Explore Python, machine learning, and the NLTK library

Develop an app to categorize RSS feeds using Python, NLTK, and machine learning

Chris Joakim (cjoakim@bellsouth.net)
Senior Software Engineer
Primedia Inc

09 October 2012

Machine learning lies at the intersection of IT, mathematics, and natural language, and is typically used in big-data applications. This article discusses the Python programming language and its NLTK library, then applies them to a machine learning project.

New to Python

This article is for software developers—particularly those coming from a Ruby or Java language background—who are facing their first machine learning implementation.

The challenge: Use machine learning to categorize RSS feeds

I was recently given the assignment to create an RSS feed categorization subsystem for a client. The goal was to read dozens or even hundreds of RSS feeds and automatically categorize their many articles into one of dozens of predefined subject areas. The content, navigation, and search functionality of the client website would be driven by the results of this daily automated feed retrieval and categorization.

The client suggested using machine learning, perhaps with Apache Mahout and Hadoop, as she had recently read articles about those technologies. Her development team and ours, however, are fluent in Ruby rather than Java™ technology. This article describes the technical journey, learning process, and ultimate implementation of a solution.

What is machine learning?

My first question was, "what exactly is machine learning?" I had heard the term and was vaguely aware that the supercomputer IBM® Watson had recently used it to defeat human competitors in a game of *Jeopardy.* As a shopper and social network participant, I was also aware that both Amazon.com and Facebook do an amazingly good job of recommending things (such as products and people) based on data about their shoppers. In short, machine learning lies at the intersection of IT, mathematics, and natural language. It is primarily concerned with these three topics, but the solution for the client would ultimately involve the first two:

- **Classification.** Assigning items to arbitrary predefined categories based on a set of training data of similar items
- **Recommendation.** Recommending items based on observations of similar items
- **Clustering.** Identifying subgroups within a population of data

The Mahout and Ruby detours

Armed with an understanding of what machine learning is, the next step was to determine how to implement it. As the client suggested, Mahout was an appropriate starting place. I downloaded the code from Apache and went about the process of learning machine learning with Mahout and its sibling, Hadoop. Unfortunately, I found that Mahout had a steep learning curve, even for an experienced Java developer, and that working sample code didn't exist. Also unfortunate was a lack of Ruby-based frameworks or gems for machine learning.

Finding Python and the NLTK

I continued to search for a solution and kept encountering "Python" in the result sets. As a Rubyist, I knew that Python was a similar object-oriented, text-based, interpreted, and dynamic programming language, though I hadn't learned the language yet. In spite of these similarities, I had neglected to learn Python over the years, seeing it as a redundant skill set. Python was in my "blind spot," as I suspect it is for many of my Rubyist peers.

Searching for books on machine learning and digging deeper into their tables of contents revealed that a high percentage of these systems use Python as their implementation language, along with a library known as the *Natural Language Toolkit* (NLTK). Further searching revealed that Python was more widely used than I had realized—such as in Google App Engine, YouTube, and websites built with the Django framework. It even comes preinstalled on the Mac OS X workstations I use daily! Furthermore, Python offers interesting standard libraries (for example, NumPy and SciPy) for mathematics, science, and engineering. Who knew?

I decided to pursue a Python solution after I found elegant coding examples. The following one-liner, for example, is all the code needed to read an RSS feed through HTTP and print its contents:

```
print feedparser.parse("http://feeds.nytimes.com/nyt/rss/Technology")
```

Getting up to speed on Python

In learning a new programming language, the easy part is often learning the language itself. The harder part is learning its ecosystem—how to install it, add libraries, write code, structure the code files, execute it, debug it, and write unit tests. This section provides a brief introduction to these topics; be sure to check out Resources for links to more information.

pip

The Python Package Index (`pip`) is the standard package manager for Python. It's the program you use to add libraries to your system. It's analogous to gem for Ruby libraries. To add the NLTK library to your system, you enter the following command:

```
$ pip install nltk
```

To display a list of Python libraries installed on your system, run this command:

```
$ pip freeze
```

Running programs

Executing a Python program is equally simple. Given a program named *locomotive_main.py* and three program arguments, you compile and execute it with the `python` program:

```
$ python locomotive_main.py arg1 arg2 arg3
```

Python uses the `if __name__ == "__main__":` syntax in Listing 1 to determine whether the file itself is being executed from the command line or just being imported by other code. To make a file executable, add `"__main__"` detection.

Listing 1. Main detection

```
import sys
import time
import locomotive

if __name__ == "__main__":
    start_time = time.time()
    if len(sys.argv) > 1:
        app = locomotive.app.Application()
        ... additional logic ...
```

virtualenv

Most Rubyists are familiar with the issue of system-wide libraries, or *gems.* A system-wide set of libraries is generally not desirable, as one of your projects might depend on version 1.0.0 of a given library, while another project depends on version 1.2.7. Likewise, Java developers are aware of this same issue with a system-wide CLASSPATH. Like the Ruby community with its `rvm` tool, the Python community uses the `virtualenv` tool (see Resources for a link) to create separate execution environments, including specific versions of Python and a set of libraries. The commands in Listing 2 show how to create a virtual environment named *p1_env* for your p1 project, which contains the `feedparser`, `numpy`, `scipy`, and `nltk` libraries.

Listing 2. Commands to create a virtual environment with virualenv

```
$ sudo pip install virtualenv
$ cd ~
$ mkdir p1
$ cd p1
$ virtualenv p1_env --distribute
$ source p1_env/bin/activate
(p1_env)[~/p1]$ pip install feedparser
(p1_env)[~/p1]$ pip install numpy
(p1_env)[~/p1]$ pip install scipy
(p1_env)[~/p1]$ pip install nltk
(p1_env)[~/p1]$ pip freeze
```

You need to "source" your virtual environment activation script each time you work with your project in a shell window. Notice that the shell prompt changes after the activation script is sourced. As you create and use shell windows on your system and to easily navigate to your project directory and activate its virtual environment, you might want to add an entry like the following to your ~/.bash_profile file:

```
$ alias p1="cd ~/p1 ; source p1_env/bin/activate"
```

The code base structure

After graduating from simple single-file "Hello World" programs, Python developers need to understand how to properly structure their code base regarding directories and file names. Each of the Java and Ruby languages has its own requirements in this regard, and Python is no different. In short, Python uses the concept of *packages* to group related code and provide unambiguous namespaces. For the purpose of demonstration in this article, the code exists within a given project root directory, such as ~/p1. Within this directory, there exists a locomotive directory for a Python package of the same name. Listing 3 shows this directory structure.

Listing 3. Example directory structure

```
locomotive_main.py
locomotive_tests.py

locomotive/
    __init__.py
    app.py
    capture.py
    category_associations.py
    classify.py
    news.py
    recommend.py
    rss.py

locomotive_tests/
    __init__.py
    app_test.py
    category_associations_test.py
    feed_item_test.pyc
    rss_item_test.py
```

Notice the oddly named *__init__.py* files. These files instruct Python to load the necessary libraries for your package as well as your specific application code files that reside in the same directory. Listing 4 shows the contents of the file locomotive/__init__.py.

Listing 4. locomotive/__init__.py

```
    # system imports; loads installed packages
    import codecs
    import locale
    import sys

    # application imports; these load your specific *.py files
    import app
    import capture
    import category_associations
    import classify
    import rss
    import news
    import recommend
```

With the locomotive package structured as in Listing 4, the main programs in the root directory of your project can import and use it. For example, file locomotive_main.py contains the following imports:

```
    import sys          # >-- system library
    import time         # >-- system library
    import locomotive   # >-- custom application code library in the "locomotive" directory
```

Testing

The Python `unittest` standard library provides a nice solution for testing. Java developers familiar with JUnit and Rubyists familiar with Test::Unit framework should find the Python `unittest` code in Listing 5 to be easily readable.

Listing 5. Python unittest

```
  class AppTest(unittest.TestCase):

      def setUp(self):
          self.app = locomotive.app.Application()

      def tearDown(self):
          pass

      def test_development_feeds_list(self):
          feeds_list = self.app.development_feeds_list()
          self.assertTrue(len(feeds_list) == 15)
          self.assertTrue('feed://news.yahoo.com/rss/stock-markets' in feeds_list)
```

The code in Listing 5 also demonstrates a distinguishing feature of Python: All code must be consistently indented or it won't compile successfully. The `tearDown(self)` method might look a bit odd at first. You might wonder why the test is hard-coded always to pass? Actually, it's not. That's just how you code an empty method in Python.

Tooling

What I really needed was an integrated development environment (IDE) with syntax highlighting, code completion, and breakpoint debugging functionality to help me with the Python learning curve. As a user of the Eclipse IDE for Java development, the `pyeclipse` plug-in was the next tool I looked at. It works fairly well though was sluggish at times. I eventually invested in the PyCharm IDE, which meets all of my IDE requirements.

Armed with a basic knowledge of Python and its ecosystem, it was finally time to start implementing a machine learning solution.

Implementing categorization with Python and NLTK

Implementing the solution involved capturing simulated RSS feeds, scrubbing their text, using a `NaiveBayesClassifier`, and classifying categories with the kNN algorithm. Each of these actions is described here.

Capturing and parsing the feeds

The project was particularly challenging, because the client had not yet defined the list of target RSS feeds. Thus, there was no "training data," either. Therefore, the feed and training data had to be simulated during initial development.

The first approach I used to obtain sample feed data was simply to fetch a list of RSS feeds specified in a text file. Python offers a nice RSS feed parsing library called `feedparser` that abstracts the differences between the various RSS and Atom formats. Another useful library for simple text-based object serialization is humorously called `pickle`. Both of these libraries are used in the code in Listing 6, which captures each RSS feed as "pickled" object files for later use. As you can see, the Python code is concise and powerful.

Listing 6. The CaptureFeeds class

```
import feedparser
import pickle

class CaptureFeeds:

    def __init__(self):
        for (i, url) in enumerate(self.rss_feeds_list()):
            self.capture_as_pickled_feed(url.strip(), i)

    def rss_feeds_list(self):
        f = open('feeds_list.txt', 'r')
        list = f.readlines()
        f.close
        return list

    def capture_as_pickled_feed(self, url, feed_index):
        feed = feedparser.parse(url)
        f = open('data/feed_' + str(feed_index) + '.pkl', 'w')
        pickle.dump(feed, f)
        f.close()

if __name__ == "__main__":
    cf = CaptureFeeds()
```

The next step was unexpectedly challenging. Now that I had sample feed data, it had to be categorized for use as training data. *Training data* is the set of data that you give to your categorization algorithm so that it can learn from it.

For example, the sample feeds I used included ESPN, the Sports Network. One of the feed items was about Tim Tebow of the Denver Broncos football team being traded to the New York Jets football team during the same time the Broncos had signed Peyton Manning as their new quarterback. Another item in the feed results was about the Boeing Company and its new jet. So, the question is, what specific category value should be assigned to the first story? The values `tebow`, `broncos`, `manning`, `jets`, `quarterback`, `trade`, and `nfl` are all appropriate. But only one value can be specified in the training data as its category. Likewise, in the second story, is the category `boeing` or `jet`? The hard part is in those details. Accurate manual categorization of a large set of training data is essential if your algorithm is to produce accurate results. The time required to do this should not be underestimated.

It soon became apparent that I needed more data to work with, and it had to be categorized already—and accurately. Where would I find such data? Enter the Python NLTK. In addition to being an outstanding library for language text processing, it even comes with downloadable sets of sample data, or a *corpus* in their terminology, as well as an application programming interface to easily access this downloaded data. To install the Reuters corpus, run the commands shown below. More than 10,000 news articles will be downloaded to your ~/nltk_data/corpora/reuters/ directory. As with RSS feed items, each Reuters news article contains a title and a body, so this NLTK precategorized data is excellent for simulating RSS feeds.

```
$ python                # enter an interactive Python shell
>>> import nltk         # import the nltk library
>>> nltk.download()     # run the NLTK Downloader, then enter 'd' Download
Identifier> reuters     # specify the 'reuters' corpus
```

Of particular interest is file ~/nltk_data/corpora/reuters/cats.txt. It contains a list of article file names and the assigned category for each article file. The file looks like the following, so the article in file 14828 in subdirectory test pertains to the topic `grain`.

```
test/14826 trade
test/14828 grain
```

Natural language is messy

The raw input to the RSS feed categorization algorithm is, of course, text written in the English language. Raw, indeed.

English, or any natural language (that is, spoken or ordinary language) is highly irregular and imprecise from a computer processing perspective. First, there is the matter of case. Is the word *Bronco* equal to *bronco*? The answer is maybe. Next, there is punctuation and whitespace to contend with. Is *bronco.* equal to *bronco* or *bronco,*? Kind of. Then, there are plurals and similar words. Are *run, running,* and *ran* equivalent? Well, it depends. These three words have a common *stem.* What if the natural language terms are embedded within a markup language like HTML? In that case, you have to deal with text like `<strong>bronco</strong>`. Finally, there is the issue of frequently used but essentially meaningless words like *a, and,* and *the.* These so-called stopwords just get in the way. Natural language is messy; it needs to be cleaned it up before processing.

Fortunately, Python and NLTK enable you to clean up this mess. The `normalized_words` method of class `RssItem`, in Listing 7, deals with all of these issues. Note in particular how NLTK cleans the raw article text of the embedded HTML markup in just one line of code! A regular expression is used to remove punctuation, and the individual words are then split and normalized into lowercase.

Listing 7. The RssItem class

```
class RssItem:
    ...
    regex = re.compile('[%s]' % re.escape(string.punctuation))
    ...
    def normalized_words(self, article_text):
        words   = []
        oneline = article_text.replace('\n', ' ')
        cleaned = nltk.clean_html(oneline.strip())
        toks1   = cleaned.split()
        for t1 in toks1:
            translated = self.regex.sub('', t1)
            toks2 = translated.split()
            for t2 in toks2:
                t2s = t2.strip().lower()
                if self.stop_words.has_key(t2s):
                    pass
                else:
                    words.append(t2s)
        return words
```

The list of stopwords came from NLTK with this one line of code; other natural languages are supported:

```
nltk.corpus.stopwords.words('english')
```

NLTK also offers several "stemmer" classes to further normalize the words. Check out the NLTK documentation on stemming, lemmatization, sentence structure, and grammar for more information.

Classification with the Naive Bayes algorithm

The Naive Bayes algorithm is widely used and implemented in the NLTK with the `nltk.NaiveBayesClassifier` class. The Bayes algorithm classifies items per the presence or absence of features in their datasets. In the case of the RSS feed items, each feature is a given (cleaned) word of natural language. The algorithm is "naive," because it assumes that there is no relationship between the features (in this case, words).

The English language, however, contains more than 250,000 words. Certainly, I don't want to have to create an object containing 250,000 Booleans for each RSS feed item for the purpose of passing to the algorithm. So, which words do I use? In short, the answer is the most common words in the population of training data that aren't stopwords. NLTK provides an outstanding class, `nltk.probability.FreqDist`, which I can use to identify these top words. In Listing 8, the `collect_all_words` method returns an array of all the words from all training articles.

This array is then passed to the `identify_top_words` method to identify the most frequent words. A useful feature of the `nltk.FreqDist` class is that it's essentially a hash, but its keys are sorted by their corresponding values, or *counts.* Thus, it is easy to obtain the top 1000 words with the `[:1000]` Python syntax.

Listing 8. Using the nltk.FreqDist class

```
def collect_all_words(self, items):
    all_words = []
    for item in items:
        for w in item.all_words:
            words.append(w)
    return all_words

def identify_top_words(self, all_words):
    freq_dist = nltk.FreqDist(w.lower() for w in all_words)
    return freq_dist.keys()[:1000]
```

For the simulated RSS feed items with the NLTK Reuters article data, I need to identify the categories for each item. I do this by reading the ~/nltk_data/corpora/reuters/cats.txt file mentioned earlier. Reading a file with Python is as simple as this:

```
def read_reuters_metadata(self, cats_file):
    f = open(cats_file, 'r')
    lines = f.readlines()
    f.close()
    return lines
```

The next step is to get the features for each RSS feed item. The `features` method of the `RssItem` class, shown below, does this. In this method, the array of `all_words` in the article is first reduced to a smaller `set` object to eliminate the duplicate words. Then, the `top_words` are iterated and compared to this set for presence or absence. A hash of 1000 Booleans is returned, keyed by `w_` followed by the word itself. Very concise, this Python.

```
def features(self, top_words):
    word_set = set(self.all_words)
    features = {}
    for w in top_words:
        features["w_%s" % w] = (w in word_set)
    return features
```

Next, I collect a training set of RSS feed items and their individual features and pass them to the algorithm. The code in Listing 9 demonstrates this task. Note that the classifier is trained in exactly one line of code.

Listing 9. Training a nltk.NaiveBayesClassifier

```
def classify_reuters(self):
      ...
      training_set = []
      for item in rss_items:
          features = item.features(top_words)
          tup = (features, item.category)  # tup is a 2-element tuple
          featuresets.append(tup)
      classifier = nltk.NaiveBayesClassifier.train(training_set)
```

The `NaiveBayesClassifier`, in the memory of the running Python program, is now trained. Now, I simply iterate the set of RSS feed items that need to be classified and ask the classifier to guess the category for each item. Simple.

```
for item in rss_items_to_classify:
    features = item.features(top_words)
    category = classifier.classify(feat)
```

Becoming less naive

As stated earlier, the algorithm assumes that there is no relationship between the individual features. Thus, phrases like "machine learning" and "learning machine" or "New York Jet" and "jet to New York" are equivalent (*to* is a stopword). In natural language context, there is an obvious relationship between these words. So, how can I teach the algorithm to become "less naive" and recognize these word relationships?

One technique is to include the common *bigrams* (groups of two words) and *trigrams* (groups of three words) in the feature set. It should now come as no surprise that NLTK provides support for this in the form of the `nltk.bigrams(...)` and `nltk.trigrams(...)` functions. Just as the top *n*-number of words were collected from the population of training data words, the top bigrams and trigrams can similarly be identified and used as features.

Your results will vary

Refining the data and the algorithm is something of an art. Should you further normalize the set of words, perhaps with stemming? Or include more than the top 1000 words? less? Or use a larger training data set? Add more stopwords or "stop-grams"? These are all valid questions to ask yourself. Experiment with them, and through trial and error, you will arrive at the best algorithm for your data. I found that 85 percent was a good rate of successful categorization.

Recommendation with the k-Nearest Neighbors algorithm

The client wanted to display RSS feed items within a selected category or similar categories. Now that the items had been categorized with the Naive Bayes algorithm, the first part of that requirement was satisfied. The harder part was to implement the "or similar categories" requirement. This is where machine learning recommender systems come into play. *Recommender systems* recommend an item based on similarity to other items. Amazon.com product recommendations and Facebook friend recommendations are good examples of this functionality.

k-Nearest Neighbors (kNN) is the most common recommendation algorithm. The idea is to provide it a set of *labels* (that is, *categories*), and a corresponding dataset for each label. The algorithm then compares the datasets to identify similar items. The dataset is composed of arrays of numeric values, often in a normalized range from 0 to 1. It can then identify the similar labels from the datasets. Unlike Naive Bayes, which produces one result, kNN can produce a ranked list of several (that is, the value of *k*) recommendations.

I found recommender algorithms simpler to comprehend and implement than the classification algorithms, although the code was too lengthy and mathematically complex to include here. Refer to the excellent new Manning book, *Machine Learning in Action,* for kNN coding examples (see Resources for a link). In the case of the RSS feed item implementation, the label values were the item categories, and the dataset was an array of values for each of the top 1000 words. Again, constructing this array is part science, part math, and part art. The values for each word in the array can be simple zero-or-one Booleans, percentages of word occurrences within the article, an exponential value of this percentage, or some other value.

Conclusion

Discovering Python, NLTK, and machine learning has been an interesting and enjoyable experience. The Python language is powerful and concise and now a core part of my developer toolkit. It is well suited to machine learning, natural language, and mathematical/scientific applications. Although not mentioned in this article, I also found it useful for charting and plotting. If Python has similarly been in your blind spot, I encourage you to take a look at it.

Resources

Learn

- Learn more about machine learning from Wikipedia.
- Check out Python's official website.
- Read Peter Harrington's Machine Learning in Action (Manning, 2012).
- Check out Natural Language Processing with Python by Steven Bird, Ewan Klein, and Edward Loper (O'Reilly, 2009).
- Check out Implement Bayesian inference using PHP (Paul Meagher, developerWorks, March-May 2009). This three-part series discusses interesting applications designed to help you appreciate the power and potential of Bayesian inference concepts.
- In the Open Source area on developerWorks, find extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM products.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools as well as IT industry trends.
- Listen to developerWorks podcasts for interesting interviews and discussions for software developers.
- Follow developerWorks on Twitter.
- Watch developerWorks demos that range from product installation and setup for beginners to advanced functionality for experienced developers.

Get products and technologies

- Explore the NLTK site and building Python programs to work with human language data.
- Download pip and learn more about this tool for installing and managing Python packages.
- Learn more about `virtualenv`, a tool to create isolated Python environments.
- Check out the Python `unittest` standard library, a Python language version of JUnit.
- Check out the `pyeclipse` plug-in for Eclipse.
- Check out the PyCharm IDE for a complete set of development tools to program with Python and capabilities the Django framework.
- Access IBM trial software (available for download or on DVD) and innovate in your next open source development project using software especially for developers.

Discuss

- Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis. Help build the Real world open source group in the developerWorks community.

About the author

Chris Joakim

Chris Joakim is Senior Software Engineer at Primedia Inc. Chris has developed software for more than 25 years in various languages, including Clojure, Ruby, Java, Objective-C, JavaScript, CoffeeScript, Flex, Smalltalk,COBOL, and now Python. He resides in Davidson, NC. In his spare time, he runs marathons and writes code. You can contact Chris at cjoakim@bellsouth.net.