O'REILLY®

Early Release
RAW & UNEDITED

Applied Text Analysis with Python

ENABLING LANGUAGE AWARE DATA PRODUCTS
WITH MACHINE LEARNING

Benjamin Bengfort,
Rebecca Bilbro & Tony Ojeda

Applied Text Analysis with Python

Enabling Language Aware Data Products with Machine Learning

Benjamin Bengfort, Rebecca Bilbro, and Tony Ojeda

Applied Text Analysis with Python

by Benjamin Bengfort , Tony Ojeda , and Rebecca Bilbro

Printed in the United States of America.

Published by O’Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Nicole Tache

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition

- 2016-12-19: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491962978 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. Applied Text Analysis with Python, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-96297-8

[FILL IN]

Chapter 1. Text Ingestion and Wrangling

As we explored the architecture of language in the previous chapter, we began to see that it is possible to model natural language in spite of its complexity and flexibility. And yet, the best language models are often highly constrained and application-specific. Why is it that models trained in a specific field or domain of the language would perform better than ones trained on general language? Consider that the term “bank” is very likely to be an institution that produces fiscal and monetary tools in an economics, financial, or political domain, whereas in an aviation or vehicular domain it is more likely to be a form of motion that results in the change of direction of an aircraft. By fitting models in a narrower context, the prediction space is smaller and more specific, and therefore better able to handle the flexible aspects of language.

The bulk of our work in the subsequent chapters will be in “feature extraction” and “knowledge engineering” - where we’ll be concerned with the identification of unique vocabulary words, sets of synonyms, interrelationships between entities, and semantic contexts. However, all of these techniques will revolve around a central text dataset: the *corpus*.

Corpora are collections of related documents that contain natural language. A corpus can be large or small, though generally they consist of hundreds of gigabytes of data inside of thousands of documents. For instance, considering that the average email inbox is 2GB, a moderately sized company of 200 employees would have around a half-terabyte email corpus. Documents contained by a corpus can also vary in size, from tweets to books. Corpora can be *annotated*, meaning that the text or documents are labeled with the correct responses for supervised learning algorithms, or *unannotated*, making them candidates for topic modeling and document clustering.

NOTE

No two corpora are exactly alike and there are many opportunities to customize the approach taken in this chapter. This chapter presents a general method for ingesting *HTML* data from the internet, a ubiquitous text markup that is easily parsed and available in a variety of domains. The HTML data is cleaned, parsed, segmented, tokenized, and tagged into a preprocessed data structure that will be used for the rest of the book.

Naturally the next question should then be “how do we construct a dataset with which to build a language model?” In order to equip you for the rest of the book, this chapter will explore the preliminaries of construction and organization of a domain-specific corpus. Working with text data is substantially different from working with purely numeric data, and there are a number of unique considerations that we will need to take. Whether it is done via scraping, RSS ingestion, or an API, ingesting a raw text corpus in a form that will support the construction of a data product is no trivial task. Moreover, when dealing with a text corpus, we must consider not only how the data is acquired, but also how it is organized on disk. Since these will be very large, often unpredictable datasets, we will need to anticipate potential performance problems and ensure memory safety through streaming data loading and multiprocessing. Finally, we must establish a systematic preprocessing method to transform our raw ingested text into a corpus that is ready for computation and modeling. By the end of this chapter, you should be able to organize your data and establish a reader that knows how to access the text on disk and present it in a standardized fashion for downstream analyses.

Acquiring a Domain-Specific Corpus

Acquiring a domain-specific corpus will be essential to producing a language-aware data product. Fortunately, the internet offers us a seemingly infinite resource with which to construct domain-specific corpora. Below are some examples of domains, along with corresponding web text data sources.

Category	Sources
Politics	*http://www.politico.com* *http://www.cnn.com/politics* *https://www.washingtonpost.com/politics* *http://www.foxnews.com/politics.html*

	http://www.huffingtonpost.com/section/politics
Business	*http://www.bloomberg.com* *http://www.inc.com* *https://www.entrepreneur.com* *https://hbr.org* *http://fortune.com*
Sports	*http://espn.go.com* *http://sports.yahoo.com* *http://bleacherreport.com* *http://www.nytimes.com/pages/sports*
Technology	*http://www.wired.com* *https://techcrunch.com* *http://radar.oreilly.com* *https://gigaom.com* *http://gizmodo.com*
Cooking	*http://blog.foodnetwork.com* *http://www.delish.com* *http://www.epicurious.com* *http://www.skinnytaste.com*

One important question to address is the degree of specificity required of a corpus for effective language modeling; how specific is specific enough? As we increase the specificity of the domain, we will necessarily reduce the volume of our corpus. For instance, it would be easier to produce a large dataset about the general category ‘sports’, but that corpus would still contain a large degree of ambiguity. By specifically targeting text data about baseball or basketball, we reduce this ambiguity, but we also reduce the overall size of our corpus. This is a significant tradeoff, because we will need a very large corpus in order to provide sufficient training examples to our language models, thus we must find a balance between domain specificity and corpus size.

Data Ingestion of Text

As data scientists, we rely heavily on structure and patterns, not only in the content of our data, but in its history and provenance. In general, good data sources have a determinable structure, where different pieces of content are organized according to some schema and can be extracted systematically via the application of some logic to that schema. If there is no common structure or schema between documents, it becomes difficult to discern any patterns for extracting the information we want, which often results in either no data retrieved at all or significant cleaning required to correct what the ingestion process got wrong.

In the specific context of text, we want the result of our ingestion process to be paragraphs of text that are ordered and complete. However, achieving this outcome is made somewhat more or less challenging depending on how the data is stored. When we ingest text data from the internet, the most common format we will encounter are HTML webpages and websites. A website is a collection of web pages that belong to the same person or organization, contain similar content, and most importantly, usually share a common domain name. In the below example, we see that districtdatalabs.com is the website, or the collection of pages, and each of the individual documents listed below it (courses, projects, etc.) represent the individual web pages.

```
districtdatalabs.com
├── /
├── /courses
├── /projects
├── /corporate-offerings
├── /about
└── blog.districtdatalabs.com
|    ├── /an-introduction-to-machine-learning-with-python
|    ├── /the-age-of-the-data-product
|    └── /building-a-classifier-from-census-data
|    └── /modern-methods-for-sentiment-analysis
...
```

The predictability of a common domain name makes systematic data collection simpler and more convenient. However, most ingested HTML does not arrive clean, ordered, and ready for analysis. For one thing, a raw HTML document collected from the web will include much that is not text: advertisements, headers and footers, navigation bars, etc. Because of its loose schema, HTML makes the systematic extraction of the text from the non-text challenging. On the other end of the spectrum is a structured format like JSON, which, while less common than HTML, is human-readable and contains substantially more schema, making text extraction easier. Somewhere in between HTML and JSON is the web syndication format RSS. RSS often provides fields, such as publication date, author, and URL, which help with systematic text extraction and also offer useful metadata that can later be used in feature engineering. On the other hand, RSS does not always contain the full text, providing instead a summary, which may not be sufficiently robust for language modeling.

In the following sections, we will explore each of these three further as we investigate a range of techniques for ingesting text from the internet, including scraping, crawling, RSS, and APIs.

Scraping and Crawling

Two of the most popular ways of ingesting data from the internet are web scraping and web crawling. Scraping (done by scrapers) refers to the automated extraction of specific information from a web page. This information is often a page's text content, but it may also include the headers, the date the page was published, what links are present on the page, or any other specific information the page contains. Crawling (done by crawlers or spiders) involves the traversal of a website's link network, while saving or indexing all the pages in that network. Scraping is done with an explicit purpose of extracting specific information from a page, while crawling is done in order to obtain information about link networks within and between websites. It is possible to both crawl a website and scrape each of the pages, but only if we know what specific content we want from each page and have information about its structure in advance.

CAUTION

There are some important precautions that need to be taken into consideration when crawling websites in order to be good web citizens and not cause any trouble. The first of these are `robot.txt` files, which are files that websites publish telling you what they do and do not allow from crawlers. A simple Google search for the website you're going to crawl and "robots.txt" should get you the file.

Let’s say we wanted to automatically fetch news stories from a variety of sources in order to quickly get a sense of what was happening today. The first step is to start with a seed list of news sites, crawl those sites, and save all the pages to disk. We can do this in Python with the help of the following libraries:

- `requests` to read the content from web pages.
- `BeautifulSoup` to extract the links.
- `awesome-slugify` to format the filenames when we save the pages to disk.

In the code snippet below, we create a function `crawl` that uses the `get` method of the `requests` library to make a call to a series of webpages and read the content of the server’s response. Then we call the `content` method on the result of this response to get its raw contents as bytes. We next create a `BeautifulSoup` object with these contents, and use the `find_all` method from `bs4` to identify the links inside the text by finding each of the `a` tags with an `href`. We then iterate over the set of those links, using `get` and `content` to retrieve and save the content of each of those webpages to a unique HTML file on disk.

```
import bs4
import requests
from slugify import slugify

sources = ['https://www.washingtonpost.com',
           'http://www.nytimes.com/',
           'http://www.chicagotribune.com/',
           'http://www.bostonherald.com/',
           'http://www.sfchronicle.com/']

def crawl(url):
    domain = url.split("//www.")[-1].split("/")[0]
    html = requests.get(url).content
    soup = bs4.BeautifulSoup(html, "lxml")
    links = set(soup.find_all('a', href=True))
    for link in links:
        sub_url = link['href']
        page_name = link.string
        if domain in sub_url:
            try:
                page = requests.get(sub_url).content
                filename = slugify(page_name).lower() + '.html'
                with open(filename, 'wb') as f:
                    f.write(page)
            except:
                pass

if __name__ == '__main__':
    for url in sources:
        crawl(url)
```

The code above creates a series of files that contain the full HTML content for each of the web pages linked to from the main list of `sources` URLs. This includes all text, images, links, and other content present on those pages. One thing to note is that the resultant files are saved without hierarchy in whichever directory the script is run from. As is, the `crawl` script does not segregate the saved HTML files by their original sources, an important data management mechanism that becomes critical as corpora increase in size, and which we will implement in later parts of this chapter. Another consideration with the above `crawl` function is that some of the resultant HTML files will contain fulltext articles, while others will merely contain article headlines and links that fall under a certain topic (e.g. street fashion or local politics). In order to collect all of the fulltext articles from all of the secondary, tertiary, etc. web pages that comprise a website, further recursion is needed.

CAUTION

Another precaution that should be taken is rate limiting, or limiting the frequency at which you ping a website. In practice, you should insert a pause for a certain amount of time (usually at least a few seconds) between each web page you call. One of the reasons for doing this is that if we hit a website with too much traffic too fast, it might bring down the website if it is not equipped to handle that level of traffic. Another reason is that larger websites might not like the fact that you are crawling their site, and they might block your IP address so that you can’t use their site anymore.

Running the code above will take several minutes. In order to speed it up, we may want to multiprocess this task. Multiprocessing means performing actions in parallel, so in this case, we’d be crawling multiple sites at the same time. We can use the `multiprocessing` library to help us do this.

In the following code snippet, we are creating a `multi_proc_crawl` function that accepts as arguments a list of URLs and a number of processes across which to distribute the work. We then create a `Pool` object, which can parallelize the execution of a function and distribute the input across processes. We then call `map`, the parallel equivalent of the Python built-in function, on our `Pool` object, which chops the `crawl` iteration into chunks and submits to the `Pool` as separate tasks. After the `crawl` tasks have been executed for all of the items in the URL list, the `close` method allows the worker processes to exit, and `join` acts as a synchronization point, reporting any exceptions that occurred among the worker processes.

```
from multiprocessing.dummy import Pool

def multi_proc_crawl(url_list, processes=2):
    pool = Pool(processes)
    pool.map(crawl, url_list)
    pool.close()
    pool.join()

multi_proc_crawl(sources, 4)
```

In effect, the above code snippet will split our original job into four processes, and then run those processes in parallel, making our `crawl` function much faster. Multiprocessing is particularly handy for large ingestion and wrangling tasks, and we will discuss it in greater detail in a later section.

At this point, we have several HTML web pages written to disk - one file per web page. Now, we will want to parse the HTML in each file in order to extract just the text we want to keep. The easiest way to do this with Python is to use the `BeautifulSoup` library. For example, if we wanted to extract the text from one of the news article pages we saved and print it to the console, we would use the code snippet below to do so. In the code below, we first identify the tags where text data is contained. Then we create an `html_to_text` function which takes a file path, reads the HTML from the file, and uses the `get_text` method to yield the text from anywhere it finds a tag that matches our tag list.

```
import bs4

TAGS = ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'p', 'li']
```

```
def html_to_text(path):
    with open(path, 'r') as f:
        html = f.read()
        soup = bs4.BeautifulSoup(html, "lxml")
        for tag in soup.find_all(TAGS):
            yield tag.get_text()
```

Just as with crawling, there are also a some considerations to take into account when scraping content from web pages. Some websites have dynamic content that is loaded via JavaScript. For these websites, you would need to take a different approach in order to obtain the content.

There are also several ways to crawl and scrape websites besides the methods we’ve demonstrated here. For more advanced crawling and scraping, it may be worth looking into the following tools.

- Scrapy - an open source framework for extracting data from websites.
- Selenium - a Python library that allows you to simulate user interaction with a website.
- Apache Nutch - a highly extensible and scalable open source web crawler.

Web crawling and scraping can take us a long way in our quest to acquire text data from the web, and the tools currently available make performing these tasks easier and more efficient. However, there is still much work left to do after initial ingestion. While formatted HTML is fairly easy to parse with packages like `BeautifulSoup`, after a bit of experience with scraping, one quickly realizes that while general formats are similar, different websites can lay out content very differently. Accounting for, and working with, all these different HTML layouts can be frustrating and time consuming, which can make using more structured text data sources, like RSS, look much more attractive.

Ingestion using RSS Feeds and Feedparser

RSS (Really Simple Syndication) is a standardized XML format for syndicated text data that is primarily used by blogs, news sites, and other online publishers who publish multiple documents (posts, articles, etc.) using the same general layout. There are different versions of RSS, all originally evolved from the Resource Description Framework (RDF) data serialization model, the most common of which is currently RSS 2.0. Atom is a newer and more standardized, but at the time of this writing, a less widely-used approach to providing XML content updates.

Text data structured as RSS is formatted more consistently than text data on a regular web page, as a content *feed*, or a series of documents arranged in the order they were published. This feed means you do not need to crawl the website in order to get other content or acquire updates, making it preferable to acquiring data through crawling and scraping. If the desired data resides in the body of blog posts or news articles and the website makes them available as an RSS feed, you can merely parse that feed.

Another feature of RSS is its ability to synchronize or retrieve the latest version of the content as articles on the source website are updated. Routine querying ensures any changes to the content are reflected in the XML. However, the RSS format also has some notable drawbacks. Most feeds give the content owner the option of displaying either the full text or just a summary of each post or article. Content owners whose revenue depends heavily on serving advertisements have an incentive to display only summary text via RSS to encourage readers to visit their website to view both the full content and the ads.

In the example below, we introduce the Python `feedparser` library to assist in ingesting the RSS feeds of a list of blogs, parsing them, extracting the text content, and then writing that content to disk as XML files. After creating a list of feeds, the `rss_parse` function uses the `parse` method to parse the XML for each of our feeds. From there, the `entries` method retrieves the feed's posts or articles. Next, we iterate through each post, extracting the title for each, using the `get_text` method to extract the text from inside any of the tags from our tag list, and writing that post's text to a file.

```
import bs4
import feedparser
from slugify import slugify

feeds = ['http://blog.districtdatalabs.com/feed',
         'http://feeds.feedburner.com/oreilly/radar/atom',
         'http://blog.kaggle.com/feed/',
         'http://blog.revolutionanalytics.com/atom.xml']

def rss_parse(feed):
    parsed = feedparser.parse(feed)
    posts = parsed.entries
    for post in posts:
        html = post.content[0].get('value')
        soup = bs4.BeautifulSoup(html, 'lxml')
        post_title = post.title
        filename = slugify(post_title).lower() + '.xml'
        TAGS = ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'p', 'li']
        for tag in soup.find_all(TAGS):
            paragraphs = tag.get_text()
            with open(filename, 'a') as f:
                f.write(paragraphs + '\n \n')
```

When the code above is run, it generates a series of XML files, one for each blog post or article belonging to the each RSS source listed in our `feeds` list. The files contain only the text content from each posts or article.

THE BALEEN INGESTION ENGINE

The actual implementation of ingestion can become complex; APIs and RSS feeds can change, and significant forethought is required to determine how best to put together an application that will conduct not only robust, autonomous ingestion, but also secure data management.

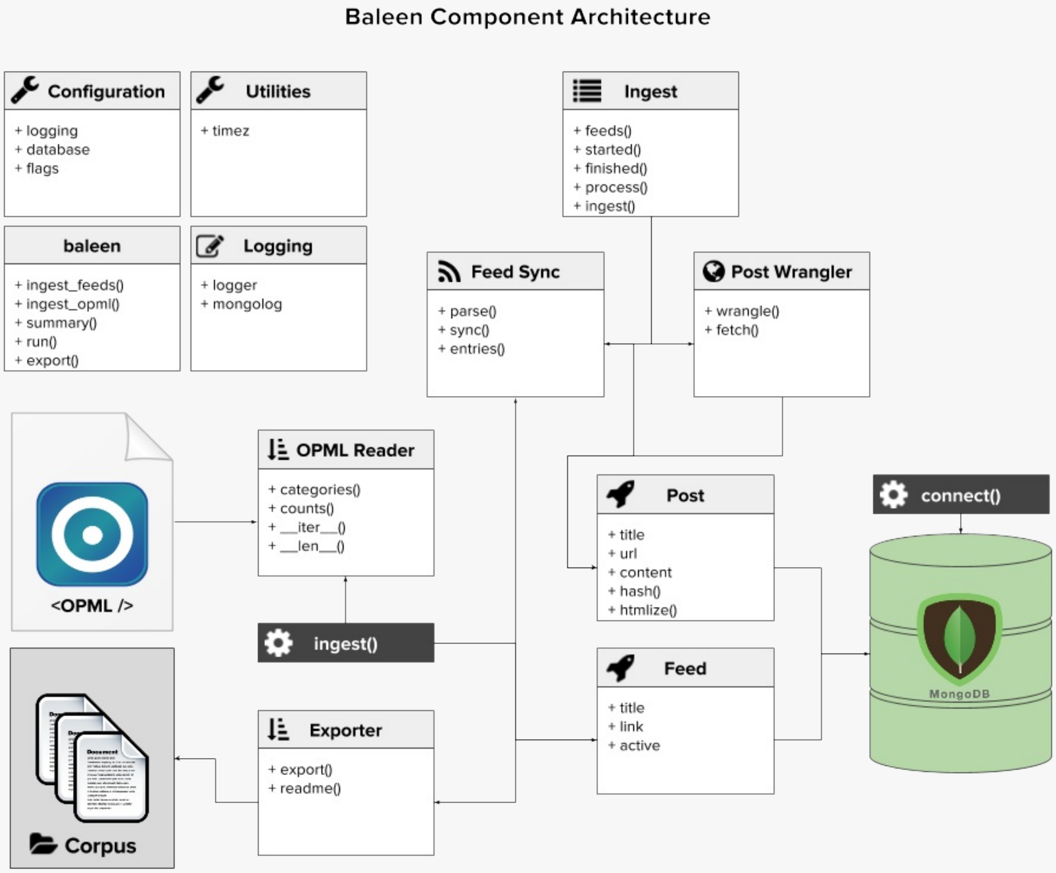


Figure 1-1. The Baleen RSS Ingestion Architecture

The complexity of routine text ingestion via RSS is shown in Figure 1-1. The fixture that specifies what feeds to ingest and how they're categorized is an OPML file that must be read from disk. Connecting and inserting posts, feeds, and other information to the MongoDB store requires an object document mapping (ODM), and tools are need to define a single ingestion job that synchronizes entire feeds then fetches and wrangles individual posts or articles.

With these mechanisms in place, other utilities are required to run the ingestion job on a routine basis (e.g. hourly). Some configuration is required to specify database connection parameters and how often to run. Since this will be a long running process, logging and other types of monitoring are required. A mechanism is needed that will schedule ingestion jobs to run every hour and deal with errors. Finally some tool is needed to export the final corpus ready for preprocessing from the database.

While RSS provides additional structure that makes ingesting text data easier, the sources available in RSS format are typically blog posts and news articles. If other types of data are needed, an alternate ingestion method, such as ingestion from an API, may be necessary.

APIs: Twitter and Search

An API (Application Programming Interface) is a set of programmatic instructions for accessing a web-based software application. Organizations frequently release their APIs to the public to enable others to develop products on top of their data. Most modern web and social media services have APIs that developers can access, and they are typically accompanied by documentation with instructions on how to access and obtain the data.

> **NOTE**
>
> As a web service evolves, both the API and the documentation are usually updated as well, and as developers and data scientists, we need to stay current on changes to the APIs we use in our data products.

A RESTful API is a type of web service API that adheres to representational state transfer (REST) architectural constraints. REST is a simple way to organize interactions between independent systems, allowing for lightweight interaction with clients such as mobile phones and other websites. REST is not exclusively tied to the web, but it is almost always implemented as such, as it was inspired by HTTP. As a result, wherever HTTP can be used, REST can also be used.

In order to interact with APIs, you must usually register your application with the service provider, obtain authorization credentials, and agree to the web service’s terms of use. The credentials provided usually consist of an API key, an API secret, an access token, and an access token secret; all of which consist of long combinations of alpha-numeric and special characters. Having a credentialing system in place allows the service provider to monitor and control use of their API. The primary reason they do this is so that they can prevent abuse of their service. Many service providers allow for registration using OAuth, which is an open authentication standard that allows a user’s information to be communicated to a third party without exposing confidential information such as their password.

APIs are popular data sources among data scientists because they provide us with a source of ingestion that is authorized, structured, and well-documented. The service provider is giving us permission and access to retrieve and use the data they have in a responsible manner. This isn't true of crawling/scraping or RSS, and for this reason, obtaining data via API is preferable whenever it is an option.

To illustrate how we can work with an API to acquire some data, let's take a look at an example. The following example uses the popular `tweepy` library to connect to Twitter's API and then, given a list of user names, retrieves the last 100 tweets from each user and saves each tweet to disk as an individual document.

In order to do this, you must obtain credentials for accessing the API, which can be done by following the steps below.

1. Go to *https://apps.twitter.com* and sign in with your Twitter account.
2. Once you've signed in, click on the **Create New App** button.
3. Fill out the required fields on the form (Name, Description, and Website) and check the checkbox indicating that you've read their Developer Agreement.
4. Click the **Create your Twitter application** button.
5. On the next page, click on the **Keys and Access Tokens** tab, and copy your API Key and API Secret tokens somewhere safe.
6. Scroll to the bottom of the page and click the **Create my access token** button.
7. Under the **Your Access Token** section, you should now see your Access Token and Access Token Secret. Copy these to a safe place also.

You can then substitute your credentials into placeholders in the code below, as well as experimenting with customizing the user list, the number of tweets to retrieve from each user's timeline, and the number of characters from the tweet to use in the file name to suit your needs.

CAUTION

It is worth noting that instead of explicitly storing your credentials in your code, you should instead save them in a JSON config file and then add that config file to a .gitignore file. This will help ensure that your credentials do not accidentally get uploaded to Github, where others can obtain them and potentially use them use them to abuse the service on your behalf.

In the code snippet below, we are using the `tweepy` library to access the Twitter API, passing it our credentials so that it can allow us to proceed. Once we are connected to the API, we pass a list of Twitter usernames to a `users` list. We then iterate through this list, using the `user_timeline` method to fetch the last 100 tweets from each user’s timeline. We then iterate through each tweet in the user’s timeline, use the `text` method to extract the text content, and save it to disk as a line in a JSON file for each user.

```
import tweepy
from slugify import slugify

API_KEY             = " "
API_SECRET          = " "
ACCESS_TOKEN        = " "
ACCESS_TOKEN_SECRET = " "

auth = tweepy.OAuthHandler(API_KEY, API_SECRET)
auth.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

api = tweepy.API(auth)

users = ["tonyojeda3","bbengfort","RebeccaBilbro","OReillyMedia",
         "datacommunitydc","dataelixir","pythonweekly","KirkDBorne"]

def get_tweets(user_list, tweets=20):
    for user in users:
        user_timeline = api.user_timeline(screen_name=user, count=tweets)
        filename = str(user) + ".json"
        with open(filename, 'w+') as f:
            for idx, tweet in enumerate(user_timeline):
                tweet_text = user_timeline[idx].text
                f.write(tweet_text + "\n")

get_tweets(users, 100)
```

The result of running the code above is a single JSON file for each username.

Each JSON file contains a line for each tweet published to that user's timeline.

> **CAUTION**
>
> It is important to read the documentation for an API and respect any specified rate limits. To access data from the Twitter API, we were required to register our application with Twitter. This ensures that they know what you are planning to do with their data, and can monitor, and in some cases control your access to the data. Web service providers like Twitter often impose limits on the amount of data you can retrieve from their service and also how quickly you can retrieve it. If you hit those limits, they will often cut off your access for a limited amount of time, and if you disregard those limits consistently and abuse their service, they may block your access permanently.

Below is a review of the different methods for ingesting text from the web, in order of our preference, and the types of data typically obtained from each.

1. APIs - text content from web services and applications.
2. RSS - text content from blog posts and news articles.
3. Web Crawling and Scraping - text content you can't get via APIs or RSS.

One of the potential problems we have encountered with the `crawl`, `rss_parse`, and `get_tweets` functions is that in each case, the resultant files are saved without any hierarchy or folder structure from which to retrieve information about the source, the corpus category, the date of ingestion, etc. While this absence of structure may simply be a nuisance with only a few dozen files, when we our corpus is likely to contain hundreds of thousands of files, we must seriously consider how to best organize and manage the text content we are ingesting. In the following section, we will explore a workflow that wraps a series of best practices for corpus data management that we have developed though experience.

Corpus Data Management

After identifying a data source to ingest application-specific language, we can then construct a corpus using the fundamental ingestion mechanics discussed in the first part of the chapter. Often, this is where data scientists will start when employing analytics: collecting a single, static set of documents, and then applying routine analyses. However, without considering routine and programmatic data ingestion, analytics will be static and unable to respond to change or new feedback. In the final section of this chapter we will discuss how to monitor corpora as our ingestion routines continue and the data change and grow.

Whether documents are routinely ingested or part of a fixed collection, some thought must go into how manage the data and prepare it for analytical processing and model computation. The first assumption we should make is that the corpora we will be dealing with will be non-trivial — that is they will contain thousands or tens of thousands of documents comprising gigabytes of data. The second assumption is that the language data will come from a source that will need to be cleaned and processed into data structures that we can perform analytics on. The former assumption requires a computing methodology that can scale, and the latter implies that we will be performing irreversible transformations on the data.

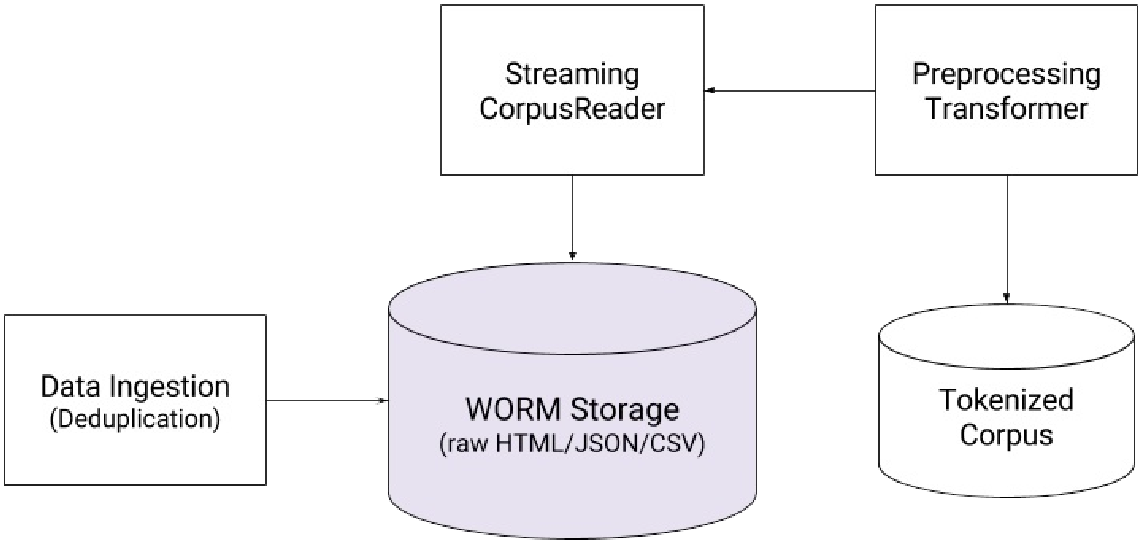

Streaming CorpusReader
Preprocessing Transformer
Data Ingestion (Deduplication)
WORM Storage (raw HTML/JSON/CSV)
Tokenized Corpus

Figure 1-2. WORM Storage Provides and Intermediate Wrangling Step

Data products often employ a write-once, read-many (WORM) storage as an intermediate data management layer between ingestion and preprocessing as shown in Figure 1-2. WORM stores (sometimes referred to as data lakes) provide streaming read accesses to raw data in a repeatable and scalable fashion, addressing the requirement for performance computing. Moreover, by keeping data in a WORM store, preprocessed data can be re-analyzed without re-ingestion; allowing new hypotheses to be easily explored on the raw data format. Because preprocessing is irreversible, having the raw data stored as a backup allows you to conduct analyses without fear.

The addition of the WORM store to our data ingestion workflow means that we need to store data in two places: the raw corpus as well as the preprocessed corpus, and leads to the question: where should that data be stored? When we think of data management, the first thought is a database. Databases are certainly valuable tools in building language aware data products, and many provide full-text search functionality and other types of indexing. However, consider the fact that most databases are constructed to retrieve or update only a couple of rows per transaction. In contrast, computational access to a text corpus will be a complete read of every single document, and will cause no in-place updates to the document, nor search or select individual documents. As such, databases tend to add overhead to computation without real benefit.

NOTE

Relational database management systems are great for transactions that operate on a small collection of rows at a time, particularly when those rows are updated frequently. Machine learning on a text corpus has a different computational profile: many sequential reads of the entire data set. As a result, storing corpora on disk (or in a document database) is often preferred.

For text data management, the best choice is often to store data in a NoSQL document storage database that allows streaming reads of the documents with minimal overhead, or to simply write each document to disk. While a NoSQL application might be worthwhile in large applications, consider the benefits of using a file-based approach: compression techniques on directories are well suited to text information and the use of a file synchronization service provides automatic replication. The construction of a corpus in a database is thus beyond the scope of this book. In order to access our text corpora, we will plan to structure our data on disk in a meaningful way, which we will explore in the next section.

Corpus Disk Structure

The simplest and most common method of organizing and managing a text-based corpus is to store individual documents in a file system on disk. By organizing the corpus into sub directories, corpora can be categorized or meaningfully partitioned by meta information like dates. By maintaining each document as its own file, readers can seek quickly to different subsets of documents and processing can be parallelized, with each process taking a different subset of documents. Text is also the most compressible format, making Zip files, which leverage directory structures on disk, an ideal distribution and storage format, in fact, NLTK `CorpusReader` objects, which we will discuss in the next section, can read from either a path to a directory or a path to a Zip file. Finally, corpora stored on disk are generally static and treated as a whole, fulfilling the requirement for WORM storage presented in the previous section.

Storing a single document per file could lead to some challenges, however. Consider smaller document sizes like emails or tweets, which don’t make sense to store as individual files. Email is typically stored in an MBox format — a plaintext format that uses separators to delimit multipart mime messages containing text, HTML, images, and attachments. The MBox format can be read in Python with the `email` module that comes with the standard library, making it easy to parse with a reader, but difficult to split into multiple documents per file. On the other hand, most email clients store an MBox file per folder (or label), e.g. the Inbox MBox, the Starred MBox, the Archive MBox and so forth, which gives us the idea that a corpus of MBox files organized by category is a good idea.

Tweets are generally small JSON data structures that include not just the text of the tweet but other meta data like user or location. The typical way to store multiple tweets is in newline delimited JSON, sometimes called the JSON lines format. This format makes it easy to read one tweet at a time by parsing only a single line at a time, but also to seek to different tweets in the file. A single file of tweets can be large, so organizing tweets in files by user, location, or day can reduce overall file sizes and again create a disk structure of multiple files. Another technique is simply to write files with a maximum size limit. E.g. keep writing data to the file, respecting document boundaries, until it reaches some size limit (e.g. 128 MB) then open a new file and continue writing there.

NOTE

A corpus on disk will necessarily contain many files that represent one or more documents in the corpus - sometimes partitioned into subdirectories that represent meaningful splits like category. Corpus and document meta information must also be stored along with its documents. As a result a standard structure for corpora on disk is vital to ensuring that data can be meaningfully read by Python programs.

Whether documents are aggregated into multi-document files or each stored as their own file, a corpus represents many files that need to be organized. If corpus ingestion occurs over time, a meaningful organization may be subdirectories for year, month, day - with documents placed into each folder respectively. If the documents are categorized, e.g. for sentiment as positive or negative, each type of document can be grouped together into their own category subdirectory. If there are multiple users in a system that generate their own subcorpora of user-specific writing, for example for reviews or tweets, then each user can have their own subdirectory. Note, however, that the choice of organization on disk has a large impact on how documents are read by `CorpusReader` objects. All subdirectories need to be stored alongside each other in a single corpus root directory. Importantly, corpus meta information such as a license, manifest, readme, or citation must also be stored along with documents such that the corpus can be treated as an individual whole.

THE BALEEN DISK STRUCTURE

The Baleen corpus ingestion engine writes a HTML corpus to disk as follows:

```
corpus
├── citation.bib
├── feeds.json
├── LICENSE.md
├── manifest.json
├── README.md
└── books
|   ├── 56d629e7c1808113ffb87eaf.html
|   ├── 56d629e7c1808113ffb87eb3.html
|   └── 56d629ebc1808113ffb87ed0.html
└── business
```

```
|   ├── 56d625d5c1808113ffb87730.html
|   ├── 56d625d6c1808113ffb87736.html
|   └── 56d625ddc1808113ffb87752.html
└── cinema
|   ├── 56d629b5c1808113ffb87d8f.html
|   ├── 56d629b5c1808113ffb87d93.html
|   └── 56d629b6c1808113ffb87d9a.html
└── cooking
    ├── 56d62af2c1808113ffb880ec.html
    ├── 56d62af2c1808113ffb880ee.html
    └── 56d62af2c1808113ffb880fa.html
```

There are a few important things to note here. First, all documents are stored as HTML files, named according to their MD5 hash (to prevent duplication) and each stored in their own category subdirectory. It is simple to identify which files are documents and which files are meta both by the directory structure and the name of each file. In the next section, we will see that a regular expression must be used to identify which files are documents vs. corpus meta. In terms of meta information, a `citation.bib` file provides attribution for the corpus and the `LICENSE.md` file specifies the rights others have to use this corpus. While these two pieces of information are usually reserved for public corpora, it is helpful to include them so that it is clear how the corpus must be used — for the same reason that you would add this type of information to a private software repository. The `feeds.json` and `manifest.json` files are two corpus-specific files that serve to identify information about the categories, and each specific file respectively. Finally, the `README.md` file is a human readable description of the corpus.

Of these files, `citation.bib`, `LICENSE`, and `README` are special files because they can be automatically read from an NLTK `CorpusReader` object with the `citation()`, `license()`, and `readme()` methods.

A structured approach to corpus management and storage means that applied text analytics follows a scientific process of reproducibility, a method that encourages the interpretability of analytics as well as confidence in their results. Moreover, structuring a corpus as above, enables us to use `CorpusReader` objects, mentioned routinely, but explained in detail in the next section.

Modifying these methods to deal with Markdown or to read corpus-specific files like the manifest is fairly simple:

```
import json

    # In a custom corpus reader class
    def manifest(self):
        """
        Reads and parses the manifest.json file in our corpus if it exists.
        """
        return json.load(self.open("README"))
```

These methods are specifically exposed programmatically to allow corpora to remain compressed, but still readable, minimizing the amount of storage required on disk. Consider that the `README` file is essential to communicating about the composition of the corpus, not just to other users or developers of the corpus, but also to “future you” who may not remember specifics; and to be able to identify which models were trained on which corpora, and what information those models have.

Corpus Readers

Once a corpus has been well structured and organized on disk, two opportunities present themselves: a systematic approach to accessing the corpus in a programming context, and the ability to monitor and manage change in the corpus. We will discuss the latter at the end of the chapter, but for now we will tackle the subject of how to load documents for use in analytics.

Most non-trivial corpora contain thousands of documents with potentially gigabytes of text data. The raw text strings loaded from the documents then need to be preprocessed and parsed into a representation suitable for analysis; an additive process whose methods may generate or duplicate data, increasing the amount of required working memory. From a computational standpoint, this is an important consideration, because without some method to stream and select documents from disk, text analytics would quickly be bound to the performance of a single machine, limiting our ability to generate interesting models. Luckily, tools for streaming accesses of a corpus from disk have been well thought out by the NLTK library, which exposes corpora in Python via `CorpusReader` objects.

NOTE

Though outside the scope of this book, it is important to note that distributed computing frameworks such as Hadoop were created in response to the amount of text data generated by web crawlers to produce search engines (Nutch in the case of Hadoop, which was inspired by two Google papers).

A `CorpusReader` is a programmatic interface to read, seek, stream, and filter documents, and furthermore to expose data wrangling techniques like encoding and preprocessing for code that requires access to data within a corpus. A `CorpusReader` is instantiated by passing a path to the directory that contains the corpus files, the `root` path, a signature for discovering document names, as well as a file encoding (by default, UTF-8). Because a corpus contains files beyond the documents meant for analysis (e.g. the README, citation, license, etc.) some mechanism must be given to the reader to identify exactly what documents are part of the corpus. This mechanism is a parameter that can be specified explicitly as a list of names or implicitly as a regular expression that will be matched upon all documents under the `root`, e.g. `\w`\.txt+, which matches one or more characters or digits in the file name preceding the file extension, `.txt`. For instance, in the following directory, this regex pattern will match the three speeches and the transcript, but not the license, README, or metadata files.

```
corpus
├── LICENSE.md
├── README.md
├── transcript.txt
└── speeches
    ├── 04102008.txt
    ├── 10142009.txt
    ├── 09012014.txt
    └── metadata.json
```

These three simple parameters then give the `CorpusReader` the ability to list the absolute paths of all documents in the corpus, to open each document with the correct encoding, and to allow programmers to access meta data such as the README, license, and citation. By default, NLTK `CorpusReader` objects can even access corpora that are compressed as Zip files, and simple extensions allow the reading of Gzip or Bzip compression as well. By itself, this is not particularly spectacular, however when dealing with a myriad of documents, the interface allows programmers to read one or more documents into memory, to seek forward and backward to particular places in the corpus without opening or reading unnecessary documents, to stream data to an analytical process holding only one document in memory at a time, and to filter or select only specific documents from the corpus at a time. These techniques are what make in-memory text analytics possible for non-trivial corpora because they apply work to only a few documents in-memory at a time.

Therefore, in order to analyze your own text corpus in a specific domain that targets exactly the language models you are attempting to build, you will need an application-specific corpus reader. This is so critical to enabling applied text analytics that we have devoted most of the remainder of this chapter to the subject! In this section we will discuss the corpus readers that come with NLTK and the possibility of structuring your corpus so that you can simply use one of them out of the box. We will then move forward into a discussion of how to define a custom corpus reader that does application-specific work, namely dealing with HTML files collected during the ingestion process.

Streaming Data Access with NLTK

NLTK comes with a variety of corpus readers (66 at the time of this writing) that are specifically designed to access the text corpora and lexical resources that can be downloaded with NLTK. For example, the `CMUDictCorpusReader` reads a dictionary of English language phonemes to support phonetic translation, and the `NombankCorpusReader` provides access to the Nombank corpus, which augments the Penn Treebank with predicate-argument annotations. While these are clearly academic corpora that show very specific properties of language, they are worth noting because you might come across a paper or technique that leverages a specific language or academic format. For example, the `NombankCorpusReader` exposes a frame-based mechanism for discovering the roles of nouns in relation to a predicate. If this technique appeared promising for your use case, transforming your application corpus into the format expected by the `NombankCorpusReader` would open up the possibility of more easily leveraging its methods.

NLTK also comes with slightly more generic utility `CorpusReader` objects.

These objects are fairly rigid in the corpus structure that they expect, but provide the opportunity to quickly create corpora and associate them with readers. They also give hints as to how to customize a `CorpusReader` for application specific purposes. To name a few notable utility readers:

- `PlaintextCorpusReader`: a reader for corpora that consist of plaintext documents, where paragraphs are assumed to be split using blank lines.
- `TaggedCorpusReader`: a reader for simple part-of-speech tagged corpora, where sentences are on their own line, and tokens are delimited with their tag.
- `BracketParseCorpusReader`: a reader for corpora that consist of parenthesis-delineated parse trees.
- `ChunkedCorpusReader`: a reader for chunked (and optionally tagged) corpora formatted with parentheses.
- `TwitterCorpusReader`: a reader for corpora that consist of Tweets that have been serialized into line-delimited JSON.
- `WordListCorpusReader`: List of words, one per line. Blank lines are ignored.
- `XMLCorpusReader`: a reader for corpora whose documents are XML files.
- `CategorizedCorpusReader`: a mixin for corpus readers whose documents are organized by category.

The tagged, bracket parse, and chunked corpus readers are *annotated* corpus readers; if you're going to be doing domain-specific hand annotation in advance of machine learning, then the formats exposed by these readers are important to understand. The Twitter, XML, and plaintext corpus readers all give hints about how to deal with data on disk that has different parseable formats, allowing for extensions related to CSV corpora, JSON, or even from a database. If your corpus is already in one of these formats, then you have little work to do. For example, consider a corpus of the plaintext scripts of the Star Wars and Star Trek movies organized as follows:

```
corpus
├── LICENSE
├── README
```

```
└── Star Trek
|   ├── Star Trek - Balance of Terror.txt
|   ├── Star Trek - First Contact.txt
|   ├── Star Trek - Generations.txt
|   ├── Star Trek - Nemesis.txt
|   ├── Star Trek - The Motion Picture.txt
|   ├── Star Trek 2 - The Wrath of Khan.txt
|   └── Star Trek.txt
└── Star Wars
|   ├── Star Wars Episode 1.txt
|   ├── Star Wars Episode 2.txt
|   ├── Star Wars Episode 3.txt
|   ├── Star Wars Episode 4.txt
|   ├── Star Wars Episode 5.txt
|   ├── Star Wars Episode 6.txt
|   └── Star Wars Episode 7.txt
└── citation.bib
```

The `CategorizedPlaintextCorpusReader` is perfect for accessing data from the movie scripts since the documents are TXT files and there are two categories, namely “Star Wars” and “Star Trek”. In order to use the `CategorizedPlaintextCorpusReader`, we need to specify a regular expression that allows the reader to automatically determine both the `fileids` and `categories`.

```
from nltk.corpus.reader.plaintext import CategorizedPlaintextCorpusReader

DOC_PATTERN = r'(?!\.)[\w_\s]+/[\w\s\d\-]+\.txt'
CAT_PATTERN = r'([\w_\s]+)/.*'

corpus = CategorizedPlaintextCorpusReader(
    '/path/to/corpus/root', DOC_PATTERN, cat_pattern=CAT_PATTERN
)
```

The document pattern regular expression specifies documents as having paths under the corpus root such that there is one or more letters, digits, spaces, or underscores, followed by the `/` character, then one or more letters, digits, spaces, or hyphens followed by `.txt`. This will match documents such as `Star Wars/Star Wars Episode 1.txt` but not documents such as `episode.txt`. The categories pattern regular expression truncates the original regular expression with a capture group that indicates that a category is any directory name, e.g. `Star Wars/anything.txt` will capture `Star Wars` as the category.

You can start to access the data on disk by inspecting how these names are captured:

```
corpus.categories()
# ['Star Trek', 'Star Wars']

corpus.fileids()
# ['Star Trek/Star Trek - Balance of Terror.txt', 'Star Trek/Star Trek - First
Contact.txt', ...]
```

Although regular expressions can be difficult, they do provide a powerful mechanism for specifying exactly what should be loaded by the corpus reader, and how. Alternatively, you could explicitly pass a list of categories and file ids, but that would make the reader a lot less flexible. By using regular expressions you could add new categories by simply creating a directory in your corpus, and add new documents by moving them to the correct directory.

Now that we have access to the `CorpusReader` objects that come with NLTK, we will explore how to modify them specifically for use with the HTML content that we have been ingesting throughout the chapter so far.

Reading an HTML Corpus

The `CategorizedPlaintextCorpusReader` in the previous section is actually very useful as it implements a standard preprocessing API that exposes the following methods:

- `paras()`: a generator of paragraphs, blocks of text delimited with double newlines.
- `sents()`: a generator of individual sentences in the text.
- `words()`: tokenizes the text into individual words.
- `raw()`: provides access to the raw text without preprocessing.

Other `CorpusReader` objects expose other language processing methods, for example automatically tagging or parsing sentences, converting annotated text into meaningful data structures like `Tree` objects, or exposing format-specific utilities like individual XML elements. In order to fit models using machine learning techniques on our text, we will need these methods as part of the feature extraction process. In the next section, we will discuss the details of preprocessing and explore what is actually going on. Before we get to that, however, we need a methodology to stream the HTML data we have ingested to programming.

So far in this chapter we have explored data ingestion from the web through a variety of techniques including web scraping, APIs and search, or by using RSS feeds or other syndication mechanisms. Because we are ingesting data from the Internet, it is a safe bet that the data we're ingesting is formatted as HTML. One option for creating a streaming corpus reader is to simply strip all the tags from the HTML, writing it as plaintext and using the `CategorizedPlaintextCorpusReader`. However, if we do that, we will lose the benefits of HTML — namely computer parseable, *structured* text, which we can take advantage of when preprocessing. Therefore, in this section we will begin to design a custom `HTMLCorpusReader` that we will extend in the preprocessing section.

```
from nltk.corpus.reader.api import CorpusReader
from nltk.corpus.reader.api import CategorizedCorpusReader

# Tags to extract as paragraphs from the HTML text
TAGS = [
    'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'p', 'li'
]

class HTMLCorpusReader(CategorizedCorpusReader, CorpusReader):
    """
    A corpus reader for raw HTML documents to enable preprocessing.
    """

    def __init__(self, root, tags=TAGS, **kwargs):
        """
        Initialize the corpus reader.  Categorization arguments
        (``cat_pattern``, ``cat_map``, and ``cat_file``) are passed to
        the ``CategorizedCorpusReader`` constructor.  The remaining
        arguments are passed to the ``CorpusReader`` constructor.
        """

        # Get the CorpusReader specific arguments
        fileids  = kwargs.pop('fileids')
        encoding = kwargs.pop('encoding')
        # Initialize the NLTK corpus reader objects
        CategorizedCorpusReader.__init__(self, kwargs)
        CorpusReader.__init__(self, root, fileids, encoding)

        # Save the tags that we specifically want to extract.
        self.tags = tags
```

Our `HTMLCorpusReader` class extends both the `CategorizedCorpusReader` and the `CorpusReader`, similarly to how the `CategorizedPlaintextCorpusReader` uses the categorization mixin. *Multiple inheritance* can by tricky, so the bulk of the code in the *`init`* function simply figures out which arguments to pass to which class. In particular, the `CategorizedCorpusReader` takes in generic keyword arguments, and the `CorpusReader` will be initialized with the root directory of the corpus, as well as the fileids and the HTML encoding scheme. However, we have also added our own customization — allowing the user to specify which HTML tags should be treated as independent paragraphs.

The next step is to augment the `HTMLCorpusReader` with a method that will allow us to *filter* how we read text data from disk, either by specifying a list of categories, or a list of file names:

```
    def resolve(self, fileids, categories):
        """
        Returns a list of fileids or categories depending on what is passed
        to each internal corpus reader function. Implemented similarly to
        the NLTK ``CategorizedPlaintextCorpusReader``.
        """
        if fileids is not None and categories is not None:
            raise ValueError("Specify fileids or categories, not both")

        if categories is not None:
            return self.fileids(categories)
        return fileids
```

This method returns a list of file ids whether or not they have been categorized. In this sense, it both adds flexibility and exposes the method signature that we will use for pretty much every other method on the reader. In our `resolve` method, if both `categories` and `fileids` are specified, it will complain, otherwise, the method will use a `CorpusReader` method to compute the file ids associated with the specific categories. Note that `categories` can either be a single category or a list of categories. Otherwise, we will simply return the `fileids` — if this is None, the `CorpusReader` will automatically read every single document in the corpus without filtering.

NOTE

Note, the ability to read only part of a corpus will become essential as we move towards machine learning, particularly for doing cross-validation where we will have to create training and testing splits of the corpus.

At the moment, the `HTMLCorpusReader` doesn't have a method for reading a stream of complete documents, one document at a time. Instead, it will expose the entire text of every single document in the corpus in a streaming fashion to our methods. However, we will want to parse one HTML document at a time, so the following method gives us access to the text on a document by document basis:

```python
def docs(self, fileids=None, categories=None):
    """
    Returns the complete text of an HTML document, closing the document
    after we are done reading it and yielding it in a memory safe fashion.
    """
    # Resolve the fileids and the categories
    fileids = self.resolve(fileids, categories)

    # Create a generator, loading one document into memory at a time.
    for path, encoding in self.abspaths(fileids, include_encoding=True):
        with codecs.open(path, 'r', encoding=encoding) as f:
            yield f.read()
```

Our custom corpus reader now knows how to deal with individual documents in the corpus, one document at a time, allowing us to filter and seek to different places in the corpus. It can handle file ids and categories, and has all the tools imported from NLTK to make disk access easier. In the next section we will extend this class with methods to preprocess the raw HTML as it is streamed in a memory safe fashion and achieve our final text data structure in advance of machine learning — a list of documents, composed of lists of paragraphs, which are lists of sentences, where a sentence is a list of tuples containing a token and its part-of-speech tag.

Preprocessing and Wrangling

A key motivation for writing this book has been the immense challenge we ourselves have encountered in our efforts to build and work with corpora large and rich enough to power meaningfully literate data products. Academic resources and toy corpora exist (many of which are thanks to the work of Steven Bird and his colleagues), yet there are few materials for the developer looking to build a custom application with a heavy-duty corpus. Unfortunately, much of the code provided in teaching materials is intended merely to illustrate the functionalities of NLTK, and the corpora provided have already been annotated; neither of these scale well to a real-world application. Any real corpus in its raw form is completely unusable for analytics without significant preprocessing and compression.

In this section, we will provide a multipurpose preprocessing framework, developed through our own experience, that can be used to systematically transform raw text into usable data. Our framework includes 5 key stages: content extraction, paragraph blocking, sentence segmentation, word tokenization, and part-of-speech tagging. For each of these stages, we have provided functions conceived as methods under the `HTMLCorpusReader` class defined in the previous section.

Readability for Accessing Core Content

Although the web is an excellent source of text with which to build novel and useful corpora, it is also a fairly lawless place in the sense that the underlying structures of webpages need not conform to any set standard. As a result, HTML content, while structured, can be produced and rendered in numerous and sometimes erratic ways. This unpredictability makes it very difficult to extract data from raw HTML text in a methodical and programmatic way. To help us in grappling with the high degree of variability, we can use the Readability-lxml library.

Readability-lxml is a Python wrapper for the Javascript Readability library by Arc90. Just as browsers like Safari and Chrome offer a reading mode, Readability takes away all the distractions from a page, leaving just the text. Given an HTML document, Readability employs a series of regular expressions to remove navigation bars, advertisements, page script tags and CSS, then builds a new Document Object Model (DOM) tree, extracts the text from the original tree and reconstructs the text within the newly restructured tree.

Here we import two readability modules, `Unparseable` and `Document`, which we can use to extract and clean the raw HTML text for the first phase of our preprocessing workflow.

```
from readability.readability import Unparseable
from readability.readability import Document as Paper

    def html(self, fileids=None, categories=None):
        """
        Returns the HTML content of each document, cleaning it using
        the readability-lxml library.
        """
        for doc in self.docs(fileids, categories):
            try:
                yield Paper(doc).summary()
            except Unparseable as e:
                print("Could not parse HTML: {}".format(e))
                continue
```

Our `html` method iterates over each file and uses the `summary` method from Readability’s `Document` class to remove any non-text content, as well as script and stylistic tags, and to correct any of the most commonly misused tags (e.g. `<div>` and `<br>`), only throwing an exception if the original HTML is found to be unparseable. The most likely reason for such an exception is if the function is passed an empty document, which has nothing to parse. The result is clean and well-structured HTML text that we will be able to incrementally decompose into paragraphs, sentences, and tokens.

Documents, Discourse, and Paragraphs

Now that we are able to filter the raw HTML text that we’ve ingested, we will move towards building a preprocessed corpus that is structured in a way that will facilitate machine learning. For this reason, as we dissect the language contained in our corpus, we must also preserve much of the original structure. Paragraphs encapsulate complete ideas, functioning as the unit of document structure, and our first step will be to isolate the paragraphs that appear within the text.

> **NOTE**
>
> The precision and sensitivity of our models will rely on how effectively we are able to link tokens with the textual contexts in which they appear.

Since we have opted to create methods that retain the structure of our HTML documents, we can isolate content that appears within paragraphs by searching for `<p>` tags, the element that formally defines an HTML paragraph. However, content can also appear in other ways, embedded inside other structures within the document like headings and lists, so we must be prepared to search broadly through the text.

Recall that we defined our `HTMLCorpusReader` class so that our reader objects have these tags as a class attribute (which could be expanded, abbreviated, or otherwise modified according to your context), and we can use BeautifulSoup to search for them.

```
import bs4

# Tags to extract as paragraphs from the HTML text
self.tags = [
    'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'p', 'li'
]

    def paras(self, fileids=None, categories=None):
        """
        Uses BeautifulSoup to parse the paragraphs from the HTML.
        """
        for html in self.html(fileids, categories):
            soup = bs4.BeautifulSoup(html, 'lxml')
            for element in soup.find_all(self.tags):
                yield element.text
            soup.decompose()
```

We iterate through each file id and pass each HTML document into the BeautifulSoup constructor, specifying that the HTML should be parsed using the lxml HTML parser. BeautifulSoup uses HTML5 by default, and while lxml is a pain to deal with, it is powerful and needed to parsing meaningfully at scale. The resulting `soup` is a nested tree structure that we can navigate using the original HTML tags and elements. For each of our document soups, we then iterate through each of the tags from our pre-defined set and yield the text from within that tag. We can then call Beautiful Soup's `decompose` method to destroy the tree when we're done working with each file to free up memory.

The result of our `paras` method is a generator with the raw text paragraphs from every document, from first to last, with no document boundaries. If passed a specific file id, `paras` will return the paragraphs from that file only. It's worth noting that the `paras` methods for many of the NLTK corpus readers, such as the standard NLTK CorpusReader, function differently, frequently doing segmentation and tokenization in addition to isolating the paragraphs. This is because NLTK methods tend to expect a corpus that has already been annotated, and are thus not concerned with reconstructing paragraphs. By contrast, our methods are designed to work on raw, un-annotated corpora and will need to support reconstruction. Though critical to our machine learning goal, our decompositional approach is far from easy and will necessitate painstaking deconstruction to isolate paragraphs, then sentences, and then tokens.

Segmentation: Breaking out Sentences

If we can think of paragraphs as the units of document structure, it is useful to see sentences as the units of discourse. Just as a paragraph within a document comprises a single idea, a sentence contains a complete language structure, one that we want to be able to identify and encode. In the next section, the method we will write to parse our text into sentences (*segmentation*) will employ NLTK methods that are trained on sentences. Segementation is a critical next step because the part-of-speech tagging methods we will employ later will rely on an internally-consistent morphology.

Thus, in order to get to the sentences, we'll write a new method that calls our `paras` method, performs sentence segmentation, and returns a generator (an iterator) yielding each sentence from every paragraph.

```
from nltk import sent_tokenize
```

```
    def sents(self, fileids=None, categories=None):
        """
        Uses the built in sentence tokenizer to extract sentences from the
        paragraphs. Note that this method uses BeautifulSoup to parse HTML.
        """
        for paragraph in self.paras(fileids, categories):
            for sentence in sent_tokenize(paragraph):
                yield sentence
```

Our `sents` method iterates through each of the paragraphs we isolated with our `paras` method, using the built-in NLTK `sent_tokenize` method to conduct segmentation. Under the hood, `sent_tokenize` employs the `PunktSentenceTokenizer`, a method that uses unsupervised learning to build a model (a series of regular expressions) for the kinds of words and punctuation (e.g. periods, question marks, exclamation points, capitalization, etc.) that signal the beginnings and ends of sentences.

The result is a generator with a list of sentences for each paragraph. This model has been trained on English text and it works well for most European languages. However, punctuation marks can be ambiguous; while periods frequently signal the end of a sentence, they can also appear in floats, abbreviations, and ellipses. In other words, identifying the boundaries between sentences can be tricky. There are alternative sentences tokenizers, and if your domain space has special peculiarities in the way that sentences are demarcated, it’s also possible to train your own using domain-specific content.

Tokenization: Identifying Individual Tokens

Whereas we have defined sentences as the units of discourse and paragraphs as the units of document structure, tokens are like the atoms of semantics. Recall that tokens are not the same as words. The tokens in our corpus will be sequences of characters that appeared in one or some of the documents and are grouped together in ways that encode some semantic information beyond just substring characters.

Now that we have found our paragraphs within our HTML documents and isolated our sentences within those paragraphs, we next need to find a way to identify the tokens within our sentences. Tokenization is the process by which we’ll arrive at those tokens, and we’ll use `WordPunctTokenizer`, a regular-expression based tokenizer that splits text on both whitespace and punctuation and returns a list of alphabetic and non-alphabetic characters.

```
from nltk import wordpunct_tokenize

    def words(self, fileids=None, categories=None):
        """
        Uses the built in word tokenizer to extract tokens from sentences.
        Note that this method uses BeautifulSoup to parse HTML content.
        """
        for sentence in self.sents(fileids, categories):
            for token in wordpunct_tokenize(sentence):
                yield token
```

As with sentence demarcation, tokenization is not always straightforward. We must consider things like: do we want to remove punctuation from tokens? Should we preserve hyphenated words as compound elements or break them apart? Should we approach contractions as one token or two, and if they are two tokens, where should they be split?

We can select different tokenizers depending on our responses to these questions. Of the many word tokenizers available in NLTK (e.g. `TreebankWordTokenizer`, `WordPunctTokenize`, `PunktWordTokenizer`, etc.), a common choice for tokenization is `word_tokenize`, which invokes the Treebank tokenizer and uses regular expressions to tokenize text as in Penn Treebank. This includes splitting standard contractions (e.g. ``wouldn't`` becomes ``would`` and ``n't``) and treating punctuation marks (like commas, single quotes, and periods followed by whitespace) as separate tokens. By contrast, `WordPunctTokenizer` is based on the `RegexpTokenizer` class, which splits strings using the regular expression \w+|[^\w\s]+ , matching either tokens or separators between tokens and resulting in a sequence of alphabetic and non-alphabetic characters. You can also use the `RegexpTokenizer` class to create your own custom tokenizer.

Part-of-Speech Tagging

Now that we can access the tokens within the sentences of our document paragraphs, we will proceed to tag each token with it's part of speech. Parts of speech (e.g. verbs, nouns, prepositions, adjectives) indicate how a word is functioning within the context of a sentence. In English as in many other languages, a single word can function in multiple ways, and we would like to be able to distinguish those uses (for example “building” can be either a noun or a verb). Part-of-speech tagging entails labeling each token with the appropriate tag, which will encode information both about the word’s definition and it’s use in context.

We’ll use the off-the-shelf NLTK tagger, `pos_tag`, which at the time of this writing uses the `PerceptronTagger()` and the Penn Treebank tagset. The Penn Treebank tagset consists of 36 parts of speech, structural tags, and indicators of tense (`NN` for singular nouns, `NNS` for plural nouns, `JJ` for adjectives, `RB` for adverbs, `PRP` for personal pronouns, etc.).

The `pos_tag` method will differentiate how words are used in context:

```
from nltk import pos_tag

    def tokenize(self, fileids=None, categories=None):
        """
        Segments, tokenizes, and tags a document in the corpus.
        """
        for paragraph in self.corpus.paras(fileids=fileid):
            yield [
                nltk.pos_tag(nltk.wordpunct_tokenize(sent))
                for sent in nltk.sent_tokenize(paragraph)
            ]
```

The `tokenize` method returns a generator of generators containing paragraphs, which are lists of sentences, which in turn are lists of part-of-speech tagged tokens. The tagged tokens are represented as `(tag, token)` tuples, where the tag is a case-sensitive string that specifies how the token is functioning in context.

NOTE

The rule of thumb for part of speech tagging is that if it starts with an *N*, it’s a noun; a *V* if it’s a verb, a *J* for an adjective, an *R* for an adverb, and if it starts with anything else, it’s some kind of a structural element. A full list of tags can be found here: *https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html*

NLTK provides several options for part-of-speech taggers (e.g. `DefaultTagger`, `RegexpTagger`, `UnigramTagger`, `BrillTagger`). Taggers can also be used in combination, such as the `BrillTagger`, which uses Brill transformational rules to improve initial tags.

Transformation

We have now successfully scripted a series of methods that will transform our raw HTML text into data that we will be able to perform machine learning on in the subsequent chapters. Unfortunately, this preprocessing isn't cheap. On a corpus of roughly 300,000 HTML news articles, these preprocessing steps took over 12 hours. Because this is not something you will want to have to do every time your run your models, we must ensure we are saving the preprocessed version of our text. So we'll write a transformer that takes our HTMLCorpusReader, executes each of the five preprocessing steps, and writes out a new text corpus to disk. This new corpus is the one on which we will perform our text analytics.

> **NOTE**
>
> We should anticipate the possibility of routine ingestion, meaning we want to be able to process only the new documents, without re-processing the old ones. This will be addressed in the monitoring section.

There are several options for how to approach this transformation. One is using a tagged corpus view, like the Brown corpus.

With the tagged corpus view, we would write out our paragraphs as double newlines with a sentence on each line, delimiting tokens and tags with a separator character, which is usually a backslash.

```
def transform(htmldir, textdir):
    """
    Pass in a directory containing HTML documents
    and an output directory for the preprocessed
    text and this function transforms the HTML to
    a text corpus that has been tagged in the Brown
    corpus style.
```

```
"""
# List the target HTML directory
for name in os.listdir(htmldir):

    # Determine paths of files to transform & write to
    inpath  = os.path.join(htmldir, name)
    outpath = os.path.join(textdir,
                           os.path.splitext(name)[0]
                           + ".txt")

    # Open the file for reading UTF-8
    if os.path.isfile(inpath):
        with codecs.open(outpath, 'w+',
                         encoding='utf-8') as f:

            # Paragraphs double newline separated,
            # sentences separated by a single newline.
            # Also write token/tag pairs.
            for paragraph in preprocess(inpath):
                for sentence in paragraph:
                    f.write(" ".join("%s/%s"
                            % (word, tag)
                            for word, tag in sentence))
                    f.write("\n")
                f.write("\n")
```

Note that this code does not employ methodology we have defined in this chapter, meaning that it is not looking for subdirectories and relies on a `preprocess` function that we have left undefined in the above. Nevertheless, the outcome is nice because it's simple text, meaning you don't need custom code in order to load the transformed corpus. Moreover, it's also human readable (if a bit weird looking).

And yet, as we gradually build up the text data structure we need (a list of documents, composed of lists of paragraphs, which are lists of sentences, where a sentence is a list of token, tag tuples), we are adding much more content to the original text than we are removing. For this reason, we would also want to apply some kind of compression method to keep disk storage under control. If we used a standard compression methodology, NLTK would still be able read it,

The tagged corpus view doesn't come with compression; though we are stripping out a lot as we go from HTML to plaintext (ads, navigation bars, etc), we actually end up significantly expanding the corpus, meaning that we'll need to add Gzip or Bzip for compression.

Writing to Pickle

Another option for transforming and saving our preprocessed corpus is using `pickle`. With this approach we would write an iterator that loads one document into memory at a time, converts it into the target data structure, and dumps a string representation of that structure to a small file on disk. Unlike the tagged corpus view, this string representation will not be human readable; you won't be able unzip and read it, but it will be more compressed, easier to load, serialize and deserialize, and thus more efficient.

```
import pickle

    def preprocess(self, fileid):
        """
        For a single file does the following preprocessing work:
            1. Checks the location on disk to make sure no errors occur.
            2. Gets all paragraphs for the given text.
            3. Segements the paragraphs with the sent_tokenizer
            4. Tokenizes the sentences with the wordpunct_tokenizer
            5. Tags the sentences using the default pos_tagger
            6. Writes the document as a pickle to the target location.
        This method is called multiple times from the transform runner.
        """
        # Compute the outpath to write the file to.
        target = self.abspath(fileid)
        parent = os.path.dirname(target)

        # Create a data structure for the pickle
        document = list(self.tokenize(fileid))

        # Open and serialize the pickle to disk
        with open(target, 'wb') as f:
            pickle.dump(document, f, pickle.HIGHEST_PROTOCOL)

        # Clean up the document
        del document

        # Return the target fileid
        return target
```

In the above `preprocess` method, once we have established a place on disk to retrieve the original files and to store their processed, pickled, and compressed counterparts, we create a temporary document variable that creates the list of lists of lists of tuples data structure. Then, after we serialize that document and write it to disk using the highest compression option, we delete that document before moving on to the next file to ensure that we are not holding extraneous content in memory.

Pickle Corpus Reader

You can then write out a `PickledHTMLCorpusReader` class that uses `pickle.load()` to quickly retrieve the Python structure:

```
class HTMLPickledCorpusReader(HTMLCorpusReader):
    """
    A corpus reader for the preprocessed pickled documents created by
    the `Preprocessor` module. This reader contains all the
    functionality of the HTMLCorpusReader but may contain less data and
    fields, but hopefully should be much faster reading and parsing data
    from disk.
    """
```

The downside is that you can’t (as a human) read the compressed data on disk, and you won’t be able to easily share the text across different applications, because it will only be readable with Python.

Corpus Monitoring

As we have established in this chapter, applied text analytics requires substantial data management and preprocessing. The methods described for data ingestion, management, and preprocessing are laborious and time-intensive, but also critical precursors to machine learning. Given the requisite time, energy, and disk storage commitments, it is good practice to include with the rest of the data some meta information about the details of how the corpus was built.

In this section, we will describe how to create a monitoring system for ingestion and preprocessing. To begin, we should consider what specific kinds of information we would like to monitor, such as the dates and sources of ingestion. Given the massive size of the corpora with which we will be working, we should, at the very least, keep track of the size of each file on disk.

```
    def sizes(self, fileids=None, categories=None):
        """
        Returns a list of tuples, the fileid and size on disk of the file.
        This function is used to detect oddly large files in the corpus.
        """
        # Resolve the fileids and the categories
        fileids = self._resolve(fileids, categories)

        # Create a generator, getting every path and computing filesize
        for path in self.abspaths(fileids):
            yield os.path.getsize(path)
```

One of our observations in working with RSS HTML corpora in practice is that in addition to text, a significant number of the ingested files came with embedded images, audio tracks, and video. These embedded media files quickly ate up memory during ingestion and were disruptive to preprocessing. The above `sizes` method is in part a reaction to these kinds of experiences with real world corpora, and will help us to be able to perform diagnostics and identify individual files within the corpus that are much larger than expected. This method will enable us to compute the complete size of the corpus, to track over time, and see how it is growing and changing.

Corpus Meta Information

In addition to tracking the size of our corpora as it grows through ingestion, preprocessing, and compression, we also want to watch how the content changes over time. Some of the measures we will track are the vocabulary, word counts, token counts, numbers of files, categories, and lexical diversity of our corpus.

```
def describe(self, fileids=None, categories=None):
    """
    Performs a single pass of the corpus and
    returns a dictionary with a variety of metrics
    concerning the state of the corpus.
    """
    # Structures to perform counting.
    counts  = nltk.FreqDist()
    tokens  = nltk.FreqDist()
    started = time.time()

    # Perform single pass over paragraphs, tokenize and count
    for para in self.paras(fileids, categories):
        counts['paras'] += 1

        for sent in self._sent_tokenizer.tokenize(para):
            counts['sents'] += 1
            for word in self._word_tokenizer.tokenize(sent):
                counts['words'] += 1
                tokens[word] += 1

    # Compute the number of files and categories in the corpus
    n_fileids = len(self._resolve(fileids, categories) or self.fileids())
    n_topics  = len(self.categories(self._resolve(fileids, categories)))

    # Return data structure with information
    return {
        'files':  n_fileids,
        'topics': n_topics,
        'paras':  counts['paras'],
        'sents':  counts['sents'],
        'words':  counts['words'],
        'vocab':  len(tokens),
        'lexdiv': float(counts['words']) / float(len(tokens)),
        'ppdoc':  float(counts['paras']) / float(n_fileids),
        'sppar':  float(counts['sents']) / float(counts['paras']),
        'secs':   time.time() - started,
    }
```

Conclusion

In this chapter, we have learned that text analytics requires a large, robust, domain-specific corpus. We have developed a systematic way of obtaining, storing, and preprocessing this corpus to prepare it for machine learning, which we will begin to do in the next chapter. But first, it will be useful to establish a common vocabulary for machine learning and discuss the ways in which machine learning on text differs from the kind of statistical programming we have done for previous applications. In the next chapter, we will consider how to frame learning problems now that our input data is text, meaning we are working in a very high dimensional space where our instances are complete documents, and our features can include word-level attributes like vocabulary and token frequency, but also metadata like author, date, and source. Next, we will prepare our preprocessed text data for machine learning by encoding it as vectors. We’ll weigh several techniques for vector encoding, and discuss how to wrap that encoding process in a pipeline to allow for systematic loading, normalization, and feature extraction. Finally, we’ll discuss how to reunite the extracted features to allow for more complex analysis and more sophisticated modeling. These steps will leave us poised to extract meaningful patterns from our corpus, and to use those patterns to make predictions about new, as-yet unseen data.

Chapter 2. Machine Learning on Text

As discussed in [Link to Come], natural language is flexible, evolves over time, and depends on context. Computation and analysis on language must also be flexible, therefore the primary computational technique for text analytics is machine learning. Learning techniques give data scientists the ability to train models in a specific context on a specific corpus, make predictions on new data, and adapt over time as the corpus grows and changes. In fact, most natural language processing uses machine learning in one form or another, from tokenization and part of speech tagging, as we saw in the previous chapter, to named entity recognition, entailment, and parsing. More recently, textual machine learning has enabled applications that utilize sentiment analysis, word sense disambiguation, automatic translation and tagging, scene recognition, captioning, chatbots, and more!

Because of Python's unique role in data science, it is rich in third party machine learning tools, from Scikit-Learn to TensorFlow, as well as language processing tools like NLTK and Gensim. In the last chapter we constructed a corpus of preprocessed documents from HTML ingested via RSS feeds, saving them as a pickled list of lists of (token, tag) tuples. The next essential preparatory step is to transform our documents into numeric features, a process called *vectorization*. Representing documents numerically gives us the ability to perform meaningful analytics and also creates the *instances* on which machine learning algorithms operate.

Instances are vector (numeric) representations of distinguishable entities in the real world that describe some predictive property. In text analysis, instances are entire documents, which can vary in length from quotes or tweets to entire books, but whose vectors are always of a uniform length. Each property of the vector representation is called a *feature*, therefore the representation of instances describes a multidimensional feature space where predictions can be made. Features represent some real world attribute of the instance, a mapping or function applied to a combination of properties, or a descriptive value that is computed by analysis of some other feature set. For text in particular, the features represent the attributes and properties of documents. Some of these features may represent a document’s content, while others may represent meta attributes such as document length, author, source, or publication date.

The essential question for this chapter therefore considers how we employ the memory-safe corpus reader we developed in the last chapter for machine learning, particularly with Scikit-Learn. In order to more clearly understand this process, we must engage the *model selection triple*, which guides machine learning workflows. We will then introduce our base vectorization model, *bag-of-words*, as well as some of its extensions, and explore how to use the vectorization process to combine linguistic techniques from NLTK with machine learning techniques in Scikit-Learn, creating repeatable and reusable Pipelines with custom transformers. Finally, we will extend our feature exploration from simple word vectors to include meta features, creating context specific feature extraction and more informative pipelines. By the end of this chapter, we will be ready to engage our preprocessed corpus, transforming documents to model space and begin modeling with both supervised and unsupervised techniques, which we’ll explore in detail in [Link to Come] and [Link to Come].

The Model Selection Triple

Discussions of machine learning are frequently characterized by a singular focus on model selection. Be it logistic regression, random forests, Bayesian methods, or artificial neural networks, machine learning practitioners are often quick to express their preference. The reason for this is mostly historical. Though modern third-party machine learning libraries have made the deployment of multiple models appear nearly trivial, traditionally the application and tuning of even one of these algorithms required many years of study. As a result, machine learning practitioners tended to have strong preferences for particular (and likely more familiar) models over others.

Now that libraries such as Scikit-Learn, NLTK, Gensim, and SpaCy have largely democratized the practice of machine learning, debates about model selection tend to truncate the challenges of machine learning into a single problem. As we will see in the next section, Scikit-Learn provides a single interface to hundreds of models, allowing them to be interchanged and compared with relative ease. While model selection is important (especially in the context of machine learning on text), successful machine learning relies on significantly more than merely having picked the “right” or “wrong” algorithm.

As we hope to have illustrated in the first two chapters, equally if not more important to the construction of language-aware data products is the foundational data layer, which requires robust ingestion, data management, and corpus preprocessing. In this chapter, we will begin to explore the next steps of the workflow, which build directly atop of that foundational layer. We will describe these next steps in the context of the *model selection triple*, a workflow that expands the often truncated perspective of model selection to include three independent, exploratory workflows: feature engineering, algorithm selection, and hyperparameter tuning, as shown in Figure 2-1.

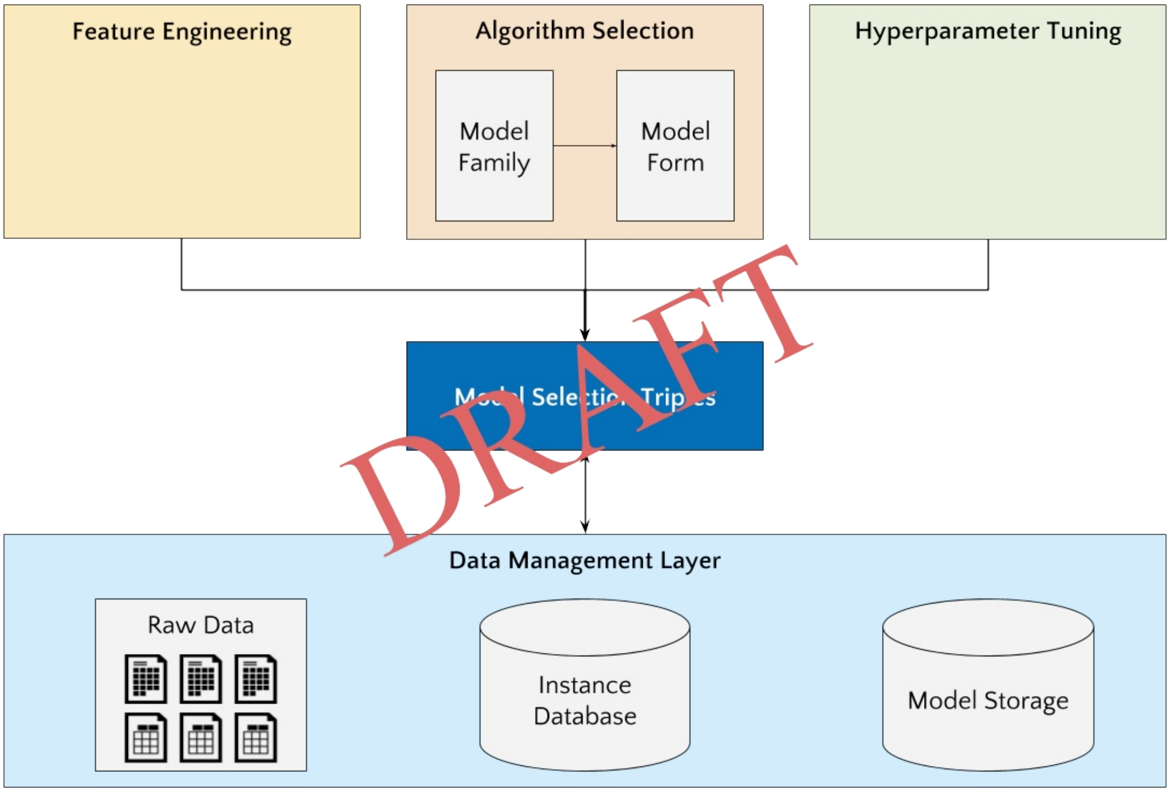


Figure 2-1. The Model Selection Triple

In the feature extraction phase, which we will begin to explore in our discussion of vectorization in this chapter, the goal is to analyze, extract, and select a sufficiently hearty set of features with which to model the data. In the second phase, a set of algorithms are selected from a *model family*, which can then be used, evaluated, and compared in parallel. Finally, we conduct *tuning* by adjusting the model hyperparameters to identify the combination that result in the most predictive fitted model.

These tasks together allow data scientists to define and describe a learning model that is able to effectively leverage specific data (feature engineering) with a specific interaction between variables and the target of interest (algorithm selection) then optimize the behavior of that model during learning and prediction (hyperparameter tuning). Applied methodologies for all three workflows usually include heuristics or rules of thumb for specific algorithms, which can loosely be described as intuition, combined with automatic optimization and search techniques.

While the workflow it describes is one with which many machine learning practitioners are likely familiar, the *model selection triple* was first explicitly described in a 2015 SIGMOD paper by Kumar et al[1]. In their paper, which concerns the development of next-generation database systems built to anticipate predictive modeling, the authors cogently express that such systems are badly needed due to the highly experimental nature of machine learning in practice. “Model selection,” they explain, “is iterative and exploratory because the space of [model selection triples] is usually infinite, and it is generally impossible for analysts to know a priori which [combination] will yield satisfactory accuracy and/or insights.”

Indeed, the process of model selection is complex, iterative, and substantially more intricate than, say, the choice of a support vector machine over a decision tree classifier. Our model selection triple workflow aims to treat these iterations as central to the science of machine learning. It is a workflow that, thanks to the robust and secure foundational data layer, can afford to enable optimization by facilitating rather than limiting those iterations.

Model Selection as Search

Just as the phrase *model selection* often masks the independent tasks of feature engineering, algorithm selection, and tuning, so too is the term *model* overloaded from the multiple domains in which it is used. In a 2015 ASA journal article, Wickham et al[2] neatly disambiguate the term by describing its three principle uses in statistical machine learning: model family, model form, and fitted model. The model family loosely describes the relationships of variables to the target of interest, e.g. a “linear model” or a “recurrent tensor neural network”. The model form is a specific instantiation of the model selection triple: a set of features, an algorithm, and specific hyperparameters. Finally the fitted model is a model form that has been fit to a specific set of training data and is available to make predictions.

Data products are composed of many fitted models, and are constructed through the model selection workflow which creates and evaluates model forms. Data products that build models from natural language are a special case of machine learning as they enable increasingly novel human computer interaction. As data products have become more successful, there has been increasing interest in generally defining a machine learning workflow for more rapid model building. Usually the discussion of machine learning techniques separate workflows and their interaction with data management because they are loosely independent or dependent only on the algorithm selected. However, by combining these workflows into a single generalization, we are able leverage a much larger machine learning space, potentially creating global optimizations and automatic analyses that can be steered (guided) by experts.

Building a machine learning model can therefore be described as the act of defining a model selection triple (MST), fitting and evaluating it, then refining by creating a new instance of the MST and comparing the result. This workflow is iterative in that many MSTs must be generated and evaluated, and exploratory because the model selection triple defines a search space that is infinite and it is impossible to know exactly which MST will perform “the best” before hand. Even if a well performing MST is discovered, we could have reached a local maxima, and another more performant model may exist elsewhere in the model selection space.

When it comes to applied text analytics, the search for the most optimal model may narrow because of the specificity required by language modeling. However the practice is the same: create a specific corpus of documents to train your model on. Select a vectorization technique to describe documents in a numeric space. Fit a model that is able to make predictions about the documents in your corpus. Evaluate the model by training it on only a portion of the data set and scoring it on a reserved, unseen portion (cross-validation). Rinse, wash, and repeat with another MST and compare. At application time, select the best result based on cross-validation and use it to make predictions.

The purpose of these past two sections is to characterize and contextualize machine learning specific terms that you'll encounter throughout the rest of the book. However, a formal approach to a discussion of machine learning is perhaps not required as we focus on the applied aspects of text analytics. In fact, machine learning formalizations are made readily available to Python developers for immediate application by Scikit-Learn. Scikit-Learn provides an API for machine learning that easily allows developers to specify many model forms, fit and evaluate them in repeatable workflows, operationalizing the model selection search process.

The Scikit-Learn API

Scikit-Learn is an extension of SciPy (a scikit) whose primary purpose is to provide machine learning algorithms as well as the tools and utilities required to engage in successful modeling. Its primary contribution is an "API for machine learning" that exposes the implementations of a wide array of model families into a single, user-friendly interface. The result is that Scikit-Learn can be used to simultaneously train a staggering variety of models, evaluate and compare them, then utilize the model to make predictions on new data. Because Scikit-Learn provides a standardized API, this can be done with little to no effort and models can be tried and evaluated by simply swapping out a few lines of code.

The API itself is object-oriented and describes a hierarchy of interfaces for different machine learning tasks. The root of the hierarchy is an `Estimator`, broadly any object that can learn from data. The primary `Estimator` objects we think of implement classifiers, regressors, or clustering algorithms - but they can include a wide array of data manipulation from dimensionality reduction to feature extraction from raw data. The concept of an `Estimator` is an interface, classes which implement `Estimator` functionality must have two methods: `fit` and `predict` as shown below:

```
from sklearn.base import BaseEstimator
```

```
class Estimator(BaseEstimator)

    def fit(self, X, y=None):
        """
        Accept input data, X, and optional target data, y. Returns self.
        """
        return self

    def predict(self, X):
        """
        Accept input data, X and return a vector of predictions for each row.
        """
        return yhat
```

The `Estimator.fit` method sets the state of the estimator based on the training data, `X` and `y`. The training data `X` is expected to be matrix-like, e.g. a two-dimensional numpy array of shape (`n_samples, +n_features`) or a Pandas `DataFrame` whose rows are the instances and whose columns are the features. Supervised estimators are also fit with a one-dimensional numpy array, `y`, that holds the correct labels. The fitting process modifies the internal state of the estimator such that it is ready or able to make predictions, this state is stored in instance variables that are usually postfixed with an underscore, e.g. `Estimator.coefs_`. For example, fitting a linear regression model would compute the optimal coefficients for the linear model, minimizing the error from the input data, and storing those coefficients as an array. Because this method modifies an internal state, it returns `self` so that the method can be chained.

The `Estimator.predict` method creates predictions using the internal, fitted state of the model on the new data, `X`. The input for the method must have the same number of columns as the training data passed to `fit`, and can have as many rows as predictions are required. This method returns a vector, `yhat`, which contains the predictions for each row in the input data.

Specializations of the `Estimator` interface may have additional methods or helpers. For example, models that use probability to make predictions have an `Estimator.predict_proba` method that returns a two-dimensional `yhat` array, whose rows are predictions and whose columns are all possible predictions; the values are the probability associated. Extending `BaseEstimator` automatically gives the `Estimator` a `fit_predict` method which allows you to combine `fit` and `predict` in one simple call.

`Estimator` objects have parameters (also called hyperparameters) that define how the fitting process is conducted. These parameters are set when the `Estimator` is instantiated (and if not specified, they are set to reasonable defaults), and can be modified with the `get_param` and `set_param` methods that are also available from the `BaseEstimator` super class. At this point, we have a fairly complete description of the `Estimator` API, and we can see how the API applies to the model selection workflow. Consider the following example:

```
from sklearn.naive_bayes import MultinomialNB

model = MultinomialNB(alpha=0.0, class_prior=[0.4, 0.6])
model.fit(documents, labels)
```

Scikit-Learn defines the model family by the package and type of the estimator. In this case we have selected the Naive Bayes model family, and a specific member of the family, a multinomial model (which is suitable for text classification). The model form is defined when the class is instantiated and hyperparameters are passed in. Here we pass an alpha parameter which is used for additive smoothing, as well as prior probabilities for each of our two classes. The model form is trained on specific data, `documents` and `labels` and at that point becomes a fitted model. This basic usage is the same for every model (`Estimator`) in Scikit-Learn from random forest decision tree ensembles to logistic regressions and beyond. It is easy to see how the API creates an ease of use that allows developers to try as many models as possible and select the best one.

Scikit-Learn also specifies utilities for performing machine learning in a repeatable fashion and we could not discuss Scikit-Learn without also discussing the `Transformer` interface. A `Transformer` is a special type of `Estimator` that creates a new data set from an old one based on rules that it has learned from the fitting process. The interface is as follows:

```
from sklearn.base import TransformerMixin

class Transfomer(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        """
        Learn how to transform data based on input data, X.
        """
        return self

    def transform(self, X):
        """
        Transform X into a new dataset, Xprime and return it.
        """
        return Xprime
```

The `Transformer.transform` method takes a dataset and returns a new dataset, `X’`, with new values based on the transformation process. There are several included transformers in Scikit-Learn including transformers to normalize or scale features, handle missing values (imputation), perform dimensionality reduction, extract or select features, or perform mappings from one feature space to another. The machine learning process often combines a series of transformers on raw data, transforming the data set each step of the way until it is passed to the fit method of a final estimator, and `Pipeline` objects (discussed later) are a mechanism for sanely combining these steps.

In the context of our model selection triple, feature engineering techniques can be described as reproducible pipelines of transformers, algorithm selection is as simple as importing an estimator from a model family package and instantiating a model form. Hyperparameter tuning can utilize reasonable defaults, as well as grid search all estimators with cross validation. Several models can be fit and evaluated in parallel, and the best results can inform the next steps for data scientists. By using the pickle module, both data and models can be serialized to a data management system for comparison and review down stream, and operationalized in a data product.

Although both NLTK, Gensim, and even newer text analytics libraries like SpaCy have their own internal APIs and learning mechanisms, the scope and comprehensiveness of Scikit-Learn models and methodologies for machine learning make it an essential part of the modeling workflow. As a result we propose to use the API to create our own `Transformer` and `Estimator` objects that implement methods from NLTK and Gensim. For example, we can create topic modeling estimators that wrap Gensim’s LDA and LSA models (which are not currently included in Scikit-Learn) or create transformers that utilize NLTK's part of speech tagging and named entity chunking methods. We will explore this in the last section of the chapter. However, the first, most critical step is to implement a mechanism that converts raw text into a feature space that can be learned on - a feature engineering and extraction process called *vectorization*.

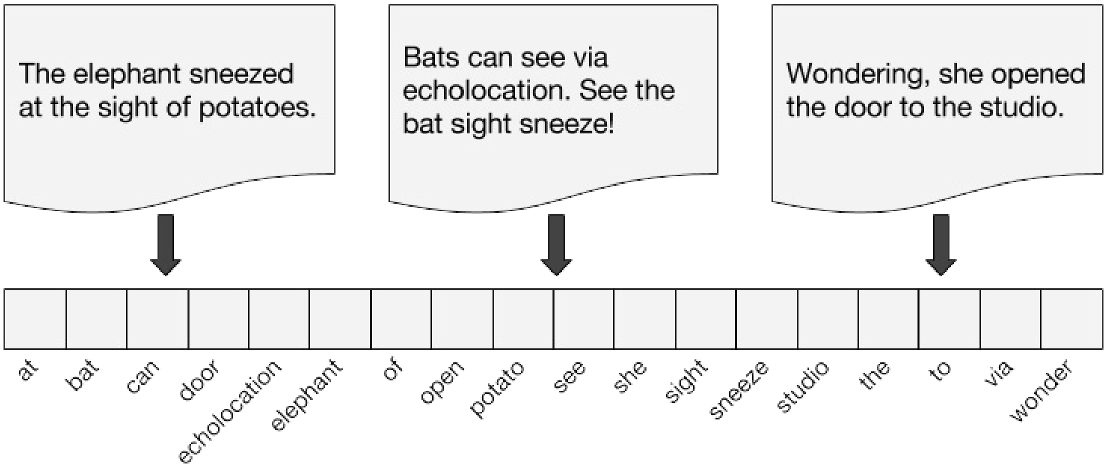
Vectorization

Machine learning algorithms operate on a numeric feature space, expecting input as a two-dimensional array where rows are instances and columns are features. In order to perform machine learning on text, we need to transform our instances, documents, into vector representations such that we can apply numeric machine learning. The process of encoding documents in a numeric feature space is called *feature extraction* or more simply, *vectorization* and is an essential first step towards language aware analysis.

In order to understand vectorization, we must shift from thinking about language as a sequence of words toward thinking about how language might occupy a high-dimensional semantic space. The term *space* implies a spatial region where each instance is represented by a point; points in space can be close together or far apart. Semantic space is therefore a mapping of meaning to space such that documents that have a similar meaning are closer together and documents that are very different are farther apart. If we can encode similarity as distance we can begin to derive the primary components of documents (ideas) and draw decision regions in semantic space.

The simplest encoding of semantic space is the "bag-of-words" model, whose primary insight is that meaning and similarity is encoded in the specific vocabulary used in each document. For example, a Wikipedia article about baseball and Babe Ruth are probably very similar because the same words will appear in both, whereas they will not share many words in common with an article about quantitative easing. This model, while simple, is extremely effective and is the starting point for more complex models.

In order to vectorize a corpus with a bag of words approach, we create a per-document representation as a vector whose length is equal to the vocabulary of the entire corpus from which the document originated as shown in Figure 2-2. In order to simplify the computation of the representation, the vector positions can be sorted in lexicographic (alphabetical) order, but this is not necessary so long as there is a dictionary which maps a token to a vector position. The entries of each element in the vector uniquely describe a single document.



The elephant sneezed at the sight of potatoes.
Bats can see via echolocation. See the bat sight sneeze!
Wondering, she opened the door to the studio.
at
bat
can
door
echolocation
elephant
of
open
potato
see
she
sight
sneeze
studio
the
to
via
wonder

Figure 2-2. Encode Documents as Vectors

The question then becomes what each element in the document vector should be, and in fact there are several choices from binary encoding to integer encoding, real number encoding, and even distributed representations. Each of these models extends or modifies the base bag of words model in order to capture more information and describe the semantic space more meaningfully. Note, however, that these vectors can be extremely sparse, particularly as vocabularies get larger, having a significant impact on machine learning methodologies in each of our three text analysis libraries.

In the next few sections we will look at several of these methods and discuss their implementations in Scikit-Learn, Gensim, and NLTK. We'll operate on a small corpus of the three sentences in the example figures. As a quick setup let's create a list of these documents and tokenize them for the proceeding vectorization examples:

```
import nltk
import string


def tokenize(text):
    stem = nltk.stem.SnowballStemmer('english')
```

```
        text = text.lower()

        for token in nltk.word_tokenize(text):
            if token in string.punctuation: continue
            yield stem.stem(token)


    corpus = [
        "The elephant sneezed at the sight of potatoes.",
        "Bats can see via echolocation. See the bat sight sneeze!",
        "Wondering, she opened the door to the studio.",
    ]
```

The tokenization method here performs some lightweight normalization, stripping punctuation using the `string.punctuation` character set and setting the text to lowercase. This function also performs some feature reduction using the `SnowballStemmer` to remove affixes such as plurality ("bats" and "bat" are the same token). The examples in the next section will utilize this example corpus and some will use the tokenization method.

Frequency Vectors

The simplest vector encoding model is to simply fill in the vector with the frequency of each word as it appears in the document, that is a vector of word counts. In this encoding scheme each document is represented as the multiset of the tokens that compose the document and the value for each word position in the vector is the number of times it appears. This representation can either be a straight count (integer) encoding as shown in Figure 2-3 or a normalized encoding where each word is weighted by the total number of words in the document.

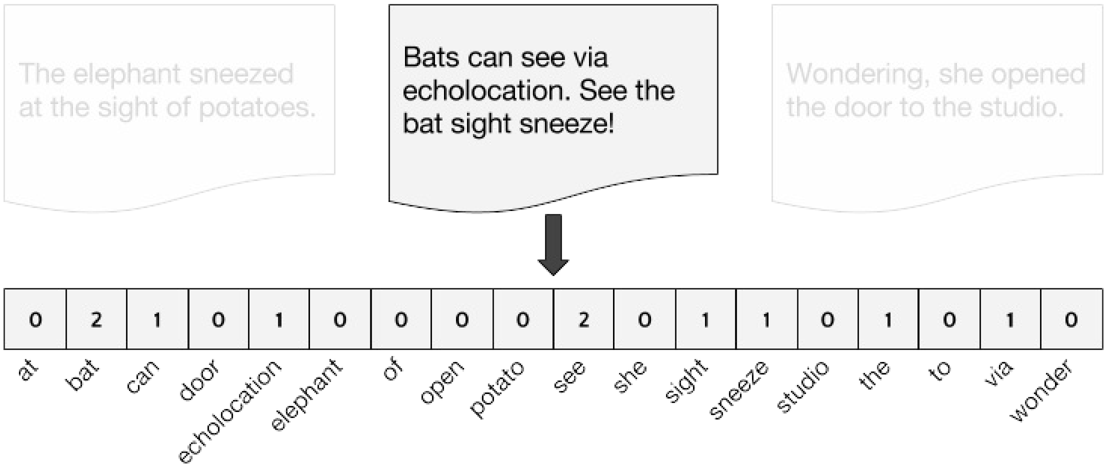

The elephant sneezed at the sight of potatoes.
Bats can see via echolocation. See the bat sight sneeze!
Wondering, she opened the door to the studio.
0
2
1
0
1
0
0
0
0
2
0
1
1
0
1
0
1
0
at
bat
can
door
echolocation
elephant
of
open
potato
see
she
sight
sneeze
studio
the
to
via
wonder

Figure 2-3. Token Frequency as Vector Encoding

NLTK provides no special mechanism for encoding documents as vectors, but instead expects features as a `dict` object whose keys are the names of the features and whose values are boolean or numeric. Because NLTK uses standard Python data structures, it also demonstrates how easy this type of encoding is. The NLTK vectorization method is:

```
from collections import defaultdict

def vectorize(doc):
    features = defaultdict(int)

    for token in tokenize(doc):
        features[token] += 1

    return features


vectors = map(vectorize, corpus)
```

The `vectorize` method simply creates a dictionary whose keys are the tokens in the document and whose values are the number of times that token appears in the document. The `defaultdict` object allows us to specify what the dictionary will return for a key that hasn’t been assigned to it yet. By setting `defaultdict(int)` we are specifying that a `0` should be returned, thus creating a simple counting dictionary. Note that the `collections` module has a `Counter` object that would work similarly. We can `map` this function to every item in the corpus using the last line of code, creating an iterable of vectorized documents.

The Scikit-Learn method involves the use of a *transformer* that vectorizes text. We will discuss the Scikit-Learn API in detail in [Link to Come]. The `CountVectorizer` transformer in the `feature_extraction` module implements two methods, `fit` and `transform`, which can be called simultaneously as follows:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectors = vectorizer.fit_transform(corpus)
```

The first step is to instantiate a blank `CountVectorizer`. The `fit` method of the vectorizer expects an iterable or list of strings or file objects. The `CountVectorizer` has its own internal tokenization and normalization methods (which we will extend to use NLTK in [Link to Come]); when `fit` is called it applies those functions to the strings and creates a dictionary of the vocabulary on the corpus. When `transform` is called, each individual document is transformed into a sparse array whose index tuple is the row (the document id) and the token id from the dictionary, and whose value is the count.

> **NOTE**
>
> For very large corpora, it is recommended to use the Scikit-Learn `HashingVectorizer`, which uses a hashing trick to find the token string name to feature index mapping. This means it uses very low memory and scales to large data sets as it does not need to store the entire vocabulary and it is faster to pickle and fit since there is no state. However, there is no inverse transform (from vector to text), there can be collisions, and there is no inverse document frequency weighting.

Finally, Gensim also has a frequency encoder called `doc2bow`. Unlike Scikit-Learn, Gensim does no work on behalf of your documents for tokenization or stemming. Additionally, Gensim also can serialize its dictionaries and references in matrix market format, making it a bit more flexible for multiple platforms. We'll see this in action in [Link to Come]. The Gensim vectorization model is as follows:

```
import gensim

corpus  = [tokenize(doc) for doc in corpus]
id2word = gensim.corpora.Dictionary(corpus)
vectors = [
    id2word.doc2bow(doc) for doc in corpus
]
```

We first create a Gensim `Dictionary` which maps tokens to particular indices based on the order it observes tokens as it reads through the corpus (eliminating the overhead of lexicographic sorting). The dictionary object can be loaded or saved to disk but also implements a `doc2bow` library which accepts a *pre-tokenized document* and returns a sparse matrix of `(id, count)` tuples where the `id` is the token's id in the dictionary. Because Gensim expects a list of tokens, it assumes that preprocessing such as tokenization, stemming, and normalization has already occurred. The `doc2bow` method also only takes a single document instance, which is why we used the list comprehension to restore the entire corpus.

Frequency representations disregard grammar as well as information about the relative position of words in the document. Bag-of-words models and the frequency representation in particular rely on the assumption that word count is a close enough approximation of the document contents, encoding enough information to differentiate it from other documents. However, this comes at the cost of highly frequent terms being represented by orders of magnitude over other terms. In the next section we will explore one-hot encoding, which eliminates this problem.

One-Hot Encoding

Frequency based encoding methods suffer from the *long tail*, or Zipfian distribution, that characterizes natural language. Because of this distribution, some features (tokens) are orders of magnitude more "significant" than other less frequent tokens and *hapax legomena*. This can have a significant impact on some models, particularly models which expect normally distributed features, generalized linear models for example.

The solution to this problem is *one-hot encoding*, a boolean vector encoding such that if the token of a particular vector index exists in the document, that element is marked as true, otherwise false. In other words, each element of a one-hot encoded vector simply reflects the presence or absence of the token in the described text as shown in Figure 2-4. This method is frequently used in artificial neural network models whose activation functions require input to be in the range of [0,1] or [-1,1].

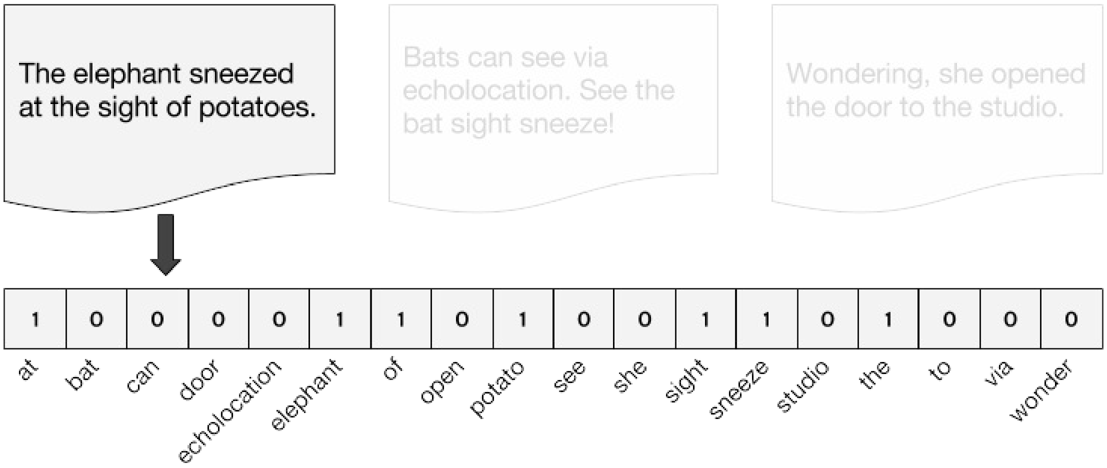

The elephant sneezed at the sight of potatoes.
Bats can see via echolocation. See the bat sight sneeze!
Wondering, she opened the door to the studio.
1
0
0
0
0
1
1
0
1
0
0
1
1
0
1
0
0
0
at
bat
can
door
echolocation
elephant
of
open
potato
see
she
sight
sneeze
studio
the
to
via
wonder

Figure 2-4. One Hot Encoding

The NLTK implementation of one-hot encoding is simply a dictionary whose keys are tokens and whose value is True:

```
def vectorize(doc):
    return {
        token: True
        for token in doc
    }

vectors = map(vectorize, corpus)
```

Dictionaries act as simple sparse matrices in the NLTK case because it is not necessary to mark every absent word False. In addition to the boolean dictionary values, it is also acceptable to use an integer value, 1 for present and 0 for absent.

In Scikit-Learn, one-hot encoding is implemented with the Binarizer transformer in the preprocessing module. The Binarizer takes only numeric data, therefore the text data must be transformed into a numeric space using the CountVectorizer ahead of one-hot encoding.

```
from sklearn.preprocessing import Binarizer

freq   = CountVectorizer()
corpus = freq.fit_transform(corpus)

onehot = Binarizer()
corpus = onehot.fit_transform(corpus.toarray())
```

The `Binarizer` class uses a threshold value (0 by default) such that all values of the vector that are less than or equal to the threshold are set to zero, while those that are greater than the threshold are set to one. Therefore by default the `Binarizer` converts all frequency values to one while maintaining the zero-valued frequencies. Note that the `corpus.toarray()` method in the code snippet above converts the sparse matrix representation to a dense one, but is optional. In corpora with large vocabularies, the sparse matrix representation is much better.

Note that in spite of its name, the `OneHotEncoder` transformer in the `sklearn.preprocessing` module is not exactly the right fit for this task. The `OneHotEncoder` treats each vector component (column) as an independent categorical variable, expanding the dimensionality of the vector for each observed value in each column. In this case, the component `(sight, 0)` and `(sight, 1)` would be treated as two categorical dimensions rather than as a single binary encoded vector component.

Gensim does not have a library-specific one-hot encoder, however the Gensim `doc2bow` method returns a list of tuples that we can manage on the fly. Extending the code from the frequency vectorization example, we can utilize the Gensim dictionary as follows for one-hot encoding:

```
corpus  = [tokenize(doc) for doc in corpus]
id2word = gensim.corpora.Dictionary(corpus)
vectors = [
    [(token[0], 1) for token in id2word.doc2bow(doc)]
    for doc in corpus
]
```

This approach uses a double list comprehension, the inner comprehension converts the list of tuples returned from the `doc2bow` method into a list of `(token_id, 1)` tuples and the outer comprehension applies that converter to all documents in the corpus.

Note that one-hot encoding is not able to encode per-word similarity, as all words are rendered equidistant, and thus are equally different. However, if we use this encoding scheme on entire documents, meaning that we represent each document as a single vector of the length of the corpus vocabulary that consists of ones and zeroes depending on whether those vocabulary words appear in the document, we can begin to encode similarity and difference at the *document* level.

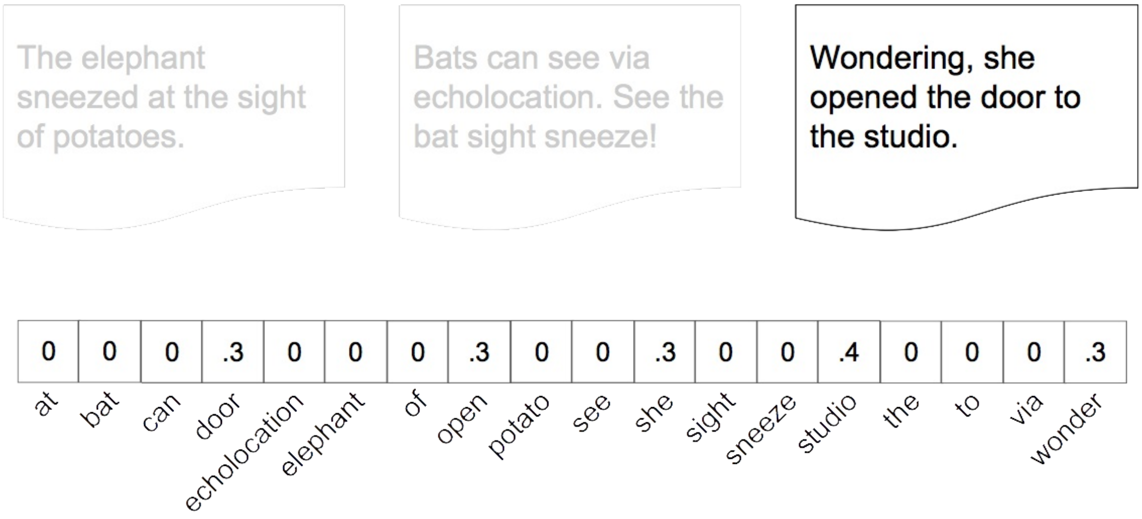
Importantly, because all words are equally distant, word form becomes incredibly important in one-hot encoded vectors. Consider the fact that the tokens `trying` and `try` are equally distant from unrelated tokens like `red` or `bicycle`. Normalizing tokens to a single word class, either through stemming or lemmatization ensures that different forms of tokens that embed plurality, case, gender, cardinality, tense, etc. are treated as a single vector component, reducing the feature space and making this model more performant. We employed stemming in the `tokenize` method in the last section; later we will explore the use of lemmatization for more accurate word class discovery.

One-hot encoding reduces the imbalance issue of the distribution of tokens in a corpus, at the cost of simplifying a document only to it’s constituent components. This is such a reduction in the amount of information that one-hot encoding is generally used for very small documents (sentences, tweets) that don’t contain very many repeated elements, and is usually applied to models that have very good smoothing properties, for example artificial neural networks. In the next section we’ll look at another normalization technique, TF-IDF that also attempts to handle the distribution problems with a frequency approach, while emphasizing semantically relevant terms to specific documents.

Term Frequency-Inverse Document Frequency

The bag-of-words representations that we have explored so far only describe a document in a stand-alone fashion, not taking into account the context of the corpus. A better approach would be to consider the relative frequency or rareness of tokens in the document against their frequency in other documents. The central insight is that tokens that appear frequently in a document have more relevance to that document, particularly if they appear infrequently in the rest of the corpus. For example in a corpus of sports text, in documents that discuss baseball tokens such as `umpire`, `base`, `dugout` will appear more frequently, whereas other tokens that appear frequently throughout the corpus, like `run`, `score`, and `play`, will be less important.

TF-IDF, *term frequency-inverse document frequency*, encoding normalizes the frequency of tokens in a document with respect to the rest of the corpus. This encoding approach accentuates terms that are very relevant to a specific instance (document), as shown in Figure 2-5, where the token `studio` has a higher relevance to this document since it only appears there. Here, the underlying assumption is that meaning is most likely encoded in the more rare terms from a document.



The elephant sneezed at the sight of potatoes.

Bats can see via echolocation. See the bat sight sneeze!

Wondering, she opened the door to the studio.

at	bat	can	door	echolocation	elephant	of	open	potato	see	she	sight	sneeze	studio	the	to	via	wonder
0	0	0	.3	0	0	0	.3	0	0	.3	0	0	.4	0	0	0	.3

Figure 2-5. TF-IDF Encoding

TF-IDF is computed on a per-term basis, such that the relevance of a token to a document is measured by the scaled frequency of the appearance of the term in the document, normalized by the inverse of the scaled frequency of the term in the entire corpus. Let's take this definition piece by piece. The term frequency of a term given a document, $tf(t, d)$, can simply be the boolean frequency (as in one-hot encoding, 1 if t occurs in d 0 otherwise), or the count. However, generally both the term frequency and inverse document frequency are scaled logarithmically to prevent bias of longer documents or terms that appear much more frequently relative to other terms: $tf(t,\ d) = 1 + \ \log f_{t,d}$.

Similarly the inverse document frequency of a term given the set of documents can be logarithmically scaled as follows: $idf(t,\ D) = \ \log 1 + \frac{N}{n_t}$ where N is the number of documents and n_t is the number of occurrences of the term t in all documents. TF-IDF is then computed completely as $tfidf(t,\ d,\ D) = tf(t,\ d) \cdot idf(t,\ D)$. Because of the ratio of the idf log function is greater or equal to 1, the TF-IDF score is always greater than or equal to zero. We interpret the score to mean, the closer the TF-IDF score of a term to a document to 1, the more information that term imparts to that document, the closer to zero, the less informative that term is.

In order to vectorize text with NLTK, we must use the `TextCollection` class, a wrapper for a list of texts or a corpus consisting of one or more texts. This class provides support for counting, concordancing, collocation discovery, and more importantly, computing `tf_idf`.

```
from nltk.text import TextCollection

def vectorize(corpus):
    corpus = [tokenize(doc) for doc in corpus]
    texts  = TextCollection(corpus)

    for doc in corpus:
        yield {
            term: texts.tf_idf(term, doc)
            for term in doc
        }
```

Because TF-IDF requires the entire corpus, this version of the `vectorize` function does not accept a single document, but rather all documents. After applying our tokenization function and creating the text collection, the function goes through each document in the corpus and yields a dictionary whose keys are the terms and whose values are the TF-IDF score for the term in that particular document.

Scikit-Learn provides a transformer called the `TfidfVectorizer` in the `feature_extraction.text` module for vectorizing documents with TF-IDF scores. Under the hood, the `TfidfVectorizer` uses the `CountVectorizer` estimator we used to produce the bag-of-words encoding to count occurrences of tokens, followed by a `TfidfTransformer`, which normalizes these occurrence counts by the inverse document frequency.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf  = TfidfVectorizer()
corpus = tfidf.fit_transform(corpus)
```

The input for a `TfidfVectorizer` is expected to be a sequence of filenames, file-like objects, or strings that contain a collection of raw documents, similar to that of the `CountVectorizer`. As a result, a default tokenization and preprocessing method is applied unless other functions are specified. The vectorizer returns a sparse matrix representation in the form of `((doc, term) tfidf)` where each key is a document and term pair and the value is the TF-IDF score.

Gensim also provides a built-in implementation of TF-IDF, highlighting the popularity of this encoding for most text analytics. In Gensim, the `TfidfModel` datastructure is similar to the `Dictionary` object in that it stores a mapping of terms and their vector positions in the order they are observed, but additionally stores the corpus frequency of those terms so it can vectorize documents on demand.

```
corpus  = [tokenize(doc) for doc in corpus]
lexicon = gensim.corpora.Dictionary(corpus)
tfidf   = gensim.models.TfidfModel(dictionary=lexicon, normalize=True)
vectors = [tfidf[lexicon.doc2bow(doc)] for doc in corpus]
```

As before, Gensim allows us to apply our own tokenization method, expecting a corpus that is a list of lists of tokens. We first construct the lexicon and use it to instantiate the `TfidfModel`, which computes the normalized inverse document frequency. We can then fetch the TF-IDF representation for each vector using a `getitem`, dictionary-like syntax, after applying the `doc2bow` method to each document using the lexicon.

Gensim provides helper functionality to write dictionaries and models to disk in a compact format, meaning you can conveniently save both the TF-IDF model and the lexicon to disk in order to load them later to vectorize new documents. It is possible though slightly more work to achieve the same result by using the `pickle` module in combination with Scikit-Learn. To save a Gensim model to disk:

```
lexicon.save_as_text('lexicon.txt', sort_by_word=True)
tfidf.save('tfidf.pkl')
```

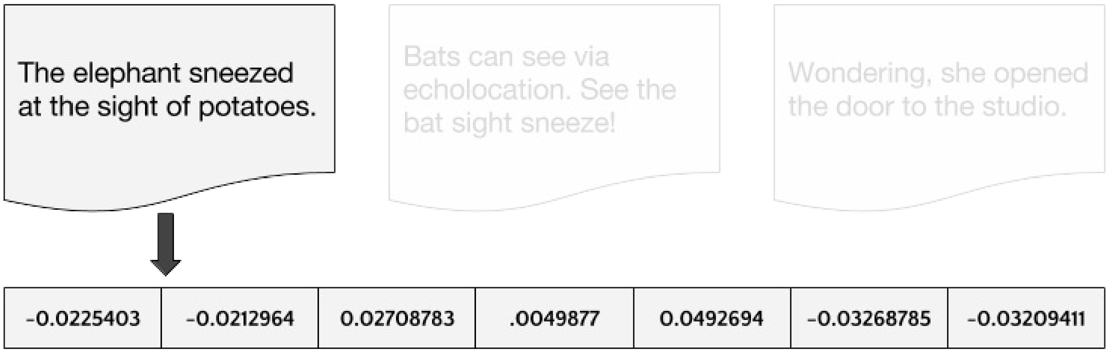
This will save the lexicon as a text-delimited text file, sorted lexicographically, and the TF-IDF model as a pickled sparse matrix. Note that the `Dictionary` object can also be saved more compactly in a binary format using its `save` method, but `save_as_text` allows easy inspection of the dictionary for later work. To load the models from disk:

```
lexicon = gensim.corpora.Dictionar.load_from_text('lexicon.txt')
tfidf = gensim.models.TfidfModel.load('tfidf.pkl')
```

One benefit of term frequency-inverse document frequency is that it naturally addresses the problem of *stopwords*, which are likely to be common across all documents in the corpus, and thus will accrue very small weights under this encoding scheme. This biases the TF-IDF model towards moderately rare words - that is words that appear in a few documents but not very commonly. Generally speaking this means that the most relevant terms to a document are the most informative features in our model. As a result TF-IDF is widely used for bag-of-words models, and is an excellent starting point for most text analytics.

Distributed Representation

The previous encoding schemes aim to represent text locally, as a series of sparse vectors encoded via a dictionary of words that either appear or do not appear in the document, possibly with some degree of frequency. However, it is also possible to encode text along a continuous scale with a distributed representation. As shown in Figure 2-6, this means that the resulting document vector is not a simple mapping from token position to token score. Instead the document is represented in a feature space that has been trained to represent word similarity based on their context. The complexity of this space (and the resulting vector length) is the product of how that representation is trained and not directly tied to the document itself.



The elephant sneezed at the sight of potatoes.

Bats can see via echolocation. See the bat sight sneeze!

Wondering, she opened the door to the studio.

-0.0225403	-0.0212964	0.02708783	.0049877	0.0492694	-0.03268785	-0.03209411

Figure 2-6. Distributed Representation

Because local document vectors such as those we've described thus far have non-negative elements, two vectors with a cosine distance of zero are orthogonal (e.g. they share no elements, terms in this case). Orthogonal vectors are not similar and it is very difficult to compare them. By using a distributed representation, documents will no longer be orthogonal — even if they do not share terms. As a result the similarity of documents in this vector space is more easily encoded.

The distributed representation implemented by Gensim, *doc2vec*[3] is an extension of the word embedding model called *word2vec* wherein documents are encoded together rather than single words in context. The word2vec algorithm was created by a team of researchers at Google, led by Tomáš Mikolov who is well known for his work in recurrent neural networks for language processing. The word2vec algorithm trains word representations based on either a continuous bag of words (CBOW) or skip-gram models, such that words are embedded in space along with similar words based on their context. In order to learn the optimal vectors for these embeddings, a two-layer recurrent neural network is used.

The doc2vec algorithm proposes a *paragraph vector* - an unsupervised algorithm that learns fixed-length feature representations from variable length documents. This representation inherits the semantic properties of words such that words such as "red" and "colorful" are more similar to words like "river". Moreover, the paragraph vector takes into consideration the ordering of words within a narrow context, similar to an n-gram model. The combined result is much more effective than a bag-of-words or bag-of-n-grams models because it generalizes better and has a lower dimensionality but still is of a fixed length so can be used in common machine learning algorithms.

Neither NLTK nor Scikit-Learn provide an implementation of either the word2vec or doc2vec algorithms. Gensim does provide an implementation, written in C that allows users to train both word2vec and doc2vec models on custom corpora. However, Gensim also conveniently comes with a model that is pre-trained on the Google news corpus, a corpus that is general enough for many language applications. In order to use that model, you have to download the model bin file, which clocks in at a whopping 1.5GB! Because doc2vec is unsupervised, we will simply train our own model as follows:

```
from gensim.models.doc2vec import TaggedDocument, Doc2Vec

corpus = [list(tokenize(doc)) for doc in corpus]
corpus = [
    TaggedDocument(words, ['d{}'.format(idx)])
    for idx, words in enumerate(corpus)
]

model = Doc2Vec(corpus, size=5, min_count=0)
print(model.docvecs[0])
# [ 0.01797447 -0.01509272  0.0731937   0.06814702 -0.0846546 ]
```

As in previous models, Gensim expects documents to be preprocessed and tokenized in advance; this allows us to use complex tokenization schemes for example n-gram models, collecting phrases as single words, or performing other types of semantic preprocessing. The input to the `Doc2Vec` model is a list of `TaggedDocument` objects, which extends the idea of a `LabeledSentence` and in turn the distributed representation of word2vec. These concepts allow us to capture both semantic similarity and word context for variable length documents.

`TaggedDocument` objects consist of *words* and *tags*. The easiest method is to instantiate the tagged document with the list of tokens along with a single tag, one which uniquely identifies the instance. In this example, we've simply labeled each document as `"d{}".format(idx)`, e.g. `d0`, `d1`, `d2` and so forth. Once we have a list of tagged documents, we can instantiate the Doc2Vec model and specify the size of the vector (usually not as low a dimensionality as 5, we selected such a small number for demonstration only) as well as the minimum count which ignores all tokens that have a frequency less than that number. In our case, we set it to zero to ensure we consider all tokens, but generally this is set between 3 and 5 depending on how much information the model needs to capture. Once instantiated, an unsupervised neural network is trained to learn the vector representations, which can then be accessed via the `docvecs` property.

Distributed representations will dramatically improve results over TF-IDF models when used correctly. The model itself can be saved to disk and retrained in an active fashion, making it extremely flexible for a variety of use cases. However, on larger corpora, training can be slow and memory intensive, and it might not be as good as a TF-IDF model with Principal Component Analysis (PCA) or Singular Value Decomposition (SVD) applied to reduce the feature space if used incorrectly. In the end, however, this representation is breakthrough work that has led to a dramatic improvement in text processing capabilities of data products in recent years.

Benefits and Limitations of Vector Encoding

In this section we've presented four types of vector encoding: frequency, one-hot, TF-IDF, and distributed representations. Each of these methods has performed the central task of converting a text document into a numeric representation that can be learned on. However, as we have discussed in each of the sections, each representation has a variety of strengths and weaknesses for different approaches. We have found that it is often best to select an encoding scheme based on the problem at hand; certain methods substantially outperform others for certain tasks.

For example, for recurrent neural network models, it is often better to use one-hot encoding, but to divide the text space. For example, one might create a combined vector for the document summary, document header, body, etc. Frequency encoding should be normalized, but different types of frequency encoding can benefit probabilistic methods like Bayesian models. TF-IDF is an excellent general purpose encoding and is often used first in modeling, but can also cover a lot of sins. Distributed representations are the new hotness, but are performance intensive and difficult to scale.

Machine learning requires a vector representation, and thus feature extraction of this form is required. The advantage is that we can embed complex representations like TF-IDF into the vector form. Furthermore feature vectorization brings us closer towards a token-concept mapping without pre-biased rules.

On the other hand, vectorization does lead to some problems. Bag-of-words models have a very high dimensionality, meaning the space is extremely sparse, leading to difficulty generalizing the data space. Word order, grammar, and other structural features are natively lost, and it is difficult to add knowledge (lexical resources, ontological encodings) to the learning process. Local encodings (e.g. non-distributed representations) require a lot of samples which could lead to over-training or underfit, but distributed representations are complex and add a layer of "representational mysticism".

In the end, much of the work for language aware applications comes from domain specific feature analysis; not just simple vectorization. In [Link to Come], we will explore some more complex methods for feature analysis and feature exploration that will assist in fine tuning our vector-based models to achieve better results. Nonetheless, using simple models that consider only of word frequencies will often be very successful; in our experience, a pure bag-of-words model works about 85% of time. In the next section, we will demonstrate how to integrate this kind of simple model into a complete pipeline in order to illustrate how to construct the core of a fully operational textual machine learning application.

Feature Extraction

In the first part of the chapter we considered a workflow for machine learning that centers around the model selection triple. In the second part of the chapter we discussed the first step toward machine learning on text: vectorization. At first glance it seems that vectorization is the feature selection part of the model selection triple, however it is only the first step to enabling feature selection and analysis on a larger scale.

The model selection triple is a method of defining a *search* - that is the belief that there exists some combination of features, a machine learning algorithm, and hyperparameters of that algorithm that will lead to a fitted model that is optimally predictive. Because the search space is large, automatic techniques for optimization are not sufficient. Instead, data scientists use intuition and generalizing techniques to fit the best possible model, usually trying several and comparing them. This style of machine learning fits very well with Scikit-Learn whose primary contribution is an API for machine learning. The API means that models and transformers can be easily swapped in and out, tested and compared until a sufficient solution is reached.

In order to engage this style of model search, though, some repeatable mechanism is required to reliably extract the same features from our corpus of documents. This mechanism is required with text in particular, because the feature extraction methodology is what allows us to apply our predictive models to new instances of text. If we don't vectorize our documents in the same exact manner, we will end up with wrong or, at the very least, unintelligible results.

The Scikit-Learn `Pipeline` object is the solution to this dilemma. `Pipeline` objects enable us to integrate a series of transformers that combine normalization, vectorization, and feature analysis into a single, well-defined mechanism. As shown in Figure 2-7 `Pipeline` objects move data from a loader (an object that will wrap our `CorpusReader` from Chapter 1) into feature extraction mechanisms to finally an estimator object that implements our predictive models. Pipelines are directed acyclic graphs (DAGs) that can be simple linear chains of transformers to arbitrarily complex branching and joining paths implemented by the `FeatureUnion` transformer.

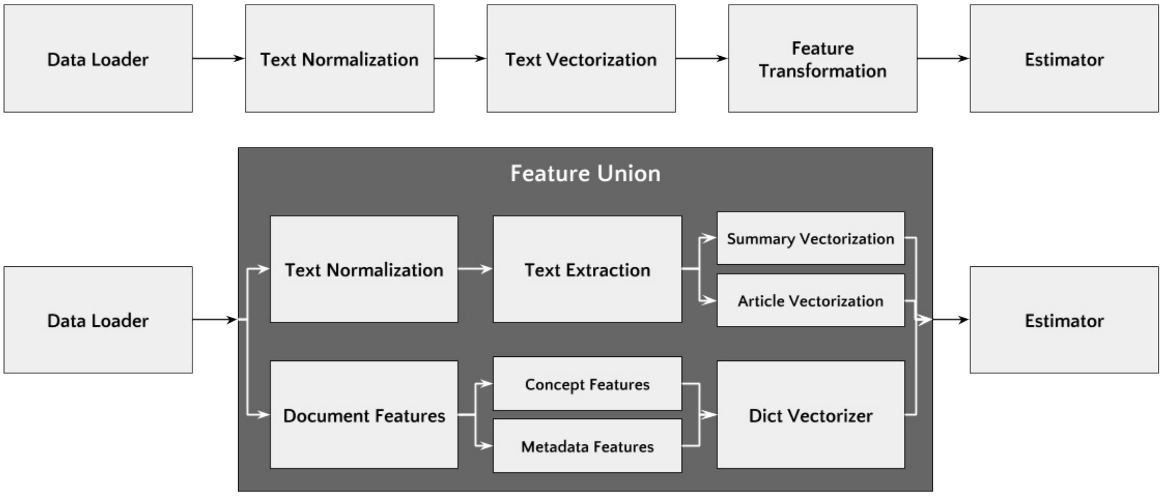


Figure 2-7. Pipelines for Text Vectorization and Feature Extraction

`Pipeline` objects are a Scikit-Learn specific utility, however they are also the critical integration point with NLTK and Gensim. Because Scikit-Learn provides so many estimators and model families, it is the primary choice for the machine learning implementations (with the exception of topic modeling with Latent Dirichlet Allocation, or LDA, and Latent Semantic Analysis, or LSA, which are the fundamental learning mechanisms of Gensim). NLTK is responsible for the preprocessing and linguistic feature extraction (such as syntax analysis) as well as corpus management. By using the `Transformer` API as discussed earlier in the chapter, we can wrap NLTK `CorpusReader` objects and other linguistic components. Gensim is responsible for vectorization, and we will similarly create transformers to wrap the Gensim utilities and estimators that Scikit-Learn does not have. Scikit-Learn is responsible for the integration via Pipelines, utilities like cross-validation, and the many models we will use from naive Bayes to logistic regression.

Pipeline Basics

The purpose of a `Pipeline` is to chain together multiple estimators representing a fixed sequence of steps into a single unit. All estimators in the pipeline, except the last one, must be transformers — that is implement the `transform` method, while the last estimator can be of any type, including predictive estimators. Pipelines provide convenience, that is `fit` and `transform` can be called for single inputs across multiple objects at once. Pipelines also provide a single interface for grid search of multiple estimators at once. Most importantly, pipelines provide *operationalization* of text models by coupling a vectorization methodology with a predictive model.

Pipelines are constructed by describing a list of `(key, value)` pairs where the `key` is a string that names the step and the `value` is the estimator object. Pipelines can be created either by using the `make_pipeline` helper function, which automatically determines the names of the steps, or by specifying them directly. Generally, it is better to specify the steps directly to provide good user documentation, whereas `make_pipeline` is used more often for automatic pipeline construction. Here is an example to create a one-hot encoder that vectorizes text in advance of a Bayesian model:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import Binarizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer


model = Pipeline([
    ('count', CountVectorizer()),
    ('onehot', Binarizer()),
    ('bayes', 'MultinomialNB()'),
])
```

The Pipeline can then be used as a single instance of a complete model. Calling `model.fit` is the same as calling `fit` on each estimator in sequence, transforming the input and passing it on to the next step. Other methods like `fit_transform` behave similarly. The pipeline will also have all the methods the last estimator in the pipeline has. If the last estimator is a transformer, so too is the pipeline. If the last estimator is a classifier, as in the example above, then the pipeline will also have `predict` and `score` methods so that the entire model can be used as a classifier.

The estimators in the pipeline are stored as a list, and can be accessed by index. For example `model.steps[1]` returns the tuple `(`*`onehot,`*
`Binarizer(copy=True, threshold=0.0))`. Common usage though is to identify estimators by their names using the `named_steps` dictionary property of

the `Pipeline` object. The easiest way to access the predictive model is to use `model.named_steps["bayes"]` and fetch the estimator directly.

Grid search can be implemented to modify the parameters of all estimators in the Pipeline as though it were a single object. In order to access the attributes of estimators, you would use the `set_params` or `get_params` pipeline methods with a dunderscore representation of the estimator and parameter names as follows: `estimator__parameter`. Let's say that we want to one-hot encode terms that appear at least three times in the corpus; we could modify the `Binarizer` as follows:

```
model.set_params(onehot__threshold=3.0)
```

Using this principle, we could define a grid search by defining the search parameters grid using the dunderscore parameter syntax. Consider the following grid search to determine the best one-hot encoded Bayesian text classification model:

```
from sklearn.model_selection import GridSearchCV

search = GridSearchCV(model, param_grid={
    'count__analyzer': ['word', 'char', 'char_wb'],
    'count__ngram_range': [(1,1), (1,2), (1,3), (1,4), (1,5), (2,3)],
    'onehot__threshold': [0.0, 1.0, 2.0, 3.0],
    'bayes__alpha': [0.0, 1.0],
})
```

The search nominates three possibilities for the `CountVectorizer` analyzer parameter (creating n-grams on word boundaries, character boundaries, or only on characters that are between word boundaries), and several possibilities for the n-gram ranges to tokenize against. We also specify the threshold for binarization, meaning that the n-gram has to appear a certain number of times before it's included in the model. Finally the search specifies two smoothing parameters (the `bayes_alpha` parameter): either no smoothing (add 0.0) or Laplacian smoothing (add 1.0). The grid search will instantiate a pipeline of our model for each combination of features, then use cross validation to score the model and select the best combination of features (in this case, the combination that maximizes the F1 score).

Pipelines do not have to be simple linear sequences of steps, in fact they can be arbitrarily complex directed acyclic graphs through the implementation of *feature unions*. The `FeatureUnion` object combines several transformer objects into a new, single transformer, similar to the `Pipline` object. However, instead of fitting and transforming data in sequence through each transformer, they are instead evaluated independently and the results are *concatenated* into a composite vector.

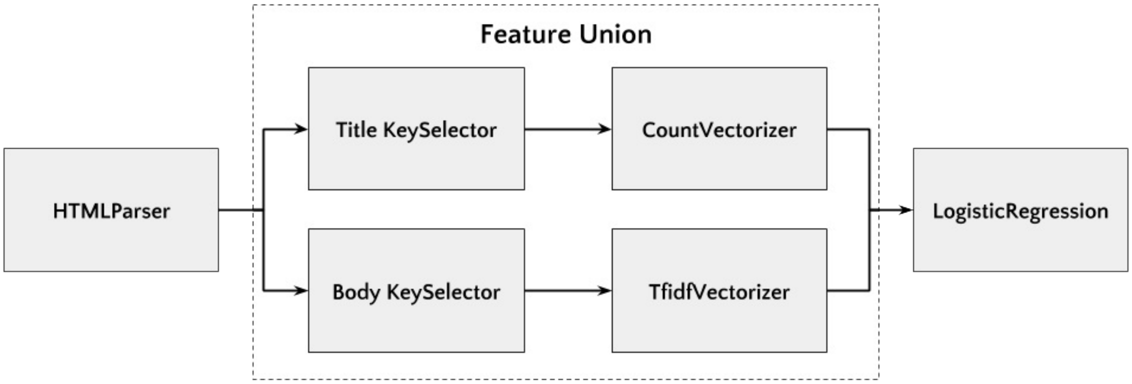


Figure 2-8. Feature Unions for Branching Vectorization

Consider the example shown in Figure 2-8. We might imagine an HTML parser transformer that uses Beautiful Soup or an XML library to parse the HTML and return a dictionary that contains the title and body of each document. When passed into the feature union, we need some method to select either the title or the body, then pass that to the appropriate vectorization method. Using frequency encoding on the title is more sensible than TF-IDF since titles are relatively small, but TF-IDF makes more sense for body text. The feature union then concatenates the two resulting vectors such that our decision space ahead of the logistic regression separates word dimensions in the title from word dimensions in the body.

`FeatureUnion` objects are similarly instantiated as `Pipeline` objects with a list of `(key, value)` pairs where the `key` is the name of the transformer, and the `value` is the transformer object. There is also a `make_union` helper function that can automatically determine names and is used in a similar fashion to the `make_pipeline` helper function — for automatic or generated pipelines. Estimator parameters can also be accessed in the same fashion, and to implement a search on a feature union, simply nest the dunderscore for each transformer in the feature union. Given the unimplemented transformers mentioned above, we can construct our pipeline as follows:

```
from sklearn.pipeline import FeatureUnion
from sklearn.linear_model import LogisticRegression

model = Pipeline([
    ('parser', HTMLParser()),
    ('text_union', FeatureUnion(

        transformer_list = [

            ('title_feature', Pipeline([
                ('title_select', KeySelector('title')),
                ('title_vect', CountVectorizer()),
            ])),

            ('body_feature', Pipeline([
                ('body_select', KeySelector('body')),
                ('body_vect', TfidfVectorizer()),
            ])),
        ],
        transformer_weights= {
            'title_feature': 0.6,
            'body_feature': 0.4,
        }
    )),
    ('clf', LogisticRegression()),
])
```

Note that the `KeySelector` and `HTMLParser` objects are currently unimplemented, though we will explore similar objects later in this section. The feature union is fit in sequence with respect to the rest of the pipeline, however each transformer within the feature union is fit *independently*, meaning that each transformer sees the same data as the input to the feature union. During transformation, each transformer is applied in parallel and the vectors that they output are concatenated together into a single larger vector, which can be optionally weighted. In the example above, we are weighting the `title_feature` transformer slightly more than the `body_feature` transformer. Using combinations of custom transformers, feature unions, and pipelines, it is possible to define incredibly rich feature extraction and transformation in a repeatable fashion.

In the rest of this section we will explore the application of pipelines, feature unions, and more with custom transformers that allow us to use NLTK and Gensim effectively within a Scikit-Learn context. We will start to build a library that will allow us to load data from our corpus reader objects, and define complex modeling. However, by collecting our methodology into a single sequence, we can repeatably apply the transformations, particularly on new documents when we want to make predictions in a production environment.

Corpus Loader

In Chapter 1 we built a `CorpusReader` that streams raw HTML data from disk so that we can preprocess it in parallel, seek to specific locations, and filter by file id. As we learned in that chapter, this component is a critical utility for our application because raw text data must be preprocessed to enable meaningful analysis and modeling, and because preprocessing is non-trivial in terms of computation time and data management. However, while the `CorpusReader` gives us memory-safe streaming access to our corpus, it is not completely ready to send into a vectorization transformer to start modeling, and the reason for this is *cross-validation*.

To conduct supervised machine learning, we can compare models trained on the same data set through cross-validation. This method divides our corpus into two smaller data sets: a training and test set. Models are fit on the training data and then evaluated on the unseen test data. This allows us to compare models without bias and prevents overfit, indicating that the model is able to generalize to unseen inputs. However, even when data is split into training and test sets, there is a potential that certain chunks of the data will have more variance than others. To handle this case, we shuffle our dataset, and divide into `k` train and test splits, averaging the scores for each split.

Note that if our natural language processing application didn't have to do any machine learning work, the `CorpusReader` would be enough; after preprocessing, the text could go directly into a transformer. However, if we use the `sklearn.cross_validation.train_test_split` function directly on the reader, the data would be loaded into memory all at once, leaving us precious little RAM for computation if any at all. Although eventually we have to load a matrix representation of our entire corpus into memory, at that point documents will be described as numeric sparse matrices, which are much more compact! Therefore in order to perform cross-validation on text data in a memory-safe manner, we will need to be able to create splits and retrieve documents by file id and category labels on demand.

It is essential to get into the habit of using cross validation to ensure that our models perform well, particularly when engaging the model selection process. We consider it so important to applied text analytics that we start by creating a `CorpusLoader` object that wraps a `CorpusReader` in order to provide streaming access to `k` splits! We'll construct the base class as follows:

```python
from sklearn.cross_validation import KFold


class CorpusLoader(object):

    def __init__(self, corpus, folds=None, shuffle=True):
        self.n_docs = len(corpus.fileids())
        self.corpus = corpus
        self.folds  = folds

        if folds is not None:
            # Generate the KFold cross validation for the loader.
            self.folds = KFold(self.n_docs, folds, shuffle)

    @property
    def n_folds(self):
        """
        Returns the number of folds if it exists; 0 otherwise.
        """
        if self.folds is None: return 0
        return self.folds.n_folds
```

In order to stay in line with the Scikit-Learn API, we'll create our loader as a class. The `CorpusLoader` object is instantiated with a `CorpusReader`, the number of folds, and whether or not to shuffle the corpus, which is true by default. If folds is not `None`, we instantiate a Scikit-Learn `KFold` object that knows how to partition the corpus by the number of documents and specified folds. We also create a property so that we can easily look up the number folds specified by the loader. The next step is to add a method that will allow us to access a listing of file ids by fold ID for either the train or the test splits.

```
def fileids(self, fold=None, train=False, test=False):

    if fold is None:
        # If no fold is specified, return all the fileids.
        return self.corpus.fileids()

    # Otherwise, identify the fold specifically and get the train/test idx
    train_idx, test_idx = [split for split in self.folds][fold]

    # Now determine if we're in train or test mode.
    if not (test or train) or (test and train):
        raise ValueError(
            "Please specify either train or test flag"
        )

    # Select only the indices to filter upon.
    indices = train_idx if train else test_idx
    return [
        fileid for doc_idx, fileid in enumerate(self.corpus.fileids())
        if doc_idx in indices
    ]
```

This rather daunting method returns a list of document ids, filtering by fold and by test or train. The first step is to check if the fold is None, if so return all documents. If a fold is specified (e.g. if we do 12 part cross-validation and request fold 3), then we create a list of train and test indices for each fold, and select the correct split. This method requires that either train or test is passed in, and so the next step is to check that one of the flags is set, which we can then use to gather the correct set of indices. Finally, we filter the file ids in the corpus, returning only those documents that match the index of the split. Once we have the file ids, we can return the documents and labels respectively as follows.

```
def documents(self, fold=None, train=False, test=False):
    for fileid in self.fileids(fold, train, test):
        yield list(self.corpus.docs(fileids=fileid))

def labels(self, fold=None, train=False, test=False):
    return [
        self.corpus.categories(fileids=fileid)[0]
        for fileid in self.fileids(fold, train, test)
    ]
```

Both the documents and labels function has the same signature as our file ids, and in fact the first thing they do is filter the corpus based on the fold and whether or not we’re accessing the train or test split. The `documents()` method returns a generator to provide memory safe access to the documents in our corpus, and yields a list of tagged tokens for each file id in the split, one document at a time. The `labels()` method uses the `corpus.categories()` to look up the label from the corpus and returns a list of labels, one per document. Usage of the `CorpusLoader` is as follows:

```
FOLDS   = 12
corpus  = PickledCorpusReader("/path/to/corpus")
loader  = CorpusLoader(corpus, folds=FOLDS)

scores  = []
for fold in range(FOLDS):
    # Get the train/test splits for the first fold
    X_train = loader.documents(fold, train=True)
    y_train = loader.labels(fold, train=True)

    X_test  = loader.documents(fold, test=True)
    y_test  = loader.labels(fold, test=True)

    # Fit a model and score
    model.fit(X_train, y_train)
    score = model.score(X_test, y_test)
    scores.append(score)
```

This methodology creates a 12-fold cross-validation that fits a model 12 times and collects 12 scores, evaluating on unseen data. These scores can then be averaged and used to compare models. The `CorpusLoader` object is not a transformer, however it is the first step to start machine learning, as it allows us to load documents from disk and add send them into the Pipeline, therefore it is often the first step in transformation pipelines.

Text Normalization

Now that we have the means with which to load data from our corpus in a memory safe manner, splitting into train/test splits for cross-validation, we can start to build transformers for our feature extraction pipeline. Transformers are *fit* so that the transformer can set an internal state based on input data and, once fit, can transform data (documents or the output of other transformers) using that state, returning a new data set. One of the first transformers that is applied (and often included in vectorization methodologies) is a text *normalization* transformer.

Normalization is another overloaded machine learning term with a few different meanings. In the case of text, normalization is intended to reduce the number of features (tokens, in a bag-of-words model) by eliminating or combining different tokens into a single class. Consider case, for example, the tokens *`friend`*, *`Friend`*, and *`FRIEND`* all have the same meaning (possibly) and therefore can be reduced to the single token, *`friend`*. Punctuation and stopwords (high frequency, structural words) may not convey much information and can therefore be eliminated. Further, words with affixes or morphologic transformations to indicate gender, plurality, tense, etc. can be collapsed into a single word class, e.g. the verbs *`throwing`*, *`throws`*, *`threw`*, and *`thrown`* are all *`throw`*.

CAUTION

The act of text normalization should be optional and applied carefully because the operation is destructive in that it removes information. Case, punctuation, stopwords, and varying word constructions are all critical to understanding language. Some models may require indicators such as case, for example a named entity recognition classifier, because in English, proper nouns are capitalized.

Many model families suffer from “the curse of dimensionality”, that is as the feature space increases in dimensions, the data becomes more sparse and less informative to the underlying decision space. Considering that any meaningfully robust corpus (such as the one we constructed in Chapter 1) will have a large vocabulary, and every vocabulary word is a feature, text analysis is extremely high dimensional. Text normalization reduces the number of dimensions, decreasing sparsity. Besides the simple filtering of tokens (removing punctuation and stopwords), there are two primary methods for text normalization: *stemming* and *lemmatization*. Both methodologies work by returning a single word class for a group of word forms thereby reducing the number of tokens in the corpus as shown in Table 2-1.

Table 2-1. Stemming vs. Lemmatization

Token	Stem	Lemma
gardener	garden	gardener
running	run	run
threw	threw	throw
foxes	fox	fox
geese	geese	goose

Stemming uses a series of rules (or a model) to slice a string to a smaller substring. The goal is to remove word affixes (particularly suffixes) that modify meaning, for example removing an *`s`* or *`es`* which generally indicates plurality in latin languages. There are several different stemmer implementations in NLTK's `nltk.stem` module. The original `PorterStemmer` and the more aggressive `LancasterStemmer` are English-only and there are other language-specific stemmer options as well. The modern option, the `SnowballStemmer` provides support for multiple languages and is used by default as follows:

```
from nltk import word_tokenize
from nltk.stem.snowball import SnowballStemmer

# Initialize the stemmer for English
stemmer = SnowballStemmer('english')
text = "The geese flying through foggy clouds passed bunnies in their dens."
tokens = word_tokenize(text)

# Create the list of stems
stems = [stemmer.stem(token) for token in tokens]
# the gees fli through foggi cloud pass bunni in their den .
```

Lemmatization on the other hand uses a dictionary to look up every token and returns the canonical "head" word in the dictionary, called a lemma. Because it is looking up tokens from a ground truth, it can handle irregular cases as well as handle tokens with different parts of speech. For example, the verb *`gardening`* should be lemmatized to *`to garden`*, while the nouns *`garden`* and *`gardener`* are both different lemmas. Stemming would capture all of these tokens into a single *`garden`* token. NLTK uses the WordNet lexicon for lookups and lemmatization as follows:

```
from nltk import pos_tag
from nltk.corpus import wordnet as wn
from nltk.stem.wordnet import WordNetLemmatizer

tag_map = {
    'N': wn.NOUN,
    'V': wn.VERB,
    'R': wn.ADV,
    'J': wn.ADJ,
}

# Instantiate the lemmatizer
lemmatizer = WordNetLemmatizer()

# Part of speech tag the text and map to wordnet tags
tagged = [(token, tag_map.get(tag[0])) for token, tag in pos_tag(tokens)]
lemmas = [lemmatizer.lemmatize(token, tag) for token, tag in tagged]
# The goose fly through foggy cloud pass bunny in their den .
```

Stemming and lemmatization have their advantages and disadvantages. Because it only requires us to splice word strings, stemming is faster. Lemmatization, on the other hand, requires a lookup to a dictionary or database, and uses part of speech tags to identify a word's root lemma, making it noticeably slower than stemming, but also more effective.

NOTE

Normalization techniques tend to be language specific and may also require methods to remove verb tense, gender, and other affixes. In Hebrew for instance, we can use consonantal templates to group similar words, whereas in Chinese, we can group pictographically-similar symbols together.

In order to add a text normalizing methodology to a Scikit-Learn Pipeline, we must write a custom transformer that puts these pieces together. We'll start by creating the base transformer class as follows:

```
import unicodedata
from sklearn.base import BaseEstimator, TransformerMixin
```

```
class TextNormalizer(BaseEstimator, TransformerMixin):

    def __init__(self, language='english'):
        self.stopwords  = set(nltk.corpus.stopwords.words(language))
        self.lemmatizer = WordNetLemmatizer()

    def is_punct(self, token):
        return all(
            unicodedata.category(char).startswith('P') for char in token
        )

    def is_stopword(self, token):
        return token.lower() in self.stopwords
```

The `TextNormalizer` takes as input a language, which is used to load the correct stopwords from the NLTK corpus. We could customize the `TextNormalizer` to allows uses to choose between stemming and lemmatization, and pass the language into the `SnowballStemmer`. For filtering extraneous tokens, we create two methods. The first, `is_punct()`, checks if every character in the token has a unicode category that starts with *P* (for punctuation), the second, `is_stopword()` determines if the token is in our set of stopwords. We can then add a `normalize()` method as follows:

```
    def normalize(self, document):
        return [
            self.lemmatize(token, tag).lower()
            for paragraph in document
            for sentence in paragraph
            for (token, tag) in sentence
            if not self.is_punct(token) and not self.is_stopword(token)
        ]
```

The `normalize()` method takes a single document that is composed of a list of paragraphs, which are lists of sentences, which are lists of `(token, tag)` tuples — the data format that we preprocessed raw HTML to in Chapter 1. This method applies the filtering functions to remove unwanted tokens, and then lemmatizes them. The `lemmatize()` method is as follows:

```
    def lemmatize(self, token, pos_tag):
        tag = {
            'N': wn.NOUN,
            'V': wn.VERB,
            'R': wn.ADV,
            'J': wn.ADJ
        }.get(pos_tag[0], wn.NOUN)

        return self.lemmatizer.lemmatize(token, tag)
```

The first step in lemmatization is to convert the Penn Treebank part of speech tags that are the default tag set in the `nltk.pos_tag` function to WordNet tag. By observing that in Penn Treebank all noun classes start with “N”, verbs with “V”, adverbs with “R”, and adjectives with “J”, we can create a simple mapping from the first character of the part of speech tag to the wordnet mappings, selecting nouns by default. Finally, we must add the `Transformer` interface so that we can add this class to a Scikit-Learn pipeline:

```
    def fit(self, X, y=None):
        return self

    def transform(self, documents):
        for document in documents:
            yield self.normalize(document)
```

By utilizing our `CorpusLoader`, we can directly pass documents into the `transform()` method, and are returned a normalized data structure for downstream processing that has fewer unique tokens than the original data set. Note, however, that this is only one methodology that utilizes NLTK very heavily. Other options include removing tokens that appear above or below a particular count threshold, removing stopwords then only selecting the first five to ten thousand most common words. Instead of doing a rules based elimination of stopwords, we could simply compute the cumulative frequency and only take words that contain 10-50% of the cumulative frequency distribution. These mtehods would allow us to ignore both the very low frequency *hapaxes* (terms that appear only once) and the most common words, enabling us to identify only the most predictive terms in the corpus.

An alternate approach is to perform dimensionality reduction with principal component analysis (PCA) or singular value decomposition (SVD), to reduce the feature space to a specific dimensionality (e.g. five or ten thousand dimensions) based on word frequency. These transformers would have to be applied following a vectorizer transformer, and would have the effect of merging together words that are similar into the same vector space.

Vectorization with Gensim

Now that we have a `CorpusLoader` that can read our data from disk as well as a `TextNormalizer` transformer that will allow us to target very specific word classes, the next phase of our pipeline is vectorization. Vectorization is a very specific dividing line from which we move from functions that apply to text to functions that apply to numeric space; in [Link to Come] we will discuss more text feature extraction. Moving to numeric space, we could apply any of the vectorization methods from the previous section, however if we want to use a non-Scikit-Learn vectorization method we will have to create a custom transformer.

Gensim vectorization techniques are an interesting opportunity because Gensim corpora can be saved and loaded from disk in such a way as to remain decoupled from the pipeline. Scikit-Learn vectorizers must be pickled to disk to be used again, but they are strongly tied to the transformer methodology. To use Gensim vectorization instead, we can build a transformer as follows:

```
import os

from gensim.corpora import Dictionary
from gensim.matutils import sparse2full


class GensimVectorizer(BaseEstimator, TransformerMixin):

    def __init__(self, path=None):
        self.path = path
        self.id2word = None

        self.load()

    def load(self):
        if os.path.exists(self.path):
            self.id2word = Dictionary.load(self.path)

    def save(self):
        self.id2word.save(self.path)
```

This transformer simply wraps a gensim `Dictionary` object that will be generated during `fit()` and whose `doc2bow` method is used during `transform()`. The `Dictionary` object (like the `TfidfModel`) can be saved and loaded from disk, so our transformer utilizes that methodology by taking a path on instantiation. If a file exists at that path, it is loaded immediately. Additionally, a `save()` method allows us to write our `Dictionary` to disk, which we can do in `fit()` as follows:

```
def fit(self, documents, labels=None):
    self.id2word = Dictionary(documents)
    self.save()
    return self

def transform(self, documents):
    for document in documents:
        docvec = self.id2word.doc2bow(document)
        yield sparse2full(docvec, len(self.id2word))
```

The `fit()` method constructs the `Dictionary` object by passing already tokenized and normalized documents to the `Dictionary` constructor. The `Dictionary` is then immediately saved to disk so that the transformer can be loaded without requiring a refit. The `transform()` method uses the `Dictionary.doc2bow` method which returns a *sparse* representation of the document as a list of `(token_id, frequency)` tuples. This representation will not work with Scikit-Learn, however, so we utilize a Gensim helper function, `sparse2full` to convert the sparse representation into a Numpy array.

It is easy to see how the vectorization methodologies that we discussed earlier in the chapter can be wrapped by Scikit-Learn transformers. This gives us more flexibility in the approaches we take, while still allowing us to leverage the machine learning utilities in each library. We leave it to the reader to investigate TF-IDF and distributed representation transformers that are implemented in the same fashion.

Document Level Features

We have so far explored a simple bag-of-words model that represents documents as a vector where each position in the vector represents a word and its value indicates the relationship of the word to the document. This representation allows us to capture documents in a numeric space to which we can apply machine learning models, and with normalization this model is sufficient to create general models that are effective approximately 85% of the time. Said another way, a general rule of thumb for the bag-of-words model is that they can achieve F1 scores for classifiers of around 0.85 (anecdotally).

While those scores are good, they are often insufficient for applications because errors are particularly noticeable when it comes to text. When we think about the model selection triple, it is apparent that we can use hyperparameter tuning and search methods to improve our performance but this typically only boosts performance by a few points. We can also enhance algorithm selection by using ensembles of weaker models that together provide stronger answers. However, by far the most effective method of boosting performance is in the final element of the MST - feature analysis.

Extending the bag-of-words model means including document, corpus, or discourse level meta-features or applying other mappings from other analyses. For example, characteristics like the size of a document or the shape of its paragraphs can be a good indicator of what kind of document it is; legal briefs are structurally very different from speeches, news stories, and medical reports. In addition, the size and shape of sentences, the range of vocabulary words used, the name of the author, etc. are frequently informative and can be used to identify the complexity of the text or the context (informal vs. formal speech).

To explore the extraction of these types of document level features, and put everything together with `FeatureUnion` objects, we will create a transformer that returns a statistics dictionary on a per document basis:

```
from collections import Counter


class TextStats(BaseEstimator, TransformerMixin):

    def fit(self, documents, labels=None):
        vocabulary = Counter(
            token for paragraph in document
            for sentence in paragraph
            for token,tag in sentence
        )

        self.corpus_vocab = len(vocabulary)
        self.corpus_count = sum(vocabulary.items())
        return self

    def transform(self, documents):
        for document in documents:
            # Collect token and vocabulary counts
            counts = Counter(
                item[0] for para in document for sent in para for item in sent
            )

            # Yield structured information about the document
            yield {
                'paragraphs': len(document),
                'sentences': sum(len(para) for para in document),
                'words': sum(counts.values()),
                'vocab': len(counts),
            }
```

The `TextStats` transformer uses a `Counter` object to count the frequency of tokens and obtain vocabulary in an efficient manner. In the `fit()` method we count the vocabulary and tokens for the entire corpus, which would allow us to compute per-document ratios or perform other normalization. The `transform()` method returns a generator of dictionaries with per-document statistics. Often applications will have many, very-specific types of these transformers that can be added or combined at will.

Building Models

In this section we have built a few techniques for extracting features including methods for vectorizing documents, extracting only valuable tokens, and identifying the structural features of documents. At this point, we need to combine our transformers into a single, repeatable pipeline of transformation that can be applied both in the build and operation phases of machine learning, in such a way that we can engage the model selection triple to find the best possible model.

The first step is to create a feature extraction pipeline using the `Pipeline` and `FeatureUnion` objects to meaningfully transform preprocessed text documents into numeric representations. In order to add our feature methodology to multiple models, we will create a function that constructs a per-estimator pipeline and returns it:

```
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction import DictVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

def construct_pipeline(estimator, **kwargs):
    return Pipeline([
        # Create a Feature Union of Text Stats and Bag of Words
        ('features', FeatureUnion(
            transformer_list = [

                # Pipeline for pulling document structure features
                ('stats', Pipeline([
                    ('stats', TextStats()),
                    ('vect', DictVectorizer()),
                ])),

                # Pipeline for creating a bag of words TF-IDF vector
                ('bow', Pipeline([
                    ('tokens', TextNormalizer()),
                    ('tfidf',  TfidfVectorizer(
                        tokenizer=identity, preprocessor=None, lowercase=False
                    )),
                    ('best', TruncatedSVD(n_components=10000)),
                ])),

            ],

            # weight components in feature union
            transformer_weights = {
                'stats': 0.15,
                'bow': 0.85,
            },
        )),

        # Append the estimator to the end of the pipeline
        ('model', estimator(**kwargs)),
    ])
```

Note the numerous transformers that we include in for feature extraction; there is a balance between the amount of work each transformer does and the modularity that allows you to combine transformers in meaningful ways. All feature extraction is conducted in the `FeatureUnion`, which passes documents both to a structural transformer and a bag-of-words extractor. Document statistics are an inner pipeline that includes a `DictVectorizer` to manage the output from the

TextStats transformer. The *bow* Pipeline uses our text normalization technique and the built in Scikit-Learn vectorizers, with a truncated singular value decomposition to further reduce the dimensionality. The features are combined with different weights, and the model is instantiated and appended at the end.

We can now create a function that builds and evaluates our model using 12 part cross validation. This function accepts a corpus, a model, and model arguments and then returns a fitted model and a score. The function is as follows:

```
def build_model(corpus, estimator, **kwargs):

    # Create a loader for scoring.
    loader = CorpusLoader(corpus, 12)
    scores = []

    # Perform 12-part cross validation
    for fold in range(12):
        # Get the train data sets
        docs_train   = loader.documents(fold, train=True)
        labels_train = loader.labels(fold, train=True)

        # Create and fit the model
        model = create_pipeline(estimator, **kwargs)
        model.fit(docs_train, labels_train)

        # Get the score of the model on the test data
        docs_test   = loader.documents(fold, test=True)
        labels_test = loader.labels(fold, test=True)
        scores.append(model.score(docs_test, labels_test))

    # Build the final model on the entire dataset
    loader = CorpusLoader(corpus, None)
    model  = create_pipeline(estimator, **kwargs)
    model.fit(loader.documents(), loader.labels())

    return model, scores
```

The first thing this function does is create a CorpusLoader that knows how to split the documents into train/test splits. For each fold in our twelve-part cross validation we fit a model on the train data and score on the unseen test data. These scores are appended to a complete list of scores. Finally we fit the model on the entire corpus, using a new loader that doesn't do any splitting. We could create a simple classifier built from our corpus created in Chapter 1 as follows:

```
import pickle
from sklearn.linear_model import LogisticRegression

corpus = PickledCorpusReader('corpus')
model, scores = build_model(corpus, LogisticRegression)

with open('model.pkl', 'wb') as fobj:
    pickle.dump(model, fobj)
```

These few lines of code engage our entire pipeline and most of the code written in the past two chapters. The `PickledCorpusReader` is able to read preprocessed documents from disk in a memory safe fashion. The `build_model()` function uses the `CorpusLoader` to split the documents into train/test splits for cross-validation and evaluation. Those splits are passed into a feature extraction pipeline that uses a `FeatureUnion` and `Pipeline` objects to send data through several transformers and whose end result is a meaningful representation of documents in numeric space. Finally these vectors are passed to a Scikit-Learn estimator, which is fit, evaluated, and saved to disk for operationalization.

Conclusion

Performing computation on natural language requires a methodology that is *flexible* and can respond to change. Machine learning gives us that flexibility by learning from example in order to make predictions on new data. In this chapter, we conducted a whirlwind overview of the requirements of machine learning and how to conduct machine learning on text. This is necessary preparation as we move forward with the rest of the book.

The workflow for machine learning surrounds the model selection triple — that is the meaningful combination of feature analysis, algorithm selection, and hyperparameter tuning. Applied text analytics focuses on the hypothesis that there is some combination of features and algorithm that will lead to effective predicability; however the algorithm selection and hyperparameter tuning elements of the model selection triple can be engaged using search mechanisms, particularly because of the effectiveness of the Scikit-Learn API and grid search.

Instead, building effective application-specific language models revolves around high quality feature extraction. We have to represent documents in a numeric space for machine learning, therefore the first consideration is a process called *vectorization* - the transformation of strings into numeric representations. The simplest and most common model is the “bag-of-words” model where elements of the vector represent words and the values represent the relationship of documents to words such as whether or not the word is in the document, the frequency of the word, or the TF-IDF score of the word. Other models include distributed representations, which may be even more effective at prediction.

Vectorization must be combined with other, document-level features as well as application-specific feature extraction. In the final section of this chapter we explored the use of `FeatureUnion` and `Pipeline` objects to create meaningful extraction methodologies by combining transformers. We also integrated NLTK and Gensim with Scikit-Learn by wrapping the other libraries in transformers and coordinating their work with Scikit-Learn in the pipelines. As we move forward, the practice of building pipelines of transformers and estimators will continue to be our primary mechanism of performing machine learning.

In Chapter 1 we performed ingestion and preprocessing of text to prepare for machine learning, this preprocessing was *transformative* but not *destructive* in that no information is lost, only added during preprocessing. Preprocessing ensures that we can do rapid prototyping on text models. In this chapter we used preprocessed documents to perform feature extraction, processes which are destructive since they remove information. In [Link to Come] we will explore classification models and applications, then in [Link to Come] we will take a look at clustering models, often called *topic modeling* in text analysis.

---

[1] Kumar, A., McCann, R., Naughton, J., Patel, J. (2015) Model Selection Management Systems: The Next Frontier of Advanced Analytics

[2] Wickham, H., Cooke, D., Hofmann, H. (2015) Visualizing statistical models: Removing the blindfold

[3] *https://arxiv.org/abs/1405.4053*