UNIT-III

Web Scraping Using Python

Advanced Scraping: Reading Documents

Advanced scraping often involves extracting data from documents that aren't traditional HTML web pages. The key challenge lies in handling different file formats and their encodings.

Document Encoding

Document encoding is the system used to represent characters in a file. The most common standard today is UTF-8, which can represent almost any character from any language. Other encodings like ASCII are more limited. When reading a document, it's crucial to specify the correct encoding to avoid errors and garbled text. You can often detect the encoding automatically, but sometimes manual specification is necessary.

Text (.txt)

Plain text files are the simplest to read. You can open and read them line by line using Python's built-in file handling functions. You need to specify the correct encoding, usually UTF-8.

Example

```
with open('document.txt', 'r', encoding='utf-8') as f:
    content = f.read()
    print(content)
```

CSV (Comma-Separated Values)

CSV files store tabular data where values are separated by a delimiter, most commonly a comma. Python's built-in csv module is the standard tool for reading these files. It handles different delimiters and properly manages quoted fields, preventing parsing errors.

Example

```
import csv
with open('data.csv', 'r', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

PDF (Portable Document Format)

PDFs are a challenging format for scraping because they can contain text, images, and other objects in a non-linear way. To extract text, you need a specialized library like PyPDF2 or pdfplumber.

- PyPDF2 is useful for splitting, merging, and extracting text from PDFs. It works well with standard text-based PDFs.
- pdfplumber is more advanced and better for extracting text and tabular data from PDFs that have a complex or structured layout.

Microsoft Word (.docx)

The .docx format is an XML-based document format. You can't simply read it as a text file. The python-docx library is a powerful tool for this purpose. It allows you to read paragraphs, tables, and other elements within a Word document.

Example

```
from docx import Document

document = Document('report.docx')

for paragraph in document.paragraphs:

    print(paragraph.text)
```

Reading and Writing Natural Languages

Reading and writing natural languages in a programmatic way is a core part of **Natural Language Processing (NLP)**. It involves teaching computers to understand, interpret, and generate human language.

Summarizing Data

Summarizing data, specifically text, is a key NLP task. The goal is to create a shorter version of a text that retains its most important information. There are two main approaches:

- **Extractive Summarization:** This method works by identifying and extracting the most important sentences or phrases from the original text and stitching them together to form the summary. It doesn't generate new words or sentences. A common technique is to score sentences based on factors like word frequency, location (e.g., sentences at the beginning are often more important), and presence of keywords.
- **Abstractive Summarization:** This is a more advanced technique that involves generating new sentences and phrases to create a more fluent and concise summary. It's similar to how a human summarizes text. This method often uses deep learning models, like sequence-to-sequence models, that learn to understand the text's meaning and then generate a new, shorter version.

Markov Models

A **Markov Model** is a statistical model used to describe a sequence of events where the probability of each event depends only on the state of the previous event. This property is known as the **Markov property**.

In NLP, a **Markov Chain** is often used to model language. A simple Markov chain for text generation might look at the probability of a word appearing given the preceding word. For example, if the previous word is "the," the model might predict that the next word is more likely to be "cat," "dog," or "house" than "run."

- **States:** The states in the model are the words themselves.
- **Transitions:** The transitions are the probabilities of moving from one word to the next.
- **N-gram Models:** A more advanced version, the **N-gram model**, considers a sequence of N words. A bigram model (N=2) considers the previous word, while a trigram model (N=3) considers the two previous words. By analyzing a large corpus of text, a Markov model can learn these transition probabilities and generate new, coherent-sounding text.

Natural Language Toolkit (NLTK)

NLTK is a powerful Python library that provides a comprehensive set of tools for working with human language data. It's widely used in academia and industry for teaching and research.

Key features of NLTK include:

- **Tokenization:** Breaking down a text into smaller units, such as words or sentences.
- **Stemming and Lemmatization:** Reducing words to their root or base form (e.g., "running," "runs," and "ran" all become "run"). This is crucial for reducing vocabulary size and improving model performance.
- **Part-of-Speech Tagging:** Identifying the grammatical role of each word in a sentence (e.g., noun, verb, adjective).
- **Corpora:** A vast collection of text data, like the Brown Corpus or the Gutenberg Corpus, which are essential for training and evaluating NLP models.
- **Parsing:** Analysing the grammatical structure of a sentence.

NLTK provides the foundational tools for many NLP tasks, from text classification and sentiment analysis to information retrieval and text summarization.

Crawling Through Forms and Logins

Crawling websites that require form submissions or user logins is a common but more advanced task than simple scraping. You can't just follow links; you have to simulate a user's interaction with the site. The **Python Requests library** is the standard tool for this, as it allows you to send HTTP requests with specific data and headers, just like a web browser.

Python Requests Library

The Requests library simplifies sending HTTP requests. Instead of manually constructing the requests, you can use simple, high-level functions like requests.get() and requests.post(). It also handles things like cookies, headers, and authentication for you.

To install it, use pip:

```
pip install requests
```

Submitting a Basic Form

When you fill out and submit a web form, your browser sends data to a server. To do this programmatically, you need to replicate this behavior.

1. **Inspect the Form:** Use your browser's developer tools to inspect the HTML <form> element. Note the following:
   - **Form action:** The URL where the data is sent (the action attribute).
   - **Method:** Whether the request is a GET or POST (the method attribute).
   - **Input fields:** The names of the input fields (the name attribute) and their values.
2. **Send the Request:** Use the requests.post() or requests.get() method and pass a dictionary of the form data to the data parameter.

Examples

```
import requests

# The URL where the form data will be submitted

form_url = 'http://example.com/login'

# A dictionary with the form data

payload = {'username': 'myuser', 'password': 'mypassword'}

# Send the POST request

response = requests.post(form_url, data=payload)

# Check the response status and content

print(response.text)
```

Radio Buttons, Checkboxes, and Other Inputs

Different HTML input types have specific ways of submitting data.

- **Radio Buttons:** Only one radio button in a group can be selected. The form data will contain the name of the group and the value of the selected button.
- **Checkboxes:** Multiple checkboxes can be selected. To represent this in your payload dictionary, you can use a list for the name of the checkbox group.
- **Dropdowns (<select>):** The value sent is the value attribute of the selected <option>.

Submitting Files and Images

When a form includes a file upload, the data is sent using the multipart/form-data encoding. The Requests library handles this easily using the files parameter. You provide a dictionary where the key is the name of the file input field and the value is a tuple containing the filename and the file object.

Examples

```
import requests

url = 'http://example.com/upload'

files = {'file_field_name': ('image.jpg', open('path/to/image.jpg', 'rb'))}

response = requests.post(url, files=files)

print(response.status_code)
```

Handling Logins and Cookies

Many websites use cookies to maintain a user's session after they log in. To crawl a logged-in session, you need to handle these cookies. The Requests library provides a Session object for this.

A Session object persists cookies across multiple requests, so you can log in once and then use the same session object to crawl other pages that require authentication.

Sample session creation

```
# Create a session object

session = requests.Session()

# Log in using a POST request

login_url = 'http://example.com/login'
```