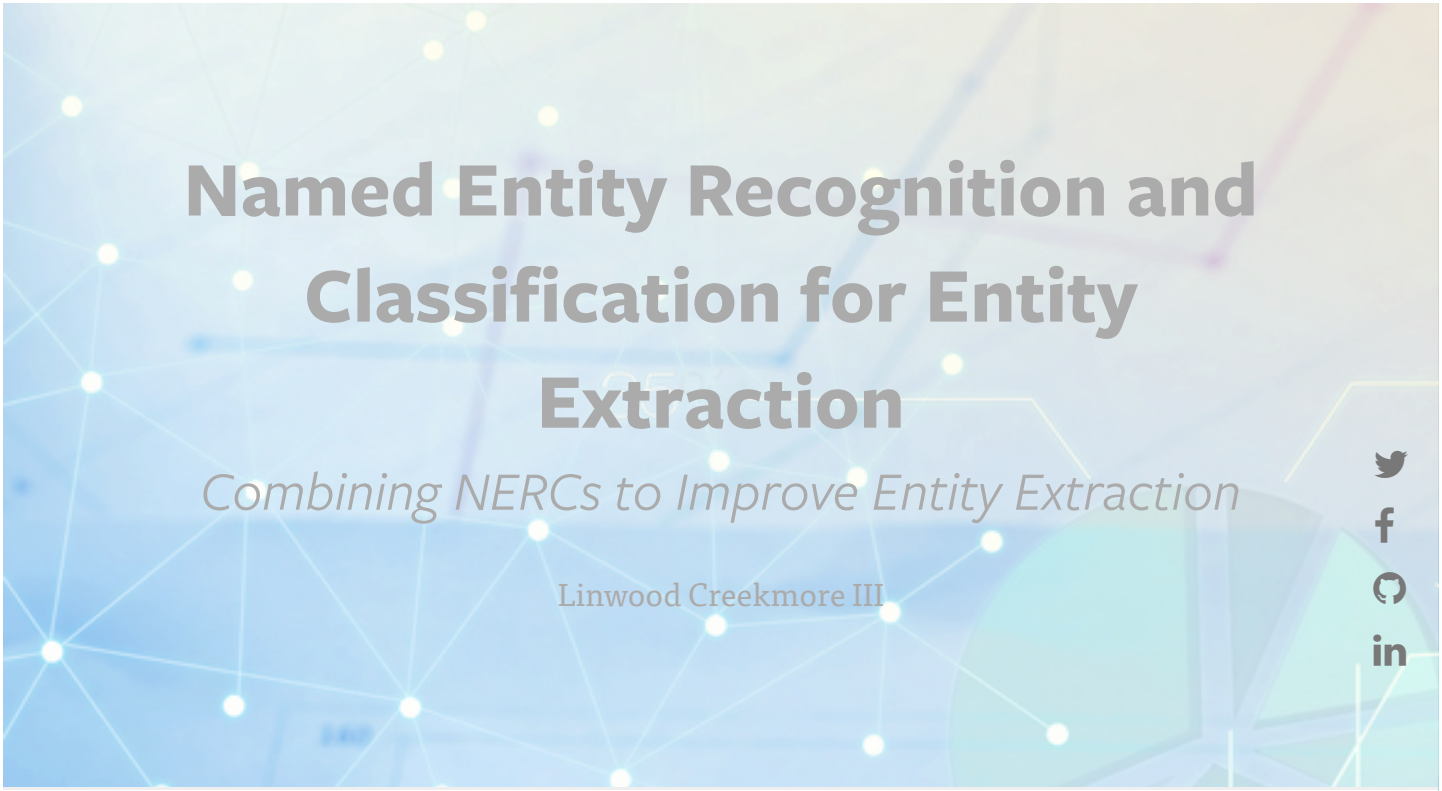


Named Entity Recognition and Classification for Entity Extraction

Combining NERCs to Improve Entity Extraction

Linwood Creekmore III

The overwhelming amount of unstructured text data available today from traditional media sources as well as newer ones, like social media, provides a rich source of information if the data can be structured. Named Entity Extraction forms a core subtask to build knowledge from semi-structured and unstructured text sources. Some of the first researchers working to extract information from unstructured texts recognized the importance of “units of information” like names (such as person, organization, and location names) and numeric expressions (such as time, date, money, and percent expressions). They coined the term “Named Entity” in 1996 to represent these.

Considering recent increases in computing power and decreases in the costs of data storage, data scientists and developers can build large knowledge bases that contain millions of entities and hundreds of millions of facts about them. These knowledge bases are key contributors to intelligent computer behavior. Not

surprisingly, Named Entity Extraction operates at the core of several popular technologies such as smart assistants (Siri, Google Now), machine reading, and deep interpretation of natural language.

This post explores how to perform Named Entity Extraction, formally known as "Named Entity Recognition and Classification (NERC). In addition, the article surveys open-source NERC tools that work with Python and compares the results obtained using them against hand-labeled data.

The specific steps include:

- Preparing semi-structured natural language data for ingestion using regular expressions; creating a custom corpus in the Natural Language Toolkit
- Using a suite of open source NERC tools to extract entities and store them in JSON format
- Comparing the performance of the NERC tools
- Implementing a simplistic ensemble classifier

The information extraction concepts and tools in this article constitute a first step in the overall process of structuring unstructured data. They can be used to perform more complex natural language processing to derive unique insights from large collections of unstructured data.

Environment Set-Up

In order to follow along with the work in this article, we recommend using Anaconda, which is an easy-to-install, free, enterprise-ready Python distribution for data analytics, processing, and scientific computing. With a few lines of code, you can have all the dependencies used in this post with, the exception of one function (email extractor).

1. Install Anaconda
2. Download the requirements.yml (remember where you saved it on your computer)
3. Follow the Use Environment from File instructions on Anaconda's website.

If you use an alternative method to set up a virtual environment, make sure you have all the files installed from the yml file. The one dependency not in the yml file is the email extractor. Cut and paste the function from this Gist, save it to a .py file, and make sure it is in your sys.path or environment path.

Data Source

The proceedings from the Knowledge Discovery and Data Mining (KDD) conferences in New York City (2014) and Sydney, Australia (2015) serve as our source of unstructured text and contain over 200 peer reviewed journal articles and keynote speaker abstracts on data mining, knowledge discovery, big data, data science, and their applications. The full conference proceedings can be purchased for $60 at the Association for Computing Machinery's Digital Library (includes ACM membership).

This post will work on a data set that is equivalent to the combined conference proceedings, but only use abstracts and extracts from the text, rather than the full proceedings, a data set that can be found on the ACM website. We will explore reading PDF data and discuss follow-on analytics if the full proceedings are available to you.

Initial Data Exploration

Visual inspection reveals that the target filenames begin with a "p" and end with "pdf." As a first step, we determine the number of files and the naming conventions by using a loop to iterate over the files in the directory and printing out the filenames. Each filename also gets saved to a list, and the length of the list tells us the total number of files in the dataset.

```
import os

## Set important paths
BASE = os.path.join(os.path.dirname(__file__), "..")
DOCS = os.path.join(BASE, "data", "docs")

def get_documents(path=DOCS):
    """
    Returns a filtered list of paths to PDF files
representing our corpus.
    """
    for name in os.listdir(path):
        if name.startswith('p') and
name.endswith('.pdf'):
            yield os.path.join(path, name)

# Print the total number of documents
print(len(list(get_documents())))
```

A total of 253 files exist in the directory. Opening one of these reveals that our data is in PDF format and that it is semi-structured (follows journal article format with separate sections for "abstract" and "title"). While PDFs provide an easily readable presentation of data, they are extremely difficult to work with in data analysis. In your work, if you have an option to get to data before conversion to a PDF format, be sure to take that option.

Creating a Custom NLTK Corpus

We used several Python tools to ingest our data, including the following libraries:

- Pdfminer - contains a command line tool called "pdf2txt.py" that extracts text contents from a PDF file (you can visit the pdfminer homepage for download instructions).
- Subprocess - a standard library module that allows us to invoke the "pdf2txt.py" command line tool within our code.
- NLTK - the Natural Language Tool Kit, or NLTK, serves as one of Python's leading platforms to analyze natural language data.
- String - provides variable substitutions and value formatting to strip non-printable characters from the output of the text extracted from our journal article PDFs.
- Unicodedata - allows Latin Unicode characters to degrade gracefully into ASCII. This is an important feature because some Unicode characters won't extract nicely.

Our task begins by iterating over the files in the directory with names that begin with "p" and end with "pdf." This time, however, we will strip the text from the pdf file, write the .txt file to a newly created directory, and use the `fname` variable to name the files we write to disk. Keep in mind that this task may take a few minutes depending on the processing power of your computer.

```
import re
import nltk
import codecs
import string
import subprocess
import unicodedata

## Create a path to extract the corpus.
CORPUS = os.path.join(BASE, "data", "corpus")

def extract_corpus(docs=DOCS, corpus=CORPUS):
    """
    Extracts a text corpus from the PDF documents and
writes them to disk.
```

```
    """

    # Create corpus directory if it doesn't exist.
    if not os.path.exists(corpus):
        os.mkdir(corpus)

    # For each PDF path, use pdf2txt to extract the text
file.
    for path in get_documents(docs):
        # Call the subprocess command (must be on your
path)
        document = subprocess.check_output(
            ['pdf2txt.py', path]
        )

        # Encode UTF-u and remove non-printable
characters
        document = filter(
            lambda char: char in string.printable,
            unicodedata.normalize('NFKD',
document.decode('utf-8'))
        )

        # Write the document out to the corpus directory
        fname = os.path.splitext(os.path.basename(path))
[0] + ".txt"
        outpath = os.path.join(corpus, fname)

        with codecs.open(outpath, 'w') as f:
            f.write(document)

# Run the extraction
extract_corpus()
```

Next, we build a custom NLTK corpus. Having our target documents loaded as an NLTK corpus brings the power of NLTK to our analysis goals.

```
# Create an NLTK corpus reader to access text data on
disk.
kddcorpus = nltk.corpus.PlaintextCorpusReader(CORPUS,
'.*\.txt')
```

We now have a semi-structured dataset in a format that we can query and analyze. First, let's see how many words (including stop words) we have in our entire corpus and the vocabulary of the corpus.

```
words = nltk.FreqDist(kddcorpus.words())
count = sum(words.values())
vocab = len(words)

print("Corpus contains a vocabulary of {} and a word
count of {}.".format(
    count, vocab
))

Corpus contains a vocabulary of 70,073 and a word count
of 2,785,178
```

The NLTK book has an excellent section on processing raw text and unicode issues. It provides a helpful discussion of some problems you may encounter.

Using Regular Expressions to Extract Specific Sections

To begin our exploration of regular expressions (a.k.a. "regex"), it's important to point out some good resources for those new to the topic. An excellent resource may be found in Johns Hopkins University's Coursera video titled Getting and Cleaning Data.

As a simple example, let's extract titles from the first 10 documents.

```
def titles(fileid=None,corpus=kddcorpus):
    """
    Use a regular expression to extract the titles from
the corpus.
    """
    pattern = re.compile(r'^(.*)[\s]+[\s]?(.*)?')

    if fileid is not None:
        match = pattern.search(kddcorpus.raw(fileid))
        yield " ".join(map(lambda s: s.strip(),
match.groups()))
    else:
        for fileid in corpus.fileids():
            # Search for a pattern match
            match = pattern.search(corpus.raw(fileid))
```

```
            if match:
                # If we find one, yield the space joined
groups.
                yield " ".join(map(lambda s: s.strip(),
match.groups()))

for idx, title in enumerate(titles()):
     print title
     if idx >= 10:
         break
```

The result is as follows:

```
Online Controlled Experiments: Lessons from Running A/B/n
Tests for 12 Years
Mining Frequent Itemsets through Progressive Sampling
with Rademacher Averages
Why It Happened: Identifying and Modeling the Reasons of
the Happening of Social Events
Matrix Completion with Queries Natali Ruchansky
Stochastic Divergence Minimization for Online Collapsed
Variational Bayes Zero Inference
Bayesian Poisson Tensor Factorization for Inferring
Multilateral Relations from Sparse Dyadic Event Counts
TimeCrunch: Interpretable Dynamic Graph Summarization
Neil Shah
Inside Jokes: Identifying Humorous Cartoon Captions Dafna
Shahaf
Community Detection based on Distance Dynamics Junming
Shao
Discovery of Meaningful Rules in Time Series Mohammad
Shokoohi-Yekta    Yanping Chen    Bilson Campana    Bing
Hu
```

This code extracts the titles, but some author names get caught up in the extraction as well.

For simplicity, let's focus on wrangling the data to use the NERC tools on two sections of the paper: the “top” section and the “references” section. The “top” section includes the names of authors and schools. This section represents all of the text above the article’s abstract. The “references” section appears at the end of the article. The regex tools of choice to extract sections are the `positive lookbehind` and `positive lookahead`

expressions. Using these, we will build two functions designed to extract the “top” and “references” sections of each document.

In addition to extracting the relevant sections of the documents, our two functions will obtain a character count for each section, extract emails, count the number of references and store that value, calculate a word per reference count, and store all the above data as a nested dictionary with filenames as the key.

```
def sectpull(fileids=None, section=None,
corpus=kddcorpus):
    """
    Uses a regular expression to pull sections from a
file:

        - "top": everything until the references section
        - "ref": the references and anything that
follows.

    Yields the text as top, ref respectively.
    """


    # Select either a single fileid or a list of fileids

    fileids = fileids or corpus.fileids()
    if section == None:
        section = None
    elif section == 'references':
        section = re.compile('(?<=' + 'REFERENCES' + ')
(.+)')
    elif section == 'body':
        section = re.compile("(.+?)(?="+'REFERENCES'+
")")
    elif section == 'top':
        section = ["ABSTRACT","Abstract","Bio","Panel
Summary"]
        for sect in section:
            try:
                section = re.compile("(.+?)(?="+sect+
")")
                break
            except:
                pass

    # Iterate through all text for each file id.
    for fileid in fileids:
        text   = corpus.raw(fileid)
```

```
        # Extract the text and search for the section
target
        if section == None:
            target = re.sub('[\s]', " ", text)
        else:
            text   = re.sub('[\s]', " ", text)
            target = section.search(text)

        if target:
            yield fileid, target.group(0),
target.group(1)


def refstats(fileids=None, section=None,
corpus=kddcorpus):
    """
    Code to pull only the references section, store a
character count, number
    of references, as well as a "words per reference"
count.

    Pass either a specific document id, a list of ids, or
None for all ids.
    """
    # Create reference number to match pattern
    refnum  = re.compile(r'\[[0-9]{1,3}\]', re.I)

    for fileid, top, refs in sectpull(fileids, section,
corpus):

        # Yield the statistics about the references
        n_refs = len(set((refnum.findall(refs))))

        words  = sum(1 for word in
nltk.word_tokenize(refs))
        wp_ref = float(words) / float(n_refs) if n_refs
else 0

        # Yield the data from the generator
        yield (fileid, len(refs), n_refs, wp_ref)
```

The above code also makes use of the `nltk.word_tokenize` tool to create the "word per reference" statistic (takes time to run).

I want to take an opportunity here to say few words about the data. When working with natural language, one should always be prepared to deal with irregularities in the data set. This corpus is no exception. It comes from a top-notch data mining organization, but human error and a lack of standardization

makes its way into the picture. For example, in one paper the header section is entitled “Categories and Subject Descriptors,” while in another the title is “Categories & Subject Descriptors.” While that may seem like a small difference, these types of differences cause significant problems. There are also some documents that will be missing sections altogether, i.e. keynote speaker documents do not contain a “references” section. When encountering similar issues in your work, you must decide whether to account for these differences or ignore them. I worked to include as much of the 253-document corpus as possible.

Next, let's test the “references” extraction function and look at the output by obtaining the first 10 entries of the dictionary created by the function. This dictionary holds all the extracted data and various calculations. The `tabulate` module is a great tool to visualize descriptive outputs in table format.

```
from tabulate import tabulate
from operator import itemgetter
import random

# Create table sorted by number of references
table   =
sorted(list(refstats(random.sample(kddcorpus.fileids(),15
),section='body')))

# Print the table with headers
headers = ('File', 'Characters', 'References', 'Words per
Reference')
print(tabulate(table, headers=headers))
```

The output is as follows:

```
File         Characters    References    Words per
Reference
---------  ------------  ------------  ------------------
---
p1005.txt         50190            22                 488
p1205.txt         47071            20
```

```
423.95
p1583.txt         46346            38
297.289
p2167.txt         45739            18
480.833
p537.txt          54152            24
445.875
p59.txt           46215            24
374.708
p715.txt          30376            18                  346
p725.txt          49690            18
555.5
p835.txt          50786            28
389.286
p865.txt          47636            16
604.875
p935.txt          58307            21
554.429
... [snip] ...
```

As you can see, this is a good start to performing bibliographic analysis with Python.

Open Source NERC Tools

Now that we have a method to obtain the corpus from the "top" and "references" sections of each article in the dataset, we are ready to perform the named entity extractions. In this post, we examine three popular, open source NERC tools. The tools are NLTK, Stanford NER, and Polyglot. A brief description of each follows.

- NLTK has a `chunk` package that uses NLTK's recommended named entity chunker to chunk the given list of tagged tokens. A string is tokenized and tagged with parts of speech (POS) tags. The NLTK chunker then identifies non-overlapping groups and assigns them to an entity class. You can read more about NLTK's chunking capabilities in the NLTK book.

- Stanford's Named Entity Recognizer, often called Stanford NER, is a Java implementation of linear chain Conditional Random Field (CRF) sequence models functioning as a Named Entity Recognizer. Named Entity Recognition (NER) labels sequences of words in a text that are the names of things, such as person and company names, or gene and protein names. NLTK contains an interface to Stanford NER written by Nitin Madnani. Details for using the Stanford NER tool are on the NLTK page and the required jar files can be downloaded here.

- Polyglot is a natural language pipeline that supports massive multilingual (i.e. language) applications. It supports tokenization in 165 languages, language detection in 196 languages, named entity recognition in 40 languages, part of speech tagging in 16 languages, sentiment analysis in 136 languages, word embeddings in 137 languages, morphological analysis in 135 languages, and transliteration in 69 languages. It is a powerhouse tool for natural language processing. We will use the named entity recognition feature for English language in this exercise. Polyglot is available via pypi.

We can now test how well these open source NERC tools extract entities from the “top” and “reference” sections of our corpus. For two documents, I hand labeled authors, organizations, and locations from the “top” section of the article and the list of all authors from the “references” section. I also created a combined list of the authors, joining the lists from the “top” and “references” sections. Hand labeling is a time consuming and tedious process. For just the two documents, this involved 295 cut-and-pastes of names or organizations.

An easy test for the accuracy of a NERC tool is to compare the entities extracted by the tools to the hand-labeled extractions. Before beginning, we take advantage of the NLTK functionality to obtain the "top" and "references" sections of the two documents used for the hand labeling:

```
# We need the top and references sections from p19.txt
and p29.txt
annotated = sectpull(['p19.txt', 'p29.txt'])
```

For each NERC tool, I created functions to extract entities and return classes of objects in different lists.

```
from collections import defaultdict

from nltk import ne_chunk
from polyglot.text import Text
from nltk.tag import StanfordNERTagger


def polyglot_entities(fileids=None, section = None,
corpus=kddcorpus):
    """
    Extract entities from each file using polyglot
    """
    results = defaultdict(lambda: defaultdict(list))
    fileids = fileids or corpus.fileids()

    for fileid in fileids:
        if section is not None:
            text =
Text((list(sectpull([fileid],section=section))[0][1]))
        else:
            text = Text(corpus.raw(fileid))



        for entity in text.entities:
            etext = " ".join(entity)

            if entity.tag == 'I-PER':
                key = 'persons'
            elif entity.tag == 'I-ORG':
                key = 'organizations'
            elif entity.tag == 'I-locations':
                key = 'locations'
            else:
```

```
                key = 'other'

            results[fileid][key].append(etext)

    return results

def stanford_entities(model, jar, fileids=None,
corpus=kddcorpus, section = None):
    """
    Extract entities using the Stanford NER tagger.
    Must pass in the path to the tagging model and jar as
downloaded from the
    Stanford Core NLP website.
    """
    results = defaultdict(lambda: defaultdict(list))
    fileids = fileids or corpus.fileids()
    tagger  = StanfordNERTagger(model, jar)
    section = section

    for fileid in fileids:
        if section is not None:
            text =
nltk.word_tokenize(list(sectpull([fileid],section=section
))[0][1])
        else:
            text  = corpus.words(fileid)

        chunk = []

        for token, tag in tagger.tag(text):
            if tag == 'O':
                if chunk:
                    # Flush the current chunk
                    etext =  " ".join([c[0] for c in
chunk])
                    etag  = chunk[0][1]
                    chunk = []

                    if etag == 'PERSON':
                        key = 'persons'
                    elif etag == 'ORGANIZATION':
                        key = 'organizations'
                    elif etag == 'LOCATION':
                        key = 'locations'
                    else:
                        key = 'other'

                    results[fileid][key].append(etext)

            else:
                # Build chunk from tags
                chunk.append((token, tag))

    return results
```

```
def nltk_entities(fileids=None, section =
None,corpus=kddcorpus):
    """
    Extract entities using the NLTK named entity chunker.
    """
    results = defaultdict(lambda: defaultdict(list))
    fileids = fileids or corpus.fileids()

    for fileid in fileids:
        if section is not None:
            text =
nltk.pos_tag(nltk.word_tokenize(list(sectpull([fileid],se
ction=section))[0][1]))
        else:
            text = nltk.pos_tag(corpus.words(fileid))



        for entity in nltk.ne_chunk(text):
            if isinstance(entity, nltk.tree.Tree):
                etext = " ".join([word for word, tag in
entity.leaves()])
                label = entity.label()
            else:
                continue

            if label == 'PERSON':
                key = 'persons'
            elif label == 'ORGANIZATION':
                key = 'organizations'
            elif label == 'LOCATION':
                key = 'locations'
            elif label == 'GPE':
                key = 'other'
            else:
                key = None

            if key:
                results[fileid][key].append(etext)

    return results
```

In this next block of code, we will apply the NLTK standard chunker, Stanford Named Entity Recognizer, and Polyglot extractor to our corpus. We pass our data, the “top” and “references” section of the two documents of interest, into the functions created with each NERC tool and build a nested dictionary of the extracted entities—author names, locations,

and organization names. This code may take a bit of time to run (30 secs to a minute).

```
# Only extract our annotated files.
fids  = ['p19.txt', 'p29.txt']

# NLTK Entities
nltkents = nltk_entities(fids, section='top')

# Polyglot Entities
polyents = polyglot_entities(fids, section='top')

# Stanford Model Loading
root  = os.path.expanduser('~/models/stanford-ner-2014-
01-04/')
model = os.path.join(root,
'classifiers/english.muc.7class.distsim.crf.ser.gz')
jar   = os.path.join(root, 'stanford-ner-2014-01-04.jar')

# Stanford Entities
stanents = stanford_entities(model, jar, fids,
section='top')
```

We will focus specifically on the "persons" entity extractions from the “top” section of the documents to estimate performance. However, a similar exercise is possible with the extractions of “organizations” entity extractions or “locations” entity extractions too, as well as from the “references” section.

To get a better look at how each NERC tool performed on the named person entities, we will use the `Pandas` dataframe. `Pandas` is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. The dataframe provides a visual comparison of the extractions from each NERC tool and the hand-labeled extractions. Just a few lines of code accomplish the task:

```
import pandas as pd
import json

p19Authors = json.load(open('./data/p19Truth.json'))
```

```
df1 = pd.Series(polyents['p19.txt']['persons'],
index=None,
dtype=None, name='Polyglot NERC Authors', copy=False,
fastpath=False)

df2=pd.Series([re.sub('\*',"",l) for l in
stanents['p19.txt']['persons']],
index=None, dtype=None, name='Stanford NERC Authors',
copy=False, fastpath=False)

df3=pd.Series([re.sub('\*',"",l) for l in
nltkents['p19.txt']['persons']],
index=None, dtype=None, name='NLTKStandard NERC Authors',
copy=False, fastpath=False)

df4 = pd.Series(p19Authors['authors'], index=None,
dtype=None, name='Hand-labeled True Authors', copy=False,
fastpath=False)

met = pd.concat([df4,df3,df2,df1], axis=1).fillna('')
met
```

	Hand-labeled True Authors	NLTKStandard NERC Authors	Stanford NERC Authors	Polyglot NERC Authors
0	Tim Althoff	Timeline	Tim Althoff	Tim Althoff
1	Xin Luna Dong	Tim Althoff	Xin Luna Dong	Xin Luna Dong
2	Kevin Murphy	Xin Luna Dong	Kevin Murphy	Kevin Murphy
3	Safa Alai	Kevin Murphy	Safa Alai	Safa
4	Van Dang	Safa Alai		Van Dang
5	Wei Zhang	Van Dang		Wei Zhang
6		Wei Zhang		
7		Stanford		
8		Mountain View		

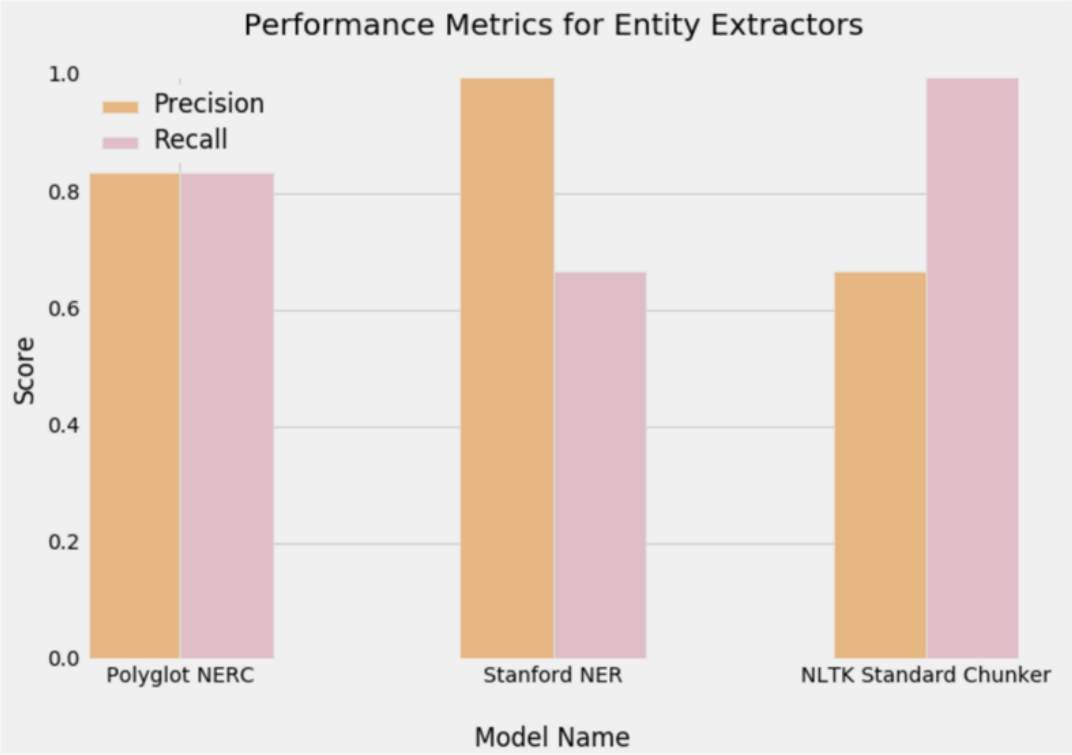
The above dataframe illustrates the mixed results from the NERC tools. NLTK Standard NERC appears to have extracted 3 false positives while the Stanford NER missed 2 true positives and the Polyglot NERC extracted all but one true positive (partially extracted; returned first name only).

Let's calculate some key performance metrics:

1. **True Negatives (TN)**: case was negative and predicted negative

2. **True Positives (TP)**: case was positive and predicted positive
3. **False Negatives (FN)**: case was positive but predicted negative
4. **False Positives (FP)**: case was negative but predicted positive

Here is a quick figure summarizing overall model performance.



The following function calculates the metrics for the three NERC tools:

```
# Calculations and logic from
http://www.kdnuggets.com/faq/precision-recall.html

def metrics(truth,run):
    truth = truth
    run = run
    TP = float(len(set(run) & set(truth)))

    if float(len(run)) >= float(TP):
        FP = len(run) - TP
    else:
        FP = TP - len(run)
    TN = 0
    if len(truth) >= len(run):
```

```
        FN = len(truth) - len(run)
    else:
        FN = 0

    accuracy = (float(TP)+float(TN))/float(len(truth))
    recall = (float(TP))/float(len(truth))
    precision = float(TP)/(float(FP)+float(TP))
    print "The accuracy is %r" % accuracy
    print "The recall is %r" % recall
    print "The precision is %r" % precision

    d = {'Predicted Negative': [TN,FN], 'Predicted
Positive': [FP,TP]}
    metricsdf = pd.DataFrame(d, index=['Negative
Cases','Positive Cases'])

    return metricsdf
```

Now let's pass our values into the function to calculate the performance metrics:

```
print
print
str1 = "NLTK Standard NERC Tool Metrics"

print str1.center(40, ' ')
print
print
metrics(p19Authors['authors'], [re.sub('\*',"",l) for l
in nltkents['p19.txt']['persons']])
```

```
NLTK Standard NERC Tool Metrics

The accuracy is 1.0
The recall is 1.0
The precision is 0.6666666666666666
```

	Predicted Negative	Predicted Positive
Negative Cases	0	3
Positive Cases	0	6

```
print
print
```

```
str2 = "Stanford NERC Tool Metrics"

print str2.center(40, ' ')
print
print
metrics(p19Authors['authors'], [re.sub('\*',"",l) for l
in stanents['p19.txt']['persons']])
```

```
Stanford NER Metrics

The accuracy is 0.6666666666666666
The recall is 0.6666666666666666
The precision is 1.0
```

	Predicted Negative	Predicted Positive
Negative Cases	0	0
Positive Cases	2	4

```
print
print
str3 = "Polyglot NERC Tool Metrics"

print str3.center(40, ' ')
print
print
metrics(p19Authors['authors'],polyents['p19.txt']
['persons'])
```

```
Polyglot NERC Tool Metrics

The accuracy is 0.8333333333333334
The recall is 0.8333333333333334
The precision is 0.8333333333333334
```

	Predicted Negative	Predicted Positive
Negative Cases	0	1
Positive Cases	0	5

The basic metrics above reveal some quick takeaways about each tool based on the specific extraction task. The NLTK

Standard Chunker has perfect accuracy and recall but lacks in precision. It successfully extracted all the authors for the document, but also extracted 3 false entities. NLTK's chunker would serve well in an entity extraction pipeline where the data scientist is concerned with identifying all possible entities

The Stanford NER tool is very precise (specificity vs sensitivity). The entities it extracts were 100% accurate, but it failed to identify half of the true entities. The Stanford NER tool would be best used when a data scientist wanted to extract only those entities that have a high likelihood of being named entities, suggesting an unconscious acceptance of leaving behind some information.

The Polyglot Named Entity Recognizer identified five named entities exactly, but only partially identified the sixth (first name returned only). The data scientist looking for a balance between sensitivity and specificity would likely use Polyglot, as it will balance extracting the 100% accurate entities and those which may not necessarily be a named entity.

A Simple Ensemble Classifier

In our discussion above, we notice the varying levels of performance by the different NERC tools. Using the idea that combining the outputs from various classifiers in an ensemble method can improve the reliability of classifications, we can improve the performance of our named entity extractor tools by creating an ensemble classifier. Each NERC tool had at least 3 named persons that were true positives, but no two NERC tools had the same false positive or false negative. Our ensemble classifier voting rule is very simple: Return all named entities that exist in at least two of the true positive named entity result sets from our NERC tools.

We implement this rule using the `set` module. We first do an `intersection` operation of the NERC results vs the hand labeled entities to get our "true positive" set.

Here is our code to accomplish the task:

```
# Create intersection of true authors from NLTK standard
output
a =set(sorted(nltkents['p19.txt']['persons'])) &
set(p19Authors['authors'])

# Create intersection of true authors from Stanford NER
output
b =set(sorted(stanents['p19.txt']['persons'])) &
set(p19Authors['authors'])

# Create intersection of true authors from Polyglot
output
c = set(sorted(polyents['p19.txt']['persons'])) &
set(p19Authors['authors'])

# Create union of all true positives from each NERC
output
(a.union(b)).union(c)
```

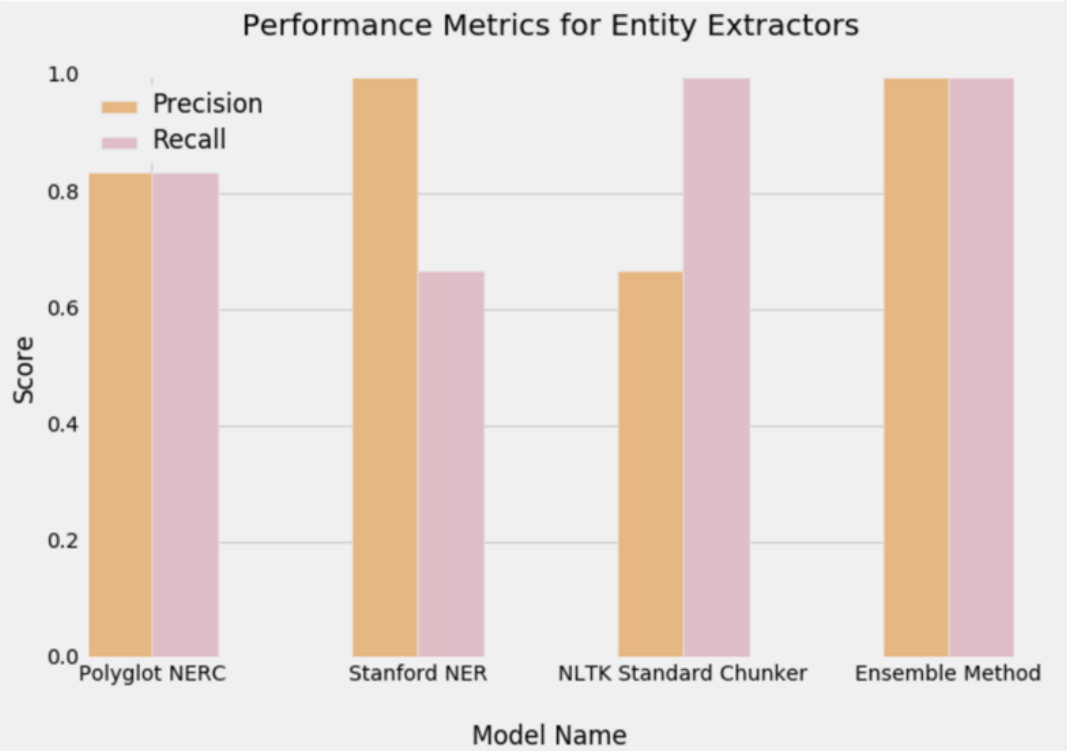
```
    {'Kevin Murphy',
     'Safa Alai',
     'Tim Althoff',
     'Van Dang',
     'Wei Zhang',
     'Xin Luna Dong'}
```

To get a visual comparison of the extractions for each tool and the ensemble set side by side, we return to our dataframe from earlier. In this case, we use the `concat` operation in pandas to append the new ensemble set to the dataframe.

```
dfensemble = pd.Series(list((a.union(b)).union(c)),
index=None, dtype=None, name='Ensemble Method Authors',
                       copy=False, fastpath=False)
met = pd.concat([df4,dfensemble,df3,df2,df1],
axis=1).fillna('')
met
```

	Hand-labeled True Authors	Ensemble Method Authors	NLTKStandard NERC Authors	Stanford NERC Authors	Polyglot NERC Authors
0	Tim Althoff	Wei Zhang	Timeline	Tim Althoff	Tim Althoff
1	Xin Luna Dong	Tim Althoff	Tim Althoff	Xin Luna Dong	Xin Luna Dong
2	Kevin Murphy	Xin Luna Dong	Xin Luna Dong	Kevin Murphy	Kevin Murphy
3	Safa Alai	Van Dang	Kevin Murphy	Safa Alai	Safa
4	Van Dang	Kevin Murphy	Safa Alai		Van Dang
5	Wei Zhang	Safa Alai	Van Dang		Wei Zhang
6			Wei Zhang		
7			Stanford		
8			Mountain View		

First, a quick visual to see how performance improved.



And we get a look at the performance metrics to see if we push our scores up in all categories:

```
print
print
str = "Ensemble NERC Metrics"

print str.center(40, ' ')
print
print
metrics(p19Authors['authors'],list((a.union(b)).union(c))
)
```

```
Ensemble NERC Metrics

The accuracy is 1.0
The recall is 1.0
The precision is 1.0
```

	Predicted Negative	Predicted Positive
Negative Cases	0	0
Positive Cases	0	6

Exactly as expected, we see improved performance across all performance metric scores and, in the end, get a perfect extraction of all named persons from this document.

Before we go any further, the idea of moving from "okay" to "perfect" is unrealistic. Moreover, this is a very small sample and only intended to show the application of an ensemble method. Applying this method to other sections of the journal articles will not lead to a perfect extraction, but it will indeed improve the performance of the extraction considerably.

Getting Your Data in Open File Format

A good rule for any data analytics project is to store the results or output in an open file format. I selected JavaScript Object Notation (JSON), which is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.

Let's take our list of persons from the ensemble results, store it as a Python dictionary, and then convert it to JSON. Alternatively, we could use the `dumps` function from the `json` module to return dictionaries, and ensure we get the open file format at every step. This way, other data scientists or users

could pick and choose what portions of code to use in their projects.

```
# Add ensemble results for author to the nested python
dictionary; use our functions
p19=
{'docName':'p19.txt','ensembleAuthors':list((a.union(b)).
union(c)),
     'body':list(sectpull(['p19.txt'], section='body'))
[0][1],'title':list(titles(['p19.txt']))[0]}

# covert nested dictionary to json for open data storage
# json can be stored in mongodb or any other disk store
output = json.dumps(p19, ensure_ascii=False,indent=3)

# print out the authors section we just created in our
json
print json.dumps(json.loads(output)
['ensembleAuthors'],indent=3)

# uncomment to see full json output
#print json.dumps((json.loads(output)),indent=3)
```

```
[
   "Wei Zhang",
   "Tim Althoff",
   "Xin Luna Dong",
   "Van Dang",
   "Kevin Murphy",
   "Safa Alai"
]
```

Conclusion

In this post, we've covered the entire data science pipeline in a natural language processing job that compared the performance of three different NERC tools. A core task in this pipeline involved ingesting plaintext into an NLTK corpus so that we could easily retrieve and manipulate the corpus. Then we used the results from the various NERC tools to create a simplistic ensemble classifier that improved the overall performance.

The techniques in this post can be applied to other domains, larger datasets or any other corpus. Everything I used in this post (with the exception of the Regular expression resource from Coursera) was not taught in a classroom or structured learning environment. It all came from online resources, posts from others, and books (that includes learning how to code in Python). If you have the motivation, you can do it.

Further Reading and Other Resources

Throughout the article, there are hyperlinks to resources and reading materials for reference, but here is a central list:

- Requirements to run this code in iPython notebook or on your machine
- Natural Language Toolkit Book (free online resource) and the NLTK Standard Chunker and a post on how to use the chunker
- Polyglot natural language pipeline for massive muliligual applications and the journal article describing the word classification model
- Stanford Named Entity Recognizer and the NLTK interface to the Stanford NER and a post on how to use the interface
- Python Pandas is a must have tool for anyone who does analysis in Python. The best book I've used to date is Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython
- Intuitive description and examples of Python's standard library set module
- Discussion of ensemble classifiers
- Nice module to print tables in standard python output called tablulate
- Regular expression training (more examples in earlier sections)

- Python library to extract text from PDF and post on available Python tools to extract text from a PDF
- ACM Digital Library to purchase journal articles to completely recreate this exercise
- My quick web scrap code to pull back abstracts and authors from KDD 2015; can apply this same analysis to web acquired dataset

District Data Labs provides data science consulting and corporate training services. We work with companies and teams of all sizes, helping them make their operations more data-driven and enhancing the analytical abilities of their employees. Interested in working with us? Let us know!

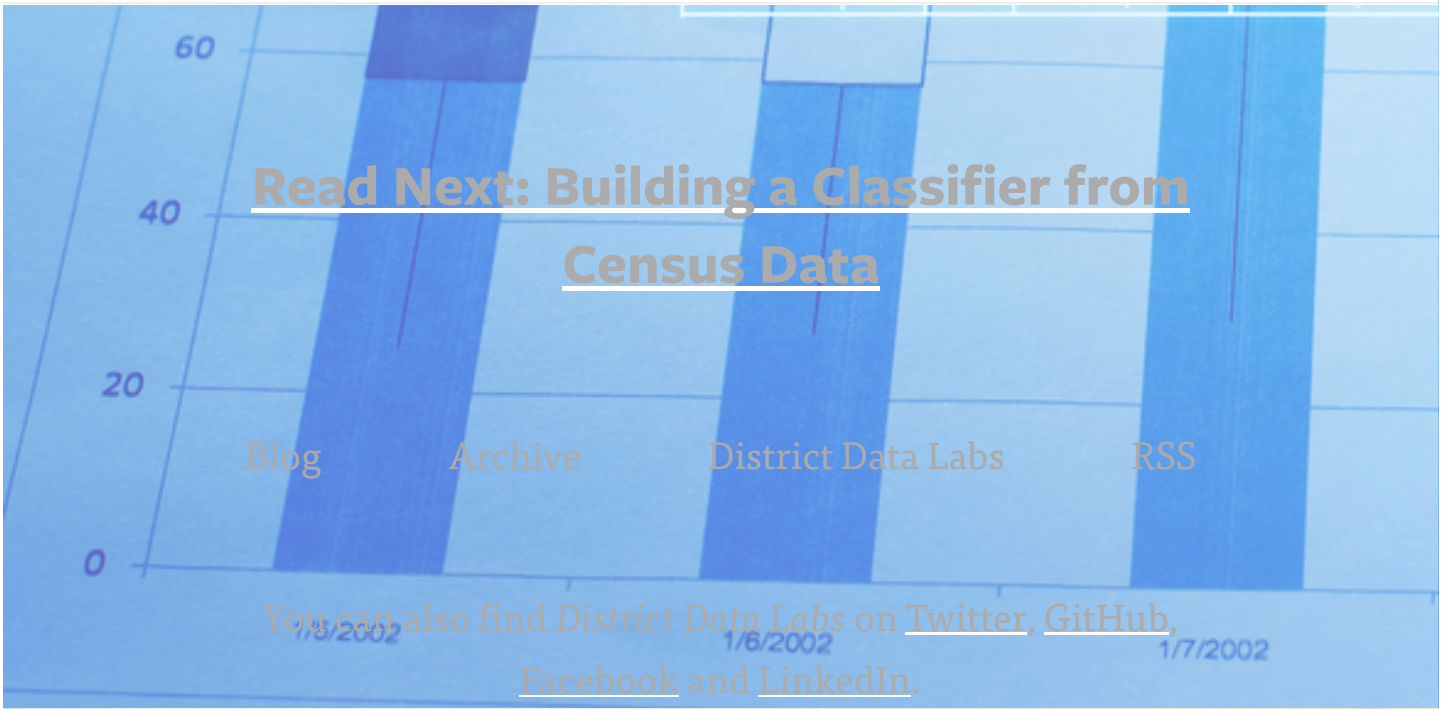
Linwood Creekmore III

May 11, 2016

2 Comments

Subscribe to this Blog

Search



Read Next: Building a Classifier from Census Data

Blog Archive District Data Labs RSS

You can also find *District Data Labs* on Twitter, GitHub, Facebook and LinkedIn.


© 2018 District Data Labs
35%
8%
120
100
80
60
40
20
0
1/5/2002
1/6/2002
1/7/2002