Chapter 7: Handling Text

Introduction

- Unstructured text data, like the contents of a book or a tweet, is both one of the most interesting sources of features and one of the most complex to handle
- In this chapter, we will cover strategies for transforming text into information-rich features

Cleaning Text

Most basic text cleaning operations should only replace Python's core string operations, in particular `strip`, `replace`, and `split`:

```
# Create text
text_data = ["   Interrobang. By Aishwarya Henriette     ",
             "Parking And Going. By Karl Gautier",
             "    Today Is The night. By Jarek Prakash   "]

# Strip whitespaces
strip_whitespace = [string.strip() for string in text_data]
```

Cleaning Text

```
# Show text
strip_whitespace
```

```
['Interrobang. By Aishwarya Henriette',
 'Parking And Going. By Karl Gautier',
 'Today Is The night. By Jarek Prakash']
```

```
# Remove periods
remove_periods = [string.replace(".", "") for string in strip_whitespace]

# Show text
remove_periods
```

```
['Interrobang By Aishwarya Henriette',
 'Parking And Going By Karl Gautier',
 'Today Is The night By Jarek Prakash']
```

Cleaning Text

We also create and apply a custom transformation function:

```
# Create function
def capitalizer(string: str) -> str:
    return string.upper()

# Apply function
[capitalizer(string) for string in remove_periods]
```

```
['INTERROBANG BY AISHWARYA HENRIETTE',
 'PARKING AND GOING BY KARL GAUTIER',
 'TODAY IS THE NIGHT BY JAREK PRAKASH']
```

Cleaning Text

Finally, we can use regular expressions to make powerful string operations:

```
# Import library
import re

# Create function
def replace_letters_with_X(string: str) -> str:
    return re.sub(r"[a-zA-Z]", "X", string)

# Apply function
[replace_letters_with_X(string) for string in remove_periods]
```

```
['XXXXXXXXXXX XX XXXXXXXXX XXXXXXXXX',
 'XXXXXXX XXX XXXXX XX XXXX XXXXXXX',
 'XXXXX XX XXX XXXXX XX XXXXX XXXXXXX']
```

Cleaning Text

Discussion

Most text data will need to be cleaned before we can use it to build features. Most basic text cleaning can be completed using Python's standard string operations. In the real world we will most likely define a custom cleaning function (e.g., `capitalizer`) combining some cleaning tasks and apply that to the text data.

Parsing and Cleaning HTML

Use Beautiful Soup's extensive set of options to parse and extract from HTML:

```
# Load library
from bs4 import BeautifulSoup

# Create some HTML code
html = """
       <div class='full_name'><span style='font-weight:bold'>
       Masego</span> Azra</div>"
       """


# Parse html
soup = BeautifulSoup(html, "lxml")

# Find the div with the class "full_name", show text
soup.find("div", { "class" : "full_name" }).text
```

```
'Masego Azra'
```

Parsing and Cleaning HTML

Discussion

Despite the strange name, Beautiful Soup is a powerful Python library designed for scraping HTML. Typically Beautiful Soup is used scrape live websites, but we can just as easily use it to extract text data embedded in HTML. The full range of Beautiful Soup operations is beyond the scope of this book, but even the few methods used in our solution show how easily we can parse HTML code to extract the data we want.

Removing Punctuation

Solution

Define a function that uses `translate` with a dictionary of punctuation characters:

```
# Load libraries
import unicodedata
import sys

# Create text
text_data = ['Hi!!!! I. Love. This. Song....',
             '10000% Agree!!!! #LoveIT',
             'Right?!?!']

# Create a dictionary of punctuation characters
punctuation = dict.fromkeys(i for i in range(sys.maxunicode)
                            if unicodedata.category(chr(i)).startswith('P'))

# For each string, remove any punctuation characters
[string.translate(punctuation) for string in text_data]
```

```
['Hi I Love This Song', '10000 Agree LoveIT', 'Right']
```

Removing Punctuation

Discussion

`translate` is a Python method popular due to its blazing speed. In our solution, first we created a dictionary, `punctuation`, with all punctuation characters according to Unicode as its keys and `None` as its values. Next we translated all characters in the string that are in `punctuation` into `None`, effectively removing them. There are more readable ways to remove punctuation, but this somewhat hacky solution has the advantage of being far faster than alternatives.

It is important to be conscious of the fact that punctuation contains information (e.g., "Right?" versus "Right!"). Removing punctuation is often a necessary evil to create features; however, if the punctuation is important we should make sure to take that into account.

Tokenizing Text

Solution

Natural Language Toolkit for Python (NLTK) has a powerful set of text manipulation operations, including word tokenizing:

```
# Load library
from nltk.tokenize import word_tokenize

# Create text
string = "The science of today is the technology of tomorrow"

# Tokenize words
word_tokenize(string)
```

```
['The', 'science', 'of', 'today', 'is', 'the', 'technology', 'of', 'tomorrow']
```

Tokenizing Text

We can also tokenize into sentences:

```
# Load library
from nltk.tokenize import sent_tokenize

# Create text
string = "The science of today is the technology of tomorrow. Tomorrow is today."

# Tokenize sentences
sent_tokenize(string)
```

```
['The science of today is the technology of tomorrow.', 'Tomorrow is today.']
```

Discussion

Tokenization, especially word tokenization, is a common task after cleaning text data because it is the first step in the process of turning the text into data we will use to construct useful features.

Removing Stop Words

Problem

Given tokenized text data, you want to remove extremely common words (e.g., *a*, *is*, *of*, *on*) that contain little informational value.

Solution

Use NLTK's `stopwords`:

```
# Load library
from nltk.corpus import stopwords

# You will have to download the set of stop words the first time
# import nltk
```

Removing Stop Words

```
# nltk.download('stopwords')

# Create word tokens
tokenized_words = ['i',
                   'am',
                   'going',
                   'to',
                   'go',
                   'to',
                   'the',
                   'store',
                   'and',
                   'park']

# Load stop words
stop_words = stopwords.words('english')

# Remove stop words
[word for word in tokenized_words if word not in stop_words]
```

```
['going', 'go', 'store', 'park']
```

Removing Stop Words

Discussion

While "stop words" can refer to any set of words we want to remove before processing, frequently the term refers to extremely common words that themselves contain little information value. NLTK has a list of common stop words that we can use to find and remove stop words in our tokenized words:

```
# Show stop words
stop_words[:5]
```

```
['i', 'me', 'my', 'myself', 'we']
```

Note that NLTK's `stopwords` assumes the tokenized words are all lowercased.

Stemming Words

Problem

You have tokenized words and want to convert them into their root forms.

Solution

Use NLTK's `PorterStemmer`:

```
# Load library
from nltk.stem.porter import PorterStemmer

# Create word tokens
tokenized_words = ['i', 'am', 'humbled', 'by', 'this', 'traditional', 'meeting']

# Create stemmer
porter = PorterStemmer()

# Apply stemmer
[porter.stem(word) for word in tokenized_words]
```

```
['i', 'am', 'humbl', 'by', 'thi', 'tradit', 'meet']
```

Stemming Words

Discussion

Stemming reduces a word to its stem by identifying and removing affixes (e.g., gerunds) while keeping the root meaning of the word. For example, both "tradition" and "traditional" have "tradit" as their stem, indicating that while they are different words they represent the same general concept. By stemming our text data, we transform it to something less readable, but closer to its base meaning and thus more suitable for comparison across observations. NLTK's `PorterStemmer` implements the widely used Porter stemming algorithm to remove or replace common suffixes to produce the word stem.

Tagging Parts of Speech

Problem

You have text data and want to tag each word or character with its part of speech.

Solution

Use NLTK's pre-trained parts-of-speech tagger:

```
# Load libraries
from nltk import pos_tag
from nltk import word_tokenize

# Create text
text_data = "Chris loved outdoor running"

# Use pre-trained part of speech tagger
text_tagged = pos_tag(word_tokenize(text_data))

# Show parts of speech
text_tagged
```

```
[('Chris', 'NNP'), ('loved', 'VBD'), ('outdoor', 'RP'), ('running', 'VBG')]
```

Tagging Parts of Speech

The output is a list of tuples with the word and the tag of the part of speech. NLTK uses the Penn Treebank parts for speech tags. Some examples of the Penn Treebank tags are:

Tag	Part of speech
NNP	Proper noun, singular
NN	Noun, singular or mass
RB	Adverb
VBD	Verb, past tense
VBG	Verb, gerund or present participle
JJ	Adjective
PRP	Personal pronoun

Encoding Text as a Bag of Words

Problem

You have text data and want to create a set of features indicating the number of times an observation's text contains a particular word.

Solution

Use scikit-learn's `CountVectorizer`:

```
# Load library
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

# Create text
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])
```

Encoding Text as a Bag of Words

Problem

You have text data and want to create a set of features indicating the number of times an observation's text contains a particular word.

Solution

Use scikit-learn's `CountVectorizer`:

```
# Load library
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

# Create text
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])
```

Encoding Text as a Bag of Words

This output is a sparse array, which is often necessary when we have a large amount of text. However, in our toy example we can use `toarray` to view a matrix of word counts for each observation:

```
bag_of_words.toarray()
```

```
array([[0, 0, 0, 2, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 1, 0, 1],
       [1, 0, 1, 0, 1, 0, 0, 0]], dtype=int64)
```

We can use the `vocabulary_` method to view the word associated with each feature:

```
# Show feature names
count.get_feature_names()
```

```
['beats', 'best', 'both', 'brazil', 'germany', 'is', 'love', 'sweden']
```

Encoding Text as a Bag of Words

Discussion

One of the most common methods of transforming text into features is by using a bag-of-words model. Bag-of-words models output a feature for every unique word in text data, with each feature containing a count of occurrences in observations. For example, in our solution the sentence `I love Brazil. Brazil!` has a value of `2` in the "brazil" feature because the word *brazil* appears two times.

The text data in our solution was purposely small. In the real world, a single observation of text data could be the contents of an entire book! Since our bag-of-words model creates a feature for every unique word in the data, the resulting matrix can contain thousands of features. This means that the size of the matrix can sometimes become very large in memory. However, luckily we can exploit a common characteristic of bag-of-words feature matrices to reduce the amount of data we need to store.

Weighting Word Importance

Problem

You want a bag of words, but with words weighted by their importance to an observation.

Solution

Compare the frequency of the word in a document (a tweet, movie review, speech transcript, etc.) with the frequency of the word in all other documents using term frequency-inverse document frequency (tf-idf). scikit-learn makes this easy with `TfidfVectorizer`:

Weighting Word Importance

```
# Load libraries
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

# Create text
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])

# Create the tf-idf feature matrix
tfidf = TfidfVectorizer()
feature_matrix = tfidf.fit_transform(text_data)

# Show tf-idf feature matrix
feature_matrix
```

Weighting Word Importance

Just as in Recipe 6.8, the output is a spare matrix. However, if we want to view the output as a dense matrix, we can use `.toarray`:

```
# Show tf-idf feature matrix as dense matrix
feature_matrix.toarray()
```

```
array([[ 0.        ,  0.        ,  0.        ,  0.89442719,  0.        ,
         0.        ,  0.4472136 ,  0.        ],
       [ 0.        ,  0.57735027,  0.        ,  0.        ,  0.        ,
         0.57735027,  0.        ,  0.57735027],
       [ 0.57735027,  0.        ,  0.57735027,  0.        ,  0.57735027,
         0.        ,  0.        ,  0.        ]])
```

Weighting Word Importance

vocabulary_ shows us the word of each feature:

```
# Show feature names
tfidf.vocabulary_
```

```
{'beats': 0,
 'best': 1,
 'both': 2,
 'brazil': 3,
 'germany': 4,
 'is': 5,
 'love': 6,
 'sweden': 7}
```

Q&A

Thank you!