

ВСТРАИВАНИЕ МОДЕЛИ МАШИННОГО ОБУЧЕНИЯ В ВЕБ-ПРИЛОЖЕНИЕ

В предшествующих главах вы ознакомились со многими концепциями и алгоритмами МО, которые могут содействовать принятию лучших и более рациональных решений. Однако приемы МО не ограничиваются автономными приложениями и анализом; они могут выступать в качестве прогнозирующих механизмов веб-служб. Например, популярные и удобные употребления моделей МО в веб-приложениях включают обнаружение спама в отправляемых формах, поисковые механизмы, системы выдачи рекомендаций для медийных и продающих порталов, а также многие другие.

В этой главе вы узнаете, как встраивать модель МО в веб-приложение, которое может не только классифицировать, но и учиться на данных в реальном времени.

В главе будут раскрыты следующие темы:

- сохранение текущего состояния обученной модели МО;
- использование баз данных SQLite для хранилищ данных;
- разработка веб-приложения с применением популярной веб-инфраструктуры Flask;
- развертывание приложения МО на публичном веб-сервере.

Сериализация подогнанных оценщиков `scikit-learn`

Как было показано в главе 8, обучение модели МО может оказаться довольно затратным с точки зрения вычислений. Ведь не хотим же мы обучать модель заново каждый раз, когда закрыли интерпретатор Python и желаем выработать прогноз или перезагрузить веб-приложение? Одним из вариантов обеспечения постоянства моделей является модуль `pickle` для Python (<https://docs.python.org/3.6/library/pickle.html>). Он делает возможными сериализацию и десериализацию структур объектов Python с целью сжатия байт-кода, чтобы мы могли сохранить классификатор в его текущем состоянии и опять загрузить его, когда нужно классифицировать новые образцы, не заставляя модель учиться на обучающих данных еще раз. Прежде чем выполнять следующий код, удостоверьтесь в том, что обучили внешнюю логистическую регрессионную модель из последнего раздела главы 8 и обеспечили ее готовность в текущем сеансе Python:

```
>>> import pickle
>>> import os
>>> dest = os.path.join('movieclassifier', 'pkl_objects')
>>> if not os.path.exists(dest):
...     os.makedirs(dest)
>>> pickle.dump(stop,
...             open(os.path.join(dest, 'stopwords.pkl'), 'wb'),
...             protocol=4)
>>> pickle.dump(clf,
...             open(os.path.join(dest, 'classifier.pkl'), 'wb'),
...             protocol=4)
```

В приведенном выше коде мы создаем каталог `movieclassifier`, где позже будем хранить файлы и данные для нашего веб-приложения. Внутри каталога `movieclassifier` мы создаем подкаталог `pkl_objects`, чтобы сохранять на локальном диске сериализованные объекты Python. С помощью метода `dump` модуля `pickle` мы затем сериализуем обученную логистическую регрессионную модель, а также набор стоп-слов из библиотеки NLTK, чтобы устранить необходимость в установке глоссария NLTK на сервере.

Метод `dump` принимает в своем первом аргументе объект, который мы хотим законсервировать, а во втором аргументе – открытый файловый объект, куда будет записываться объект Python.

Посредством аргумента `wb` внутри функции `open` мы открываем файл в двоичном режиме для консервации и устанавливаем `protocol=4`, чтобы выбрать самый последний и эффективный протокол консервации, который появился в Python 3.4 и совместим с последующими версиями Python. В случае возникновения проблем с использованием `protocol=4` проверьте, работаете ли вы с самой последней версией Python 3. Или же можете обдумать вопрос применения протокола с меньшим номером.



Наша логистическая регрессионная модель содержит несколько массивов NumPy, таких как весовой вектор, а более рациональный способ сериализации массивов NumPy предусматривает использование альтернативной библиотеки `joblib`. Чтобы гарантировать совместимость с серверной средой, которая будет применяться позже в главе, мы воспользуемся стандартным подходом к консервации. Дополнительные сведения о библиотеке `joblib` доступны по ссылке <https://joblib.readthedocs.io/en/latest/>.

Консервировать объект `HashingVectorizer` нет никакой необходимости, потому что он не нуждается в подгонке. Взамен мы можем создать файл сценария Python, из которого импортировать векторизатор в текущий сеанс Python. Скопируем следующий код и сохраним его в файле `vectorizer.py` внутри каталога `movieclassifier`:

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
    os.path.join(cur_dir,
        'pkl_objects',
        'stopwords.pkl'), 'rb'))

def tokenizer(text):
    text = re.sub('<[^\>]*>', '', text)
    emoticons = re.findall('(?:::|;|=) (?:-)?(?:\)|\(|D|P)',
        text.lower())
```

```

text = re.sub('[\W]+', ' ', text.lower()) \
        + ' '.join(emoticons).replace('-', '')
tokenized = [w for w in text.split() if w not in stop]
return tokenized

vect = HashingVectorizer(decode_error='ignore',
                        n_features=2**21,
                        preprocessor=None,
                        tokenizer=tokenizer)

```

После консервации объектов Python и создания файла `vectorizer.py` имеет смысл перезапустить интерпретатор Python или ядро IPython для Jupyter Notebook, чтобы проверить, можем ли мы безошибочно десериализовать объекты.



На заметку!

Тем не менее, имейте в виду, что расконсервирование из не заслуживающего доверия источника может быть сопряжено с риском в плане безопасности, т.к. модуль `pickle` не защищен против злонамеренного кода. Поскольку `pickle` проектировался для сериализации произвольных объектов, процесс расконсервирования будет выполнять код, который сохранился в файле консервации. Таким образом, если вы получаете файлы консерваций из не заслуживающего доверия источника (скажем, загружая их из Интернета), тогда проявите дополнительную осторожность и расконсервируйте элементы в виртуальной среде и/или на второстепенной машине, не хранящей важные данные, к которым никто кроме вас не должен иметь доступ.

В терминальном окне перейдем в каталог `movieclassifier`, запустим новый сеанс Python и выполним приведенный ниже код для проверки, можем ли мы импортировать `vectorizer` и расконсервировать классификатор:

```

>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(
...     os.path.join('pkl_objects',
...                   'classifier.pkl'), 'rb'))

```

После успешной загрузки `vectorizer` и расконсервирования классификатора мы можем использовать эти объекты для предварительной обработки образцов документов и выработки прогнозов об их отношениях:

```
>>> import numpy as np
>>> label = {0: 'негативный', 1: 'позитивный'}
>>> example = ['I love this movie']
>>> X = vect.transform(example)
>>> print('Прогноз: %s\nВероятность: %.2f%%' % \
...       (label[clf.predict(X)[0]],
...       np.max(clf.predict_proba(X))*100))
Прогноз: позитивный
Вероятность: 91.56%
```

Поскольку наш классификатор возвращает метки классов как целые числа, мы определили простой словарь Python для отображения таких целых чисел на их отношения. Затем мы применили `HashingVectorizer` для трансформации простого примера документа в вектор слов `X`. Наконец, мы использовали метод `predict` классификатора на основе логистической регрессии для прогнозирования метки класса, а также метод `predict_proba` для возвращения соответствующей вероятности прогноза. Обратите внимание, что в результате вызова метода `predict_proba` возвращается массив со значениями вероятностей для всех уникальных методов классов. Так как метка класса с наибольшей вероятностью соответствует метке класса, возвращаемой методом `predict`, для возвращения вероятности спрогнозированного класса мы применили функцию `np.max`.

Настройка базы данных SQLite для хранилища данных

В текущем разделе мы настроим простую базу данных SQLite для сбора дополнительных отзывов о прогнозах от пользователей веб-приложения. Мы можем использовать такую обратную связь для обновления классификационной модели. SQLite – это механизм баз данных SQL с открытым кодом, не требующий для своей работы отдельного сервера, что делает его идеальным вариантом при разработке небольших проектов и простых веб-приложений. По существу базу данных SQLite можно считать одиночным самодостаточным файлом базы данных, который позволяет напрямую обращаться к хранилищу.

Кроме того, SQLite не требует конфигурирования, специфичного для системы, и поддерживается во всех распространенных операционных системах. Механизм SQLite известен своей высокой надежностью и применяется популярными компаниями, среди которых Google, Mozilla, Adobe, Apple, Microsoft и многие другие. Дополнительная информация о механизме SQLite доступна на официальном веб-сайте <http://www.sqlite.org>.

К счастью, согласно принятой в Python философии “батарейки в комплекте” стандартная библиотека Python предлагает API-интерфейс `sqlite3`, который позволяет работать с базами данных SQLite (больше сведений о модуле `sqlite3` можно получить по ссылке <https://docs.python.org/3.6/library/sqlite3.html>).

Выполнив следующий код, мы создадим внутри каталога `movieclassifier` новую базу данных SQLite и сохраним в ней два примера рецензий на фильмы:

```
>>> import sqlite3
>>> import os

>>> if os.path.exists('reviews.sqlite'):
...     os.remove('reviews.sqlite')
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('CREATE TABLE review_db\'
...           ' (review TEXT, sentiment INTEGER, date TEXT)')
>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO review_db\'
...           ' (review, sentiment, date) VALUES\'
...           ' (?, ?, DATETIME('now'))", (example1, 1))
>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO review_db\'
...           ' (review, sentiment, date) VALUES\'
...           ' (?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()
```

В коде мы создаем подключение (`conn`) к файлу базы данных SQLite, вызывая метод `connect` из библиотеки `sqlite3`, который создает в каталоге `movieclassifier` файл базы данных `reviews.sqlite`, если он пока не существует. Обратите внимание, что в SQLite не реализована функция замены для существующих таблиц; если нужно выполнить код еще раз, тогда придется вручную удалить файл базы данных с помощью проводника файлов.

Затем посредством метода `cursor` мы создаем курсор, который даст возможность проходить по записям базы данных, используя универсальный синтаксис SQL. Далее с применением первого вызова `execute` мы создаем новую таблицу базы данных `review_db`, которую будем использовать для сохранения и последующего доступа к записям. В таблице `review_db` мы создаем три столбца: `review`, `sentiment` и `date`. Они будут применяться для хранения примеров рецензий на фильмы и соответствующих меток классов (отношений).

С использованием SQL-команды `DATETIME('now')` мы добавляем к записям дату и отметку времени. Знаки вопроса (?) применяются для передачи методу `execute` текстов рецензий на фильмы (`example1` и `example2`) и связанных с ними меток классов (1 и 0) в виде позиционных аргументов как членов кортежа. В заключение мы вызываем метод `commit` для сохранения изменений, внесенных в базу данных, и закрываем подключение посредством метода `close`.

Чтобы проверить, корректно ли сохранились записи в базе данных, мы снова откроем подключение к базе данных и с помощью SQL-команды `SELECT` извлечем из таблицы все строки, зафиксированные в период между началом 2017 года и сегодняшним днем:

```
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM review_db WHERE date\"
...         \" BETWEEN '2017-01-01 00:00:00' AND DATETIME('now')")
>>> results = c.fetchall()

>>> conn.close()
>>> print(results)
[('I love this movie', 1, '2017-04-24 00:14:38'),
 ('I disliked this movie', 0, '2017-04-24 00:14:38')]
```

В качестве альтернативы мы могли бы также воспользоваться бесплатным подключаемым модулем Firefox под названием SQLite Manager (Диспетчер SQLite), доступным по ссылке <https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/>, который предлагает удобный графический пользовательский интерфейс для работы с базами данных SQLite (рис. 9.1).

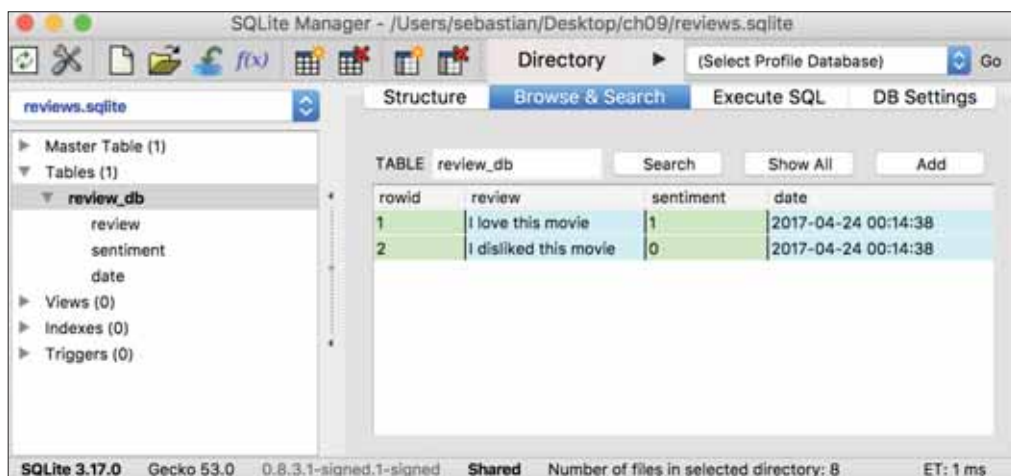


Рис. 9.1. Графический пользовательский интерфейс SQLite Manager

Разработка веб-приложения с помощью Flask

Теперь, когда код для классификации рецензий на фильмы готов, давайте обсудим основы веб-инфраструктуры Flask, необходимые для разработки нашего веб-приложения. После выпуска Армином Ронахером первоначальной версии Flask в 2010 году эта инфраструктура с годами обрела огромную популярность и применяется в таких известных приложениях, как LinkedIn и Pinterest. Поскольку инфраструктура Flask написана на языке Python, она снабжает программистов на Python удобным интерфейсом для встраивания существующего кода Python, подобного классификатору рецензий на фильмы.



На заметку!

Flask также называют *микроинфраструктурой*, имея в виду тот факт, что ее ядро сохранено небольшим и простым, но может легко расширяться с использованием других библиотек. Хотя кривая обучения легковесного API-интерфейса Flask не настолько крута, как у других популярных веб-инфраструктур вроде Django, рекомендуется заглянуть в официальную документацию по ссылке <http://flask.pocoo.org/docs/0.12/>, чтобы получить больше сведений о ее функциональности.

Если библиотека Flask в текущей среде Python отсутствует, тогда ее можно установить с помощью команды `conda` или `pip` в терминальном окне (на момент написания главы последним стабильным выпуском был 1.0.2):

```
conda install flask
# или pip install flask
```

Первое веб-приложение Flask

В настоящем подразделе мы разработаем очень простое веб-приложение с целью ознакомления с API-интерфейсом Flask перед тем, как заняться реализацией классификатора рецензий на фильмы. Первое приложение будет состоять из простой веб-страницы с полем формы, которое позволит вводить имя. После отправки формы веб-приложению визуализируется новая страница. Несмотря на свою простоту, этот пример веб-приложения поможет получить представление о том, как переменные сохраняются и передаются между разными частями кода внутри инфраструктуры Flask.

Прежде всего, мы создаем дерево каталогов:

```
1st_flask_app_1/
  app.py
  templates/
    first_app.html
```

Файл `app.py` содержит основной код, который будет выполняться интерпретатором Python для запуска веб-приложения Flask. В каталоге `templates` инфраструктура Flask будет искать статические HTML-файлы для визуализации в веб-браузере. Теперь взглянем на содержимое файла `app.py`:

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def index():
    return render_template('first_app.html')
if __name__ == '__main__':
    app.run()
```

Давайте обсудим индивидуальные части кода по очереди.

1. Мы запускаем приложение как одиночный модуль. Соответственно, мы инициализируем новый экземпляр `Flask` с аргументом `__name__`, чтобы инфраструктуре Flask было известно, что подкаталог HTML-шаблонов (`templates`) она может отыскать в том же каталоге, где находится приложение.
2. Затем мы применяем декоратор маршрута (`@app.route('/')`), чтобы указать URL, который должен инициировать выполнение функции `index`.
3. Далее функция `index` просто визуализирует HTML-файл `first_app.html`, находящийся в подкаталоге `templates`.
4. Наконец, мы используем функцию `run` для запуска приложения на сервере, когда этот сценарий непосредственно выполняется интерпретатором Python, что гарантируется оператором `if` с условием `__name__ == '__main__'`.

Рассмотрим содержимое файла `first_app.html`:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
    <div>Hi, this is my first Flask web app!</div>
  </body>
</html>
```



На
заметку!

На тот случай, если синтаксис HTML вам не знаком, по ссылке <https://developer.mozilla.org/ru/docs/Web/HTML> доступно удобное руководство по основам HTML.

Здесь мы просто заполняем пустой файл HTML-шаблона с помощью элемента `<div>` (элемент блочного уровня), который содержит предложение: `Hi, this is my first Flask web app!`.

Инфраструктура Flask позволяет запускать приложения локально, что удобно для разработки и тестирования веб-приложений перед их развертыванием на публичном веб-сервере. Давайте запустим наше веб-приложение,

выполнив в терминальном окне следующую команду из текущего каталога `1st_flask_app_1`:

```
python3 app.py
```

Мы должны увидеть в терминальном окне строку такого вида:

```
* Running on http://127.0.0.1:5000/  
* Выполняется на http://127.0.0.1:5000/
```

Строка содержит адрес локального сервера. Мы можем ввести этот адрес в адресной строке веб-браузера, чтобы посмотреть на веб-приложение в действии. Если все было сделано корректно, тогда отобразится простой веб-сайт с содержимым `Hi, this is my first Flask web app!` (рис. 9.2).



Рис. 9.2. Пример веб-приложения *Flask* в действии

Проверка достоверности и визуализация форм

В этом подразделе мы расширим наше простое веб-приложение *Flask* элементами HTML-формы, чтобы научиться собирать данные от пользователя с применением библиотеки *WTForms* (<https://wtforms.readthedocs.org/en/latest/>), которую можно установить через `conda` или `pip`:

```
conda install wtforms  
# или pip install wtforms
```

Веб-приложение будет предлагать пользователю ввести свое имя в текстовом поле (рис. 9.3).

После щелчка на кнопке отправки (`Say Hello` (Поприветствовать)) и проверки достоверности формы будет визуализирована новая HTML-страница для отображения имени пользователя (рис. 9.4).



Рис. 9.3. Запрос имени у пользователя



Рис. 9.4. Отображение имени пользователя

Настройка структуры каталогов

Новая структура каталогов, которую нам нужно настроить для данного приложения, выглядит следующим образом:

```
1st_flask_app_2/  
  app.py  
  static/  
    style.css  
  templates/  
    _formhelpers.html  
    first_app.html  
    hello.html
```

Ниже показано модифицированное содержимое `app.py`:

```
from flask import Flask, render_template, request  
from wtforms import Form, TextAreaField, validators  
app = Flask(__name__)  
class HelloForm(Form):  
    sayhello = TextAreaField('', [validators.DataRequired()])
```

```

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)

```

Давайте проанализируем приведенный код шаг за шагом.

1. Используя `wtforms`, мы расширяем функцию `index` текстовым полем, которое будет встраиваться в стартовую страницу с применением класса `TextAreaField`, автоматически проверяющего наличие допустимого ввода от пользователя.
2. Кроме того, мы определяем новую функцию `hello`, которая будет визуализировать HTML-страницу `hello.html` после проверки достоверности HTML-формы.
3. Далее мы используем метод `POST` для передачи данных формы серверу в теле сообщения. В заключение установкой аргумента `debug=True` внутри метода `app.run` мы дополнительно активизируем отладчик Flask, который является полезным средством при разработке новых веб-приложений.

Реализация макроса с использованием механизма шаблонизации Jinja2

Теперь с применением механизма шаблонизации Jinja2 мы реализуем обобщенный макрос в файле `_formhelpers.html`, который позже импортируем в файле `first_app.html`, чтобы визуализировать текстовое поле:

```

{% macro render_field(field) %}
<dt>{{ field.label }}
<dd>{{ field(**kwargs)|safe }}
{% if field.errors %}
<ul class=errors>
{% for error in field.errors %}
<li>{{ error }}</li>

```

```
{% endfor %}
</ul>
{% endif %}
</dd>
</dt>
{% endmacro %}
```

Подробное обсуждение языка шаблонизации Jinja2 выходит за рамки этой книги. Однако на веб-сайте <http://jinja.pocoo.org> можно найти всеобъемлющую документацию по синтаксису Jinja2.

Добавление стиля CSS

Далее мы создадим простой CSS-файл (Cascading Style Sheet – каскадная таблица стилей), `style.css`, для демонстрации того, как можно изменять внешний вид и восприятие HTML-документов. Мы должны сохранить CSS-файл со следующим содержимым, которое просто удваивает размер шрифта HTML-элементов `body`, в подкаталоге по имени `static` – стандартном месте, где инфраструктура Flask ищет статические файлы наподобие CSS:

```
body {
    font-size: 2em;
}
```

Вот модифицированное содержимое файла `first_app.html`, которое теперь визуализирует текстовое поле для ввода пользователем своего имени:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
    <link rel="stylesheet" href="{{ url_for('static',
      filename='style.css') }}">
  </head>
  <body>
    {% from "_formhelpers.html" import render_field %}
    <div>What's your name?</div>
    <form method=post action="/hello">
      <dl>
        {{ render_field(form.sayhello) }}
      </dl>
      <input type=submit value='Say Hello' name='submit_btn'>
    </form>
  </body>
</html>
```

Мы загружаем CSS-файл в разделе заголовка HTML-разметки `first_app.html`. Теперь размер всех текстовых элементов внутри HTML-элемента `body` должен измениться. В разделе тела HTML мы импортируем макрос формы из `_formhelpers.html` и визуализируем форму `sayhello`, которую указали в файле `app.py`. Кроме того, мы добавили кнопку к тому же самому элементу `form`, чтобы пользователь мог отправлять запись из текстового поля.

Создание результирующей страницы

Наконец, мы создадим файл `hello.html`, который будет визуализироваться посредством строки `return render_template('hello.html', name=name)` внутри функции `hello`, определенной в сценарии `app.py`, с целью отображения отправленного пользователем текста. Вот содержимое файла `hello.html`:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
    <link rel="stylesheet" href="{{ url_for('static',
      filename='style.css') }}">
  </head>
  <body>
    <div>Hello {{ name }}</div>
  </body>
</html>
```

После модификации веб-приложения Flask мы можем запустить его локально, выполнив следующую команду из главного каталога приложения, и посмотреть результат в веб-браузере по адресу `http://127.0.0.1:5000/`:

```
python3 app.py
```



На заметку!

Если вы – новичок в области разработки веб-приложений, то на первый взгляд некоторые концепции могут выглядеть очень сложными. В таком случае рекомендуется поместить предшествующие файлы в каталог на своем жестком диске и внимательно их исследовать. Вы увидите, что веб-инфраструктура Flask относительно прямолинейна и гораздо проще, чем может показаться поначалу. Вдобавок не забывайте обращаться к великолепной документации и примерам Flask по ссылке <http://flask.pocoo.org/docs/0.12/>.

Превращение классификатора рецензий на фильмы в веб-приложение

После ознакомления с основами разработки веб-приложений с помощью Flask давайте продвинемся на шаг вперед и реализуем наш классификатор рецензий на фильмы в виде веб-приложения. В текущем разделе мы разработаем веб-приложение, которое сначала предложит пользователю ввести рецензию на фильм (рис. 9.5).



Рис. 9.5. Запрос рецензии на фильм у пользователя

После отправки рецензии пользователь увидит новую страницу, которая отобразит спрогнозированную метку класса и вероятность прогноза. К тому же пользователь получит возможность оставить отзыв о выработанном прогнозе, щелкая на кнопке **Correct** (Правильный) или **Incorrect** (Неправильный), как показано на рис. 9.6.

Если пользователь щелкнет на одной из кнопок **Correct** или **Incorrect**, тогда наша классификационная модель будет обновлена согласно пользовательскому отзыву. Кроме того, мы также сохраним введенный пользователем текст рецензии на фильм и предполагаемую метку класса, которую можно вывести из щелчка на кнопке, в базе данных SQLite для ссылки в будущем. (В качестве альтернативы пользователь мог бы пропустить шаг обновления и щелкнуть на кнопке **Submit another review** (Отправить еще одну рецензию), чтобы отправить еще одну рецензию.)

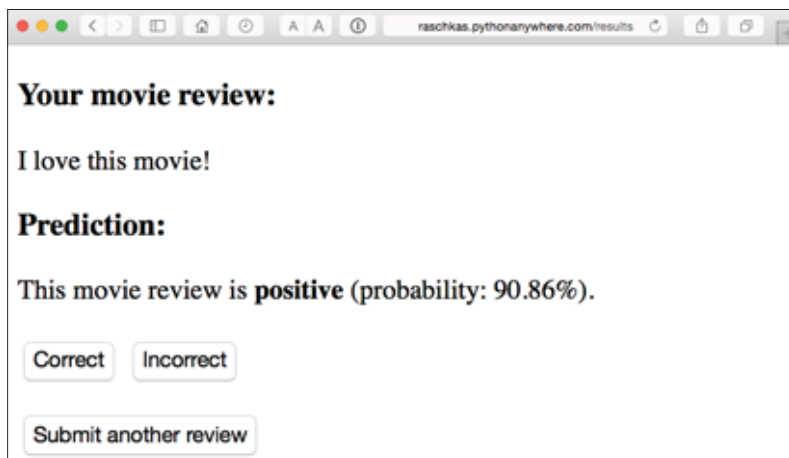


Рис. 9.6. Отображение выработанного прогноза и возможность отправки отзыва о нем

Третьей страницей, которую пользователь увидит после щелчка на одной из кнопок обратной связи, будет простой экран благодарности с кнопкой **Submit another review**, щелчок на которой перенаправляет пользователя обратно на стартовую страницу (рис. 9.7).

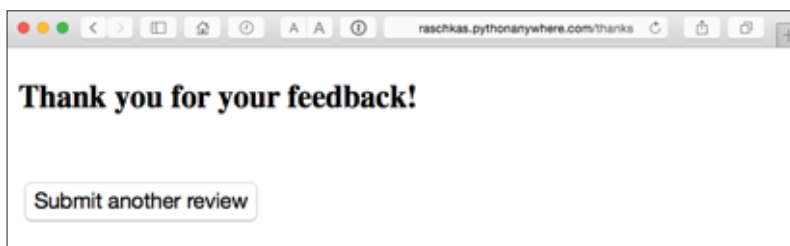


Рис. 9.7. Экран благодарности



На заметку!

Прежде чем мы более пристально взглянем на код реализации рассматриваемого веб-приложения, рекомендуется поработать с живой демонстрацией, доступной по ссылке <http://raschkas.pythonanywhere.com>, чтобы лучше понимать, чего мы пытаемся достигнуть в данном разделе.

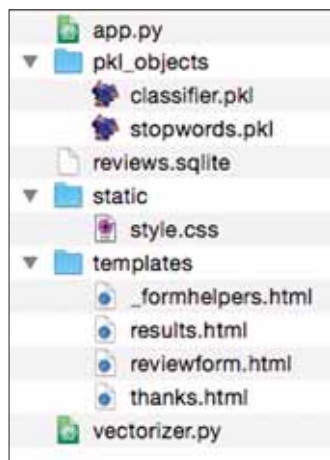


Рис. 9.8. *Дерево каталогов для приложения классификации рецензий на фильмы*

Файлы и подкаталоги – дерево каталогов

Чтобы начать с общей картины, давайте посмотрим на дерево каталогов, которое мы собираемся создать для приложения классификации рецензий на фильмы (рис. 9.8).

В предыдущем разделе главы мы уже создали файл `vectorizer.py`, базу данных SQLite в файле `reviews.sqlite` и подкаталог `pkl_objects` с законсервированными объектами Python.

Файл `app.py` в главном каталоге – это сценарий Python, который содержит код Flask, а файл базы данных `review.sqlite` (созданный ранее в главе) будет использоваться для хранения рецензий на фильмы, отправленные нашему веб-приложению. В подкаталоге `templates` находятся HTML-шаблоны, которые будут визуализироваться инфраструктурой Flask и отображаться в веб-браузере, а в подкаталоге `static` – простой CSS-файл, предназначенный для подстройки внешнего вида визуализируемой HTML-разметки.



На заметку!

Отдельный каталог, который содержит приложение классификатора рецензий на фильмы с кодом, обсуждаемым в настоящем разделе, входит в состав архива примеров кода для книги. Архив доступен для загрузки на веб-сайте издательства и на GitHub по ссылке <https://github.com/rasbt/python-machine-learning-book-2nd-edition/>. Код, относящийся к текущему разделу, находится в подкаталоге `PythonMachineLearningSecondEdition_Code/code/Chapter09/movieclassifier`.

Реализация главного приложения как `app.py`

Поскольку содержимое файла `app.py` довольно длинное, мы разберем его в два этапа. Первая часть `app.py` импортирует модули и объекты Python, которые нам понадобятся, а также код для расконсервирования и настройки классификационной модели:

```

from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np

# импортировать HashingVectorizer из локального каталога
from vectorizer import vect

app = Flask(__name__)

##### Подготовка классификатора
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                   'pkl_objects',
                                   'classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'негативный', 1: 'позитивный'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = np.max(clf.predict_proba(X))
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"
              " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()

```

К этому времени первая часть сценария `app.py` должна выглядеть хорошо знакомой. Мы просто импортируем `HashingVectorizer` и расконсервируем классификатор на основе логистической регрессии. Затем мы определяем функцию `classify` для возвращения спрогнозированной метки класса и вероятности прогноза, относящиеся к заданному текстовому документу. Функция `train` может применяться для обновления классификатора при условии, что предоставлены документ и метка класса.

С использованием функции `sqlite_entry` мы можем сохранять в базе данных отправленную рецензию на фильм вместе с ее меткой класса и от-

меткой времени в регистрационных целях. Обратите внимание, что при перезапуске веб-приложения объект `clf` будет сбрасываться в исходное законсервированное состояние. В конце главы вы узнаете, как применять накопленную в базе данных SQLite информацию для постоянного обновления классификатора.

Концепции во второй части сценария `app.py` также должны выглядеть довольно знакомыми:

```
##### Flask
class ReviewForm(Form):
    moviereview = TextAreaField('',
                                [validators.DataRequired(),
                                 validators.length(min=15)])

@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():
    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                               content=review,
                               prediction=y,
                               probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']
    inv_label = {'negative': 0, 'positive': 1}
    y = inv_label[prediction]
    if feedback == 'Incorrect':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(debug=True)
```