Содержание

9

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 стали прогнозирующими механизмами веб-служб. Например, популярные и удобные случаи 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 способно 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 популярного веб-фреймворка 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/3.7/library/pickle.html`). Он делает возможными сериалиацию и десериалиацию структур объектов Python с целью сжатия байт-кода, чтобы мы могли сохранить классификатор в текущем состоянии и опять загрузить его, когда нужно классифицировать новые непомеченные 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 жестком диске или твердотельном накопителе сериализированные объекты Python. С помощью метода `dump` модуля `pickle` мы затем сериализируем 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. Во втором аргументе предоставляется открытый

файловый объект, куда будет записываться объект Python. Посредством аргумента `wb` внутри функции `open` мы открываем файл в двоичном режиме для консервирования и устанавливаем `protocol=4`, чтобы выбрать самый последний и эффективный протокол консервирования, 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 — в этой книге рекомендуется Python 3.7. Или же можете обдумать вопрос применения протокола с меньшим номером.

Также имейте в виду, что если вы имеете дело со специальным веб-сервером, то должны также проверить, совместима ли установленная на нем копия Python с версией протокола.

Сериализация массивов NumPy с помощью библиотеки joblib

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 обеспечить совместимость с серверной средой, которая будет применяться позже в главе, мы воспользуемся стандартным подходом к консервированию. Дополнительные сведения о библиотеке `joblib` доступны по ссылке `https://joblib.readthedocs.io`.

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 консервирования объектов Python и создания файла `vectorizer.py` имеет смысл перезапустить интерпретатор Python или ядро Jupyter Notebook, чтобы проверить, можем ли мы безошибочно десериализировать объекты.

Консервирование может быть сопряжено с риском в плане безопасности

Тем не менее, имейте в виду, что расконсервирование данных 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 выполнит код, который был сохранен в консервированном файле. Таким образом, если вы получаете консервированные файлы из не заслуживающего доверия источника (скажем, загружая их из Интернета), тогда предусмотрите дополнительные меры предосторожности — расконсервируйте элементы в виртуальной среде и/или на второстепенной машине, где отсутствуют 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. It's amazing."]
>>> X = vect.transform(example)
>>> print('Прогноз: %s\nВероятность: %.2f%%' %\
...       (label[clf.predict(X)[0]],
...        np.max(clf.predict_proba(X))*100))
Прогноз: позитивный
Вероятность: 95.55%
```

Поскольку наш классификатор возвращает спрогнозированные метки классов как целые числа, мы определили простой словарь Python для отображения таких целых чисел на их отношения (`'негативный'` или `'позитивный'`). Хотя в итоге получилось простое приложение только с двумя классами, необходимо отметить, что подход с отображением посредством словаря также обобщается на многоклассовые конфигурации. Кроме того, такой словарь отображения должен архивироваться вместе с моделью.

В этом случае из-за того, что определение словаря включает лишь одну строку кода, мы не будем прилагать усилия по его сериализации с применением `pickle`. Однако в реальных приложениях с более обширными словарями отображения вы можете задействовать те же самые команды `pickle.dump` и `pickle.load`, которые использовались в предыдущем примере кода.

Продолжая обсуждение предыдущего примера кода, мы затем 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 метод 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/3.7/library/sqlite3.html`.)

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

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

>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('DROP TABLE IF EXISTS review_db')
>>> 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.

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 текущей датой:

```
>>> 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, '2020-04-15 17:53:46'), ('I disliked
this movie', 0, '2020-04-15 17:53:46')]
```

В качестве альтернативы мы могли бы также воспользоваться бесплатным приложением DB Browser for SQLite (Программа просмотра баз данных SQLite), доступным по ссылке https://sqlitebrowser.org/dl/; оно предлагает удобный графический пользовательский интерфейс для работы с базами данных SQLite (рис. 9.1).

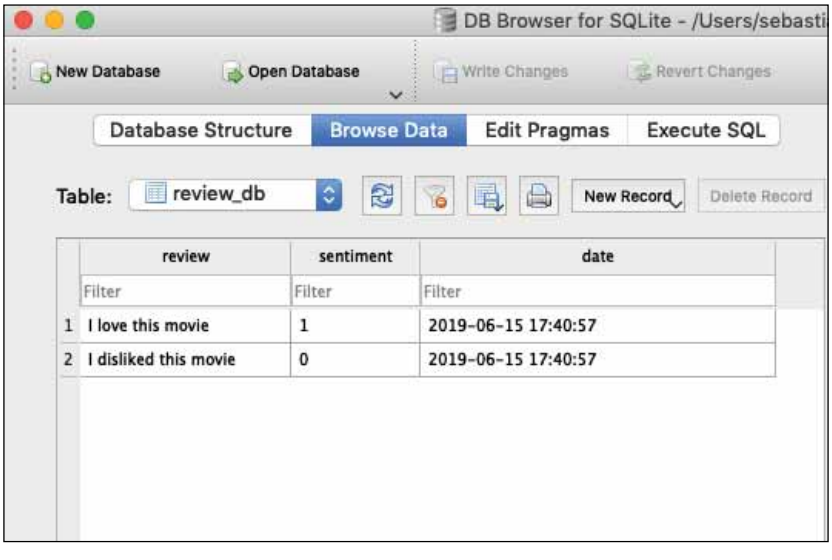

Рис. 9.1. *Приложение DB Browser for SQLite*

Разработка веб-приложения с помощью Flask

Теперь, когда код для классификации рецензий на фильмы готов, давайте обсудим основы веб-фреймворка Flask, необходимого для разработки нашего веб-приложения. После выпуска Армином Ронахером первоначальной версии Flask в 2010 году этот фреймворк с годами обрел огромную популярность и применяется в таких известных приложениях, как LinkedIn и Pinterest. Поскольку фреймворк Flask написан на языке Python, он снабжает программистов на Python удобным интерфейсом для встраивания существующего кода Python, подобного классификатору рецензий на фильмы.