащее вместе с его левосторонними дочерними элементами: *The* и *product*. В этом примере роль подлежащего играет *sales*, так что нам нужно выбрать следующий именной фрагмент: *The product sales*. Сделать это можно с помощью кода:

```
❶ for tok in token.lefts:
  ❷ if tok.dep_ == 'nsubj':
    ❸ phrase = ' '.join([tok.text for tok in tok.lefts]) + ' ' +
      tok.text + ' ' + phrase
      break
❹ print(phrase)
```

Начинаем с обхода в цикле дочерних элементов смыслового глагола ❶ в поисках подлежащего ❷. Затем присоединяем к началу дочерние элементы подлежащего и подлежащее фразы ❸, после чего выводим полученную фразу ❹.

В результате должно быть выведено следующее:

```
The product sales hit 18.6 million units sold.
```

Этот результат — сжатая версия исходного предложения.

Попробуйте сами

Напишите сценарий для краткого изложения финансовых отчетов так, чтобы в них были выделены только предложения, содержащие фразы, которые относятся к денежным суммам. Кроме того, этот сценарий должен сократить выбранные предложения до такой степени, чтобы они содержали только подлежащее, смысловой глагол, относящуюся к денежной сумме фразу и токены, которые можно выбрать при обходе по главным элементам, начиная со смыслового слова относящейся к деньгам фразы и заканчивая смысловым глаголом предложения. Например, получив предложение:

```
The company, whose profits reached a record high this year, largely
attributed to changes in management, earned a total revenue of
$4.26 million.
```

ваш сценарий должен вернуть следующее:

```
The company earned revenue of $4.26 million.
```

Здесь роль смыслового слова фразы *$4.26 million* играет *million*. Главный элемент для токена *million* — *of*, который является дочерним элементом *revenue*, а тот, в свою очередь, является дочерним элементом *earned*, смыслового глагола предложения.

Усовершенствование чат-бота для бронирования билетов с помощью учета контекста

Как вы, вероятно, уже поняли, единого решения для всех задач интеллектуальной обработки текста не существует. Например, приведенный ранее в этой главе сценарий бронирования билетов сможет найти пункт назначения, только если входное предложение содержит слово *to*.

Чтобы отдача от этих сценариев была больше, необходимо учитывать контекст для выбора подходящего ответа. Расширим функциональность сценария бронирования билетов, чтобы он мог обрабатывать

более широкий спектр вводимых пользователями данных, включая высказывания, не содержащие пару *to* + `GPE` в каком-либо сочетании. Рассмотрим следующее высказывание:

```
I am attending the conference in Berlin.
```

В нем пользователь выражает намерение отправиться в Берлин без *to*. В предложении присутствует только сущность типа `GPE`. В подобных случаях чат-бот может задать подтверждающий вопрос наподобие такого:

```
You want a ticket to Berlin, right?
```

Этот усовершенствованный чат-бот для бронирования билетов реагирует по-разному в трех различных ситуациях.

- Пользователь выражает ясное намерение забронировать билет в определенное место.
- Сразу не понятно, хочет пользователь забронировать билет в упомянутый им пункт назначения или нет.
- Текст пользователя не содержит упоминания о каком-либо пункте назначения.

Чат-бот генерирует подходящий ответ в зависимости от того, к какой из этих категорий относится вводимый пользователем текст. Рисунок 6.5 иллюстрирует варианты обработки вводимого пользователем текста.

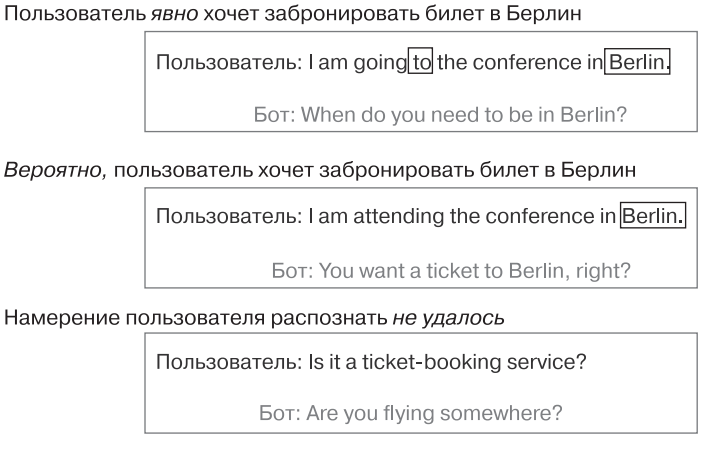


Рис. 6.5. Пример обработки вводимого пользователем текста

Эта схема работы реализована в следующем сценарии. Для удобства код разбит на несколько частей.

Первый фрагмент кода содержит функцию `guess_destination`, предназначенную для поиска в предложении сущности типа `GPE`. В него необходимо вставить функцию `det_destination`, которая ищет в предложении паттерн *to* + `GPE` (эта функция была описана и обсуждалась в подразделе «Проход в цикле по главным элементам токенов» на с. 132). Функции `guess_destination` и `det_destination` необходимы для обработки первого и второго вариантов ввода пользователя соответственно.

```
import spacy
nlp = spacy.load('en')
# Вставьте сюда функцию det_destination из предыдущего листинга
# ...
def guess_destination(doc):
  for token in doc:
  ❶ if token.ent_type != 0 and token.ent_type_ == 'GPE':
    ❷ return token.text
❸ return 'Failed to determine'
```

Код в функции `guess_destination` проходит в цикле по токенам предложения в поисках сущности типа `GPE` ❶. После обнаружения функция возвращает ее вызывающему коду ❷. Если же найти такую сущность не удалось, функция возвращает сообщение `'Failed to determine'` ❸, означающее, что предложение не содержит сущности типа `GPE`.

В следующей функции `gen_response` генерируется ответ на основе кода, возвращаемого функцией, описанной в предыдущем фрагменте:

```
def gen_response(doc):
❶ dest = det_destination(doc)
  if dest != 'Failed to determine':
  ❷ return 'When do you need to be in ' + dest + '?'
❸ dest = guess_destination(doc)
  if dest != 'Failed to determine':
  ❹ return 'You want a ticket to ' + dest +', right?'
❺ return 'Are you flying somewhere?'
```

Код функции `gen_response` начинается с вызова функции `det_destination` ❶, которая определяет, содержит ли данное высказывание пару *to* + `GPE`. В случае обнаружения такой пары считаем, что пользователь хочет забронировать билет в указанный пункт назначения, поэтому уточняем время вылета ❷.

Если функции `det_destination` не удалось найти в высказывании пару *to* + `GPE`, вызывается функция `guess_destination` ❸, которая пытается найти в нем сущность типа `GPE`. При обнаружении такой сущности пользователю задается подтверждающий вопрос, действительно ли он хочет лететь в этот пункт назначения ❹. В противном случае сценарий спрашивает пользователя, хочет ли он вообще куда-либо полететь ❺.

Для проверки кода в деле применяем к предложению конвейер, а затем передаем объект `Doc` в функцию `gen_response`, описанную в предыдущем листинге:

```
doc = nlp(u'I am going to the conference in Berlin.')
print(gen_response(doc))
```

Для приведенного в этом примере высказывания результат работы сценария должен выглядеть следующим образом:

```
When do you need to be in Berlin?
```

Можете поэкспериментировать с примером высказывания, чтобы получить различные варианты ответов.

Повышаем IQ чат-бота за счет поиска подходящих модификаторов

Чтобы сделать чат-бот умнее, можно воспользоваться деревьями зависимостей для поиска модификаторов для конкретных слов. Например, научив наше приложение распознавать прилагательные, подходящие для заданного существительного, достаточно будет просто сказать чат-боту: *I'd like to read a book.*, на что он ответит нечто вроде *Would you like a fiction book?*.

Модификатор (modifier) — необязательный элемент фразы или простого предложения, меняющий смысл другого элемента. Удаление модификатора не всегда меняет основной смысл предложения, но делает его более общим. В качестве небольшого примера рассмотрим следующие два предложения:

```
I want to read a book.
I want to read a book on Python.
```

В первом предложении модификаторы не используются. Во втором используется модификатор *on Python*, конкретизирующий запрос.

Для конкретизации необходимо использовать модификаторы. Например, чтобы должным образом отреагировать на сообщение пользователя, полезно знать, какие модификаторы можно использовать в сочетании с определенным существительным или глаголом.

Возьмем такую фразу:

```
That exotic fruit from Africa.
```

В этой именной группе главный элемент — *fruit*, а *that* и *exotic* — *премодификаторы* (premodifiers), то есть модификаторы, располагающиеся перед уточняемым словом. *from Africa* представляет собой фразу-*постмодификатор* (postmodifier) — модификатор, следующий за словом, значение которого он сужает или уточняет. На рис. 6.6 приведено дерево зависимостей для данной фразы.

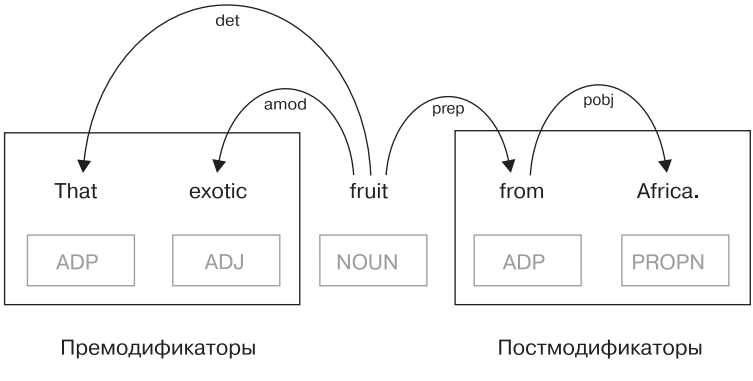


Рис. 6.6. Примеры пре- и постмодификаторов

Пусть нам нужно установить возможные прилагательные-модификаторы для слова *fruit*. (Прилагательные-модификаторы — это всегда премодификаторы.) Кроме того, хотелось бы знать, какие сущности типа `GPE` можно найти в числе постмодификаторов того же слова. Данная информация в дальнейшем окажется полезной при генерации высказываний во время беседы о фруктах.

Схема работы реализована в следующем сценарии:

```
  import spacy
  nlp = spacy.load('en')
❶ doc = nlp(u"Kiwano has jelly-like flesh with a refreshingly fruity taste.
  This is a nice exotic fruit from Africa. It is definitely worth trying.")
❷ fruit_adjectives = []
❸ fruit_origins = []
  for token in doc:
  ❹ if token.text == 'fruit':
    ❺ fruit_adjectives = fruit_adjectives + [modifier.text for modifier
      in token.lefts if modifier.pos_ == 'ADJ']
    ❻ fruit_origins = fruit_origins + [doc[modifier.i + 1].text for modifier
      in token.rights if modifier.text == 'from' and
      doc[modifier.i + 1].ent_type != 0]
  print('The list of adjectival modifiers for word fruit:',
    fruit_adjectives)
  print('The list of GPE names applicable to word fruit as postmodifiers:',
    fruit_origins)
```

Начинаем с применения конвейера к короткому тексту, содержащему слово *fruit* с пре- и постмодификаторами ❶. Объявляем два пустых списка: `fruit_adjectives` ❷ и `fruit_origins` ❸. В первом будут храниться все найденные для слова *fruit* прилагательные-модификаторы. Во втором — все найденные среди постмодификаторов слова *fruit* сущности типа `GPE`.

Далее ищем в цикле по токенам всего текста слово *fruit* ❹. Найдя его, сначала определяем прилагательные-премодификаторы этого слова, проходя по его левосторонним дочерним элементам и выбирая только прилагательные (определители и составные слова также могут быть премодификаторами). Добавляем прилагательные-модификаторы в список `fruit_adjectives` ❺.

Затем ищем постмодификаторы — в частности, именованные сущности, — проходя по правосторонним синтаксическим дочерним элементам слова *fruit*, и добавляем их в список `fruit_origins` ❻.

Сценарий выводит два списка:

```
The list of adjectival modifiers for word fruit: ['nice', 'exotic']
The list of GPE names applicable to word fruit as postmodifiers: ['Africa']
```

Теперь наш бот «знает», что фрукт может быть приятным на вкус, экзотическим (или же и приятным, и экзотическим одновременно), а также может поставляться из Африки.

Резюме

Когда необходимо обработать высказывание (или просто фразу), важно изучить его структуру и выяснить, каким общим паттернам оно следует. Выявить паттерны можно с помощью лингвистических признаков spaCy. Обладая этой информацией, программа начинает «понимать» намерения пользователя и реагировать на них должным образом.

Паттерны на основе лингвистических признаков хорошо подходят для распознавания общей структуры предложений, имеющих подлежащее, вспомогательный модальный глагол, смысловой глагол и прямое дополнение. Но на практике в приложениях приходится распознавать более сложные варианты структур и обрабатывать более широкий спектр текстовых данных, вводимых пользователями. Здесь пригодятся деревья зависимостей предложений. Дерево зависимостей можно обходить различными способами, выделяя из него необходимые элементы информации. Например, с помощью деревьев зависимостей можно находить модификаторы конкретных слов, а затем использовать эту информацию для генерации осмысленного текста.

7 Визуализация

Вероятно, простейший способ почерпнуть из данных полезную информацию — представить их визуально. С помощью наглядной визуализации, наподобие приведенной на рис. 7.1, можно сразу же выявить паттерны в данных.

В этой главе вам предстоит научиться визуализировать синтаксическую структуру предложений и именованных сущностей в документе с помощью встроенных средств визуализации библиотеки spaCy — средства визуализации зависимостей displaCy и средства визуализации именованных сущностей displaCy.

Начнем с изучения интерактивных демонстрационных версий средств визуализации (они доступны на сайте создателя displaCy — компании Explosion AI) и разберемся, на что эти средства способны. Далее посмотрим, как запустить веб-сервер displaCy на своей машине для визуализации программным способом объектов `Doc` из кода spaCy. Вы научитесь настраивать визуализации под свои потребности и узнаете, как использовать displaCy для визуализации подготовленных вручную данных без передачи объекта `Doc`.

Знакомство с встроенными средствами визуализации spaCy

Как работают средства визуализации зависимостей и именованных сущностей displaCy? Простейший способ познакомиться со встроенными средствами визуализации библиотеки spaCy — воспользоваться

их интерактивными демонстрациями, доступными на сайте https://explosion.ai/demos/. На этой странице вы найдете ссылки на страницы демонстраций двух средств визуализации displaCy, а также другие ссылки на демонстрации.

Средство визуализации зависимостей displaCy

Средство визуализации зависимостей displaCy предназначено для визуализации синтаксической структуры введенного текста. Для работы с интерактивной демонстрацией перейдите по ссылке https://explosion.ai/demos/displacy. Замените пример предложения в поле Text to parse (Текст для синтаксического разбора) вашим текстом, после чего щелкните на пиктограмме поиска (увеличительное стекло), чтобы сгенерировать визуализацию. Результат будет выглядеть так, как показано на рис. 7.1.

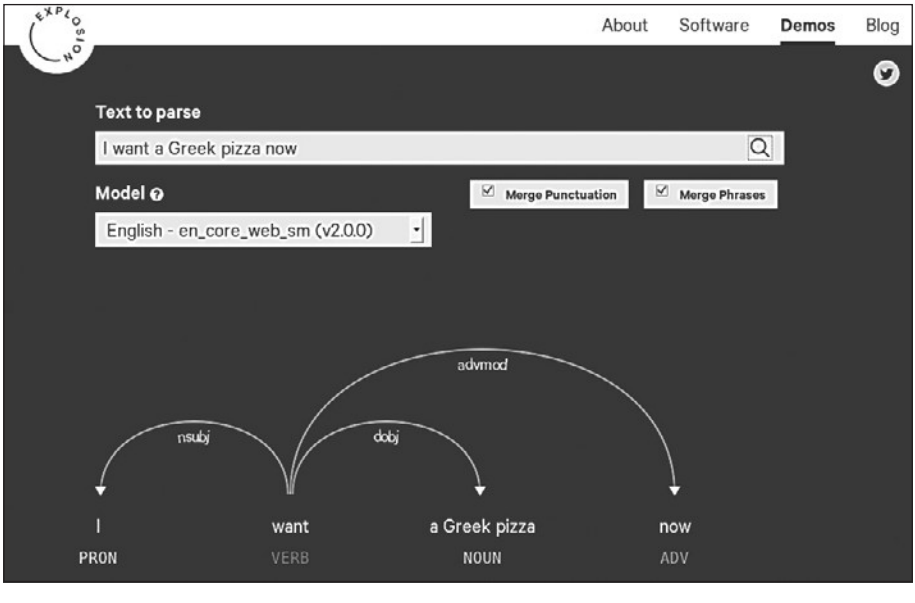


Рис. 7.1. Средство визуализации зависимостей displaCy на сайте компании Explosion AI

Средство визуализации зависимостей сразу же отображает синтаксическую структуру введенного текста, в том числе теги частей речи и синтаксические зависимости в нем.

Средство визуализации позволяет гибко настраивать визуальное представление с помощью кнопок-флажков Merge Punctuation (Присоединять знаки препинания) и Merge Phrases (Производить слияние фраз). Нажав кнопку Merge Punctuation, вы присоедините знак препинания к предыдущему токену, что сделает визуализацию более компактной, а значит, и более удобочитаемой. При нажатии кнопки Merge Phrases производится слияние всех именных групп в единые токены, как показано на рис. 7.1. По умолчанию обе эти опции включены.

Вы можете изменить одну или обе эти настройки путем снятия соответствующего флажка (флажков). Например, если снять флажок Merge Phrases для предложения *I want a Greek pizza now.*, вы увидите более подробный разбор зависимостей для данного предложения, демонстрирующий зависимости внутри фразы *a Greek pizza*.

Если установить флажок Merge Phrases, можно получить более компактное дерево зависимостей, что особенно удобно при работе с предложениями, содержащими несколько именных групп. Например, рассмотрим следующее предложение: *I see a few young people working in their vegetable field.* Оно содержит две именные группы: *a few young people* и *their vegetable field.* Первая представляет собой прямое дополнение глагола *see*, вторая — предложное дополнение, модифицирующее глагол *work*, с метками зависимостей `dobj` и `pobj` соответственно. Иными словами, эти метки зависимости относятся скорее к существительным в соответствующих фразах, а не к предложению в целом.

Помимо опций Merge Punctuation и Merge Phrases, есть и опция выбора из выпадающего списка доступных моделей нужной вам статистической модели языка. Благодаря ей можно использовать модели для разбора синтаксических зависимостей без скачивания и установки в рабочей среде. В настоящее время доступны следующие модели[1]: `en_core_web_sm`, `en_core_web_md` и `en_core_web_lg`, а также маленькие (`sm`) модели для других европейских языков, среди которых немецкий, испанский, португальский, французский, итальянский и нидерландский.

[1] В настоящее время (версия 2.3.0) для английского языка также доступна только маленькая модель.

Средство визуализации именованных сущностей displaCy

Средство визуализации именованных сущностей displaCy предназначено для визуализации именованных сущностей введенного текста. Его интерактивную демонстрационную версию можно найти по ссылке https://explosion.ai/demos/displacy-ent/. С точки зрения пользователя эта версия работает аналогично демонстрационной версии средства визуализации зависимостей displaCy, о которой мы говорили в предыдущем разделе. Чтобы сгенерировать визуализацию текста, введите его в текстовое поле, а затем щелкните на пиктограмме поиска. Средство визуализации обработает запрос и выведет исходный текст внизу окна, выделив найденные именованные сущности вместе с метками, как показано на рис. 7.2.

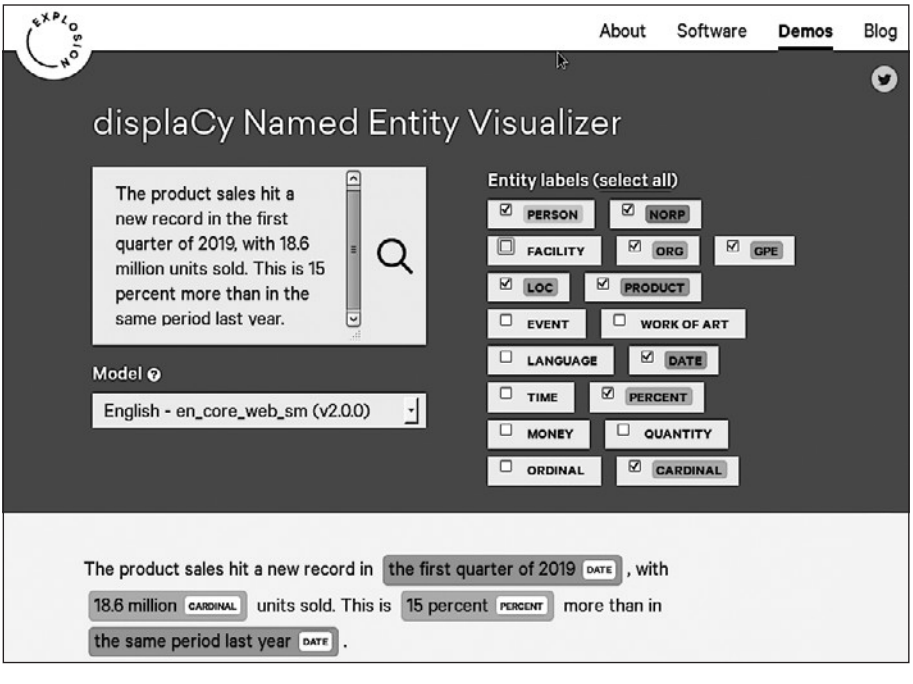


Рис. 7.2. Средство визуализации именованных сущностей displaCy на сайте компании Explosion AI

При этом путем выбора/снятия флажков в блоке Entity labels (Метки сущностей) можно выбирать, какие именованные сущности приложение должно распознавать в полученном тексте. В приведенном на рис. 7.2

примере к списку выбранных по умолчанию типов меток сущностей были добавлены PERCENT и CARDINAL. В результате добавления типа PERCENT средство визуализации будет распознавать фразы, выражающие процентное соотношение либо включающие символ %. Добавление типа сущностей CARDINAL гарантирует, что средство визуализации будет распознавать во введенном тексте фразы, относящиеся к числительным.

Какие флажки выбирать — зависит от контекста. При обработке финансового отчета имеет смысл выбрать флажки MONEY и DATE. Если же отчет содержит данные о финансовых операциях более чем одной компании, можно выбрать флажок метки сущности ORG, чтобы средство визуализации выделило в тексте названия этих компаний.

Визуализация из кода spaCy

Начиная со spaCy v2.0, средства визуализации displaCy включены в состав ядра библиотеки. Это значит, что в коде Python их можно использовать сразу же после установки spaCy.

Для этого необходимо выполнить следующее: запустить встроенный веб-сервер, отправить ему объект Doc (или список объектов Doc) для визуализации. Сервер сгенерирует визуализацию для полученного объекта Doc. Увидеть ее можно будет в браузере. В данном разделе мы рассмотрим несколько примеров.

Визуализация разбора зависимостей

Следующий сценарий демонстрирует простейший способ визуализации дерева зависимостей предложения:

```
import spacy
nlp = spacy.load('en')
❶ doc = nlp(u"I want a Greek pizza.")
❷ from spacy import displacy
❸ displacy.serve(doc, ❹style='dep')
```

В этом коде создаем объект Doc для отправки в displaCy ❶. Затем импортируем библиотеку displaCy из ядра библиотеки spaCy ❷, после чего запускаем веб-сервер displaCy и передаем ему объект Doc. Обе операции

производятся с помощью вызова одного метода `displacy.serve()` ❸. Значение `'dep'` параметра `style` означает, что displaCy должен использовать средство визуализации зависимостей ❹ и визуализировать дерево зависимостей текста, содержащегося в объекте `Doc`. Если вам интересно реализовать опции кнопок-флажков, которые уже встречались в этой главе, обратитесь к разделу «Попробуйте сами» далее.

Вне зависимости от того, как выполняется код — в сеансе Python или в качестве отдельного сценария, выполнение входит в бесконечный цикл и демонстрирует сообщения от веб-сервера displaCy. Первые сообщения, которые вы увидите, должны выглядеть так:

```
Serving on port 5000...
Using the 'dep' visualizer
```

Это значит, что веб-сервер сгенерировал визуализацию дерева зависимостей переданного ему текста и обслуживает HTTP-запросы на порте 5000 (по умолчанию) локальной машины. На практике — для просмотра визуализации достаточно перейти по ссылке http://localhost:5000. В данном примере визуализация должна выглядеть так, как на рис. 7.3.

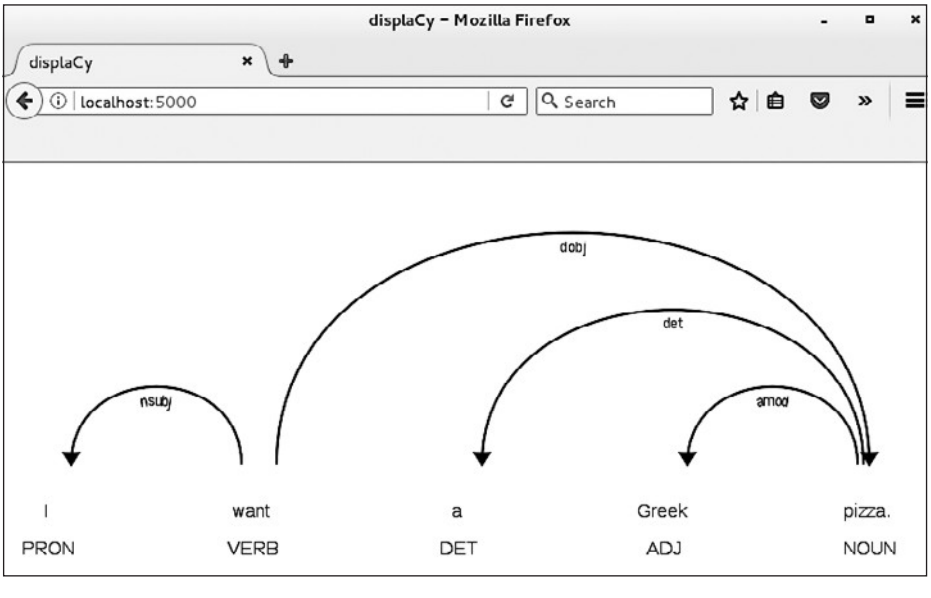


Рис. 7.3. Пример визуализации дерева зависимостей, которую можно сгенерировать из кода на языке Python, а затем просмотреть в браузере

Чтобы остановить сервер displaCy, введите `Ctrl-C` в терминале, где выполняется запустивший сервер сценарий. В результате должно быть выведено следующее заключительное сообщение сервера:

```
Shutting down server on port 5000.
```

После останова сервера вы не сможете генерировать новые визуализации в браузере, но уже сгенерированные будут для вас доступны.

Попробуйте сами

Попробуйте в работе средство визуализации именованных сущностей, для чего внесите соответствующие изменения в сценарий средства визуализации зависимостей из предыдущего раздела. Чтобы displaCy использовал средство визуализации именованных сущностей, необходимо в методе `displacy.serve()` присвоить параметру `style` значение `'ent'`.

Чтобы визуализация была интереснее, возьмите более длинный текст, состоящий, скажем, из нескольких предложений. Можете попробовать такой:

```
Microsoft Windows is a family of proprietary operating systems
developed and sold by Microsoft. Bill Gates announced Microsoft
Windows on November 10, 1983. Microsoft first released Windows
for sale on November 20, 1985. Windows 1.0 was initially sold for
$100.00, and its sales surpassed 500,000 copies in April 1987. For
comparison, more than a million copies of Windows 95 were sold in
just the first 4 days.
```

После вызова метода `displacy.serve()` объекта `Doc` перейдите в браузере по адресу http://localhost:5000 для просмотра визуализации. Обратите внимание на распознанные средством распознавания именованные сущности в этом тексте и их типы. Вы увидите, что средство распознавания сущностей нашло в тексте имена людей, товары и компании, а также фразы, относящиеся к датам, числам и денежным суммам.

Визуализация по отдельным предложениям

Визуализация деревьев зависимостей удобна при работе с одним предложением. Но при визуализации длинного текста графическое представление оказывается громоздким и занимает много места, из-за чего его неудобно читать при отображении в одну строку. displaCy,

получив содержащий несколько предложений объект Doc, генерирует для каждого предложения отдельную визуализацию, но при этом все визуализации располагает в одну строку.

Вместо того чтобы передавать объект Doc, можно производить визуализацию по отдельным предложениям. Это бывает удобно, когда нужно выделить смысл связного текста в целом и исследовать последовательность предложений. Начиная с версии 2.0.12, в displaCy можно передавать объекты Span и располагать визуализации по строкам. Если бы вы хотели создать визуализацию для каждого предложения в doc.sents, вы смогли бы передать doc.sents в виде списка, таким образом:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"I have a relaxed pair of jeans. Now I want a skinny pair.")
❶ spans = list(doc.sents)
from spacy import displacy
displacy.serve(❷spans, style='dep')
```

Как было сказано в главе 3, свойство doc.sents объекта Doc представляет собой итератор для обхода содержащихся в этом объекте Doc предложений. Поэтому с его помощью нельзя ссылаться на предложения по индексу, но можно проходить по ним в цикле. Или можно создать список объектов Span, в котором каждый интервал будет соответствовать предложению. В этом коде преобразуем содержащиеся в объекте Doc предложения в список объектов Span ❶. Затем передадим этот список интервалов методу displacy.serve() для визуализации ❷.

В результате в отдельной строке будут сгенерированы и выведены визуализации для каждого предложения, благодаря чему их можно будет просматривать путем вертикальной прокрутки.

Настройка визуализаций под свои задачи с помощью аргумента options

Помимо аргументов docs и style, встречавшихся в примерах выше, у метода displacy.serve() есть несколько других параметров. Вероятно, самый интересный из них — аргумент options, который позволяет задавать ассоциативный массив параметров при настройке макета визуализации под свои задачи. В этом разделе рассмотрим некоторые из наиболее полезных настроек options.

Использование аргумента options средства визуализации зависимостей

Просматривать длинные предложения при выводе в одну строку довольно неудобно. Для подобных случаев предусмотрена возможность создания визуализаций в *компактном режиме*, что занимает меньше места. Для этого необходимо установить значение опции `'compact'` в аргументе `options` в `True`, как показано в следующем сценарии. Шрифт, используемый средством визуализации, также меняется. (Полный список доступных опций можно найти в документации по API displaCy по адресу https://spacy.io/api/top-level/#options-dep/.)

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"I want a Greek pizza.")
from spacy import displacy
options = {❶'compact': True, ❷'font': 'Tahoma'}
displacy.serve(doc, style='dep', ❸options=options)
```

Метод `displacy.serve()` ожидает, что аргумент `options` представляет собой ассоциативный массив. В этом примере задаем только две опции: `'compact'` со значением `True` ❶ и `'font'` со значением `'Tahoma'` ❷, в обоих случаях меняя используемые по умолчанию значения. (Средство визуализации позволяет использовать стандартные веб-шрифты, например Arial, Courier и др.) Далее передаем ассоциативный массив опций в виде аргумента `options` ❸.

На рис. 7.4 показано, что должен отобразить браузер, когда вы перейдете по адресу http://localhost:5000 после запуска сценария.

Прямоугольные дуги на рисунке выглядят немного необычно, но они делают визуализацию более сжатой, и поэтому всю схему можно просмотреть без необходимости прокрутки.

Попробуйте сами

В интерактивной демонстрации средства визуализации зависимостей, о которой шла речь в подразделе «Средство визуализации зависимостей displaCy» на с. 145, использовались кнопки-флажки Merge Punctuation

и Merge Phrases. В spaCy можно объединять именные группы в один токен с помощью опции `collapse_phrases` (на рис. 7.1 показана эквивалентная опция на сайте средства визуализации displaCy), по умолчанию равной `False`. Опция `collapse_punct`, отвечающая за присоединение знаков препинания к токенам, по умолчанию равна `True`.

Измените код в предыдущем сценарии так, чтобы `collapse_phrases` в наборе передаваемых опций устанавливалась равной `True`. Запустите сценарий и просмотрите полученную визуализацию в браузере с целью убедиться, что именные группы отображаются как единый токен.

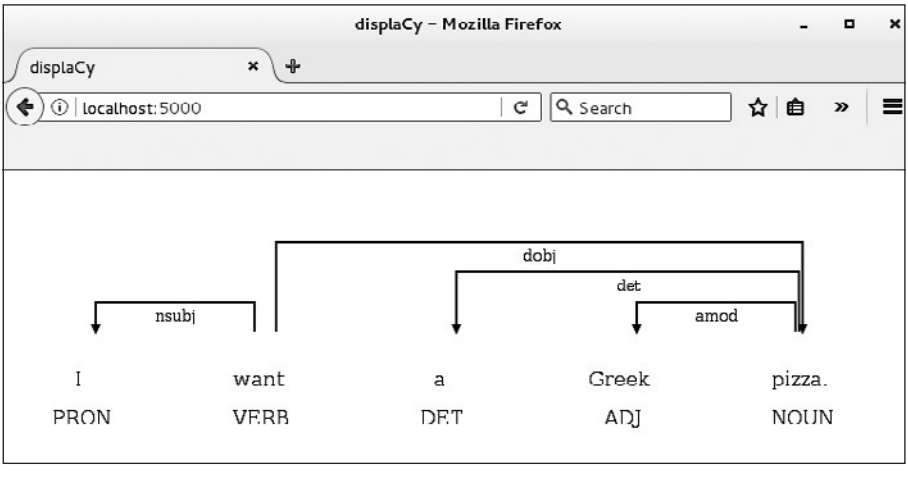


Рис. 7.4. Пример настройки средства визуализации под задачи пользователя

Использование аргумента options средства визуализации именованных сущностей

Список опций средства визуализации именованных сущностей (его можно найти по адресу https://spacy.io/api/top-level/#displacy_options-ent/) намного короче, чем у средства визуализации зависимостей. Используя средство визуализации сущностей, можно с помощью опции `'ent'` выбрать, какие типы сущностей выделять, а с помощью опции `'colors'` — переопределить используемые по умолчанию цвета.

Более важной из двух опций является первая, поскольку она позволяет указать средству визуализации на то, что нужно выделять только сущности определенных типов. В следующем примере приведен случай, в котором имеет смысл ограничивать число отображаемых типов сущностей.

В этом сценарии средству визуализации сущностей не передаются никакие опции, следовательно, оно будет выделять сущности всех типов, содержащиеся в передаваемом объекте `Doc`:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"In 2011, Google launched Google +, its fourth foray
  into social networking.")
❶ doc.user_data['title'] = "An example of an entity visualization"
from spacy import displacy
displacy.serve(doc, style=❷'ent')
```

Мы воспользовались атрибутом `user_data` объекта `Doc` для задания его названия ❶. Средство визуализации displaCy автоматически выводит содержащийся в этом атрибуте текст в качестве заголовка визуализации. Добавлять заголовок к визуализации не обязательно, но это стоит сделать для снабжения визуализаций комментариями.

Мы задаем параметр `style` метода `displacy.serve()` равным `'ent'` ❷, чтобы displaCy использовал средство визуализации именованных сущностей. В итоге должна получиться визуализация наподобие приведенной на рис. 7.5 (в книге изображение приведено в оттенках серого, а на сайте оно полноцветное).

В целом эта визуализация выглядит неплохо. Но в данном контексте выделять порядковое числительное *fourth*, вероятно, не обязательно — неясно, зачем в данном контексте может понадобиться выделение этой информации. С помощью опции `'ents'` выбираем типы сущностей, которые должны быть выделены в визуализации. Реализация данной возможности показана в следующем коде. Чтобы увидеть это в действии, замените последнюю строку кода из предыдущего сценария следующими двумя строками, после чего запустите модифицированный сценарий:

```
options = {❶'ents': ["ORG", "PRODUCT", "DATE"], ❷'colors': {"ORG": "aqua",
  "PRODUCT": "aqua"}}
displacy.serve(doc, style='ent', options=options)
```

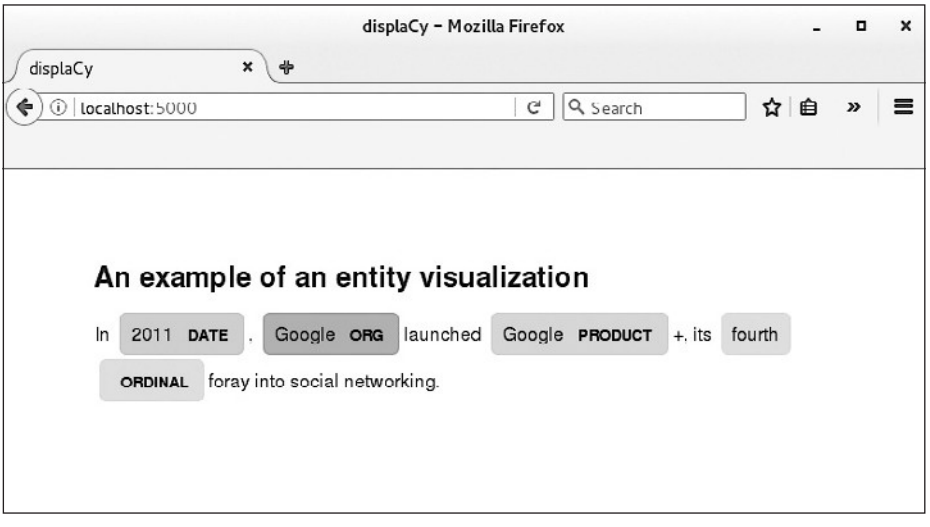


Рис. 7.5. Пример визуализации именованных сущностей с опциями по умолчанию

На этот раз средство визуализации не распознает никаких сущностей, за исключением тех, что относятся к типам `ORG`, `PRODUCT` и `DATE` ❶. Здесь продемонстрировано и использование опции `'colors'`, с помощью которой можно менять назначаемые по умолчанию для различных типов сущностей цвета: для типов `ORG` и `PRODUCT` назначаем цвет `"aqua"` ❷.

ПРИМЕЧАНИЕ

При использовании опции 'colors' можно указывать либо названия веб-цветов для типов сущностей, либо шестнадцатеричные коды цветов. Так, в примере на с. 154 вместо названия цвета aqua можно было использовать шестнадцатеричный код #00FFFF.

На рис. 7.6 показано, как выглядит итоговая визуализация.

Как вы можете заметить, эта визуализация практически идентична приведенной на рис. 7.5. Но в данном случае средство визуализации не выделяет сущности типа `ORDINAL`, поскольку переданный список опций `'ents'` не включает этот тип.

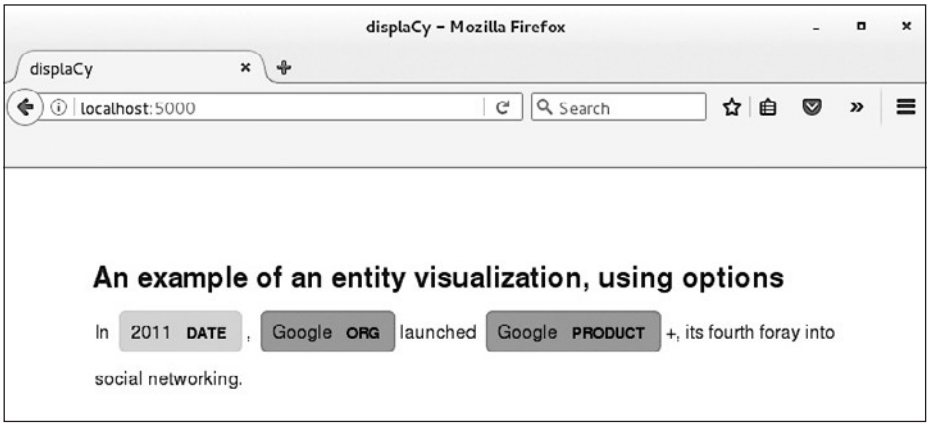


Рис. 7.6. Пример визуализации сущностей с указанием опций 'ents' и 'colors'

Экспорт визуализации в файл

В вышеприведенных примерах для просмотра сгенерированных визуализаций использовался веб-сервер displaCy. Как вы поняли, сгенерированную с использованием метода `displacy.serve()` визуализацию можно открывать до тех пор, пока запущенный `displacy.serve()` работает.

С помощью метода `displacy.render()` это ограничение можно обойти, чтобы создать визуализацию для использования в будущем. Метод `displacy.render()` позволяет визуализировать разметку в виде HTML-страницы и сохранять ее в отдельном файле, который позднее можно будет открыть в любом браузере без необходимости запуска веб-сервера.

В следующем сценарии показано использование метода `displacy.render()` для визуализации именованных сущностей, которая отображена на рис. 7.5:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u"In 2011, Google launched Google +, its fourth foray
  into social networking.")
❶ doc.user_data["title"] = "An example of an entity visualization"
  # В следующем блоке кода мы указываем displaCy визуализировать разметку
  # в виде полной HTML-страницы
  from spacy import displacy
❷ html = displacy.render(doc, style='ent', ❸page=True)
  # В следующем блоке кода мы сохраняем сгенерированный displacy.render()
  # HTML-файл на диск локальной машины
```

```
❹ from pathlib import Path
❺ output_path = Path("/visualizations/ent_visual.html")
❻ output_path.open("w", encoding="utf-8").write(html)
```

Код в этом сценарии можно разбить на три части. Первый блок вам уже знаком: в нем создается конвейер обработки текста, который потом мы применяем к тексту. Заголовок для объекта `Doc` указываем с помощью его атрибута `user_data` ❶.

Вторая и третья части начинаются со строки комментария. Во втором блоке отображаем визуализацию именованных сущностей для созданного на предыдущем шаге объекта `Doc` с помощью метода `displacy.render()` ❷. В отличие от метода `displacy.serve()` метод `displacy.render()` не запускает веб-сервер, а генерирует HTML-разметку для визуализации. Благодаря тому что аргумент `page` имеет значение `True`, `displacy.render()` знает о необходимости сгенерировать разметку в виде полной HTML-страницы ❸.

В последнем блоке кода импортируем класс `Path` из модуля `pathlib`, появившегося в Python 3.4 ❹. Класс `Path` позволяет выполнять системные вызовы для объектов `Path`. В данном примере создаем экземпляр этого класса для следующего пути: `/visualizations/ent_visual.html` ❺, предполагая, что в локальной файловой системе уже есть каталог `/visualizations` (в противном случае будет сгенерировано исключение). Далее открываем файл `ent_visual.html` (если он не существует, то будет создан) в каталоге `/visualizations` и записываем в него сгенерированную на предыдущем шаге HTML-страницу ❻.

Итого, сценарий генерирует HTML-файл, содержащий визуализацию именованных сущностей для указанного текста, и сохраняет его в локальной файловой системе. Найдя HTML-файл в каталоге, в котором он был сохранен, и дважды щелкнув на нем, вы откроете файл в браузере.

Использование displaCy для отображения данных в ручном режиме

Средства визуализации displaCy позволяют создавать наборы данных для дальнейшего отображения вручную, а не передавать данные в виде объекта `Doc` или `Span`. Это удобно, когда, например, нужно визуализировать результаты работы других библиотек NLP или

создать визуализацию с набором пользовательских тегов или пользовательских меток зависимости. (Подробнее о том, как создать такие пользовательские теги и метки, см. в главе 10.)

В качестве примера вручную визуализируем предложение *I want a Greek pizza*.

Форматирование данных

Сначала необходимо перевести данные в формат displaCy — ассоциативный массив, содержащий два списка: `"words"` и `"arcs"`, как показано в следующем коде:

```
sent = {
    "words": [
     ❶ {"text": "I", "tag": "PRON"},
        {"text": "want", "tag": "VERB"},
        {"text": "a", "tag": "DET"},
        {"text": "Greek", "tag": "ADJ"},
        {"text": "pizza", "tag": "NOUN"}
    ],
    "arcs": [
     ❷ {"start": 0, "end": 1, "label": "nsubj", "dir": "left"},
        {"start": 2, "end": 4, "label": "det", "dir": "left"},
        {"start": 3, "end": 4, "label": "amod", "dir": "left"},
        {"start": 1, "end": 4, "label": "dobj", "dir": "right"}
    ]
}
```

Ассоциативный массив `sent` содержит два списка: `"words"` и `"arcs"`, каждый из которых, в свою очередь, включает набор ассоциативных массивов. Ассоциативный массив в списке `"words"` задает теги для токенов предложения ❶, а словарь в списке `"arcs"` описывает дуги дерева зависимостей, которые соединяют пары синтаксически связанных слов в предложении ❷. В этом примере предложение содержит пять слов, для которых определены четыре синтаксических отношения. Именно поэтому в списке `"words"` ассоциативного массива содержится пять элементов, а в списке `"arcs"` — четыре.

Теперь, имея ассоциативный массив с данными, мы можем сгенерировать визуализацию зависимостей для нашего образца предложения. Для ее отображения воспользуемся следующим кодом:

```
❶ from spacy import displacy
  displacy.serve(❷sent, style="dep", ❸manual=True)
```

Обратите внимание, что импортировать всю библиотеку spaCy не нужно. Достаточно импортировать из нее модуль `displacy` ❶. Далее вызываем метод `displacy.serve()`, передавая в него ассоциативный массив `sent` в качестве первого параметра вместо объекта `Doc` ❷. Третий параметр, `manual`, сообщает displaCy, что нами был создан набор данных для визуализации вручную ❸, так что displaCy не нужно извлекать данные из объекта `Doc`.

Попробуйте сами

При создании ассоциативного массива с данными для отображения визуализации вручную можно применить пользовательские теги и указать, например, что средство визуализации должно использовать теги уточненных частей речи вместо тегов общих частей речи по умолчанию.

Сделать это можно, установив опцию `fine_grained` равной `True` при передаче объекта `Doc` для визуализации, но для практики попробуйте произвести действия вручную.

В примере из подраздела «Форматирование данных» на с. 158 поменяйте теги в списке `"words"` ассоциативного массива `"sent"` так, чтобы их значения представляли собой теги уточненных частей речи. Далее запустите сервер displaCy и попросите его сгенерировать визуализацию на основе заданных в ассоциативном массиве `"sent"` данных. Перейдите в браузере по адресу http://localhost:5000 для просмотра визуализации.

Резюме

В предыдущих главах вы познакомились с визуализациями синтаксической структуры, а в этой главе уже научились их генерировать с помощью средства визуализации зависимостей displaCy. Кроме того, вы узнали, как генерировать визуальные схемы информации об именованных сущностях с помощью средства визуализации именованных сущностей displaCy.

8 Распознавание намерений

Чат-бот должен быть настоящим интеллектуалом. Например, диалоговый чат-бот должен уметь распознавать намерение пользователя, чтобы поддерживать с ним разговор, чат-бот, предназначенный для заказа еды, должен понимать требования пользователя, чтобы принять заказ. И хотя задача распознавания намерений уже рассматривалась в предыдущих главах, в этой главе мы обсудим ее более подробно.

Начнем с распознавания намерений пользователя с помощью выделения переходного глагола и прямого дополнения высказывания. Далее обсудим, как выяснить намерение пользователя по последовательности предложений, как распознать синонимы, выражающие различные возможные намерения, и определить намерение пользователя на основе семантического подобия.

Распознавание намерений с помощью выделения переходного глагола и прямого дополнения

Обычно распознавание намерения пользователя состоит из трех этапов: синтаксического разбора предложения на токены, соединения токенов маркированными дугами, отражающими синтаксические отношения, и прохода по этим дугам для выделения соответствующих токенов. Но во многих случаях для распознавания намерений пользователя достаточно выделить переходный глагол и прямое дополнение, как видно из разбора синтаксических зависимостей на рис. 8.1.

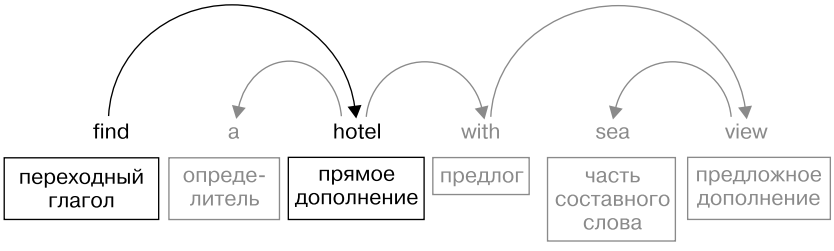


Рис. 8.1. Пример визуального представления синтаксической структуры предложения

Дуга, соединяющая переходный глагол с прямым дополнением, указывает, что пользователь намерен найти гостиницу (если объединить переходный глагол и прямое дополнение в одно слово, получится `findHotel`). В последующем коде эту структуру можно использовать в качестве *идентификатора намерения* (intent identifier), как показано ниже:

```
intent = extract_intent(doc)
if intent == 'orderPizza':
  print('We need you to answer some questions to place your order.')
  ...
elif intent == 'showPizza':
  print('Would you like to look at our menu?')
  ...
```

ПРИМЕЧАНИЕ

В главе 11 мы рассмотрим более детальные примеры использования идентификаторов намерения в коде приложения для чат-бота.

Иногда определить смысл пары «переходный глагол/прямое дополнение» не так и просто. Чтобы найти глагол и существительное, лучше всего описывающие намерение, приходится изучать синтаксические отношения переходного глагола и прямого дополнения.

В других случаях бывает, что пользователь не выражает свое намерение явным образом, так что приходится вычислять его предполагаемые намерения. В этом разделе обсудим стратегии выделения намерений с помощью структуры синтаксических зависимостей.

Получение пары «переходный глагол/прямое дополнение»

Начнем с выделения из предложения пары «переходный глагол/прямое дополнение» путем поиска для всех токенов меток зависимости, равных `dobj`. Найдем прямое дополнение и легко получим соответствующий переходный глагол в виде синтаксического главного элемента этого прямого дополнения, как показано в следующем сценарии:

```
import spacy
nlp = spacy.load('en')
❶ doc = nlp(u'show me the best hotel in berlin')
❷ for token in doc:
    if token.dep_ == 'dobj':
      print(❸token.head.text + token.text.capitalize())
```

В данном сценарии применяем конвейер к образцу предложения ❶, после чего проходим в цикле по токенам в поисках токена с меткой зависимости `dobj` ❷. Найдя его, определяем соответствующий переходный глагол как главный элемент прямого дополнения ❸. В этом примере также объединяем переходный глагол и его прямое дополнение, чтобы выразить намерение одним словом.

Результаты работы сценария выглядят так:

```
showHotel
```

Учтите, что не все предложения, содержащие пару «переходный глагол/прямое дополнение», выражают какое-либо намерение. Например, предложение *He gave me a book* просто констатирует факт. Подобные предложения можно отфильтровать путем проверки характеристик глагола и отбора только тех предложений, в которых есть глаголы настоящего времени и не третьего лица. Впрочем, такие предложения — редкость при общении пользователя с чат-ботом, принимающим заказы.

Выделение множественных намерений с помощью token.conjuncts

Иногда встречаются предложения, выражающие несколько намерений. Например, такое:

```
I want a pizza and cola.
```

В данной ситуации пользователь хочет заказать пиццу и кока-колу. Однако в большинстве случаев эти намерения можно считать частью одного составного намерения. Хотя пользователь хочет заказать разнотипные товары, подобное предложение обычно считают одним заказом с несколькими позициями. В этом примере можно распознать намерение, состоящее из пары «переходный глагол/прямое дополнение» `orderPizza`, но при этом из него можно выделить `pizza` и `cola` в качестве отдельных позиций размещаемого заказа.

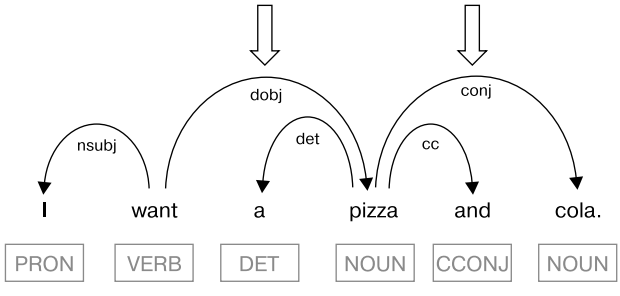


Рис. 8.2. Дерево зависимостей предложения, содержащего прямое дополнение и его конъюнкт

На схеме есть две стрелки, указывающие на дуги прямого дополнения *pizza* и его конъюнкта *cola*. *Конъюнкт* (conjunct) существительного — это другое существительное, присоединенное к первому посредством союза, такого как «и», «или» и т. д. Для выделения прямого дополнения и его конъюнкта можно воспользоваться следующим кодом:

```
doc = nlp(u'I want a pizza and cola.')
# Выделяем прямое дополнение и его конъюнкт
for token in doc:
  if token.dep_ == 'dobj':
    dobj = [token.text]
    conj = [t.text for t in ❶token.conjuncts]
# Составляем список выделенных элементов
❷ dobj_conj = dobj + conj
print(dobj_conj)
```

Выделяем конъюнкт прямого дополнения с помощью свойства `conjuncts` объекта `Token`, соответствующего этому прямому дополнению ❶. После получения прямого дополнения и его конъюнктов объединяем их в один список ❷.

Результат работы сценария выглядит так:

```
['pizza', 'cola']
```

Чтобы сформулировать намерение, необходимо выделить и глагол. Простейший способ его получения, при уже выделенном прямом дополнении, — найти синтаксический главный элемент этого прямого дополнения (с подобным примером вы уже встречались в подразделе «Получение пары “переходный глагол/прямое дополнение”» на с. 162):

```
verb = dobj.head
```

Далее, с помощью свойства `text` глагола и прямого дополнения можно составить идентификатор намерения.

Попробуйте сами

В приведенном на с. 163 сценарии вы получили доступ к конъюнкту прямого дополнения через свойство `conjuncts` объекта `Token`. В новом сценарии замените эту строку кодом, который бы извлекал конъюнкт в процессе поиска дуги с меткой `conj` при обходе наружу от прямого дополнения. Сделать это можно внутри цикла, в котором ищется прямое дополнение посредством поиска дуги с меткой `dobj`. Не забудьте убедиться, что главный элемент дуги `conj` соответствует прямому дополнению.

Выделение намерения с помощью списков слов

В некоторых случаях намерение пользователя лучше всего описывают не переходный глагол и прямое дополнение, а другие токены. Обычно они связаны отношением с переходным глаголом или прямым дополнением: просто нужно будет сделать еще один шаг и найти слова, лучше всего описывающие намерение, путем исследования синтаксических отношений переходного глагола и прямого дополнения.

В качестве примера рассмотрим следующее высказывание:

```
I want to place an order for a pizza.
```

В этом предложении намерение лучше всего описывают слова *want* и *pizza*, хотя ни одно из них не является переходным глаголом или прямым дополнением. Впрочем, из дерева зависимостей данного высказывания видно, что *want* и *pizza* относятся к переходному глаголу *place* и прямому дополнению *order* соответственно. Упомянутое дерево зависимостей приведено на рис. 8.3.

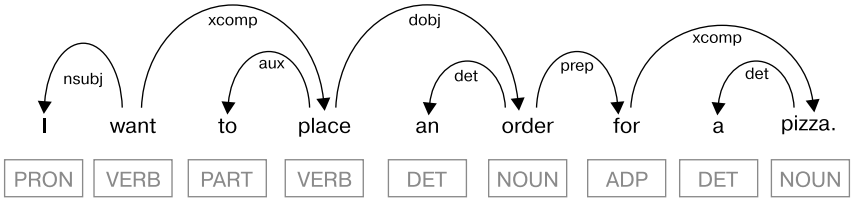


Рис. 8.3. Дерево зависимостей высказывания, переходный глагол и прямое дополнение которого не передают намерение пользователя

Для выделения слов *want* и *pizza* из высказывания воспользуемся списком заранее определенных слов и найдем их в высказывании.

Опытный программист может усомниться в эффективности «жесткого» прописывания в коде подобного списка, ведь есть вероятность, что список окажется очень длинным, особенно для использования в ряде различных контекстов. Но предназначенный для конкретной задачи, например заказа пиццы, список, скорее всего, будет на удивление коротким, так что такой подход довольно эффективен. Он реализован в следующем коде:

```
# Применяем конвейер к образцу предложения
doc = nlp(u'I want to place an order for a pizza.')
# Выделяем прямое дополнение и его переходный глагол
dobj = ''
tverb = ''
for token in doc:
❶ if token.dep_ == 'dobj':
    dobj = token
    tverb = token.head
# Выделяем глагол для описания намерения
intentVerb = ''
verbList = ['want', 'like', 'need', 'order']
❷ if tverb.text in verbList:
  intentVerb = tverb
❸ else:
```

```
    if tverb.head.dep_ == 'ROOT':
      intentVerb = tverb.head
  # Выделяем дополнение для описания намерения
  intentObj = ''
  objList = ['pizza', 'cola']
❹ if dobj.text in objList:
    intentObj = dobj
  else:
    for child in dobj.children:
      if child.dep_ == 'prep':
        ❺ intentObj = list(child.children)[0]
          break
    ❻ elif child.dep_ == 'compound':
          intentObj = child
          break
  # Выводим в консоль высказанное в образце предложения намерение
  print(intentVerb.text + intentObj.text.capitalize())
```

Как всегда, начинаем с поиска и выделения прямого дополнения и его переходного глагола ❶. Затем проверяем, присутствуют ли они в соответствующем списке предопределенных слов. Здесь, конечно, используем упрощенные списки: `verbList` содержит глаголы, которыми может воспользоваться покупатель при размещении заказа, а `objList` содержит прямые дополнения — возможные позиции меню. Начинаем с проверки переходного глагола ❷. Если его нет в списке допустимых глаголов ❸, проверяем смысловой глагол (`ROOT`) предложения, играющий роль главного элемента для переходного глагола. Эта реализация работает, даже если переходный глагол является смысловым глаголом предложения, поскольку главным элементом для смыслового глагола (`ROOT`) является он сам.

Далее обращаемся к прямому дополнению ❹. Если оно не входит в список допустимых слов, проверяем его синтаксические дочерние элементы. Начинаем с предлога этого прямого дополнения. Если таковой существует, берем его дочерний элемент (он у него может быть только один) ❺, который и будет всегда предложным дополнением.

Чтобы данный подход можно было применять к широкому спектру случаев, недостаточно только искать предлоги среди дочерних элементов прямого дополнения. Такая логика не годится, например, для следующего высказывания: *I want to place a pizza order*. Здесь отсутствует ветвь дочернего элемента-предлога. Вместо него у прямого до-

полнения есть левосторонний дочерний элемент *pizza*, который spaCy помечает тегом *compound*. Поэтому части составных элементов ищем среди дочерних элементов прямого дополнения ❻.

Наконец, выводим строку — идентификатор намерения. Она должна выглядеть так:

```
wantPizza
```

Обратите внимание, что идентификатором намерения у нас служит `wantPizza`, а не `placeOrder` (такой результат получился бы при использовании пары «переходный глагол/прямое дополнение»). Подобный подход позволяет уменьшить число идентификаторов намерений в приложении.

Поиск значений слов с помощью синонимов и семантического подобия

На английском, как и на многих других естественных языках, одни и те же мысли и намерения можно выражать различными способами благодаря *синонимам* — словам и фразам, обозначающим примерно одно и то же.

Разработчикам приложений для чат-ботов приходится учитывать тот факт, что выражать намерения, которые приложение должно поддерживать, пользователи могут с помощью широкого диапазона фраз. Это значит, что приложение должно распознавать синонимичные фразы во вводимом пользователем тексте. На самом деле при создании чат-бота на основе таких популярных платформ, как Dialogflow компании Google, приходится указывать набор фраз для каждого возможного намерения — модель обучается на этих высказываниях.

Существует несколько подходов к распознаванию синонимов. Один из них — использовать заранее заданные списки синонимов, искать в них нужные слова и определять значение слов в соответствии с тем, в каком списке они оказались. Еще один способ — распознавать синонимы по степени семантического подобия. Эта задача подробно обсуждалась в главе 5. В следующих разделах будут рассмотрены оба этих подхода.

Распознавание синонимов с помощью заранее заданных списков

Как вы уже знаете, в большинстве случаев намерение во фразе лучше всего выражают переходный глагол и его прямое дополнение. Простейший способ выяснить, сколько намерений выражают две фразы — одно или два, — проверить, синонимичны ли в них переходные глаголы и прямые дополнения.

Например, эти три фразы выражают одно намерение — `orderPizza`:

```
I want a dish. I'd like to order a pizza. Give me a pie.
```

Обработка этих высказываний состоит из следующих шагов.

1. Разбор зависимостей для выделения из предложения переходного глагола и его прямого дополнения.
2. Поиск переходного глагола и его прямого дополнения в заранее заданных списках синонимов и замена (в случае успеха) их словами, понятными приложению.
3. Составление строки, идентифицирующей намерение.

Эти шаги кратко проиллюстрированы на рис. 8.4 на примере предложения *I want a dish*.

После разбора зависимостей (происходящего неявно во время применения конвейера к высказыванию) выделяем прямое дополнение и переходный глагол, а затем сверяемся с соответствующим списком синонимов. Если они в нем обнаружились, заменяем слово первым в списке синонимом, который приложение умеет распознавать.

Посмотрим, как выглядит реализация этой схемы на языке Python:

```
# Применяем конвейер к примеру предложения
doc = nlp(u'I want a dish.')
# Выделяем из дерева зависимостей переходный глагол
# и его прямое дополнение
❶ for token in doc:
    if token.dep_ == 'dobj':
      verb = token.head.text
      dobj = token.text
# Создаем список кортежей возможных синонимов глагола
❷ verbList = [('order','want','give','make'),('show','find')]
```

```
  # Находим кортеж с выделенным из примера текста переходным глаголом
❸ verbSyns = [item for item in verbList if verb in item]
  # Создаем список кортежей возможных синонимов прямого дополнения
❹ dobjList = [('pizza','pie','dish'),('cola','soda')]
  # Находим кортеж с выделенным из примера текста прямым дополнением
  dobjSyns = [item for item in dobjList if dobj in item]
  # Заменяем переходный глагол и прямое дополнение понятными приложению
  # синонимами и формируем строку-идентификатор намерения
❺ intent = verbSyns[0][0] + dobjSyns[0][0].capitalize()
  print(intent)
```

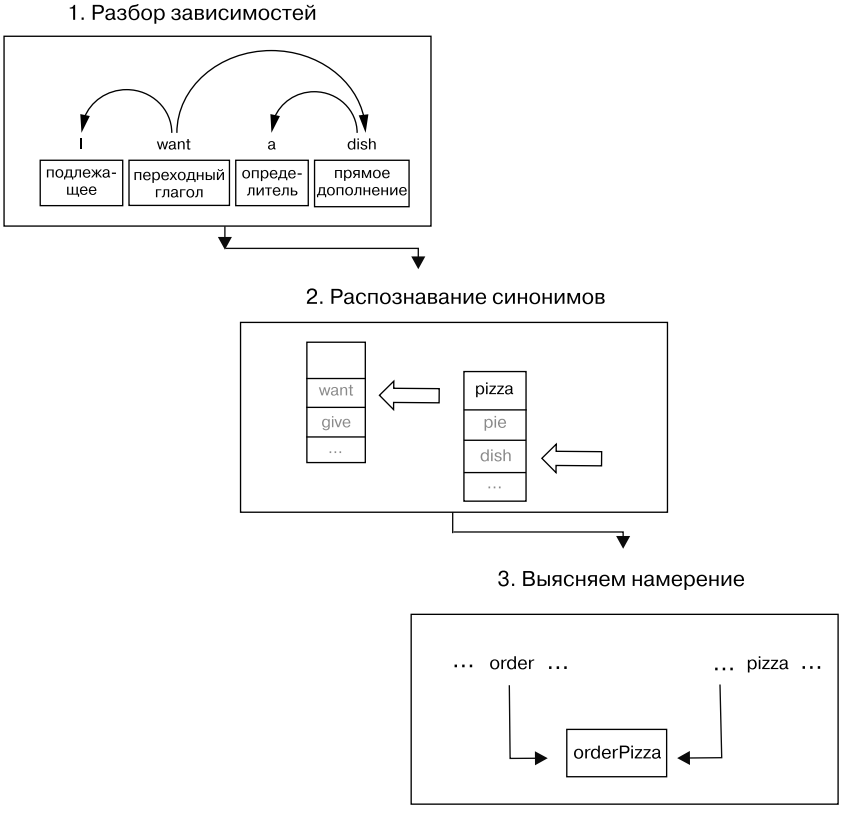


Рис. 8.4. Распознавание намерения с помощью списков синонимов

Начинаем с создания объекта `Doc` для нашего примера предложения. Проходим по дереву зависимостей, доступному через объект `Doc`, и выделяем переходный глагол вместе с его прямым дополнением ❶. Далее создаем список кортежей, содержащий все допустимые переходные

глаголы и их синонимы ❷. Первый элемент в каждом кортеже — понятный приложению переходный глагол, остальные — его синонимы.

Теперь, после описания допустимых переходных глаголов и их синонимов в виде списка кортежей, можно пройти в цикле по этому списку и найти кортеж, который содержит выделенный из примера предложения переходный глагол ❸.

Аналогичным образом создаем список кортежей для допустимых прямых дополнений и их синонимов, после чего находим в нем кортеж, содержащий выделенное из примера предложения прямое дополнение ❹.

Наконец, производим конкатенацию первых элементов найденных кортежей, составляя из них название намерения ❺. В результате команда `print` должна вывести следующую строку:

```
orderPizza
```

Учтите, что набор синонимов для заданного глагола зависит в основном от типа создаваемого приложения. Например, в контексте чат-бота, принимающего заказы на изготовление пиццы, глаголы *make* и *give* допускается считать синонимами, поскольку пользователи могут употреблять для заказа пиццы как фразу *Make me a pizza*, так и фразу *Give me a pizza*.

Попробуйте сами

На основе примера кода со с. 168 создайте новый сценарий, который сохранит функциональность кода, но в случае, если в соответствующих списках не удастся найти переходный глагол, его прямое дополнение или ни то и ни другое, сгенерирует *unrecognized* в качестве названия намерения. Для проверки работоспособности кода попробуйте различные примеры предложений, изменяя их так, чтобы увидеть в действии новую функциональность. Например, можете использовать такое предложение:

```
I want an apple.
```

Проверьте этот сценарий и на предложении, в котором есть глагол, отсутствующий в списке.

Кроме того, попробуйте решить с помощью списков синонимов проблему с конъюнктами, обсуждавшуюся в предыдущих разделах.

Распознавание неявных намерений с помощью семантического подобия

Логика настоящих приложений, применяемых на практике, как правило, сложнее, чем в описанных в этой главе примерах. Даже большой набор заранее заданных списков синонимов не всегда помогает. Дело в том, что пользователи могут выражать свои намерения множеством различных и не всегда явных способов.

При распознавании неявных намерений важную роль играет контекст. Если бот предназначен для решения конкретного типа задач, то он должен распознавать фразы, подразумевающие соответствующий запрос. Например, бот для заказа еды воспримет фразу *I feel like eating a pie* как намерение заказать пиццу.

Один из широко известных способов заставить пользователя выразить намерение яснее — задать уточняющий вопрос. Для выбора задаваемого вопроса необходимо вычислить семантическое подобие предыдущего высказывания пользователя.

Рисунок 8.5 иллюстрирует реализацию решения этой задачи.

Первый шаг — анализ дерева зависимостей входного высказывания и выделение прямого дополнения и его переходного глагола. Если прямое дополнение не удалось найти в заранее заданном списке синонимов (как обсуждалось в подразделе «Распознавание синонимов с помощью заранее заданных списков» на с. 168), можно попытаться определить, насколько оно близко к включенным в этот список словам. После этого на основе вычисления семантического подобия можно будет сгенерировать уточняющий вопрос пользователю.

Реализуем эту методику в коде, который разобьем на несколько частей. Как обычно, начинаем с применения конвейера обработки текста к примеру предложения:

```
doc = nlp(u'I feel like eating a pie.')
```

Далее извлекаем прямое дополнение:

```
for token in doc:
  if token.dep_ == 'dobj':
    dobj = token
```

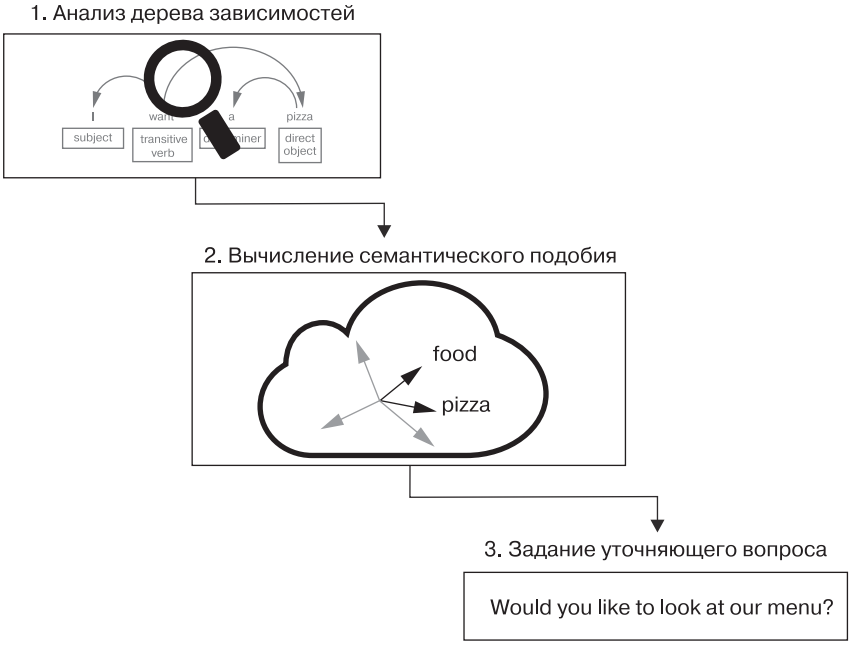


Рис. 8.5. Распознавание неявного намерения путем вычисления семантического подобия и задавания уточняющих вопросов

Создаем токен для слова *food*. Вычисляем семантическое подобие этого токена с токеном найденного прямого дополнения:

```
tokens = nlp(u'food')
```

Если степень семантического подобия превышает заранее заданное пороговое значение, приложение считает, что пользователь хочет разместить заказ, и задает ему уточняющий вопрос для подтверждения:

```
if dobj.similarity(tokens[0]) > 0.6:
  question = 'Would you like to look at our menu?'
```

Как вы помните из главы 5, библиотека spaCy вычисляет семантическое подобие токенов на основе векторов слов. Чем ближе два вектора в векторном пространстве, тем выше степень их подобия. В данном примере будем считать, что прямое дополнение имеет отношение к пищевому продукту, если значение степени подобия не ниже 0,6.

Попробуйте сами

Конечно, нам, как и нашему приложению, заранее не известно, какие фразы применит пользователь и насколько легко будет распознать его намерения. Именно поэтому на практике в приложениях сочетается несколько подходов для распознавания намерения. Для обработки более широкого спектра случаев попробуйте совместить подход на основе распознавания синонимов и подход, основанный на выявлении неявных намерений. Сначала ваш код должен попытаться определить намерение высказывания с помощью подхода, основанного на распознавании синонимов, а если не получилось — использовать и подход на основе семантического подобия. Если же результат все равно не достигнут, высказывание можно пометить как выражающее нераспознанное намерение.

Выделение намерения из последовательности предложений

Слова, выражающие намерение пользователя, могут встречаться в различных предложениях связного текста. Например:

```
I have finished my pizza. I want another one.
```

В подобном случае чат-бот должен быть готов к выделению намерений пользователя из текста в целом. В этом разделе рассмотрим методики решения данной задачи.

Обход структуры зависимостей связного текста

Начнем с синтаксического разбора зависимостей текста с выявлением пар «переходный глагол/прямое дополнение» в каждом из предложений, как показано на рис. 8.6.

Серая стрелка на схеме соответствует зависимости, которая нас интересует. Другими словами, было бы желательно заменить слово-заместитель *one* на обозначаемое им существительное *pizza*. Но средство

анализа зависимостей в spaCy не отображает эту связь, поскольку не может связать токены из разных предложений, и вы сами должны определить зависимости.

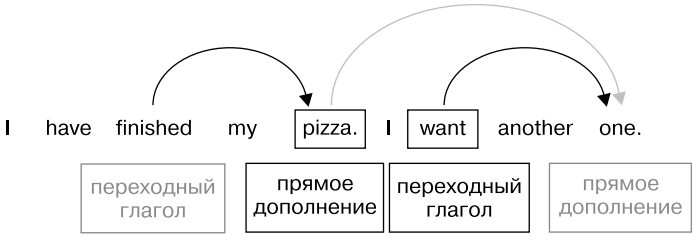


Рис. 8.6. Наглядное представление разбора синтаксических зависимостей текста в целом

Замена местоименных элементов их антецедентами

Антецедент (antecedent) — выражение (слово или простое предложение), обозначаемое *местоименным элементом* (pro-form), например местоимением или местоглаголием. При такой разновидности выделения намерений необходимо определить антецеденты и заменить ими соответствующие местоименные элементы. Для этого нужно проделать следующее.

1. Произвести синтаксический разбор зависимостей всего текста.
2. Разбить связный текст на предложения.
3. Найти антецедент (который далее укажем при формировании идентификатора намерения) для местоимения, служащего прямым дополнением переходного глагола.

На рис. 8.7 эти шаги показаны графически.

В spaCy можно реализовать первые два шага с помощью строки кода:

```
doc = nlp(u'I have finished my pizza. I want another one.')
```

Мы преобразовали объект, возвращаемый свойством `doc.sents`, в список, чтобы иметь возможность ссылаться на отдельные предложения по индексу. (Можно также пройти по непосредственно содержащейся в `doc.sents` последовательности предложений, используя цикл `for`.)

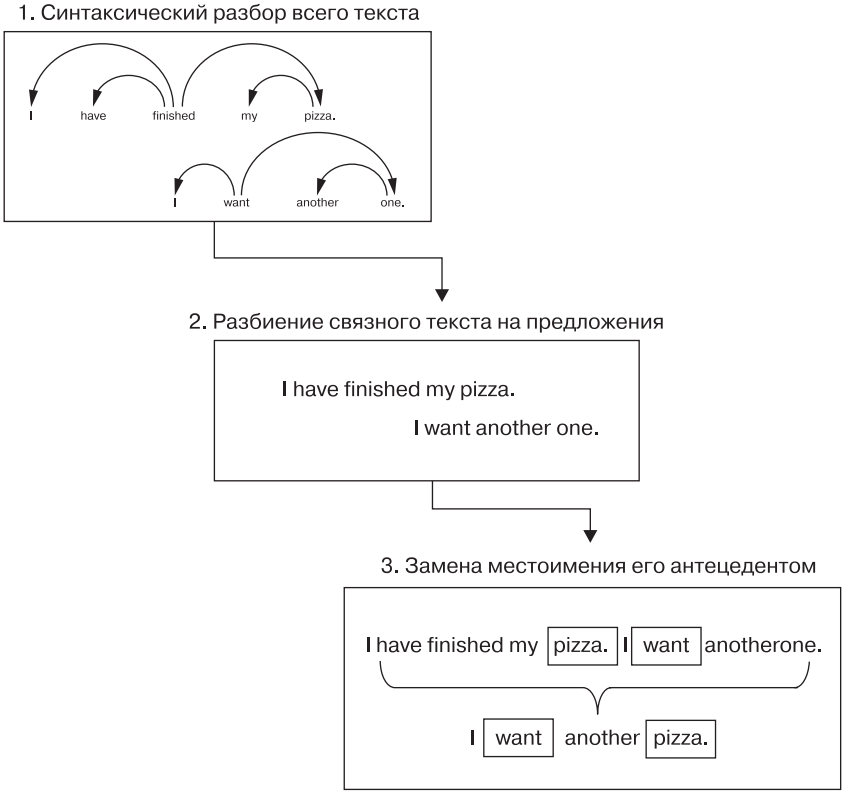


Рис. 8.7. Наглядная схема процесса выделения намерения из последовательности предложений

Далее объявляем два списка для допустимых переходных глаголов и прямых дополнений соответственно:

```
verbList = [('order','want','give','make'),('show','find')]
dobjList = [('pizza','pie','pizzaz'),('cola','soda')]
```

Эти списки содержат кортежи синонимов (см. подробности в подразделе «Распознавание синонимов с помощью заранее заданных списков» на с. 168).

Необходимо также задать список для допустимых слов-заместителей. Для этого нужно определить, какими местоименными элементами может быть прямое дополнение. Но сначала выясним, какие еще фразы

можно использовать вместо последнего предложения, а затем в каждой из них выделим прямое дополнение.

Возможные варианты:

```
I want another one. I want it again. I want the same. I want more.
```

Список слов-заместителей будет иметь следующий вид:

```
substitutes = ('one','it','same','more')
```

В отличие от списков переходных глаголов и прямых дополнений структура списка слов-заместителей проста, поскольку группировать их не нужно. Одно и то же слово-заместитель может относиться к любому из прямых дополнений.

Помимо списков, нам понадобится ассоциативный массив, в который будут заноситься части описания намерений по мере их выделения:

```
intent = {'verb': '', 'dobj': ''}
```

Теперь можно приступить к процессу распознавания намерений:

```
for sent in doc.sents:
  for token in sent:
    if token.dep_ == 'dobj':
      verbSyns = [item for item in verbList if token.head.text in item]
    ❶ dobjSyns = [item for item in dobjList if token.text in item]
      substitute =  [item for item in substitutes if token.text in item]
      if ❷(dobjSyns != [] or substitute != []) and ❸verbSyns != []:
          intent['verb'] = verbSyns[0][0]
    ❹ if dobjSyns != []:
          intent['dobj'] = dobjSyns[0][0]
```

Во внешнем цикле проходим по последовательности предложений из объекта `Doc`, а во внутреннем обходим найденные в предложении токены и проверяем, являются ли они прямыми дополнениями. Если да, выясняем, входит токен в список синонимов прямых дополнений или в список слов-заместителей ❶, а также проверяем, входит ли соответствующий переходный глагол в список синонимов переходных глаголов.

Нам необходимо выделить только те прямые дополнения, которые входят в список синонимов прямых дополнений или в список слов-

заместителей ❷. Например, не нужно выделять переходный глагол из следующей фразы (если, конечно, вы не продаете яблоки):

```
I want an apple.
```

Не интересует нас и переходный глагол, не входящий в список допустимых ❸, даже если его прямое дополнение этому условию удовлетворяет, как в следующей фразе:

```
I like it.
```

Именно поэтому, прежде чем выбирать переходный глагол, стоит проверить не только тот факт, что прямое дополнение включено в список синонимов прямых дополнений или в список слов-заместителей, но и то, входит ли переходный глагол в список синонимов переходных глаголов.

Наконец, для выбора прямого дополнения, описывающего намерение, необходимо убедиться, что оно входит в список синонимов прямых дополнений ❹. После этого формируем идентификатор намерения:

```
intentStr = intent['verb'] + intent['dobj'].capitalize()
```

Его также можно вывести в консоль и убедиться, что программа работает корректно:

```
print(intentStr)
```

Результат, указывающий, что пользователь намерен заказать пиццу, должен выглядеть следующим образом:

```
orderPizza
```

Попробуйте сами

В некоторых текстах антецедент и местоименный элемент могут быть разделены несколькими предложениями. Взгляните на следующую последовательность предложений:

```
I have finished my pizza. It was delicious.
I want another one.
```

Отредактируйте сценарий со с. 176 так, чтобы он корректно обрабатывал эту последовательность предложений и другие ей подобные.

Резюме

Распознавание намерений — сложная задача, для решения которой порой необходим целый комплекс подходов. В этой главе вы научились выделять наиболее важные части дерева зависимостей высказывания для распознавания намерений. Кроме того, вы узнали, как их анализировать с помощью заранее заданных списков, с помощью семантического подобия либо же с использованием обоих подходов. Помимо всего прочего, вы научились выделять намерение последовательности предложений путем замены местоименных элементов их антецедентами.

9 Сохранение данных, введенных пользователем, в базе данных

Многим коммерческим приложениям рано или поздно приходится отправлять свои данные в базу данных (БД). Например, чат-бот для заказа еды может сохранять форму заказа после того, как в ней окажется вся информация, извлеченная из разговора с пользователем. После внесения в базу данных заказ можно подвергать дальнейшей обработке, результатом которой станет отправка товара покупателю.

В этой главе обсуждаются вопросы преобразования информации, выделенной из введенного текста, в структурированный формат, подходящий для сохранения в *реляционной* (табличной) базе данных и дальнейших операций с ней. Вы увидите на примерах, как чат-бот может разрезать входной текст на части и сформировать на его основе подходящую для БД структуру.

Преобразование неструктурированных данных в структурированные

Структурированные данные упорядочиваются на основе заранее заданной схемы данных в репозитории четко заданного формата. Если вам уже приходилось работать с реляционными базами данных, то вы знаете, что все данные, вводимые в БД, необходимо сначала преобразовать в структурированный формат, который можно разместить в таблице или в наборе связанных таблиц.

Проблема в том, что данные на естественном языке, которые вводит пользователь, являются *неструктурированными*, то есть не имеют заранее заданной схемы организации. Типичные примеры неструктурированных данных — текст и мультимедийный контент из сообщений электронной почты, веб-страниц, деловых документов, видео, фотографии и т. д. И хотя в БД можно хранить неструктурированные данные, их необходимо предварительно обработать перед тем, как занести в базу: фотографии промаркировать для классификации в базе данных, а текстовым документам присвоить идентификаторы, чтобы база данных могла их различить.

Иногда неструктурированное текстовое содержимое приходится преобразовывать более радикально — например, выделять из него конкретные элементы информации, прежде чем формировать структуру определенного формата.

Чат-бот обычно производит синтаксический разбор высказываний покупателя для заполнения определенной формы, а какое-нибудь другое приложение может выделять из веб-страниц только определенные элементы, маркировать их, а затем преобразовывать в таблицу, как показано на рис. 9.1.

Компания	Сфера деятельности	Местоположение	Штат	Годовой доход
Apelsin	Software development	San Francisco	26	2 000 000

Рис. 9.1. Пример преобразования неструктурированного контента в структурированные данные

Утилиты наподобие spaCy позволяют раскрыть внутреннюю структуру текста путем лингвистической разметки всех его токенов. Подобная предварительная обработка позволяет выделять в тексте конкретные элементы, проверяя метки синтаксической зависимости.

На рис. 9.2 показано, каким образом чат-бот заказа еды может распознавать и выделять необходимые элементы из высказываний пользователя на основе меток синтаксической зависимости, присваиваемых библиотекой spaCy каждому из токенов в процессе применения к нему конвейера обработки текста.

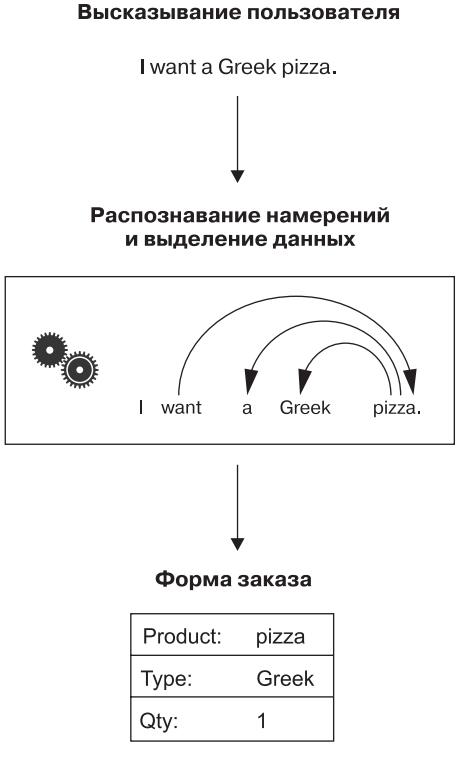


Рис. 9.2. Укрупненная схема преобразования неформатированного текста в табличные данные

После выделения этих элементов их можно структурировать нужным образом и вставить в базы данных в виде строки таблицы.

Выделение данных в формате обмена данными

Многие современные реляционные базы данных предоставляют нативную поддержку множества распространенных форматов обмена данными. MySQL поддерживает XML и JSON — два самых часто используемых формата обмена данными в Интернете.

От формата данных нередко зависит способ их сбора. Например, если используемая база данных поддерживает JSON, данные можно извлекать непосредственно в объект JSON и отправлять его в базу для дальнейшей обработки. Объект JSON представляет собой данные типа «ключ/значение», заключенные в фигурные скобки. Выглядит он так:

```
{"product": "pizza", "type": "Chicago", "qty": 1}
```

Значениями в JSON, помимо таких простейших значений, как строки и числа, могут служить и более сложные элементы данных — массивы и другие объекты JSON. Как это выглядит на практике, вы увидите в разделе «Создание чат-бота, использующего базу данных» на с. 185.

Фактически формат JSON существенно упрощает процесс формирования структуры данных для БД в сценарии Python. Прежде всего, код меньше привязан к конкретному типу базы данных, как было бы в случае использования менее распространенного формата. Во-вторых, элементы в объекте JSON могут идти в любом порядке, благодаря чему число ограничений, налагаемых на процесс определения и выделения необходимых элементов из входного текста, гораздо меньше.

На рис. 9.3 показана возможная схема взаимодействия чат-бота для заказа еды с базой данных на основе JSON.

На шаге 1 пользователь отправляет чат-боту запрос на греческую пиццу. На шаге 2 чат-бот обрабатывает полученное высказывание с помощью spaCy, генерируя объект JSON, содержащий необходимую для принятия заказа информацию. На шаге 3 соответствующий заказу объект JSON отправляется в базу данных, которая сохраняет форму заказа и генерирует ответ чат-боту. На шаге 4 чат-бот сообщает пользователю, был ли размещен заказ.

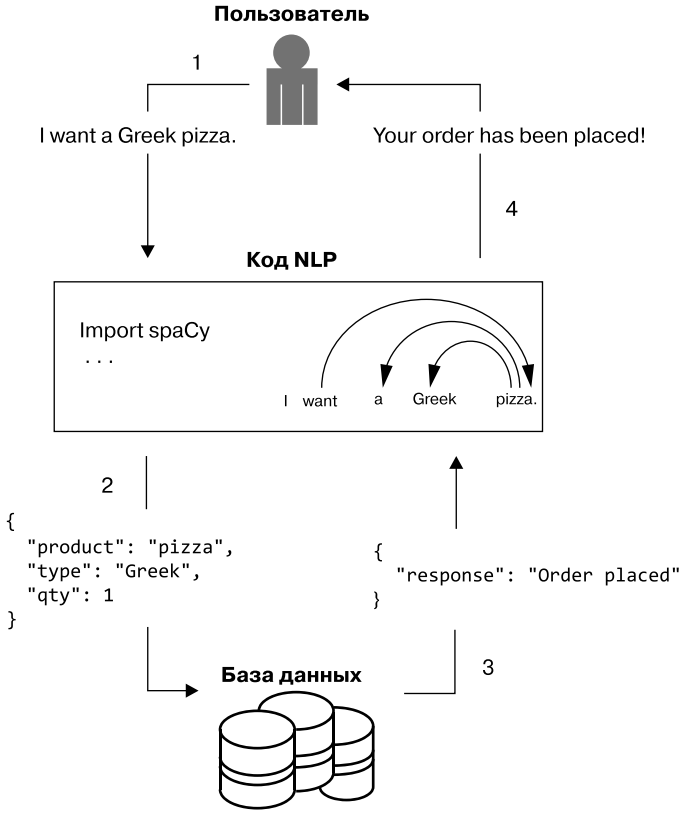


Рис. 9.3. Технологический процесс работы приложения чат-бота для заказа еды

Перенос логики приложения в базу данных

Обратите внимание, что база данных в приложении чат-бота на рис. 9.3 не только сохраняет переданный объект JSON, но и сообщает приложению, была ли операция сохранения заказа успешной. Это происходит потому, что база данных отвечает за часть логики приложения.

Использующие БД приложения достаточно часто выносят связанную с обработкой данных логику в базу данных. Такой подход позволяет избежать лишнего перемещения данных между слоем логики приложения и базой данных, исключая тем самым избыточность данных, повышая эффективность их обработки и обеспечивая их безопасность.

На рис. 9.4 показана более подробная схема относящейся к базе данных части приложения чат-бота, приведенного на рис. 9.3.

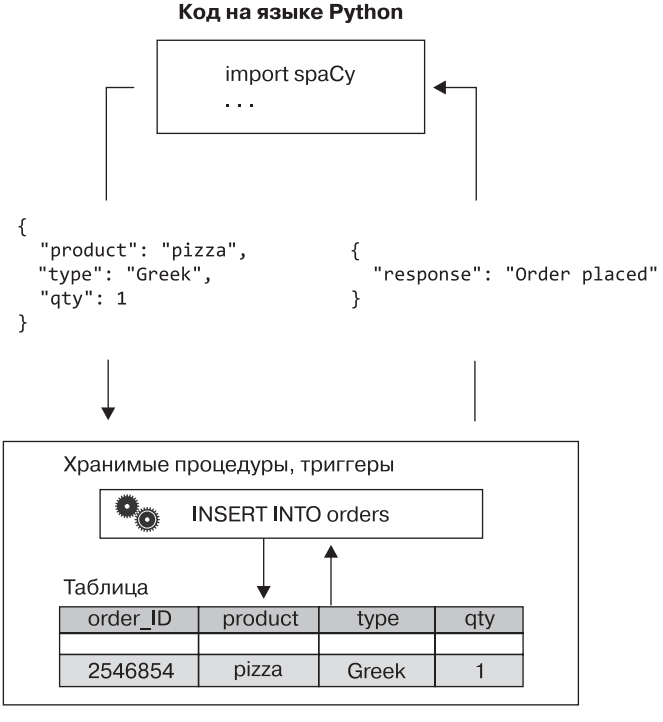


Рис. 9.4. Более подробная схема работы базы данных, используемой в приложении чат-бота с рис. 9.3

В этом приложении база данных преобразует входной объект JSON в реляционные данные и сохраняет их в реляционной таблице таким образом, чтобы гарантировать полноту и корректность вставляемых данных. При отсутствии значения в одном или нескольких полях заказчик получит сообщение о том, какую информацию необходимо предоставить.

Проверить значения всех полей до перемещения входных данных в таблицу можно с помощью хранимых процедур, предложений `ON ERROR` операторов SQL или триггеров, привязанных к таблице, куда все эти данные отправляются. Более полное обсуждение SQL выходит за рамки данной книги, но в подразделе «Подготовка среды базы

данных» на с. 189 вы найдете пример создания инфраструктуры базы данных для приложения с помощью SQL, а также взаимодействия с этой инфраструктурой на языке Python.

ПРИМЕЧАНИЕ

Если база данных, которую вы используете, не поддерживает преобразование JSON в реляционные данные, вам придется самостоятельно реализовать на Python логику проверки полноты данных. Впрочем, этот вопрос выходит за рамки данной книги.

Создание чат-бота, использующего базу данных

Реализуем знания о том, как создать использующий базу данных чат-бот, на простом примере — для приведенного на рис. 9.3 приложения. Приложение должно обработать высказывание пользователя, а также выделить необходимую для формы заказа информацию: название товара, его тип, количество и т. п. Далее эта информация упаковывается в объект JSON, который отправляется в базу данных. База данных должна преобразовать объект в реляционные данные и сообщить приложению, являются ли данные полными.

Сбор данных и формирование объекта JSON

Начнем с разработки слоя логики приложения. Для формирования объекта JSON, который можно будет отправить в любую базу данных, воспользуемся Python. Один из вариантов реализации приведен в следующем коде:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I want a Greek pizza.')
❶ orderdict ={}
❷ for token in doc:
  ❸ if token.dep_ == 'dobj':
      dobj = token
    ❹ orderdict.update(product = dobj.lemma_)
    ❺ for child in dobj.lefts:
```

```
❻ if child.dep_ == 'amod' or child.dep_ == 'compound':
      orderdict.update(ptype = child.text )
❼ elif chi/ld.dep_ == 'det':
      orderdict.update(qty = 1 )
❽ elif child.dep_ == 'nummod':
      orderdict.update(qty = child.text)
  break
```

Сначала объявляем ассоциативный массив `orderdict` — контейнер для создаваемого объекта JSON ❶. Позднее его можно будет легко преобразовать в строку JSON.

Проходим в цикле по токенам высказывания ❷ в поисках прямого дополнения ❸. Пользователь либо хочет пиццу, либо просит кого-то сделать пиццу. В любом случае *pizza* будет играть в высказывании роль прямого дополнения, так что найти нужно именно его. Конечно, в приложении, пригодном для реальной эксплуатации, проверок будет больше.

Найдя прямое дополнение, задаем в ассоциативном массиве `orderdict` пару «ключ/значение» с `product` в качестве ключа и леммой прямого дополнения в качестве значения ❹. Сокращаем число возможных словоформ названия товара до одной, базовой формы, применив лемматизацию (которая в большинстве случаев сводится к преобразованию из множественного числа в единственное).

Далее проходим в цикле по синтаксическим левосторонним дочерним элементам прямого дополнения ❺, рассчитывая найти в них информацию о типе заказываемого товара. В плане меток синтаксических зависимостей `type` товара может представлять собой либо часть составного существительного, либо прилагательное-модификатор (`amod`) ❻. Например, spaCy считает слово *Greek* во фразе *a Greek pizza* прилагательным-модификатором, а слово *Chicago* во фразе *a Chicago pizza* — частью составного существительного.

Затем ищем модификатор среди дочерних элементов или определитель среди частей составного существительного. Наличие определителя *a* означает, что покупатель заказывает одну единицу товара ❼. И, напротив, слово с меткой зависимости `nummod` указывает на конкретное число единиц ❽.

Выводим в консоль ассоциативный массив `orderdict`:

```
print(orderdict)
```

Результат ее выполнения должен выглядеть следующим образом:

```
{'product': 'pizza', 'ptype': 'Greek', 'qty': 1}
```

Мы получили строку JSON, которую теперь можно отправить в базу данных для дальнейшей обработки.

Преобразование числительных в числа

Прежде чем перейти к коду отправки строки JSON в базу данных, обсудим, что произойдет, если пользователь явным образом укажет количество товара, как в следующем высказывании:

```
I want two Greek pizzas.
```

При обработке этого высказывания с помощью предыдущего сценария получится такой результат:

```
{'product': 'pizza', 'ptype': 'Greek', 'qty': two}
```

В первом примере предложения значение ключа `qty` представляет собой число. Во втором — число, описанное словами. На данном этапе подобное различие не кажется существенным, но проблема в том, что при создании каждого из столбцов реляционной таблицы нужно указать тип данных. Попытка вставить в столбец данные другого типа приведет к ошибке.

Необходимо подготовить чат-бот к тому, что пользователи могут вводить количество товара самым разным способом. Для решения проблемы преобразуем строки, содержащие числительные, в соответствующие целочисленные значения.

Для этого опишем список чисел, выраженных словами, отсортируем их в порядке возрастания, после чего пройдем по списку в цикле в поисках нужного числа.

Описываем список для числительных от нуля *(zero)* до двадцати *(twenty)*, то есть предполагаем, что покупатели не станут заказывать более 20 единиц одинакового товара в одной покупке.

Нам нужно реализовать этот процесс преобразования в виде функции, принимающей в качестве параметра числительное или число (в последнем случае никакого преобразования не требуется) и возвращающей число. Затем потребуется соответствующим образом модифицировать код сценария из предыдущего раздела. Реализация этой функции может выглядеть, например, так:

```
❶ def word2int(numword):
    num = 0
  ❷ try:
    ❸ num = int(numword)
    ❹ return num
    except ValueError:
     ❺ pass
  ❻ words = ["zero", "one", "two", "three", "four", "five", "six", "seven",
      "eight","nine", "ten", "eleven", "twelve", "thirteen", "fourteen",
      "fifteen", "sixteen", "seventeen", "eighteen", "nineteen", "twenty"]
  ❼ for idx, word in enumerate(words):
        ❽ if word in numword:
              num = idx
  ❾ return num
```

Функция `word2int()` принимает один параметр: числительное (для преобразования в соответствующее число) либо готовое число (в этом случае ничего преобразовывать не нужно) ❶. Функция должна корректно обрабатывать оба случая, поскольку заранее не известно, какой из них может встретиться в высказывании пользователя.

Для обработки случаев, когда преобразование не требуется, используем блок `try/except` ❷. Проверяем, являются ли входные данные целым числом ❸, и если да, то возвращаем его в неизменном виде ❹. В противном случае игнорируем ошибку, вызванную попыткой работать с нечисловым значением как с числом, и переходим к преобразованию нечислового значения в число ❺.

Объявляем список числительных в порядке возрастания, начиная с нуля (`zero`) ❻. Проходим в цикле по этому списку с помощью метода `enumerate()` ❼ и ищем в нем слово, полученное функцией на входе ❽. Найдя, возвращаем значение счетчика цикла (индекс слова в списке) в качестве цифрового представления входного числительного ❾.

Добавляем описание функции `word2int()` в предыдущий сценарий. Переходим в конец этого сценария и находим там следующие строки кода:

```
elif child.dep_ == 'nummod':
  orderdict.update(qty = child.text)
```

Модифицируем их так, чтобы воспользоваться описанной в этом разделе функцией `word2int()`:

```
elif child.dep_ == 'nummod':
  orderdict.update(qty = word2int(child.text))
```

Проверяем, как наш сценарий справится с вышеупомянутым предложением:

```
I want two Greek pizzas.
```

На этот раз получаем следующий результат:

```
{'product': 'pizza', 'ptype': 'Greek', 'qty': 2}
```

Значение поля `'qty'` теперь представляет собой число, и данные находятся в единообразном формате, подходящем для отправки в базу данных.

Подготовка среды базы данных

Для подготовки среды базы данных необходимо инсталлировать базу данных или получить к ней доступ; создать в базе данных необходимые компоненты (например, схему базы данных, таблицы и т. д.); инсталлировать модуль Python, обеспечивающий взаимодействие с базой данных.

В этом разделе будем использовать MySQL, хотя подошла бы любая база данных, способная получать и обрабатывать данные в формате JSON, например Oracle. База данных MySQL давно известна поддержкой наиболее популярных форматов обмена данными — XML и JSON. Кроме того, MySQL — самая популярная в мире база данных с открытым исходным кодом, доступная для большинства современных операционных систем, включая Linux, Windows, Unix и macOS. Для различных бизнес-требований у MySQL существуют бесплатная и коммерческая версии.

Для этой главы вам подойдет MySQL Community Edition — бесплатная версия с лицензией GPL. Узнать больше о MySQL Community Edition можно на ее официальном веб-сайте по адресу https://www.mysql.com/products/community/.

Сначала необходимо инсталлировать MySQL в своей системе. На момент написания данной книги текущей версией являлась MySQL 8.0. Загляните в главу Installing and Upgrading MySQL («Инсталляция и обновление MySQL») справочного руководства MySQL 8.0 по адресу https://dev.mysql.com/doc/refman/8.0/en/installing.html либо в аналогичную главу руководств для будущих версий MySQL. Там вы найдете подробные инструкции по инсталляции для вашей операционной системы.

После инсталляции можете запустить сервер MySQL с помощью команды, указанной в руководстве для соответствующей операционной системы. Прежде чем начать работу с базой данных, необходимо получить генерируемый во время установки пароль суперпользователя MySQL (`'root'@'localhost'`). Найти его можно в журнале ошибок инсталляции.

При наличии пароля суперпользователя к базе данных можно подключить MySQL из системного терминала с помощью следующей команды:

```
$ mysql -uroot -p
Enter password: ******
mysql>
```

Если вы предпочитаете графический интерфейс, можете воспользоваться MySQL Workbench (https://www.mysql.com/products/workbench/) — унифицированной графической утилитой, предназначенной для моделирования баз данных MySQL и управления ими.

После подключения к серверу прежде всего задайте новый пароль для суперпользователя и замените им случайный, сгенерированный во время инсталляции. Для этого воспользуйтесь следующей командой:

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'Your-pswd';
```

Теперь можем приступать к разработке серверной инфраструктуры, необходимой для нашего приложения. Начнем с создания базы

данных, которая послужит контейнером для прочих необходимых приложению объектов. Для этого в командной строке вводим следующую команду `mysql>`:

```
mysql> CREATE DATABASE mybot;
Query OK, 1 row affected (0.03 sec)
```

Выбираем только что созданную базу данных для использования:

```
mysql> USE mybot;
Database changed
```

Переходим к созданию структуры нашей базы данных. Для этого примера нам достаточно одной таблицы, которую можно создать с помощью следующей команды:

```
CREATE TABLE orders (
  id INT NOT NULL AUTO_INCREMENT,
  product VARCHAR(30),
  ptype VARCHAR(30),
  qty INT,
  PRIMARY KEY (id)
);
```

Для взаимодействия кода на языке Python с данной инфраструктурой базы данных необходимо инсталлировать драйвер MySQL Connector/Python. В любой операционной системе это можно сделать с помощью следующей команды `pip`:

```
pip install mysql-connector-python
```

Дальнейшие подробности относительно установки драйвера можно найти в документации по адресу https://dev.mysql.com/doc/connector-python/en/.

С помощью следующего простого сценария проверьте, как установился Connector/Python:

```
import mysql.connector
cnx = mysql.connector.connect(user='root', password='Your_pswd',
                              host='127.0.0.1',
                              database='mybot')
cnx.close()
```

Если установка прошла успешно, вы не должны увидеть никаких сообщений об ошибках.

Отправка данных в БД

Вернемся к сценарию со с. 191. Следующий код выполняет соединение с базой данных и вносит данные о заказе в таблицу `orders`. Добавляем этот код в конец вышеупомянутого сценария:

```
  import json
❶ json_str = json.dumps(orderdict)
  import mysql.connector
  from mysql.connector import errorcode
  try:
  ❷ cnx = mysql.connector.connect(user='root', password='Your_pswd',
                                  host='127.0.0.1',
                                  database='mybot')
  ❸ query = ("""INSERT INTO orders (product, ptype, qty)
      SELECT product, ptype, qty FROM
          JSON_TABLE(
        ❹ %s,
            "$" COLUMNS(
              qty    INT PATH '$.qty',
              product   VARCHAR(30) PATH "$.product",
              ptype     VARCHAR(30) PATH "$.ptype"
            )
          ) AS jt1""")
  ❺ cursor = cnx.cursor()
  ❻ cursor.execute(query, ❼(json_str,))
  ❽ cnx.commit()
❾ except mysql.connector.Error as err:
    print("Error-Code:", err.errno)
    print("Error-Message: {}".format(err.msg))
  finally:
    cursor.close()
    cnx.close()
```

Сначала преобразуем ассоциативный массив `orderdict` в строку JSON ❶. Затем подключимся к базе данных ❷ и опишем оператор вставки SQL для дальнейшей его передачи базе данных и выполнения ❸. Обратите внимание на «заполнитель» — *переменную связывания* (bind variable) — в операторе ❹. Благодаря подобным «заполнителям» можно писать операторы SQL, принимающие данные во время выполнения.

Прежде чем выполнять оператор, создадим объект курсора ❺ для работы с объектами базы данных, к которой подключились. Теперь можно выполнить оператор `INSERT` ❻, связав с «заполнителем» в операторе

полученную в начале этого фрагмента кода строку JSON ❼. Обратите внимание на функцию `JSON_TABLE`, превращающую данные, переданные в формате JSON, в табличные, которые подходят для вставки в реляционную таблицу.

После выполнения оператора `INSERT` необходимо явным образом зафиксировать внесенные изменения с помощью метода `commit()` ❽. В противном случае при закрытии соединения произойдет откат операции вставки (либо явным образом при вызове `cnx.close()`, либо по завершении выполнения сценария).

В случае возникновения ошибки на стороне базы данных начнет выполняться блок `except` ❾. В следующем разделе вы узнаете, как воспользоваться этой функциональностью, если переданная строка JSON не содержит все нужные поля.

Теперь выполните сценарий. Если сообщения об ошибках возвращены не были, вернитесь в командную строку `mysql`, в которой работали в предыдущем разделе, и введите оператор `SELECT`:

```
mysql> SELECT * FROM orders;
ID   PRODUCT     PTYPE    QTY
---- ----------- -------- ---
1    pizza       Greek    2
```

Если вы получите такие же результаты, значит, сценарий Python работает корректно.

Что делать, если запрос пользователя содержит недостаточно информации

Иногда запрос пользователя содержит слишком мало информации, чтобы заполнить все поля в форме заказа. В качестве примера рассмотрим следующее высказывание:

```
I want two pizzas.
```

Таблица 9.1 демонстрирует форму заказа, которую приложение сгенерирует на основе этого высказывания.

Таблица 9.1. Форма заказа, в которой отсутствует часть информации

Product	ptype	quantity
Pizza		2

Значение для поля `ptype` отсутствует, поскольку пользователь не указал желаемый тип пиццы. Для решения этой проблемы расширим оператор `INSERT` из предыдущего сценария следующим образом:

```
query = ("""INSERT INTO orders (product, ptype, qty)
SELECT product, ptype, qty FROM
    JSON_TABLE(
      %s,
      "$"
      COLUMNS(
           qty    INT PATH '$.qty' ❶ERROR ON EMPTY,
           product   VARCHAR(30) PATH "$.product" ❶ERROR ON EMPTY,
           ptype     VARCHAR(30) PATH "$.ptype" ❶ERROR ON EMPTY
      )
    ) AS jt1""");
```

В каждый столбец таблицы `JSON_TABLE` была добавлена опция `ERROR ON EMPTY` ❶, которая позволяет обрабатывать ошибки, вызванные попыткой вставить строку JSON, содержащую не все необходимые поля.

Если теперь выполнить сценарий с примером предложения *I want two pizzas.*, появится следующее сообщение:

```
Error-Code: 3665
Error-Message: Missing value for JSON_TABLE column 'ptype'
```

Можно усовершенствовать сценарий так, чтобы в подобных случаях чат-бот просил пользователя уточнить заказ с помощью следующего вопроса:

```
What type of pizza do you want?
```

Ответ может выглядеть, например, так:

```
I want Greek ones.
```

Структура предложения, содержащего ответ, аналогична структуре исходного предложения. Следовательно, для его анализа можно использовать тот же код, что и для анализа исходного предложения.

Конечно, такой подход основывается на определенных допущениях касательно ответа пользователя, но настоящая реализация должна начинаться именно с этого подхода, а затем по необходимости следует переходить к рассмотрению других возможных структур ответа. Например, пользователь может ответить одним словом *Greek*. В таком случае будет достаточно просто проверить, входит ли оно в список типов пиццы.

Попробуйте сами

Приведенное сообщение об ошибке указывает на отсутствие значения конкретного поля. Но нам все равно необходимо выделить название этого поля из сообщения, чтобы попросить пользователя уточнить конкретную часть заказа. Один из способов сделать это — найти в сообщении предложное дополнение. Например, в сообщении `ErrorMessage: Missing value for JSON_TABLE column 'ptype'` в роли предложного дополнения выступает `'ptype'`.

Резюме

В этой главе вы научились разрезать необработанный текст на кусочки для вставки его в реляционную базу данных. Вы использовали формат JSON для взаимодействия с БД, которая умееет обрабатывать данные такого формата и преобразовывать их в реляционные. Помимо того, вы узнали, как можно реализовать часть логики приложения внутри базы данных с помощью чистого SQL, перенося, таким образом, обработку данных ближе к месту их хранения. Для реализации более сложных вариантов обработки можно использовать триггеры и хранимые процедуры (подробности вы найдете в документации к используемой базе данных).

10 Обучение моделей

В главе 1 вы узнали, что библиотека spaCy включает в себя статистические нейросетевые модели, предобученные для распознавания именованных сущностей, частеречной разметки, синтаксического разбора зависимостей и предсказания семантического подобия. Но вас никто не ограничивает одними лишь предобученными готовыми моделями. При желании можно обучить модель на собственных примерах, настроив компоненты ее конвейера под потребности своего приложения.

В этой главе обсуждаются вопросы обучения средств распознавания именованных сущностей и синтаксического разбора зависимостей — компонентов конвейера. Их чаще всего приходится настраивать так, чтобы модель отвечала требованиям конкретного сценария использования. Дело в том, что определенная предметная область обычно требует конкретного набора сущностей, а иногда и четкого способа разбора зависимостей. Вы научитесь обучать как уже существующую модель на новых примерах данных, так и совершенно новую. Кроме того, вы узнаете, как сохранить настроенный компонент конвейера на диск для последующей загрузки его в другом сценарии или модели.

Обучение компонента конвейера модели

Для настройки под цели и задачи конкретного приложения обучать модель с нуля приходится редко. Обычно можно воспользоваться уже существующей моделью и обновить только определенный компонент

конвейера. Этот процесс, как правило, состоит из двух шагов: подготовки *обучающих примеров данных* (набора предложений с лингвистической разметкой, подходящих для обучения модели) и передачи их нужному компоненту конвейера, как показано на рис. 10.1.

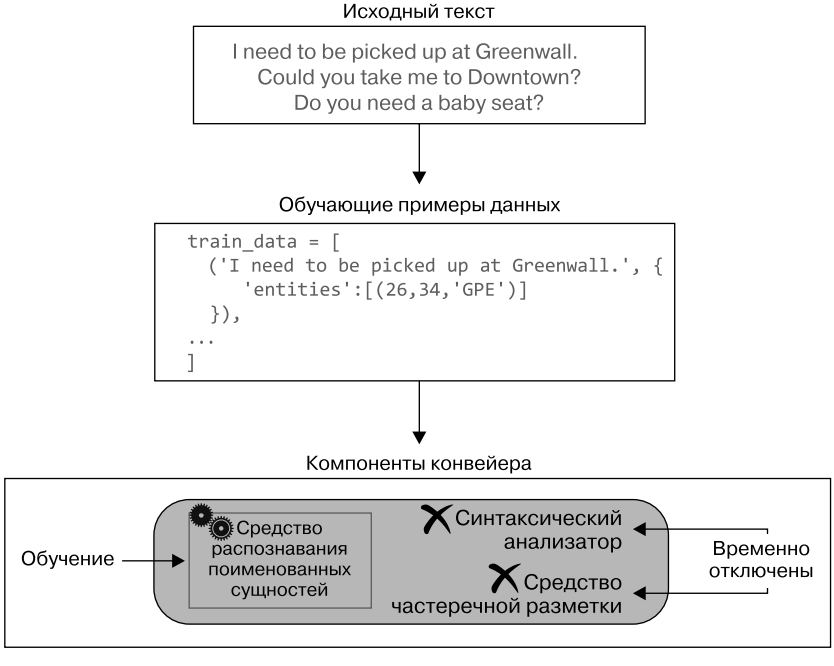


Рис. 10.1. Процесс обучения компонента конвейера

Для подготовки обучающих примеров данных необходимо преобразовать исходные текстовые данные в обучающий пример, в котором будет само предложение и разметка всех токенов. Во время обучения библиотека spaCy корректирует весовые коэффициенты модели на основе обучающих примеров данных, стремясь минимизировать *погрешность*, еще называемую *потерями/функцией потерь* (loss), предсказаний, выполняемых моделью. Проще говоря, алгоритм вычисляет соотношение токена и метки, определяя вероятность того, что данному токену следует присвоить именно эту метку.

На практике для полноценного обучения компонента модели могут понадобиться сотни или даже тысячи обучающих примеров данных.

Прежде чем обучать определенный компонент, необходимо временно отключить все остальные компоненты конвейера модели, чтобы защитить их от ненужных изменений.

Обучение средства распознавания именованных сущностей

Представьте, что вы разрабатываете чат-бот для такси, который, само собой, должен правильно распознавать все географические названия в городе и окрестностях. Для этого необходимо обновить систему распознавания именованных сущностей модели на основе своих примеров данных. Например, слово Solnce, обозначающее название микрорайона в городе, система должна определять как географическую сущность. В следующих разделах описан способ решения этой задачи.

Определяем, нужно ли обучать средство распознавания именованных сущностей

Сначала посмотрим, как распознает интересующие нас сущности уже имеющееся средство распознавания именованных сущностей из модели английского языка, которая используется по умолчанию (модель `en_core_web_sm`). Может, его и не придется обновлять. Используем типичные фразы для заказа такси, например:

```
Could you pick me up at Solnce?
```

Чтобы узнать, как средство распознавания классифицирует *Solnce* в этом предложении, выведем именованные сущности предложения с помощью следующего сценария:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'Could you pick me up at Solnce?')
  for ent in doc.ents:
    print(ent.text, ent.label_)
```

В этом примере единственной именованной сущностью является *Solnce*, так что сценарий выдаст следующую строку:

```
Solnce LOC
```

Учтите, что выводимый для сущности результат может различаться в зависимости от используемой модели и предложения. Получить описание метки `LOC` сущности можно с помощью функции `spacy.explain()`:

```
>>> print(spacy.explain('LOC'))
'Non-GPE locations, mountain ranges, bodies of water'
```

Получается, что средство распознавания именованных сущностей классифицировало *Solnce* как местоположение, не являющееся сущностью типа `GPE`. Значит, чтобы *Solnce* распознавалось как сущность типа `GPE`, средство распознавания нужно обновить. О том, как это сделать, поговорим в следующих разделах.

ПРИМЕЧАНИЕ

Для простоты в этом примере используется только одна именованная сущность. Но вы можете создать дополнительные названия микрорайонов для обучения средства распознавания именованных сущностей.

Вместо того чтобы обновлять уже существующее средство распознавания, можно заменить его пользовательской версией. Впрочем, в этом случае понадобится гораздо больше обучающих примеров данных для сохранения не связанной с сущностями типа `GPE`, но все же необходимой функциональности.

Создание обучающих примеров данных

Когда вы убедились в том, что средство распознавания именованных сущностей необходимо обучить для выполнения ваших задач, можно переходить к следующему шагу — созданию набора обучающих примеров данных. Для этого вам понадобится соответствующий текст.

Оптимальный источник данных для создания подобного обучающего набора — реальные данные, введенные когда-либо пользователями. Выберите высказывания, в которых есть нужные вам именованные сущности. Обычно вводимые пользователем данные сохраняются в файл в виде простого текста. Например, журнал пользовательского ввода для приложения такси может содержать следующие высказывания:

```
Could you send a taxi to Solnce?
Is there a flat rate to the airport from Solnce?
How long is the wait for a taxi right now?
```

Чтобы на их основе создать обучающие примеры данных, необходимо преобразовать высказывания в список кортежей, где каждому обучающему примеру данных будет соответствовать отдельный кортеж, как показано ниже:

```
train_exams = [
  ❶ ('Could you send a taxi to Solnce?', {
      ❷ 'entities': [(25, 32, 'GPE')]
    }),
    ('Is there a flat rate to the airport from Solnce?', {
        'entities': [(41, 48, 'GPE')]
    }),
    ('How long is the wait for a taxi right now?', {
        'entities': []
    })
]
```

Каждый кортеж состоит из двух значений: строки с высказыванием ❶ и ассоциативного массива для разметки найденных в этом высказывании сущностей. Разметка сущности включает ее начальную и конечную позиции, которые отсчитываются по составляющим высказывание символам, и присвоенную этой сущности метку ❷.

Автоматизация процесса создания примеров данных

Вы уже могли убедиться в том, что создание обучающего набора данных вручную требует немало времени. К тому же оно чревато ошибками, особенно если требуется обработать сотни или тысячи высказываний. Автоматизировать этот утомительный процесс можно с помощью следующего сценария, который быстро создает набор обучающих примеров данных на основе заданного текста:

```
  import spacy
  nlp = spacy.load('en')
❶ doc = nlp(u'Could you send a taxi to Solnce? I need to get to Google.
  Could you send a taxi an hour later?')
❷ #f = open("test.txt","rb")
  #contents =f.read()
  #doc = nlp(contents.decode('utf8'))
❸ train_exams = []
```

```
❹ districts = ['Solnce', 'Greenwal', 'Downtown']
  for sent in doc.sents:
    entities = []
    for token in sent:
      if token.ent_type != 0:
        ❺ start = token.idx - sent.start_char
          if token.text in districts:
            entity = (start, start + len(token), 'GPE')
          else:
            entity = (start, start + len(token), token.ent_type_)
          entities.append(entity)
    tpl = (sent.text, {'entities': entities})
  ❻ train_exams.append(tpl)
```

Для большей удобочитаемости мы «зашили» несколько высказываний прямо в сценарий ❶. В комментированных строках кода показано, как можно было извлечь высказывания из файла, не прибегая к такой манипуляции ❷.

После получения высказываний — из файла или непосредственно из сценария — можно приступать к генерации на их основе списка обучающих примеров данных. Начнем с создания пустого списка ❸, а затем объявим список, содержащий названия сущностей, которые наша модель должна распознавать не так, как она это делает сейчас ❹. (В данном примере это список микрорайонов.)

Учтите, что во вводимом пользователем тексте уже могут быть сущности, которые модель распознает правильно (скажем, Google или London), поэтому менять поведение средства распознавания при их классификации не требуется. Наша цель — создание обучающих примеров данных для всех встречающихся в высказываниях сущностей и обработка всех, а не только новых сущностей. Обучающий набор данных для настоящей реализации должен включать многочисленные примеры данных для сущностей различных типов. В зависимости от требований приложения обучающий набор данных может состоять из нескольких сотен примеров.

Проходим в цикле по заданным высказываниям, создавая на каждом проходе новый пустой список сущностей. Далее проходим в цикле по токенам в высказывании в поисках сущностей и заполняем этот список. Для каждой найденной сущности определяем индекс первого

ее символа в высказывании ❺, а затем вычисляем конечный индекс путем прибавления к начальному `len(token)`.

Кроме того, необходимо проверить, входит ли найденная сущность в список сущностей, которым необходимо присвоить метки. Если да, присваиваем ей метку `GPE`, или средство распознавания использует текущую метку из разметки сущности. После этого объявляем кортеж для обучающего примера и добавляем его в конец обучающего набора данных ❻.

Данный сценарий отправляет генерируемые примеры данных в список `train_exams`, который после выполнения сценария должен выглядеть следующим образом:

```
>>> train_exams
[
❶ ('Could you send a taxi to Solnce?', {'entities': [(25, 31, 'GPE')]}),
❷ ('I need to get to Google.', {'entities': [(17, 23, 'ORG')]}),
❸ ('Could you send a taxi an hour later?', {'entities': []})
]
```

Для простоты этот обучающий набор данных состоит всего из нескольких обучающих примеров. Обратите внимание, что только первый из них содержит сущность, входящую в список сущностей, с которыми нам нужно «познакомить» средство распознавания (в нашем примере — список микрорайонов) ❶. Впрочем, это не означает, что второй и третий обучающие примеры данных бесполезны. Второй обучающий пример ❷ вносит еще один тип сущности, чтобы средство распознавания не «забыло» то, что знало ранее.

Третий обучающий пример данных не содержит никаких сущностей ❸. Для улучшения результатов обучения в обучающий набор необходимо включать не только примеры прочих типов сущностей, но и примеры, не содержащие никаких сущностей. В подразделе «Процесс обучения» на с. 203 обо всем этом рассказано более подробно.

Отключение лишних компонентов конвейера

Документация spaCy рекомендует перед запуском процесса обучения конкретного компонента конвейера отключать все прочие его компоненты, чтобы модифицировать только тот компонент, который

необходимо обновить. Следующий код отключает все компоненты конвейера, за исключением средства распознавания именованных сущностей. Этот код необходимо добавить в конец приведенного в предыдущем разделе сценария либо выполнить его в том же сеансе Python, что и упомянутый сценарий (оставшийся последний фрагмент кода будет добавлен в следующем разделе, когда начнется обсуждение процесса обучения):

```
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != 'ner']
nlp.disable_pipes(*other_pipes)
```

Теперь можно начинать обучение средства распознавания именованных сущностей нахождению новых сущностей, описанных в обучающих примерах данных.

Процесс обучения

В процессе обучения придется перетасовать обучающие примеры данных и пройти по ним в цикле, подстраивая весовые коэффициенты модели так, чтобы они как можно лучше отражали отношения токенов и их разметки. Загляните в главу 1 за более подробным рассказом о нейросетевых моделях и понятии весовых коэффициентов.

Существует несколько методик, использование которых в цикле обучения позволяет повысить точность. Следующий код иллюстрирует обработку обучающих примеров данных по отдельным порциям (так называемым *батчам*). При этом обучающие примеры демонстрируются модели в различных представлениях во избежание усвоения ненужных обобщений, которые могут встретиться в обучающем корпусе.

Добавьте следующий код в сценарий, который сначала был создан в подразделе «Создание обучающих примеров данных» на с. 199, а затем модифицирован в предыдущем разделе:

```
import random
from spacy.util import minibatch, compounding
❶ optimizer = nlp.entity.create_optimizer()
for i in range(25):
  ❷ random.shuffle(train_exams)
      max_batch_size = 3
```

```
    ❸ batch_size = compounding(2.0, max_batch_size, 1.001)
    ❹ batches = minibatch(train_exams, size=batch_size)
      for batch in batches:
            texts, annotations = zip(*batch)
        ❺ nlp.update(texts, annotations, sgd=optimizer)
❻ ner = nlp.get_pipe('ner')
❼ ner.to_disk('/usr/to/ner')
```

Прежде чем приступить к обучению, необходимо создать *оптимизатор* ❶ — функцию, отвечающую за хранение промежуточных результатов (состояния) между обновлениями весов модели. Можно было бы создать оптимизатор с помощью метода `nlp.begin_training()`, но он очищает список типов сущностей. А в этом примере при обновлении уже существующей модели нежелательно, чтобы она «забывала» имеющиеся типы сущностей, поэтому воспользуемся методом `nlp.entity.create_optimizer()`, который создает оптимизатор для средства распознавания именованных сущностей без потери имеющегося набора типов сущностей.

Во время обучения примеры данных демонстрируются модели в цикле в случайном порядке. Это необходимо для избежания ненужных обобщений на основе упорядоченности примеров ❷. Кроме того, сценарий обрабатывает обучающие примеры данных по батчам — документация spaCy утверждает, что таким образом можно повысить эффективность процесса обучения при достаточно большом количестве примеров. Для варьирования размера батчей на каждом шаге используется метод `compounding()`, который возвращает генератор размеров батчей: по сути, он генерирует бесконечный ряд нарастающих значений. Начинается ряд с указанного в качестве первого параметра значения, а каждое последующее значение образуется путем умножения предыдущего на третий параметр, но так, чтобы не превышалось заданное вторым параметром максимальное значение ❸. Затем обучающие примеры данных организуются в батчи с помощью метода `minibatch()`. Размер батча (параметр `size`) при этом задается итератором, который вернул в предыдущей строке кода метод `compounding()` ❹.

Далее надо пройти в цикле по батчам, обновляя модель средства распознавания именованных сущностей на каждой итерации. Для каждого батча необходимо обновить модель путем вызова метода `nlp.update()` ❺,

который выполняет предсказание для всех сущностей, найденных в примерах данных батча, а затем проверяет правильность переданной ему разметки. Если предсказание неправильно, в процессе обучения весовые коэффициенты модели корректируются, чтобы в следующий раз вес правильного предсказания был выше.

Наконец, необходимо сериализовать обновленный компонент распознавания именованных сущностей на диск, чтобы в будущем иметь возможность загрузить его в другом сценарии (или другом сеансе Python). Для этого необходимо сначала извлечь компонент из конвейера ❻, а затем сохранить его на диск с помощью метода `to_disk()` ❼, предварительно убедившись, что в системе есть каталог `/usr/to`.

Оценка работы обновленного средства распознавания именованных сущностей

Теперь можно протестировать обновленное средство распознавания именованных сущностей. Если вы выполняете обсуждавшийся в этой главе пример, закройте соответствующий сеанс Python, откройте новый и введите следующий код, чтобы гарантировать правильные обобщения модели (если же вы создали отдельный сценарий из обсуждавшегося в предыдущих разделах кода и запустили его, можете выполнить следующий код либо в качестве отдельного сценария, либо из сеанса Python):

```
import spacy
from spacy.pipeline import EntityRecognizer
❶ nlp = spacy.load('en', disable=['ner'])
❷ ner = EntityRecognizer(nlp.vocab)
❸ ner.from_disk('/usr/to/ner')
❹ nlp.add_pipe(ner, "custom_ner")
❺ print(nlp.meta['pipeline'])
❻ doc = nlp(u'Could you pick me up at Solnce?')
for ent in doc.ents:
  print(ent.text, ent.label_)
```

Сначала загружаем компоненты конвейера, кроме компонента распознавания именованных сущностей ❶. Дело в том, что обучение компонента конвейера существующей модели не перекрывает навсегда исходное поведение компонента. При загрузке модели по умолчанию загружаются исходные версии компонентов, составляющих конвейер

модели. Поэтому, чтобы использовать обновленную версию, ее необходимо явно загрузить с диска. Таким образом, можно работать с несколькими пользовательскими вариантами одного и того же компонента конвейера, загружая при необходимости нужный.

Создание нового компонента состояло из двух шагов: формирования нового экземпляра конвейера из класса `EntityRecognizer` ❷ и загрузки в него данных с диска (при этом в качестве параметра указывался каталог, в который был сериализован компонент распознавания именованных сущностей ❸).

Теперь нужно добавить загруженный компонент распознавания именованных сущностей в текущий конвейер, указав при этом собственное название (что не обязательно) ❹. Если вывести названия доступных в настоящий момент компонентов конвейера ❺, то, помимо названий `'tagger'` и `'parser'`, там будет и ваше нестандартное.

Осталось протестировать загруженный компонент распознавания именованных сущностей. Разумеется, следует взять не то предложение, что было включено в обучающий набор данных ❻.

В результате должно получиться следующее:

```
Available pipe components: ['tagger', 'parser', 'custom_ner']
Solnce GPE
```

Теперь обновленный компонент распознавания именованных сущностей правильно распознает пользовательские названия именованных сущностей.

Создание нового синтаксического анализатора

В следующих разделах рассмотрим, как создать свой синтаксический анализатор, приспособленный для конкретной задачи. В частности, обучим анализатор, способный раскрывать в предложении семантические отношения, а не синтаксические зависимости. *Семантические отношения* (semantic relations) складываются в предложении между значениями слов и фраз.

Понимание входного текста с помощью нестандартного синтаксического разбора зависимостей

Для чего могут понадобиться семантические отношения? Представьте, что чат-бот должен понимать запросы пользователя, сформулированные на английском языке, и преобразовывать их в SQL-запросы для передачи в базу данных. Для этого он производит синтаксический разбор, выделяя смысл и разрезая входной текст на части, которые затем используются при формировании запроса к базе данных.

Например, нам нужно разобрать следующее предложение:

```
Find a high paid job with no experience.
```

Один из возможных SQL-запросов, сгенерированных на его основе, выглядит так:

```
SELECT * FROM jobs WHERE salary = 'high' AND experience = 'no'
```

Сначала посмотрим, как этот пример предложения обработает стандартный синтаксический анализатор. Для этого можно воспользоваться таким сценарием:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'Find a high paid job with no experience.')
print([(t.text, t.dep_, t.head.text) for t in doc])
```

Сценарий выводит текст, метки зависимостей и синтаксические главные элементы всех токенов. При использовании модели `en_core_web_sm` результат выполнения выглядит следующим образом:

```
[
  ('Find', 'ROOT', 'Find'),
  ('a', 'det', 'job'),
  ('high', 'amod', 'job'),
  ('paid', 'amod', 'job'),
  ('job', 'dobj', 'Find'),
  ('with', 'prep', 'Find'),
  ('no', 'det', 'experience'),
  ('experience', 'pobj', 'with'),
  ('.', 'punct', 'Find')
]
```

Схематически этот разбор зависимостей можно изобразить так, как показано на рис. 10.2.

Такой синтаксический разбор вряд ли позволит сгенерировать из предложения желаемый запрос к базе данных. Приведенный ранее в этом разделе SQL-запрос выбирает работу, удовлетворяющую требованиям «высокая зарплата» и «нет опыта работы». По этой логике слово *job* должно соединяться не только с *high paid*, но и с *no experience*, однако в этом синтаксическом разборе *job* с *no experience* не связано.

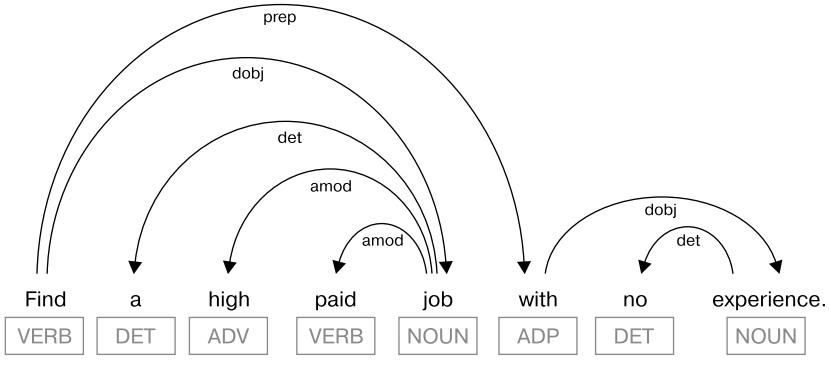


Рис. 10.2. Синтаксический разбор зависимостей примера предложения

Для удовлетворения требований приложения разметку придется поменять так, чтобы упростить задачу генерации запросов базы данных. Поэтому необходимо реализовать нестандартный синтаксический анализатор, который отображал бы семантические отношения, а не синтаксические зависимости. В данном случае нам нужна дуга между словами *job* и *experience*. В следующих разделах описано, как это сделать.

Выбор используемых типов семантических отношений

Во-первых, необходимо выбрать множество маркируемых типов отношений. Документация spaCy включает пример нестандартного средства разбора сообщений (https://spacy.io/usage/training/#intent-parser) со следующими семантическими отношениями: `ROOT`, `PLACE`, `ATTRIBUTE`,

QUALITY, TIME и LOCATION. Например, можно присвоить метку PLACE месту, где что-то происходит, — месту *hotel* в высказывании *I need a hotel in Berlin*. В этом же предложении *Berlin* получит метку LOCATION, что позволит различать географические объекты и пункты меньшего размера.

Для соответствия требованиям семантики в список необходимо добавить еще один тип — ACTIVITY. Он послужит меткой для слова *job* в нашем примере предложения. (Хотя и исходного множества типов отношений может быть достаточно, ведь работа обычно ассоциируется с конкретным местом, для которого можно использовать тип PLACE.)

Создание обучающих примеров данных

Как обычно, для процесса обучения компонента конвейера сначала следует подготовить обучающие примеры данных. Для обучения синтаксического анализатора необходима информация о метке зависимости каждого токена и главном элементе каждого отношения. В данном примере ради краткости и упрощения изложения используется лишь одна пара обучающих примеров данных. Конечно, на практике для обучения компонента анализатора их понадобится гораздо больше.

```
TRAINING_DATA = [
    ('find a high paying job with no experience', {
        'heads': [0, 4, 4, 4, 0, 7, 7, 4],
        'deps': ['ROOT', '-', 'QUALITY', 'QUALITY', 'ACTIVITY', '-',
                 'QUALITY', 'ATTRIBUTE']
    }),
    ('find good workout classes near home', {
        'heads': [0, 4, 4, 4, 0, 6, 4],
        'deps': ['ROOT', '-', 'QUALITY', 'QUALITY', 'ACTIVITY',
                 'QUALITY', 'ATTRIBUTE']
    })
]
```

Учтите, что в новом анализаторе синтаксически связанные слова не всегда будут связаны семантическим отношением. Чтобы в этом убедиться, можете выполнить следующий тест, в котором генерируется список главных элементов *синтаксических* зависимостей, найденных в примере предложения из первого обучающего примера списка TRAINING_DATA:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'find a high paying job with no experience')
heads = []
for token in doc:
    heads.append(token.head.i)
print(heads)
```

При использовании модели `en_core_web_sm` код выводит следующий список индексов главных элементов токенов:

```
[0, 4, 4, 4, 0, 4, 7, 5]
```

Если сравнить его с главными элементами, указанными для того же предложения в списке `TRAINING_DATA`, станут заметны отличия. В частности, в обучающем примере слово *with* является дочерним элементом слова *experience*, в то время как согласно стандартным синтаксическим правилам *with* в этом предложении — дочерний элемент слова *job*. Такое отклонение от правил будет понятнее, если немного изменить наше предложение:

```
find a high paying job without any experience
```

В плане семантики *without* можно считать модификатором слова *experience*, поскольку *without* меняет смысл слова *experience*. Модификаторы, в свою очередь, всегда зависят от модифицируемого слова. Следовательно, учитывая семантику, в приведенном примере логично считать, что *without* — дочерний элемент в паре *without/experience*.

Обучение анализатора

В следующем сценарии показано обучение синтаксического анализатора с нуля при использовании пустой модели. В этом примере создать совершенно новый анализатор логичнее, чем обновлять уже существующий: обучить уже существующий анализатор синтаксических зависимостей распознавать еще и семантические отношения весьма непросто, поскольку эти два типа отношений часто противоречат друг другу. Но это не значит, что нельзя использовать свой нестандартный анализатор с уже существующими моделями. Его

можно загрузить в любую модель, заменив исходный анализатор синтаксических зависимостей.

Для обучения анализатора в следующем сценарии используются обучающие примеры данных из объявленного в предыдущем разделе списка `TRAINING_DATA`. Не забудьте вставить список `TRAINING_DATA` перед следующим кодом:

```
import spacy
❶ nlp = spacy.blank('en')
❷ parser = nlp.create_pipe('parser')
❸ nlp.add_pipe(parser, first=True)
❹ for text, annotations in TRAINING_DATA:
  ❺ for d in annotations.get('deps', []):
    ❻ parser.add_label(d)
❼ optimizer = nlp.begin_training()
   import random
❽ for i in range(25):
    ❾ random.shuffle(TRAINING_DATA)
        for text, annotations in TRAINING_DATA:
            nlp.update([text], [annotations], sgd=optimizer)
❿ parser.to_disk('/home/oracle/to/parser')
```

Сначала создаем пустую модель ❶, затем — пустой компонент анализатора ❷ и добавляем его в конвейер модели ❸.

В этом примере получаем набор меток для анализатора из добавленного нами к коду списка `TRAINING_DATA`. Операция реализована в виде двух циклов: во внешнем цикле проходим по обучающим примерам, выделяя из каждого кортеж, содержащий главный элемент и разметку зависимостей ❹; во внутреннем — проходим по кортежу разметки, извлекая метки из списка `deps` ❺ и добавляя их в анализатор ❻.

Теперь можно приступить к процессу обучения. Сначала обзаводимся оптимизатором ❼, после чего реализуем простой цикл обучения ❽, перетасовывая обучающие примеры данных в случайном порядке ❾. Далее проходим в цикле по обучающим примерам, обновляя на каждой итерации модель анализатора.

Наконец, сериализуем наш нестандартный анализатор на диск, чтобы позднее его можно было загружать и использовать в других сценариях ❿.

Тестирование нестандартного анализатора

Загрузить нестандартный анализатор с диска в конвейер уже существующей модели можно с помощью следующего сценария:

```
import spacy
from spacy.pipeline import DependencyParser
❶ nlp = spacy.load('en', disable=['parser'])
❷ parser = DependencyParser(nlp.vocab)
❸ parser.from_disk('/home/oracle/to/parser')
❹ nlp.add_pipe(parser, "custom_parser")
print(nlp.meta['pipeline'])
doc = nlp(u'find a high paid job with no degree')
❺ print([(w.text, w.dep_, w.head.text) for w in doc if w.dep_ != '-'])
```

Отмечу, что этот сценарий аналогичен сценарию для загрузки нестандартного средства распознавания именованных сущностей, который был приведен в подразделе «Оценка работы обновленного средства распознавания именованных сущностей» на с. 205. Загружаем обычную модель, отключая определенный компонент: в данном примере — синтаксический анализатор ❶. Далее создаем анализатор ❷ и загружаем в него данные, предварительно сериализованные на диск ❸. Чтобы использовать анализатор, его необходимо добавить в конвейер модели ❹. После этого анализатор можно протестировать ❺.

Результаты работы сценария должны выглядеть следующим образом:

```
['tagger', 'ner', 'custom_parser']
[
  ('find', 'ROOT', 'find'),
  ('high', 'QUALITY', 'job'),
  ('paid', 'QUALITY', 'job'),
  ('job', 'ACTIVITY', 'find'),
  ('no', 'QUALITY', 'degree'),
  ('degree', 'ATTRIBUTE', 'job')
]
```

Исходный компонент синтаксического анализатора в обычной модели был заменен нашей пользовательской версией, в то время как остальные компоненты конвейера не поменялись. Позднее можно будет восстановить исходный компонент, загрузив модель с помощью вызова `spacy.load('en')`.

Попробуйте сами

Обучив наш нестандартный синтаксический анализатор раскрывать семантические отношения, можете попробовать его в деле. Продолжите пример из этого раздела и напишите сценарий для генерации SQL-запроса на основе запроса на обычном английском языке. В сценарии проверьте элементы `ROOT` каждого запроса, чтобы определить, нужно ли формировать оператор `SELECT`. Затем на основе элемента `ACTIVITY` выберите, к какой таблице базы данных будет выполняться сгенерированный запрос. В предложении (clause) `WHERE` оператора используйте элементы `QUALITY` и `ATTRIBUTE`.

Резюме

Из библиотеки spaCy можно скачать набор предобученных статистических моделей и сразу же их использовать. Но часто эти модели не удовлетворяют всем заданным требованиям. Для большего соответствия задачам конкретного приложения иногда имеет смысл усовершенствовать один из компонентов конвейера уже существующей модели или создать новый в пустой модели.

В этой главе вы познакомились с тем, как обучить существующий компонент распознавателя именованных сущностей распознавать еще и дополнительный набор сущностей, которые по умолчанию были неправильно маркированы. Вы узнали, как обучить нестандартный компонент синтаксического анализатора предсказывать определенные древовидные структуры, связанные с входным текстом, которые отражают семантические отношения, а не синтаксические зависимости.

В обоих случаях первый и, вероятно, важнейший, требующий больше всего времени шаг состоит в подготовке обучающих данных. После этого, чтобы реализовать цикл обучения нестандартного компонента, необходимо написать всего несколько строк кода.

11 Развертывание собственного чат-бота

В предыдущих главах вам приходилось жестко «зашивать» весь входной текст в NLP-сценарии, вручную присваивая текст объекту `Doc`. Но с созданием чат-ботов для таких задач, как прием заказов, не все однозначно. Необходимо развернуть приложение в бот-канале, например в Telegram, который обеспечивает обмен сообщениями между ботом и пользователем.

Эта глава начинается с обзора устройства чат-бота. Я познакомлю вас с процессом подготовки платформы для чат-бота на основе Telegram и с этапами его развертывания на данной платформе. Вы научитесь обрабатывать разнообразные виды вводимых пользователями данных с помощью API Telegram и хранить беседы для отслеживания уже заданных вопросов.

Схема реализации и развертывания чат-бота

В этом разделе более подробно рассматривается процесс обмена информацией между обычным чат-ботом и пользователем, а также необходимая для этого обмена структура.

Обычный чат-бот включает в себя несколько уровней. Реализовав на своей машине логику обработки вводимых пользователем данных, вам придется озаботиться приложением-мессенджером для создания учетных записей, с которыми будут работать ваши программы. Пользователи не взаимодействуют с реализацией бота на вашей машине

напрямую: они общаются с ботом через мессенджер. Помимо мессенджера, чат-боту могут понадобиться дополнительные сервисы, например база данных или какое-либо другое хранилище данных.

Схема на рис. 11.1 демонстрирует сочетание уровней в типовом чат-боте.

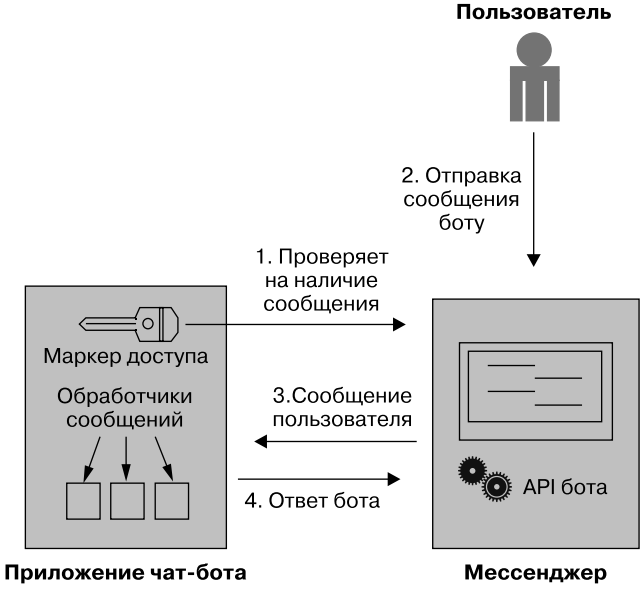


Рис. 11.1. Типичные взаимодействия между пользователем и интегрированным в мессенджер ботом

Сначала бот отправляет запросы мессенджеру в бесконечном цикле, проверяя, не начал ли пользователь разговор. Эти запросы включают маркер аутентификации, который генерируется при создании бота в мессенджере. У каждого бота — свой аутентификационный маркер (authentication token), также называемый маркером доступа или API-ключом, что позволяет мессенджеру распознавать, от какого бота поступают запросы.

Когда пользователь отправляет сообщение боту, мессенджер его обрабатывает, после чего перенаправляет адресату. Бот выбирает подходящий *обработчик* (handler) — процедуру, генерирующую ответы на конкретный тип сообщений пользователя — и отправляет сгенерированный ответ пользователю.

Для взаимодействия с пользователями используется программа-посредник. Обычно ею служит бот-платформа, которую предоставляет приложение-мессенджер (Skype, Facebook Messenger или Telegram). С точки зрения мессенджера бот является сторонним приложением, которое работает внутри мессенджера.

В следующем разделе мы детально обсудим процесс развертывания чат-бота, реализованного на языке Python, для бот-платформы Telegram. Вы увидите некоторые характерные для нее нюансы реализации ботов и научитесь использовать ее возможности для упрощения разработки ботов.

Я выбрал бот-платформу Telegram, потому что она предоставляет обширные ресурсы для Python-разработчиков, включая документацию по созданию ботов для Telegram на Python, различные руководства и справочники, а также примеры на GitHub. Другими словами, Telegram имеет все необходимое для создания чат-бота на Python. В других приложениях-мессенджерах, например Facebook Messenger, приходится использовать сторонние приложения (наподобие Flask или Ngrok), что усложняет реализацию бота и заставляет разработчика отвлекаться от задач NLP.

Telegram как платформа для бота

Telegram — облачный мессенджер и одно из наиболее распространенных приложений-мессенджеров в мире. Помимо прочих возможностей, он предоставляет платформу для создания ботов, а также библиотеку Python с удобным интерфейсом. Telegram можно использовать на платформах Android, iOS, Windows, Linux и macOS. В основном же он рассчитан на смартфоны.

Создание учетной записи Telegram и авторизация чат-бота

Прежде чем создать бот в Telegram, необходимо зарегистрировать учетную запись Telegram. Для этого вам понадобятся смартфон или планшет с iOS или Android. Версия Telegram для ПК не подойдет. Впрочем, после создания учетной записи Telegram ее можно использовать и на ПК.

Пошаговое руководство по созданию учетной записи Telegram можно найти по адресу https://telegramguide.com/create-a-telegram-account/. После создания учетной записи Telegram можно приступать к созданию бота. Сделать это можно как со смартфона, так и с ПК. Пошаговая инструкция приведена ниже.

1. В приложении Telegram введите в поиск `@BotFather` или откройте URL https://telegram.me/botfather. BotFather — это бот Telegram, предназначенный для управления всеми прочими ботами в вашей учетной записи.
2. На странице BotFather нажмите кнопку Start для вывода списка команд, с помощью которых можно настраивать боты Telegram.
3. Для создания нового бота введите команду `/newbot` в поле Write a message. При этом у вас будут запрошены название и имя пользователя создаваемого бота. Далее вы получите для него маркер авторизации. На рис. 11.2 показан снимок экрана смартфона во время этого процесса.

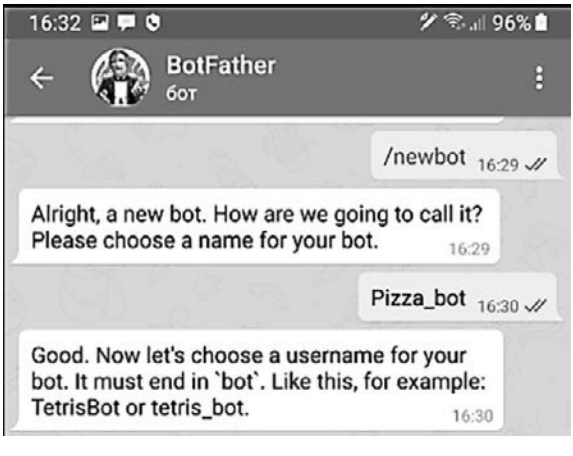


Рис. 11.2. Создание нового бота в Telegram с помощью смартфона

Теперь реализованную на вашей машине (на Python) функциональность бота можно интегрировать с ботом, только что созданным в Telegram, как описано в следующем разделе.

ПРИМЕЧАНИЕ

Важно понимать: бот, только что созданный вами в Telegram, не реализует логику обработки вводимых пользователем данных. Фактически он представляет собой обертку для настоящего бота, который вам придется реализовать самостоятельно.

Знакомство с библиотекой python-telegram-bot

Для подключения реализованной на Python функциональности чат-бота вам понадобится библиотека `python-telegram-bot`, основанная на API создания ботов Telegram. Она имеет удобный интерфейс для создателей ботов. Благодаря ей можно сосредоточиться на коде бота, а не на нюансах взаимодействия мессенджера с реализацией бота.

Библиотека `python-telegram-bot` представляет собой свободное ПО, которое распространяется по лицензии LGPLv3. Инсталлировать или обновить его посредством `pip` можно с помощью следующей команды:

```
$ pip install python-telegram-bot --upgrade
```

ПРИМЕЧАНИЕ

В приведенных далее примерах предполагается, что вы используете версию 12.0 или выше библиотеки python-telegram-bot.

После установки библиотеки проверьте возможность доступа к своему боту Telegram из Python с помощью следующих строк кода (для осуществления этой проверки необходимо соединение с Интернетом):

```
import telegram
bot = telegram.Bot(token='XXXXXX...')
```

Вместо `XXXXXX...` вставьте полученный при создании бота маркер. Затем проверьте свои учетные данные с помощью следующей строки:

```
print(bot.get_me())
```

Если функция `bot.get_me()` вернет ваши учетные данные, значит, указанный вами ранее аутентификационный маркер допустим.

Использование объектов telegram.ext

Для создания настоящего бота вам понадобятся объекты `telegram.ext`, включая `telegram.ext.Updater` и `telegram.ext.Dispatcher` — два самых важных объекта библиотеки, необходимых в любой реализации. Если коротко, объект `Updater` получает сообщения от Telegram и передает их объекту `Dispatcher`. В свою очередь, объект `Dispatcher` передает данные соответствующему обработчику. Следующий код иллюстрирует использование этих объектов в простом боте, отвечающем на любое сообщение сообщением с тем же текстом:

```
from telegram.ext import Updater, MessageHandler, Filters
# Функция, реализующая обработчик сообщений
❶ def echo(update, context):
    update.message.reply_text(update.message.text)
# Создание экземпляра Updater
❷ updater = Updater('TOKEN', use_context=True)
# Регистрация обработчика входных текстовых сообщений
updater.dispatcher.add_handler(MessageHandler(Filters.text, echo))
# Начинаем опрашивать мессенджер на предмет обновлений
updater.start_polling()
updater.idle()
```

Начинаем с импорта модулей `Updater` и `MessageHandler` из пакета `telegram.ext`. Затем описываем функцию `echo`, принимающую в качестве параметров два объекта: `update` и `context` ❶. Объект `update` представляет собой входящее сообщение — текст, фотографию, стикер и т. д. Объект `context` содержит атрибуты, хранящие данные из того же чата и от того же пользователя. Как `update`, так и `context` создаются неявным образом и передаются *функции обратного вызова* (callback) — функции обработки сообщений, связанной с определенным обработчиком. В этом примере роль функции обратного вызова обработчика текстовых сообщений играет `echo()`: она содержит одну-единственную строку кода, указывающую Telegram вернуть текстовое сообщение пользователя без каких-либо изменений.

Далее создаем объект `Updater` ❷, с помощью которого координируем процесс выполнения бота на протяжении всего сценария. При создании объекта `Updater` автоматически создается объект `Dispatcher`, что позволяет регистрировать обработчиков для различных типов входных данных, например для текста и фотографий. В данном примере

регистрируем один обработчик, предназначенный исключительно для текстовых сообщений, и передаем ему реализованную ранее в этом сценарии функцию обратного вызова. В результате чат-бот станет вызывать функцию обратного вызова при каждом получении сообщения Telegram, содержащего текст.

Далее запускаем бот и вызываем метод `start_polling()` объекта `Updater`, который начинает процесс опроса на предмет новых сообщений из мессенджера. Поскольку метод `start_polling()` — неблокирующий, нам приходится вызывать также метод `idle()` объекта `Updater`, блокирующий сценарий до момента получения сообщения или ввода пользователем сочетания клавиш выхода (Ctrl+C). Больше подробностей о классах и методах библиотеки `python-telegram-bot` вы найдете в документации по созданию ботов для Telegram на Python.

Для тестирования сценария запустите его на машине, подключенной к Интернету. После этого любой пользователь Telegram может вступить с ним в разговор. В приложении Telegram начните поиск по имени пользователя (@<*имя_пользователя*>) и введите вместо *имя_пользователя* имя пользователя, которое вы дали боту при его создании; затем выберите его. Для начала разговора нажмите кнопку START или введите команду `/start`. После этого можно отправлять боту сообщения. Поскольку реализованный бот является эхо-ботом, все его ответы будут содержать тот же текст, который был ему отправлен.

Создание чат-бота Telegram с использованием spaCy

В предыдущем разделе вы создали простой интегрированный в Telegram сценарий с помощью библиотеки `python-telegram-bot`. Теперь усовершенствуем эту реализацию и с помощью библиотеки spaCy сделаем созданный в Telegram бот полнофункциональным.

В следующем коде создается простой бот, который обрабатывает высказывание пользователя и определяет, содержит ли оно прямое дополнение. На основе этой информации генерируется ответ пользователю. Сам по себе такой код особой пользы не несет, но

демонстрирует, как можно связать реализованный с помощью spaCy код обработки текста с кодом, реализованным на основе библиотеки `python-telegram-bot`.

```
import spacy
from telegram.ext import Updater, MessageHandler, Filters
# Использующая spaCy функция обратного вызова
❶ def utterance(update, context):
    msg = update.message.text
    nlp = spacy.load('en')
    doc = nlp(msg)
    for token in doc:
      if token.dep_ == 'dobj':
        update.message.reply_text('We are processing your request...')
        return
    update.message.reply_text('Please rephrase your request. Be as
                               specific as possible!')
# Код, отвечающий за взаимодействие с Telegram
updater = Updater('TOKEN', use_context=True)
updater.dispatcher.add_handler(MessageHandler(Filters.text, utterance))
updater.start_polling()
updater.idle()
```

Обратите внимание, что код, отвечающий за взаимодействие с Telegram, такой же, как и в предыдущем сценарии. Единственное отличие состоит в реализации функции обратного вызова ❶. В данном случае функция `utterance()` использует для обработки ввода пользователя библиотеку spaCy.

В этой функции из переданного в нее объекта `update` извлекается текст сообщения, который далее преобразуется в объект `Doc` библиотеки spaCy. После этого необходимо проверить, содержит ли заключенное в объекте `Doc` высказывание прямое дополнение. Если нет — просим пользователя уточнить свой запрос. Например, пользователь может сказать: *I am hungry*, подразумевая, что хочет заказать какую-то еду. Но для размещения заказа необходимо уточнить, какую именно еду, например: *I want a pizza*.

Вероятно, самый интересный нюанс этого примера — иллюстрация получения обрабатываемых spaCy высказываний из приложения-бота. В примерах из предыдущих глав использовались «зашитые» в код высказывания. Здесь же впервые показано, как получают входные данные настоящие боты.

Расширение возможностей чат-бота

Разобравшись с интеграцией в Telegram чат-бота, использующего spaCy, можем заняться созданием более интересного варианта. Например, расширим функциональность бота из предыдущего раздела и научим его выделять намерение из сообщения пользователя, а не просто выводить сообщение о том, что запрос обрабатывается. Для этого подойдет сценарий, приведенный в одной из предыдущих глав.

Вернемся к сценарию из подраздела «Распознавание синонимов с помощью заранее заданных списков» на с. 168: в нем из введенного высказывания выделяется намерение с помощью списков синонимов. Вставьте код из этого сценария в отдельную функцию, скажем `extract_intent()`, которая принимает один параметр — текст введенного пользователем сообщения в виде объекта `Doc` (не забудьте убрать строку кода с «зашитым» высказыванием в начале сценария, а также строку в его конце, выводящую выделенное намерение). Функция среди прочего должна возвращать распознанное намерение в виде строкового значения. Вставьте в создаваемом сценарии эту новую функцию над функцией обратного вызова и отредактируйте функцию обратного вызова так, чтобы она выглядела следующим образом:

```
...
def extract_intent(doc):
  # Вставьте сюда код из главы 8

def utterance(update, context):
  msg = update.message.text
  nlp = spacy.load('en')
  doc = nlp(msg)
  for token in doc:
    if token.dep_ == 'dobj':
   ❶ intent = extract_intent(doc)
   ❷ if intent == 'orderPizza':
        update.message.reply_text('We need some more information
                                  to place your order.')
      elif intent == 'showPizza':
        update.message.reply_text('Would you like to look at our menu?')
      else:
        update.message.reply_text('Your intent is not recognized.')
      return
  update.message.reply_text('Please rephrase your request. Be as
                            specific as possible!')
...
```

Чтобы получить информацию о намерении пользователя, вызовите только что написанную функцию `extract_intent()` из функции обратного вызова `utterance` ❶. Набор дальнейших действий зависит от полученного намерения. В данном примере просто отправьте пользователю соответствующее сообщение ❷.

Конечно, можно было вставить код из главы 8 непосредственно в функцию обратного вызова, но это отрицательно сказалось бы на удобочитаемости кода. Делать подобное не рекомендуется в принципе.

Сохранение состояния текущего чата

Созданный нами бот способен не только на простую обработку сообщений пользователя — теперь он умеет распознавать намерения. Но все равно этого мало для принятия заказов от пользователей. Основной недостаток заключается в том, что чат-бот использует одну и ту же функцию обратного вызова `utterance` для всего последующего ввода пользователя, даже если намерение уже было распознано и пришло время задать пользователю дополнительные вопросы.

Для решения этой проблемы необходимо *сохранять состояние* текущего чата, чтобы чат-бот знал, на какие вопросы пользователь уже ответил и что еще необходимо спросить. Функцию обратного вызова следует модифицировать так, чтобы она обрабатывала сообщения пользователя в соответствии с текущим состоянием чата.

Схема работы бота при этом будет следующей: если чат-бот еще не распознал намерение, он попросит пользователя выразить намерение явным образом. После выяснения намерения бот перейдет к другому вопросу, который соответствует текущему состоянию разговора.

Библиотека `python-telegram-bot` включает объект `ConversationHandler`, упрощающий такую реализацию. С его помощью можно описывать точки входа и состояния разговора, привязывая их к разным обработчикам.

Возьмем такой пример: точка входа — команда Telegram `/start` — связывается с обработчиком, инициирующим начало диалога. Функция обратного вызова обработчика должна возвращать начальное состояние разговора, определяющее, какой обработчик использовать для

очередного сообщения пользователя. После отправки ответа пользователю функция обратного вызова обработчика возвращает новое состояние, меняя, таким образом, состояние разговора.

Следующий код включает фрагменты сценария, которые иллюстрируют изменение состояния разговора между чат-ботом и пользователем с помощью объекта `ConversationHandler`:

```
def start(update, context):
...
 ❶ return 'ORDERING'
def intent_ext(update, context):
...
 ❷ if context.user_data.has_key('intent'):
        return 'ADD_INFO'
    else:
        update.message.reply_text('Please rephrase your request.')
        return 'ORDERING'
def add_info(update, context):
...
    return ConversationHandler.END
def cancel(update, context):
...
    return ConversationHandler.END
...
def main():
...
    disp = updater.dispatcher
    conv_handler = ConversationHandler(
        entry_points=[CommandHandler('start', start)],
        states={
         ❸ 'ORDERING': [MessageHandler(Filters.text, intent_ext)
                        ],
           'ADD_INFO': [MessageHandler(Filters.text, add_info)
                        ],
        },
        fallbacks=[CommandHandler('cancel', cancel)]
    )
    disp.add_handler(conv_handler)
...
```

Объект `ConversationHandler` позволяет описывать несколько функций обратного вызова и задавать порядок их вызова. Функция обратного вызова обрабатывает сообщение пользователя и, в зависимости от результатов обработки, меняет (или не меняет) состояние диалога.

В данном примере функция обратного вызова для команды `/start` переводит разговор в состояние `ORDERING` ❶. Это значит, что следующее сообщение от пользователя будет обработано функцией `intent_ext()`, поскольку функция `intent_ext()` теперь относится к обработчику состояния `ORDERING` ❸, как указано в ассоциативном массиве `states` объекта `ConversationHandler`.

Обратите внимание, что чат-бот может переключаться между состояниями, опираясь на логику условного перехода, как показано в функции `intent_ext()`: состояние разговора меняется на `ADD_INFO` (состояние, в котором производится сбор дополнительной информации) только в случае, если намерение распознано ❷.

Собираем все части чат-бота вместе

Теперь, когда в общих чертах понятно, как должен быть организован бот Telegram, ведущий разговор по заранее заданной схеме, обратимся к полной реализации подобного сценария. Для заполнения формы заказа наш бот должен задать пользователю ряд вопросов один за другим.

В нашем упрощенном примере чат-бот будет обрабатывать только одно намерение, `orderPizza`, и просить пользователя, заполняющего форму заказа, уточнить разновидность заказываемой пиццы.

Сценарий разбит на фрагменты, соответствующие описаниям отдельных функций:

```
import logging
import sys
import spacy
from telegram.ext import Updater, CommandHandler, MessageHandler,
Filters, ConversationHandler
# Служит для получения общей отладочной информации
logger = logging.getLogger(__name__)
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)

def extract_intent(doc):
  # Сюда необходимо вставить код, созданный согласно описанному выше,
  # в разделе «Расширение возможностей чат-бота»
  ...
  return intent
```

Функция `extract_intent()` выделяет из введенного высказывания намерение. Она вызывается из описанной далее функции обратного вызова `intent_ext()`. Код `extract_intent()` здесь не приведен, и вы можете использовать код, описанный в подразделе «Расширение возможностей чат-бота» на с. 222.

```
def details_to_str(user_data):
    details = list()
    for key, value in user_data.items():
        details.append('{} - {}'.format(key, value))
    return "\n".join(details).join(['\n', '\n'])
```

Функция `details_to_str()` просто преобразует содержимое ассоциативного массива `user_data` в строковый вид. Ассоциативный массив `user_data` содержит информацию, которую чат-бот извлекает из разговора: например, тип пиццы и заказываемое количество. Эту информацию бот включает в итоговое сообщение, которое отправляется пользователю.

До этого момента описывались вспомогательные функции, которые напрямую или косвенно должны вызываться из функций обратного вызова нашего бота. Теперь опишем сами функции обратного вызова:

```
def start(update, context):
    update.message.reply_text('Hi! This is a pizza ordering app. Do you
                              want to order something?')
    return 'ORDERING'
```

Функция `start()` представляет собой функцию обратного вызова для команды `/start` Telegram. Другими словами, чат-бот вызывает эту функцию в начале разговора. Она возвращает состояние `ORDERING`, которое означает, что следующее полученное сообщение должно быть обработано функцией обратного вызова, связанной с обработчиком состояния `ORDERING` (в данном примере ею является функция `intent_ext()`).

```
def intent_ext(update, context):
  msg = update.message.text
  nlp = spacy.load('en')
  doc = nlp(msg)
  for token in doc:
    if token.dep_ == 'dobj':
      intent = extract_intent(doc)
      if intent == 'orderPizza':
        context.user_data['product'] = 'pizza'
        update.message.reply_text('We need some more information to place
          your order. What type of pizza do you want?')
```

```
        return 'ADD_INFO'
      else:
        update.message.reply_text('Your intent is not recognized. Please
        rephrase your request.')
        return 'ORDERING'
      return
  update.message.reply_text('Please rephrase your request. Be as specific
  as possible!')
```

Для простоты примера приведенная здесь функция `intent_ext()` способна распознавать только одно намерение: `orderPizza`. При обнаружении такого намерения она возвращает состояние `ADD_INFO`. В противном случае функция возвращает состояние `ORDERING`, в результате чего для обработки следующего сообщения пользователя она будет вызвана снова. Обработчик состояния `ADD_INFO` можно реализовать следующим образом:

```
def add_info(update, context):
  msg = update.message.text
  nlp = spacy.load('en')
  doc = nlp(msg)
  for token in doc:
    if token.dep_ == 'dobj':
      dobj = token
      for child in dobj.lefts:
        if child.dep_ == 'amod' or child.dep_ == 'compound':
          context.user_data['type'] = child.text
          user_data = context.user_data
          update.message.reply_text("Your order has been placed."
                                    "{}"
                                    "Have a nice day!".format(
                                     details_to_str(user_data)))
      return ConversationHandler.END
  update.message.reply_text("Cannot extract necessary
                            info. Please try again.")
  return 'ADD_INFO'
```

Функция `add_info()` представляет собой обратный вызов для обработчика состояния `ADD_INFO`. В этой реализации она ждет, когда пользователь уточнит, какая именно пицца ему нужна, после чего переводит состояние в `ConversationHandler.END` — последнее состояние:

```
def cancel(update, context):
    update.message.reply_text("Have a nice day!")
    return ConversationHandler.END
```

Приведенная выше функция `cancel()` просто прощается с пользователем и переводит состояние в `ConversationHandler.END`.

Наконец, функция `main()` должна выглядеть примерно так:

```
def main():
    # Замените TOKEN на фактический маркер авторизации
    updater = Updater("TOKEN", use_context=True)
    disp = updater.dispatcher
    conv_handler = ConversationHandler(
        entry_points=[CommandHandler('start', start)],
        states={
            'ORDERING': [MessageHandler(Filters.text,
                                        intent_ext)
                         ],
            'ADD_INFO': [MessageHandler(Filters.text,
                                        add_info)
                         ],
        },
        fallbacks=[CommandHandler('cancel', cancel)]
    )
    disp.add_handler(conv_handler)
    updater.start_polling()
    updater.idle()
if __name__ == '__main__':
    main()
```

Как обычно, весь процесс выполнения бота координирует функция `main()` его сценария.

Проверить сценарий можно с помощью веб-приложения Telegram. На рис. 11.3 приведен скриншот веб-приложения Telegram при выполнении этого сценария.

Попробуйте сами

Модифицируйте сценарий из предыдущего раздела так, чтобы он распознавал и другие намерения, а не только `orderPizza`. Еще одно возможное намерение: `showPizza`, означающее, что пользователь хотел бы взглянуть на меню. Для реализации нужно будет внести изменения в функцию `intent_ext()`, добавив условие `if intent == 'showPizza'` в цикл обработки объекта `Doc`. Кроме того, необходимо добавить новое состояние в ассоциативный массив `states` объекта `ConversationHandler` — с названием, скажем, `SHOW_MENU` — и реализовать для него функцию обратного вызова.

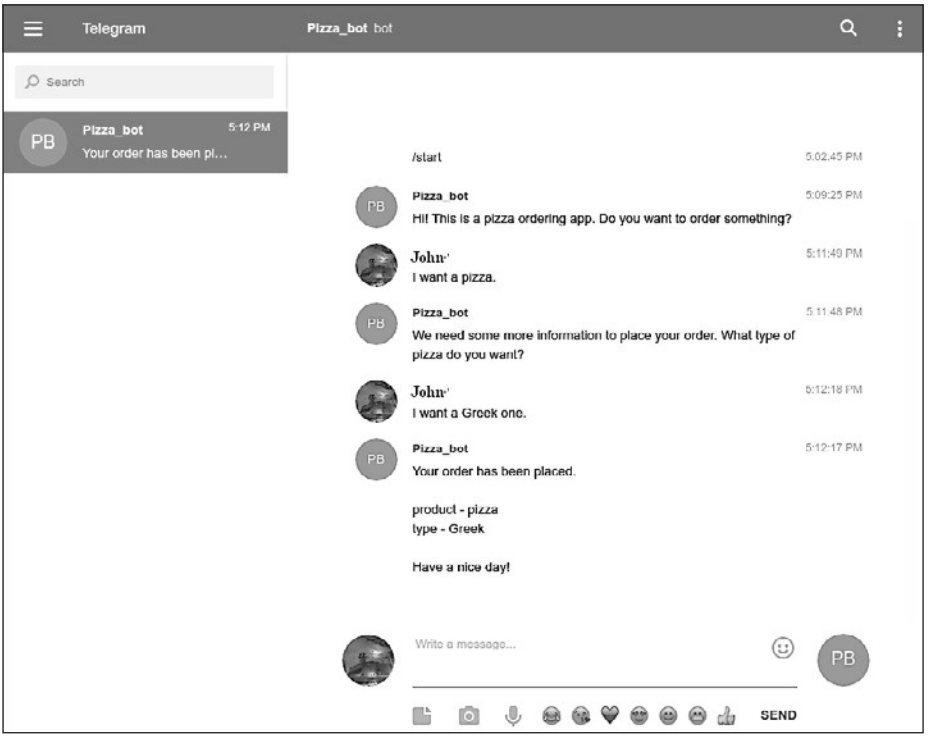


Рис. 11.3. Тестирование бота с помощью веб-приложения Telegram

Резюме

В этой главе вы научились реализовывать и развертывать простой чат-бот с помощью бот-платформы Telegram — популярного приложения-мессенджера. Вы научились описывать и сохранять состояния разговора, а также, что немаловажно, увидели, откуда в действительности могут поступать сообщения пользователей, которые затем обрабатываются с помощью spaCy.

12 Реализация веб-данных и обработка изображений

Чат-боты, используемые на практике, должны адекватно реагировать на самые разные виды вводимых пользователем данных — от вопросов на неожиданные темы до изображений, отправляемых через мессенджеры. Пользователи чат-ботов могут отправлять не только текстовые сообщения, но и фотографии, и бот должен реагировать соответствующим образом и на то и на другое.

В этой главе приведены примеры использования прочих библиотек из экосистемы ИИ языка Python для разработки чат-бота. Первым делом для поиска информации о ключевых словах, извлеченных из вопроса пользователя, свяжем spaCy с «Википедией». Затем покажем, как получить описательные теги для изображения с помощью Clarifai — утилиты распознавания изображений и видеофайлов, благодаря которой ваше приложение сможет понимать визуальный контент.

Далее соберем все эти компоненты воедино и создадим бот Telegram, способный адекватно реагировать на текст и изображения с помощью извлечения нужной информации из «Википедии».

Схема работы

На рис. 12.1 приведена схема работы бота, который предстоит создать в этой главе: он будет понимать текстовые сообщения и картинки, отвечая на них текстом из «Википедии».

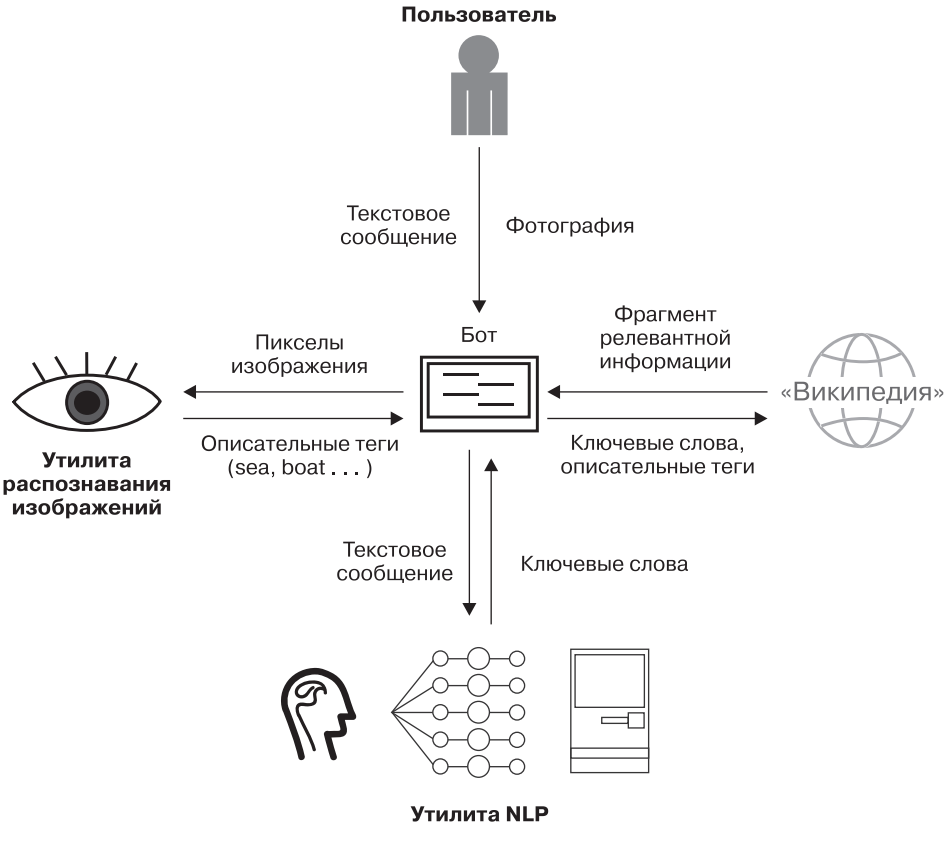


Рис. 12.1. Схема работы бота, способного обрабатывать текстовые сообщения и изображения

Пользователь может отправлять этому боту как текстовые сообщения, так и изображения. Картинки бот посылает для обработки утилите распознавания изображений, которая возвращает словесное описание изображения в виде описательных тегов. Из текстовых сообщений бот выделяет ключевые слова или фразы с помощью NLP-утилит наподобие spaCy. Далее бот находит по тегу или ключевой фразе наиболее подходящий контент в «Википедии» (или где-нибудь еще в Интернете) и возвращает часть его пользователю. Такую схему работы можно использовать в чат-ботах для поддержания разговора на различные темы и с разной целью: для развлечения, учебы или личных нужд.

Учим бот искать ответы на вопросы в «Википедии»

Начнем с обсуждения того, какие методики необходимо реализовать в нашем боте, чтобы он мог интерпретировать широкий спектр текстовых сообщений. В предыдущих главах говорилось о том, что боты, используемые для коммерческой деятельности, обычно запрашивают у пользователя определенную информацию, заполняя на основе его ответа форму заказа или бронирования. Бот, предназначенный для поддержания неформальных разговоров, должен уметь отвечать на самые разнообразные вопросы пользователей.

Для ответа на вопросы пользователя чат-бот может, например, выбирать в вопросе ключевые слова или фразы, которые указывают, какую информацию следует включить в ответ. Ими затем можно пользоваться для поиска ответа с помощью API «Википедии» для языка Python. Этот API позволяет обращаться к содержимому «Википедии» программным образом и производить его разбор, извлекая контент из наиболее подходящих статей по ключевым словам. В следующих разделах описано, как это реализовать.

Прежде чем перейти к примерам, убедитесь, что используемая вами модель spaCy — достаточно свежая, поскольку точность синтаксического разбора в более новых версиях выше. Проверить версию вашей текущей модели можно с помощью следующей команды:

```
nlp.meta['version']
```

После этого зайдите на страницу интерактивной демонстрации https://explosion.ai/demos/displacy (она обсуждалась в главе 7), чтобы просмотреть список новых стабильных версий моделей spaCy. Или можете заглянуть в документацию spaCy, перейдя по адресу https://spacy.io/usage, и найти там текущую версию spaCy. Схемы версий библиотеки spaCy и ее моделей аналогичны. В зависимости от выясненной информации можете обновить используемую модель. Подробная информация о скачивании и установка модели spaCy приводилась в главе 2.

Выясняем, чему посвящен вопрос

Для понимания того, о чем говорит собеседник, в вопросе есть более важные и менее важные слова. Иногда бывает достаточно проанализировать одно слово — например, существительное, которое следует за предлогом. Например, пользователь может попросить бот выяснить информацию о носорогах с помощью любого из следующих вопросов:

```
Have you heard of rhinos? Are you familiar with rhinos? What could you
tell me about rhinos?
```

Посмотрим на результаты синтаксического разбора этих предложений. На рис. 12.2 приведено графическое представление синтаксического разбора первого предложения.

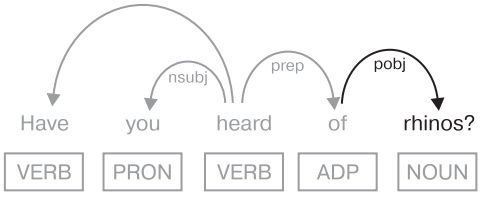


Рис. 12.2. Синтаксический разбор предложения, содержащего предложное дополнение

Этот синтаксический разбор демонстрирует, что в подобных предложениях слово *rhinos* можно получить путем выделения предложного дополнения. Именно *rhinos* является наиболее полезным словом в вопросе для поиска ответа. Следующий фрагмент кода демонстрирует, как можно выделить первое вхождение предложного дополнения в вопросе:

```
doc = nlp(u"Have you heard of rhinos?")
for t in doc:
  if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = (' '.join([child.text for child in t.lefts]) + ' ' +
              t.text).lstrip()
    break
```

В этом коде также извлекаются левосторонние дочерние элементы предложного дополнения, поскольку у дополнения могут быть важные

модификаторы, как в следующем примере: *What can you say about wild mountain goats?* При получении подобного вопроса код должен присвоить переменной `phrase` значение *wild mountain goats*.

Обратите внимание на использование в конце кода оператора `break`, благодаря которому из предложения выбирается только первое предложное дополнение. Например, в предложении *Tell me about the United States of America*. выбирается только фраза *the United States*, а не слово *America*.

Впрочем, подобный подход не всегда помогает. Если пользователь попросит, допустим: *Tell me about the color of the sky*., потребуется более изощренная логика. В таком случае можно выбрать все предложные дополнения, следующие за первым предложным дополнением, при условии, что последнее зависит от первых.

Эту логику можно реализовать следующим образом:

```
doc = nlp(u"Tell me about the color of the sky.")
for t in doc:
  if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = (' '.join([child.text for child in t.lefts]) + ' ' +
              t.text).lstrip()
    if bool([prep for prep in t.rights if prep.dep_ == 'prep']):
      prep = list(t.rights)[0]
      pobj = list(prep.children)[0]
      phrase = phrase + ' ' + prep.text + ' ' + pobj.text
    break
```

Обратите внимание, что код обрабатывает предложное дополнение, зависимое от первого предложного дополнения, только если последнее вообще существует в предложении. В противном случае этот код работает так же, как и приведенный выше фрагмент кода.

Теперь рассмотрим другой тип вопросов, в которых два слова — глагол и его подлежащее — лучше всего описывают, что пользователь хочет получить в ответ:

```
Do you know what an elephant eats? Tell me how dolphins sleep. What is
an API?
```

На рис. 12.3 показано, как выглядит синтаксический разбор одного из этих предложений.

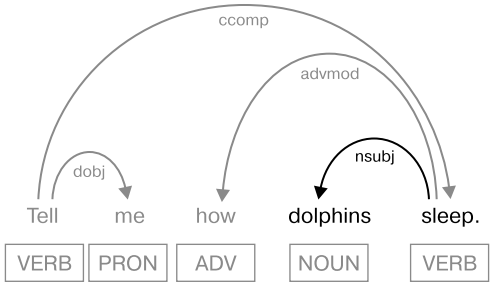


Рис. 12.3. Синтаксический разбор предложения, в котором наиболее информативным элементом для понимания собеседника является пара «подлежащее/глагол»

Как видно из приведенного на рисунке синтаксического разбора, наиболее информативным для понимания собеседника элементом является пара «подлежащее/глагол», встречающаяся в конце предложения. Выделить из предложения пару «подлежащее/глагол» можно с помощью такого кода:

```
doc = nlp(u"Do you know what an elephant eats?")
for t in reversed(doc):
  if t.dep_ == 'nsubj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = t.text + ' ' + t.head.text
    break
```

При изучении кода обратите внимание, что нужно пройти в цикле от конца предложения к началу с помощью функции `reversed()` языка Python. Дело в том, что необходимо извлечь последнюю в предложении пару «подлежащее/глагол», как в следующем примере: *Do you know what an elephant eats?* В этом предложении нас больше интересует фраза *elephant eats*, а не *you know* — еще одна пара «подлежащее/глагол».

Кроме того, в некоторых вопросах последнее существительное предложения является прямым дополнением глагола, который также важен для определения того, чему посвящен вопрос. Как в следующем примере:

```
How to feed a cat?
```

Просто выделить прямое дополнение *cat* в этом предложении недостаточно, поскольку для понимания вопроса необходимо также слово *feed*. Оптимальный вариант — сгенерировать фразу *feeding*

a cat, то есть заменить неопределенную (*to*) форму глагола герундием, добавив окончание *-ing*, в результате чего ключевая фраза приобретет оптимальную форму для поиска в Интернете. Синтаксический разбор этого предложения приведен на рис. 12.4.

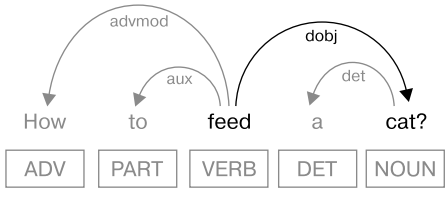


Рис. 12.4. Синтаксический разбор предложения, в котором наиболее информативной является пара «глагол/прямое дополнение»

Этот синтаксический разбор демонстрирует, насколько просто можно выделить требуемую пару, поскольку прямое дополнение и его переходный глагол находятся в прямой связи.

Реализация описанного выделения в виде кода выглядит следующим образом:

```
doc = nlp(u"How to feed a cat?")
for t in reversed(doc):
  if t.dep_ == 'dobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = t.head.lemma_ + 'ing' + ' ' + t.text
    break
```

В этом случае также необходимо пройти от конца предложения к началу. Чтобы понять, зачем это нужно, рассмотрим следующее предложение: *Tell me something about how to feed a cat*. Оно содержит две пары «глагол/прямое дополнение», но нас интересует только вторая, расположенная ближе к концу предложения.

Попробуйте сами

Модифицируйте код из предыдущего раздела, выделяющий фразу *elephant eats*, таким образом, чтобы выделяемая из предложения ключевая фраза включала в себя возможные модификаторы подлежащего,

за исключением возможного определителя. Например, для предложения *Tell me how a female cheetah hunts.* ваш сценарий должен вернуть *female cheetah hunts*, отбросив определитель *a* из именной группы. В качестве примера возможной реализации такого сценария взгляните на код, следующий за рис. 12.2: в нем выбирались модификаторы выделяемого предложного дополнения.

Кроме того, добавьте в код проверку наличия прямого дополнения у глагола, включенного в выделяемую фразу. В этом случае необходимо присоединить прямое дополнение в конец ключевой фразы. Например, в результате обработки вопроса *Do you know how many eggs a sea turtle lays?* должна получиться ключевая фраза *sea turtle lays eggs*.

Ответы на вопросы пользователей с помощью «Википедии»

У нас уже есть ключевая фраза, по которой можно найти информацию, необходимую для генерации ответа на вопрос пользователя. Осталось эту информацию извлечь. Бот может получать ответы на вопросы пользователей из различных источников (в зависимости от приложения), но «Википедия» — достаточно удачный вариант для старта. С помощью библиотеки Python `wikipedia` (https://pypi.org/project/wikipedia/) можно получить доступ к статьям «Википедии» из кода на Python.

Установить эту библиотеку с помощью `pip` можно так:

```
pip install wikipedia
```

Для проверки только что установленной библиотеки воспользуйтесь следующим сценарием, в котором для выделения ключевого слова из предложения используется фрагмент кода из предыдущего раздела. Далее по этому ключевому слову производится поиск в «Википедии».

```
import spacy
import wikipedia
nlp = spacy.load('en')
doc = nlp(u"What do you know about rhinos?")
for t in doc:
  if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
```

```
        ❶ phrase = (' '.join([child.text for child in t.lefts]) + ' ' +
                    t.text).lstrip()
          break
    ❷ wiki_resp = wikipedia.page(phrase)
      print("Article title: ", wiki_resp.title)
      print("Article url: ", wiki_resp.url)
      print("Article summary: ", wikipedia.summary(phrase, sentences=1))
```

В этом сценарии выделяем ключевое слово или фразу из полученного предложения ❶, после чего передаем его функции `wikipedia.page()`, возвращающей статью, лучше всего подходящую к данному ключевому слову ❷. Затем выводим название URL и первое предложение этой статьи.

Результат работы сценария должен выглядеть таким образом:

```
Article title:  Rhinoceros
Article url:  https://en.wikipedia.org/wiki/Rhinoceros
Article summary:  A rhinoceros (, from Greek  rhinokeros, meaning
'nose-horned', from  rhis, meaning 'nose', and  keras, meaning 'horn'),
commonly abbreviated to rhino, is one of ...
```

Попробуйте сами

Расширьте сценарий из предыдущего раздела так, чтобы он видел дочерние элементы первого предложного дополнения и зависимые от него предложные дополнения. Например, из предложения *Have you heard of fried eggs with yellow tomatoes?* он должен выделить ключевую фразу *fried eggs with yellow tomatoes.*

Реагируем на отправляемые в чаты изображения

Помимо текстовых сообщений, пользователи мессенджеров часто отправляют в чат изображения, а другие пользователи комментируют то, что изображено на рисунке. Например, кто-то отправляет фотографию винограда и получает в ответ такой комментарий: *I love fruit. It contains lots of fiber and vitamins.* Можно ли обучить бот делать нечто подобное? Один из способов — сгенерировать описательные теги для изображений, которые бот будет использовать при обработке. Для

этого необходима утилита распознавания изображений наподобие Clarifai: со встроенными моделями, обученными на фотографиях из различных сфер жизни и по разным темам, таким как одежда, путешествия, знаменитости и т. д.

С помощью Clarifai и ее набора категорий бот в результате «поймет», что изображено на заданной фотографии. Для этого, во-первых, необходимо получить описательные теги (объекты, содержащие вероятности) с помощью общей модели распознавания изображений Clarifai, чтобы для начала в общих чертах опознать изображенное на фотографии. Например, наличие тега *no person* указывает, что на фотографии нет людей.

Во-вторых, после изучения полученных тегов можно применить к той же фотографии более конкретные модели утилиты Clarifai (для продуктов питания, для одежды и т. д.), а также получить набор других, уточненных тегов, которые позволят лучше разобраться в том, что изображено. Полный список моделей распознавания изображений Clarifai находится на странице Models по адресу https://www.clarifai.com/models/.

Генерация описательных тегов для изображений с помощью Clarifai

Clarifai включает клиент Python для взаимодействия с его API распознавания. Установить свежий стабильный пакет можно с помощью `pip`:

```
pip install clarifai --upgrade
```

Прежде чем использовать библиотеку Clarifai, необходимо получить ключ API (для этого нужно создать учетную запись и нажать соответствующую кнопку на сайте https://www.clarifai.com/).

Получив ключ, вы можете приступать к работе с библиотекой Clarifai. Следующий простой сценарий передает изображение модели Clarifai и выводит список тегов, выражающих возможные категории, к которым относится изображение:

```
from clarifai.rest import ClarifaiApp, client, Image
app = ClarifaiApp(api_key='YOUR_API_KEY')
❶ model = app.public_models.general_model
```

```
filename = '/your_path/grape.jpg'
❷ image = Image(file_obj=open(filename, 'rb'))
response = model.predict([image])
❸ concepts = response['outputs'][0]['data']['concepts']
for concept in concepts:
  print(concept['name'], concept['value'])
```

В этом примере вызывался API Predict библиотеки Clarifai с универсальной моделью ❶. В качестве входных данных Clarifai принимает только пикселы, так что не забудьте указать режим `'rb'` ❷, чтобы открыть файл с изображением для чтения в двоичном формате. API Predict генерирует для указанной фотографии список описательных тегов, например *fruit*, *grape*, *health* и т. д. ❸, благодаря чему наш код сможет понять, что на ней изображено.

Использовавшийся в этом примере файл `grape.jpg` содержит фото, показанное на рис. 12.5.

Рис. 12.5. Фото, отправленное в Clarifai в предыдущем сценарии

Список концептов, сгенерированных сценарием для этой фотографии:

```
no person 0.9968359470367432
wine 0.9812138080596924
fruit 0.9805494546890259
juicy 0.9788177013397217
health 0.9755384922027588
grow 0.9669009447097778
grape 0.9660607576370239
...
```

Каждая запись содержит название категории и вероятность того, что изображение относится к этой категории. Так, первый тег в списке сообщает, что на данном фото с вероятностью 0,99 не изображены люди. Обратите внимание, что далеко не все теги напрямую описывают изображенное на фотографии. Например, тег *wine* в этот список включен, вероятно, потому, что вино делают из винограда. Наличие в списке непрямых тегов повышает гибкость интерпретации изображения ботом.

Генерация текстовых реакций на изображения на основе тегов

Теперь вы знаете, как получить описательные теги: осталось разобраться, как ими воспользоваться для адекватной реакции на изображение и как выбрать из списка наиболее важные теги. Вот несколько общих советов, к которым стоит прислушаться.

- Учитывать только теги с высоким значением вероятности. Для этого можно выбрать пороговое значение вероятности. Например, рассматривать только пять или десять тегов с максимальной вероятностью.
- Выбирать только те теги, которые подходят к контексту текущего чата. В главе 11 рассматривался пример сохранения контекста текущего чата в боте Telegram с помощью ассоциативного массива `context.user_data`.
- Проходить в цикле по сгенерированным тегам в поисках конкретного тега. Например, искать теги *fruit* или *health*, чтобы определить, следует ли продолжать разговор на данную тему.

Бот, о котором мы поговорим в следующем разделе, реализует третий из этих вариантов.

Собираем все части воедино в боте Telegram

В оставшейся части данной главы поработаем над созданием чат-бота Telegram, который будет использовать API «Википедии» и API библиотеки Clarifai, чтобы адекватно реагировать на тексты и изображения продуктов питания. Подробное описание процесса создания нового бота Telegram приводилось в главе 11.

Импорт библиотек

Раздел импортов кода должен включать все библиотеки, используемые в боте. В этом примере в код будут включены библиотеки, необходимые для доступа к API ботов Telegram, «Википедии», Clarifai и spaCy.

```
import spacy
import wikipedia
from telegram.ext import Updater, CommandHandler, MessageHandler, Filters
from clarifai.rest import ClarifaiApp, Image
```

Если вы строго следовали инструкциям из этой главы и главы 11, то все перечисленные библиотеки должны быть доступны в вашей системе.

Написание вспомогательных функций

Далее нужно реализовать вспомогательные функции, которые вызываются из функций обратного вызова бота. Функция `keyphrase()` принимает на входе предложение в виде объекта `Doc` и пытается выделить из него наиболее информативное слово или фразу, как обсуждалось ранее в подразделе «Выясняем, чему посвящен вопрос» на с. 233. Следующая реализация использует уже знакомые вам фрагменты кода, но с изменениями, необходимыми для применения их в одной функции:

```
def keyphrase(doc):
  for t in doc:
    if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
      return (' '.join([child.text for child in t.lefts]) + ' ' + t.text).
lstrip()
  for t in reversed(doc):
```

```
    if t.dep_ == 'nsubj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
      return t.text + ' ' + t.head.text
  for t in reversed(doc):
    if t.dep_ == 'dobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
      return t.head.text + 'ing' + ' ' + t.text
  return False
```

Обратите внимание, что условные выражения в этом коде располагаются в порядке приоритета. Так, обнаружив предложное дополнение, выделяем его и выходим из функции, не проверяя остальные условные выражения. Конечно, некоторые запутанные вопросы могут удовлетворять нескольким условиям, но подобная проверка усложнила бы реализацию функции.

Аналогично функции `keyphrase()` функция `photo_tags()` предназначена для определения слова, которое лучше всего описывает то, что ввел пользователь. Но, в отличие от `keyphrase()`, она анализирует фотографию. Анализ производится с помощью библиотеки Clarifai, генерирующей для указанной фотографии набор описательных тегов. В этой реализации используются только две модели Clarifai: общая и модель для продуктов питания:

```
def photo_tags(filename):
  app = ClarifaiApp(api_key=CLARIFAI_API_KEY)
  model = app.public_models.general_model
  image = Image(file_obj=open(filename, 'rb'))
  response = model.predict([image])
  concepts = response['outputs'][0]['data']['concepts']
  for concept in concepts:
    if concept['name'] == 'food':
      food_model = app.public_models.food_model
      result = food_model.predict([image])
      first_concept = result['outputs'][0]['data']['concepts'][0]['name']
      return first_concept
  return response['outputs'][0]['data']['concepts'][1]['name']
```

Код начинается с применения общей модели. Если в сгенерированном списке обнаружен тег `'food'`, то для того, чтобы получить теги, точнее других описывающие продукты на картинке, применяется модель для продуктов питания. В качестве ключевого слова для поиска в этой реализации используется только первый из найденных тегов.

После определения ключевого слова или ключевой фразы в функции `keyphrase()` или в функции `photo_tags()` необходимо получить соответствующий им фрагмент информации. Эту задачу решает функция `wiki()`:

```
def wiki(concept):
  nlp = spacy.load('en')
  wiki_resp = wikipedia.page(concept)
  doc = nlp(wiki_resp.content)
  if len(concept.split()) == 1:
    for sent in doc.sents:
      for t in sent:
        if t.text == concept and t.dep_ == 'dobj':
          return sent.text
  return list(doc.sents)[0].text
```

Используемый алгоритм ищет в извлеченном содержимом предложение, в котором ключевое слово играет роль прямого дополнения.

Но такая простая реализация способна адекватно обработать ввод, состоящий только из одного слова. При вводе слова приведенный алгоритм всего лишь извлекает первое подходящее предложение из найденной по этому слову статьи в «Википедии».

Написание функций обратного вызова и main()

Далее необходимо написать функции обратных вызовов бота. Функция `start()` просто приветствует пользователя в ответ на команду `/start`:

```
def start(update, context):
    update.message.reply_text('Hi! This is a conversational bot. Ask
                              me something.')
```

Функция `text_msg()` представляет собой обратный вызов для обработчика текстовых сообщений пользователя:

```
def text_msg(update, context):
  msg = update.message.text
  nlp = spacy.load('en')
  doc = nlp(msg)
  concept = keyphrase(doc)
  if concept != False:
    update.message.reply_text(wiki(concept))
  else:
    update.message.reply_text('Please rephrase your question.')
```

Прежде всего к сообщению пользователя применяется конвейер spaCy, который преобразует его в объект `Doc`. Далее объект `Doc` передается в описанную выше функцию `keyphrase()` для выделения из сообщения ключевого слова или фразы. Возвращенные ей ключевое слово или фраза затем передаются в функцию `wiki()` для получения соответствующего фрагмента информации (в данной реализации он представляет собой одно предложение).

Приведенная в следующем коде функция `photo()` представляет собой обратный вызов для обработчика посланных пользователем фотографий:

```
def photo(update, context):
  photo_file = update.message.photo[-1].get_file()
  filename = '{}.jpg'.format(photo_file.file_id)
  photo_file.download(filename)
  concept = photo_tags(filename)
  update.message.reply_text(wiki(concept))
```

Эта функция извлекает полученное изображение в виде файла и передает его для дальнейшей обработки вспомогательным функциям, которые обсуждались в подразделе «Написание вспомогательных функций» на с. 242.

Наконец, добавляем функцию `main()`, в которой регистрируем обработчиков текстовых сообщений и фотографий:

```
def main():
    updater = Updater("YOUR_TOKEN", use_context=True)
    disp = updater.dispatcher
    disp.add_handler(CommandHandler("start", start))
    disp.add_handler(MessageHandler(Filters.text, text_msg))
    disp.add_handler(MessageHandler(Filters.photo, photo))
    updater.start_polling()
    updater.idle()
if __name__ == '__main__':
    main()
```

Функция `main()` нашего бота довольно лаконична: создается объект `Updater`, в который передается маркер бота, затем получается объект `Dispatcher` для регистрации обработчиков. В этом примере регистрируются только три обработчика. Первый из них представляет собой обработчик команды `/start`. Второй обрабатывает поступающие от

пользователя текстовые сообщения. Третий обрабатывает выставляемые пользователем фотографии. После регистрации обработчиков бот запускается с помощью вызова `updater.start_polling()` и последующего вызова `updater.idle()` для блокировки сценария в ожидании поступления от пользователя сообщения или команды выхода (Ctrl+C).

Тестирование бота

После создания бот необходимо протестировать. Это можно сделать либо на смартфоне, либо на компьютере. На смартфоне введите в поиск приложения Telegram название вашего бота с предшествующим символом `@` и далее команду `/start` для начала чата. На компьютере воспользуйтесь веб-приложением Telegram, перейдя по адресу https://web.telegram.org.

После получения от чат-бота приветствия отправьте ему простой запрос, например *Tell me about fruit*. Бот должен ответить одним предложением, выделенным из соответствующей статьи «Википедии». Для простоты выберите предложение, в котором роль ключевого слова играет прямое дополнение (в данном случае *fruit*).

Можете также отправить фото и посмотреть, каким комментарием на него ответит бот. На рис. 12.6 приведен снимок экрана при подобной проверке.

Напоминаю, что эта реализация способна обрабатывать должным образом только фотографии продуктов питания.

Попробуйте сами

Учтите, что приведенная в предыдущем разделе реализация бота не способна генерировать разумные ответы на большое количество различных типов данных, вводимых пользователем. Используемая функция `wiki()` может обрабатывать должным образом только те запросы, на которые функция `keyphrase()` возвращает одно слово, и лучше всего она работает, если это слово является прямым дополнением. Бот может адекватно комментировать только изображения пищи.

Расширьте функцию `wiki()` так, чтобы она могла обрабатывать не отдельные слова, а фразы: например *dolphins sleep*. Чтобы найти подходящие для подобных предложений фразы, необходимо использовать метки зависимости, так как речь идет о паре «подлежащее/глагол». Кроме того, необходимо свести слова к их леммам. Например, критерию поиска должны удовлетворять фразы *dolphins sleep* и *dolphins sleeps*.

Можете также расширить функциональность функции `photo_tags()`, чтобы она обрабатывала фотографии не только пищевых продуктов, но и чего-то другого, например одежды.

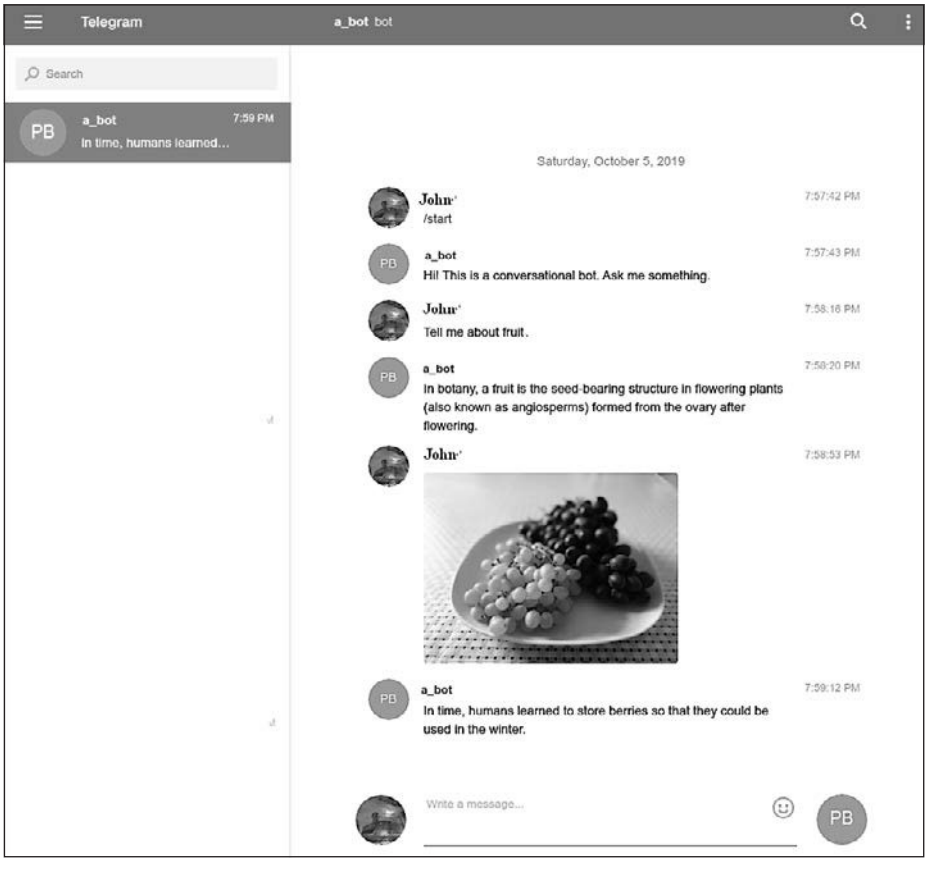


Рис. 12.6. Снимок экрана для созданного нами бота

Резюме

В этой главе мы рассмотрели совместное использование spaCy и других библиотек экосистемы ИИ языка Python для создания приложения с элементами искусственного интеллекта, способного обрабатывать данные различных типов. С помощью API для Python «Википедии» и Clarifai был разработан чат-бот, умеющий реагировать на изображения и извлекать текст из «Википедии» (подобные умения делают бот гораздо более интересным собеседником).

После прочтения книги вам, вероятно, захочется расширить и усовершенствовать полученные знания. Самый лучший способ это сделать — продолжить экспериментировать с чат-ботами. Начните с создания сценария Python для Telegram в соответствии с приведенными в главе 11 инструкциями; затем расширьте функциональность согласно имеющимся в этой главе инструкциям; займитесь усовершенствованием изложенных в книге алгоритмов и адаптацией их к вашим сценариям использования.

Приложение. Начальное руководство по лингвистике

В большинстве глав этой книги особое внимание уделялось поиску паттернов последовательностей слов путем анализа структуры предложений с помощью библиотеки spaCy. Чтобы понимать суть анализа предложений и паттернов слов, необходимы базовые знания в области лингвистики. Данное приложение представляет собой начальное справочное руководство по лингвистике.

Грамматика зависимостей и грамматика с фразовой структурой

По умолчанию spaCy использует грамматику зависимостей, а не более распространенную в лингвистике грамматику с фразовой структурой. В этом разделе я расскажу о различиях двух типов грамматик. Данная информация будет полезна тем, у кого есть лингвистическое образование.

Грамматика с фразовой структурой, известная также как *грамматика составляющих* (constituent-based grammar), моделирует естественный язык на основе формирования в предложении *составляющих* (constituent) из сочетаний слов. В синтаксисе составляющей называется группа слов, выступающих в предложении как единое целое. Правила фразовой структуры разбивают предложение на составляющие, формируя древовидную структуру, которая начинается с отдельных слов и постепенно дорастает до все больших составляющих.

Грамматика зависимостей представляет собой грамматику, упор в которой делается на отношения между отдельными словами, а не между составляющими предложения. Поэтому при синтаксическом разборе зависимостей, подобно тем, что многократно встречались в книге, формируется дерево, отражающее связи слов в предложении друг с другом.

На рис. П.1 приведен пример разбора предложения на основе каждого из этих типов грамматик.

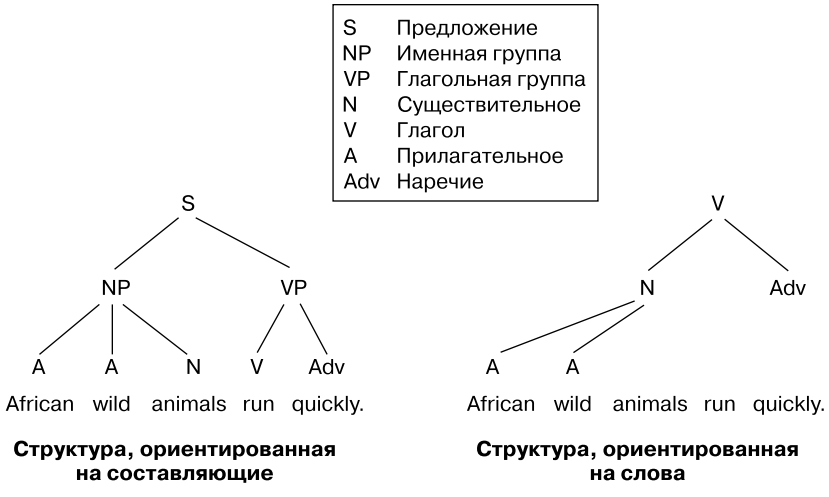


Рис. П.1. Пример древовидных структур для грамматики с фразовой структурой, ориентированной на составляющие *(слева)*, и грамматики зависимостей, ориентированной на слова *(справа)*

Дерево фразовой структуры разбивает предложение, исходя из того, что предложение состоит из именной и глагольной групп. Эти фразы видны на втором уровне иерархии, сразу под меткой предложения (S) — номинальным верхним уровнем. На нижнем уровне располагаются отдельные слова, из которых состоят фразы.

В структуре зависимостей структурным центром предложения является глагол, а прочие слова напрямую или косвенно связаны с ним с помощью направленных связей — *зависимостей* (dependencies). Грамматики зависимостей, используемые в spaCy по умолчанию, выражают грамматическую структуру предложения в виде множества взаимно однозначных соответствий слов друг другу.

Каждое из этих соответствий выражает грамматическую функцию, в которой одно слово играет роль *дочернего элемента* (child), то есть зависимого слова, а второе — главного (head), то есть управляющего, эле-

мента. Например, в паре *blue sky* главным словом является *sky*, а *blue* — его модификатор, то есть подчиненное слово. Главный элемент можно рассматривать как слово с наибольшей относительной «значимостью», без которого дочерний элемент не имеет смысла. И напротив, главный элемент отношения в предложении зачастую имеет смысл сам по себе, без дочерних элементов (например, в предложении, приведенном на рис. П.1, слова *African* и *wild*, а также слово *quickly* необязательны).

На рис. П.2 эта концепция показана наглядно.

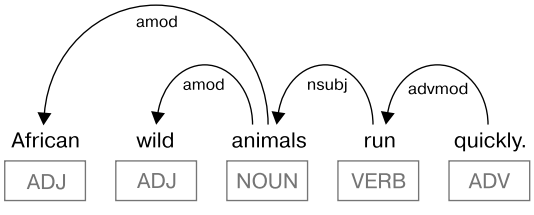


Рис. П.2. Пример структуры дерева зависимостей, основанного на идее главных/дочерних элементов

Обратите внимание, что дерево зависимостей на рис. П.2 аналогично дереву, представленному справа на рис. П.1. Между этими представлениями есть лишь одно внешнее различие: форма дерева на рис. П.1 — пирамидальная, а в дереве связи на рис. П.2 главный элемент/дочерний элемент выделен с помощью маркированных направленных дуг.

Каждое слово в предложении должно быть связано только с одним главным элементом. При этом дочерних элементов у слова может быть как несколько, так и один или ни одного. Грамматика spaCy предполагает, что главным элементом для главного элемента предложения (токена `ROOT`) служит он сам. В данном примере главным элементом предложения является глагол *run*, так что свойство `head` объекта `Token`, соответствующего этому слову, будет ссылаться на него же.

Обратите внимание, что отношение «главный элемент/дочерний элемент» не связано с линейным порядком слов в предложении. Например, дочерний элемент *wild* предшествует своему главному элементу *animals*, а главный элемент *quickly* стоит после его главного элемента *run*.

Общие грамматические понятия

В книге встречаются достаточно сложные понятия, такие как переходные глаголы и прямые дополнения, предложные дополнения, вспомогательные модальные глаголы и личные местоимения.

Переходные глаголы и прямые дополнения

Прямое дополнение (direct object) — это существительное (или именная группа), которое обозначает непосредственный предмет действия глагола. *Переходный глагол* (transitive verb) — глагол, вступающий в отношение с прямым дополнением. В большинстве случаев для распознавания намерения важно выделить из предложения именно прямое дополнение и переходный глагол. Дело в том, что эти слова лучше других описывают действие и объект, над которым это действие совершается. Например, в предложении *I want a pizza* намерение высказывания выражают слова *want* и *pizza*.

Предложные дополнения

Предлоги (prepositions) служат для соединения именных групп с другими словами в предложении. Некоторые из них, например *in*, *above*, *under*, *after* и *before*, выражают пространственные или временны́е отношения. Другие, в частности *to*, *of* и *for*, указывают на семантические роли. Например, в предложении *You'll find the envelope under the book* предлог *under* выражает пространственное отношение между *envelope* и *book*. А в предложении *I will deploy it to a channel* предлог *to* указывает на цель, выраженную предложным оборотом *a channel*.

Предложное дополнение (object of a preposition), называемое в теории лингвистики комплементом (complement), — это существительное, местоимение или именная группа, следующие за предлогом. В предложении *I wrote a series of articles* слово *articles* является предложным дополнением.

В некоторых вопросах, выделив предложное дополнение, можно получить слово или фразу, которые будут наиболее информативны в плане

поиска ответа на этот вопрос. Например, в высказывании *What can be done about climate change?* фраза *climate change* — ключевая для определения того, чему посвящен вопрос.

Средство разбора синтаксических зависимостей библиотеки spaCy отмечает предлоги меткой `'prep'`, а предложные дополнения — меткой `'pobj'`.

Вспомогательные модальные глаголы

В число вспомогательных модальных глаголов (modal auxiliary verbs) входят *may*, *might*, *can*, *could*, *must*, *ought*, *shall*, *should*, *will*, *would* и др. В сочетании со смысловым глаголом в неопределенной форме они указывают на *модальность* (modality), то есть вероятность, способность, необходимость, желание, разрешение или совет сделать что-либо.

Средство частеречной разметки библиотеки spaCy распознает вспомогательные модальные глаголы, помечая их тегом уточненной части речи `'MD'`. Анализатор синтаксических зависимостей помечает их меткой `'aux'`. Проверка предложения на наличие в нем вспомогательного модального глагола может понадобиться, например, при составлении утвердительного высказывания на основе вопроса.

Личные местоимения

Личное местоимение (personal pronoun) обозначает конкретного человека, объект или несколько людей/объектов. В английском языке существует несколько форм личных местоимений, которые различаются грамматической ролью в предложении.

- Форма именительного падежа (*I*, *you*, *he*, *she*, *it*, *we*, *they*) обычно бывает у субстантивированного подлежащего.
- Форма винительного падежа (*me*, *you*, *him*, *her*, *it*, *us*, *them*) — у прямого или предложного дополнения.
- Возвратные местоимения (*myself*, *yourself/yourselves*, *himself*, *herself*, *itself*, *ourselves*, *themselves*) обычно относятся к подлежащему, указанному в пределах того же простого предложения.

Синтаксический анализатор spaCy присваивает личным местоимениям различные метки в зависимости от их формы. Так, личному местоимению в форме именительного падежа присваивается метка зависимости `'nsubj'`, которая означает субстантивированное подлежащее (*n*ominal *subj*ect). Стоит отметить, что роль подлежащего во многих создаваемых пользователями чат-ботах играет личное местоимение *I*.

Личным местоимениям в форме винительного падежа присваивается метка `'dobj'` или `'iobj'` (прямое (*d*irect *obj*ect) и непрямое дополнение (*i*ndirect *obj*ect) соответственно). Возвратные местоимения обычно играют роль прямых или непрямых дополнений.

Юлий Васильев

Обработка естественного языка. Python и spaCy на практике

Перевел с английского И. Пальти

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*М. Куклис*
Художественный редактор	*В. Мостипан*
Корректоры	*Н. Кудрейко, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214,
тел./факс: 208 80 01.

Подписано в печать 31.03.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 500. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт».
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. I, ком. 6.3-23Н.

Дэвид Фостер

ГЕНЕРАТИВНОЕ ГЛУБОКОЕ ОБУЧЕНИЕ. ТВОРЧЕСКИЙ ПОТЕНЦИАЛ НЕЙРОННЫХ СЕТЕЙ

Генеративное моделирование — одна из самых обсуждаемых тем в области искусственного интеллекта. Машины можно научить рисовать, писать и сочинять музыку. Вы сами можете посадить искусственный интеллект за парту или мольберт, для этого достаточно познакомиться с самыми актуальными примерами генеративных моделей глубокого обучения: вариационными автокодировщиками, генеративно-состязательными сетями, моделями типа кодер-декодер и многими другими.

Дэвид Фостер делает понятными и доступными архитектуру и методы генеративного моделирования, его советы и подсказки сделают ваши модели более творческими и эффективными в обучении. Вы начнете с основ глубокого обучения на базе Keras, а затем перейдете к самым передовым алгоритмам.