Codeword Development Dataset

January 26, 2016

Done by Linan Qiu (github.com/linanqiu) for TextIQ
In this notebook, we generate a development / test dataset for the codeword detection problem.
The codeword detection problem can be framed as the following:

- A codeword is a word that is used in a different context from its usual context. (eg. using `cheeseburger` to denote `equities` in an email between equities traders)
- Given a corpus containing codewords (**codeword corpus**) and a reference corpus not containing codewords (**reference corpus**), find codewords in the codeword corpus.

As far as the author knows, no dataset specifically containing a known list of codewords exist. Hence, a synthetic dataset is needed to facilitate experiments for this problem.

1 Approach Overview

To generate a synthetic codeword corpus, we do the following steps

1. Get a reference corpus containing little to no codewords
2. Select $|\mathbf{x}|$ words as codewords. Denote this set as $\mathbf{x}$ where each x_i is a codeword
3. Generate a set $\mathbf{y}$ where each $y_i = x_j | j \neq i$ ie. each word in $\mathbf{y}$ maps to a random word in $\mathbf{x}$ that is not the same as itself
4. Generate a codeword corpus by replacing every x_i in reference corpus with y_i (which will not be the same as x_i)

Then, to verify a detection method's validity, we can

1. Provide the detector both corpuses (codeword and reference)
2. Make detector find codewords in codeword corpus
3. Compare against $\mathbf{y}$ for retrieval rate

However, this still leaves some questions open, including what reference corpus to choose, how adequate words can be chosen as codeword candidates, and how a baseline for codeword detection can be made. This will be addressed along the way.

First we get some hygiene `python` stuff out of the way and make our graphs pretty.

In [1]:
```
%matplotlib inline

import cPickle as pickle
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

2 Synthetic Codeword Corpus Creation

2.1 Reference Corpus Choice

We choose the WSJ Corpus from the PennTreebank to be the reference corpus due to the similarity in the content of the text and the data of potential TextIQ clients.

A helper function is made in `lib.corpus_parser` to use `nltk` to parse the WSJ corpus. Tip to future replications: use a symlink to link `LDC99T42-Treebank-3/package/treebank_3/parsed/mrg/wsj` and `LDC99T42-Treebank-3/package/treebank_3/parsed/mrg/brown` to `~/nltk_data/corpora/ptb/wsj` and `~/nltk_data/corpora/ptb/brown` so that `nltk` can parse the WSJ corpus directly.

```
In [ ]: # reads wsj corpus and saves all lines to all-lines.pkl
        from lib.corpus_parser import *

        all_lines = parse_wsj()
        pickle.dump(all_lines, open('all-lines.pkl', 'wb'))
```

2.2 Codeword Selection

We need a way to select $|\mathbf{x}|$ codewords (in this case, $|\mathbf{x}| = 100$ an arbitrary number base on intuition about the size of the corpus and usage patterns.)

2.2.1 TF-IDF

A naive way to do this would be TF-IDF as was initially used where

- T_{x_i} is the number of times word x_i appeared in the entire corpus
- D_{x_i} is the number of documents x_i appeared in the entire corpus
- N_D is the number of documents in the entire corpus ie.
- $\text{Rank}_{x_i} = \text{TFIDF}_{x_i} = (\log(T_{x_i} + 1) * (\log(1 + N_D/D_{x_i}))$ where the left term is the term frequency smoothed such that 0s won't occur, and the right term is the inverse document frequency.

We select the top $|\mathbf{x}|$ words. The intuition behind this method was to downweight terms that appeared in every document (such as "the", "a", "he"), and upweight rarer terms.

However, this method was disappointing since it upweighted rare terms too much. In particular, note that the inverse document frequency portion of TFIDF is a decreasing function. That means words appearing only 1 time is upweighted the most. Hence, a lot of names and esoteric words that only occur once were selected. The top words consisted of "yeargin", "steinhardt", "corry", "psyllium". We needed an alternative method that downweighted very frequent words, upweighted rarer words, but also **downweighted extremely rare words**.

2.2.2 Gamma Distribution

A good way to do this was to use a Gamma distribution instead of the inverse document frequency measure. That means

$\text{Rank}_{x_i} = (\log(T_{x_i} + 1) * (g(D_{x_i}))$ where $g(y)$ is the Gamma distribution function fitted on a given α and β. Specifically, we can set α and β such that the mode of the distribution is a certain proportion of documents (ie. this is the proportion of documents that "important words" should belong in) and the mean to be 2 times of that. Choosing Mode $= 0.075 * N_D = (\alpha - 1) * \beta$ and hence Mean $= 0.15 * D = \alpha * \beta$, we make the assumption that important words should be present in around 7.5% to 15% of articles.

This Gamma distribution could would like this:

```
In [ ]: d = len(all_lines)
        beta = 0.075 * d
        alpha = 0.15 * d / beta
        loc = 0
```

```
from scipy.stats import gamma
import numpy as np

rv = gamma(a=alpha, loc=loc, scale=beta)
x = np.linspace(0, 2400)
plt.plot(x, rv.pdf(x))
plt.show()
```

This function upweights words that appear not so frequently, but downweights words that appear very rarely (along with words that appear frequently). The results turned out to be very reasonable for the WSJ corpus. Top words include "bonds", "index", "japanese", "oil", "traders".

This result should not be surprising given that TF-IDF is a measure meant for weighting words in a single document among many documents, not an aggregate measure across documents.

This allows us to select $\mathbf{x}$ by taking the top $|\mathbf{x}| = 100$ words using Rank.

2.3 Generate Substitution Key

Now that we have $\mathbf{x}$ we want to generate a set $\mathbf{y}$ permuted version of $\mathbf{x}$ such that $y_i = x_j | j \neq i$ ie. each word in $\mathbf{y}$ maps to a random word in $\mathbf{x}$ that is not the same as itself. This is rather trivial, and we save the key in a pickle and a `.json` so that we can read it easily. We also present the substituted dictionary in the output below.

```
In [ ]: from lib.substitute import *

        substitute_key = generate_substitute_key(all_lines)
        pickle.dump(substitute_key, open('substitute-key.pkl', 'wb'))
        import simplejson
        f = open('substitute-key.json', 'wb')
        simplejson.dump(substitute_key, f)
        f.close()

        print(substitute_key)
```

2.4 Codeword Corpus Generation

Now we can generate a codeword corpus by going through the reference corpus and replacing every occurence of word x_i with y_i. We retain the original reference corpus and save both in pickles.

```
In [ ]: all_lines_substituted = generate_substitute_corpus(all_lines, substitute_key)
        pickle.dump(all_lines_substituted, open('corpus-substituted.pkl', 'wb'))
        pickle.dump(all_lines, open('corpus-original.pkl', 'wb'))
```

3 Verification of Baseline Codeword Detection

3.1 Generate Embeddings

Now we run both corpuses over `word2vec`. The code below by default reads in the models from storage. This is because I usually run the training on a computer other than my own, and usually on two different computers. However, if you decide to torture your computer, feel free to take around 12 hours on this step (based on the default parameters in `w2v.w2v`). The default parameters used are `model = gensim.models.Word2Vec(sentences, min_count=5, workers=8, iter=300, window=15, size=300, negative=25)` as recommended by Jasneet.

```
In [2]: from w2v.w2v import *

        # all_lines_original = build_sentences(all_lines_original_filename)
        # all_lines_substituted = build_sentences(all_lines_substituted_filename)
        #
        # original_model = model_from_sentences(all_lines_original)
        # substituted_model = model_from_sentences(all_lines_substituted)
        #
        # original_model.save_word2vec_format('./models/corpus-original-w2v.mdl', binary=True)
        # substituted_model.save_word2vec_format('./models/corpus-substituted-w2v.mdl', binary=True)

        original_model = model_from_saved("./models/corpus-original-w2v.mdl", binary=True)
        substituted_model = model_from_saved("./models/corpus-substituted-w2v.mdl", binary=True)

        substitute_key = pickle.load(open('substitute-key.pkl', 'rb'))
```

3.2 Finding Common Words

Then we find the common words in both corpuses. Since we are using a synthetic dataset, and there is always a one to one matching between $\mathbf{x}$ and $\mathbf{y}$, the two corpuses should contain the same vocabularies. However, this won't always be the case in actual situations.

```
In [3]: from comparison.similarity import *
        intersect_vocab = set(substituted_model.vocab).intersection(set(original_model.vocab))
```

3.3 Detecting Codewords

We detect codewords by measuring, from the two corpuses, what a word is similar to.

- For each word w_i, we find $\mathbf{z}_{w_i,x,n}$, the top n words that are similar to w_i as predicted by `word2vec` in gensim's `most_similar` method using the **reference corpus**
- For that same word w_i we find $\mathbf{z}_{w_i,y,n}$ the top n words that are similar to w_i as predicted by the same method but using the **codeword corpus**
- We find $\mathbf{S}$, the intersection of $\mathbf{z}_{w_i,x,n}$ and $\mathbf{z}_{w_i,y,n}$. In other words, every word in $S_i \in \mathbf{S}$ is defined by $S_i \in \{\mathbf{z}_{w_i,x,n} \cap \mathbf{z}_{w_i,y,n}\}$.
- We record the length of the intersection, $|\mathbf{S}|$, for each word w_i.

```
In [4]: vocabs = generate_similarity_counts(original_model, substituted_model, intersect_vocab)
        pickle.dump(vocabs, open('vocabs-compared.pkl', 'wb'))
```

Intuition tells us that codewords are used differently in the codeword corpus than in the reference corpus. In that case $|\mathbf{S}|$ should be small for codewords and large for non-codewords. Non-codewords are used similarly in both reference corpus and codeword corpus, while codewords are used differently in both corpuses. We can use this to detect codewords.

4 Results

4.1 Codeword Isolation

We find that we were able to isolate words that have few similarity intersection counts. The distribution of similarity intersection counts across all words look normal with the exception of a cluster of codewords at the left tail.

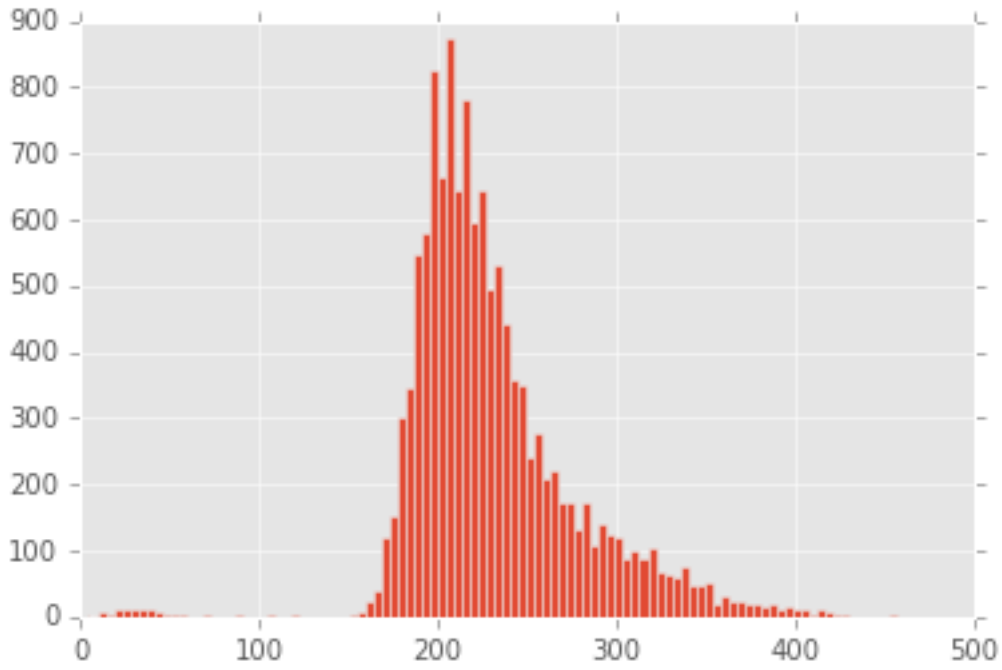
```
In [5]: # plot results
        codewords = sorted(vocabs, key=vocabs.get)
```

```
positive = 0
false_positive = 0
count = 1
results = {}

for word in codewords:
    if word in substitute_key:
        positive += 1
    else:
        false_positive += 1
    results[count] = {'count': count, 'positive': positive, 'false_positive': false_positive, '
    count += 1

import pandas as pd
import matplotlib
import matplotlib.pyplot as plot
matplotlib.style.use('ggplot')

# distribution of comparison counts
plot.hist(vocabs.values(), bins=100)
plot.show()
```

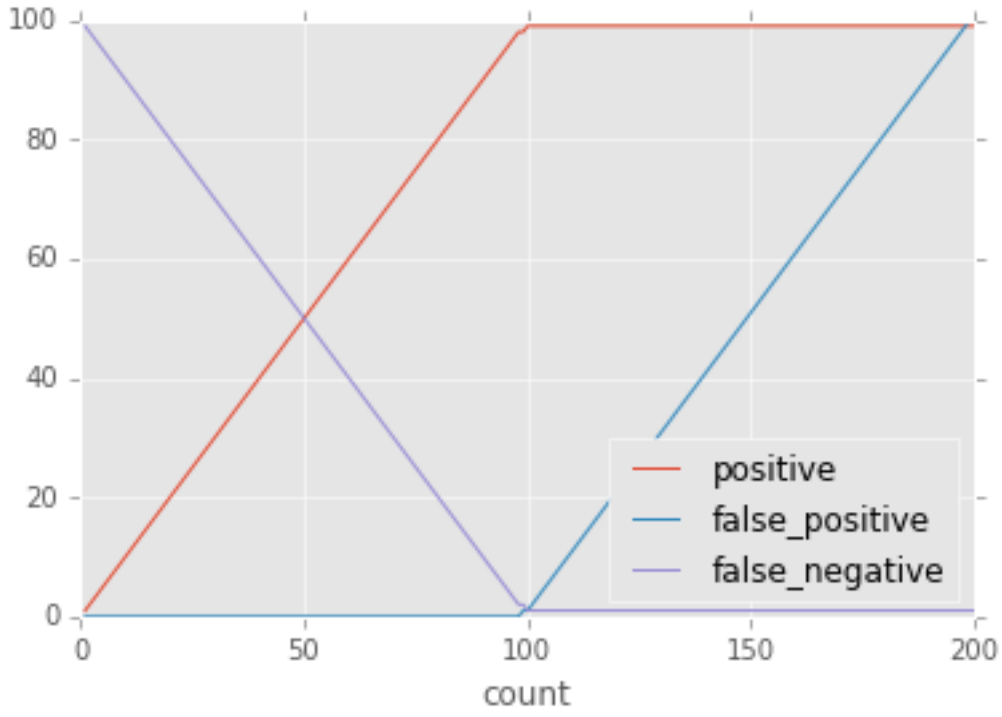


We can check that the cluster of words at the left tail are indeed codewords. We first sort the all words by similarity count in ascending order. The words with fewest similarity counts will be at the front.

Then, we iterate through the words and record (for every count)

- `positive`: number of actual codewords correctly identified
- `false_positive`: number of words wrongly classified as codeword
- `false_negative`: number of actual codewords that were not detected

In [6]:
```
# plot results
dataframe = pd.DataFrame.from_dict(results, orient='index')
dataframe.plot(x='count', y=['positive', 'false_positive', 'false_negative'], xlim=[0, 200], yl
plot.show()
```



As we increase the counts. We find that at a count of around 100, we have a very high positive rate with a very low false positive and false negative rate. Even lower false positive rates can be achieved by sacrifising some false negative rates.

In [7]:
```
threshold = 100
positive = 0
false_positive = 0
count = 1

for word in codewords:
    if word in substitute_key:
        positive += 1
    else:
        false_positive += 1
    count += 1

    if count > 100:
        break

print('positive: %d\nfalse positive: %d\nfalse negative: %d' % (positive, false_positive, len(su
```

```
positive: 99
false positive: 1
false negative: 1
```

4.2 Codeword Recovery

It is also meaningful to see if we can successfully recover the original words given the codewords. That is, given $\mathbf{y}$ and $\mathbf{x}$, can we successfully produce a function that matches each y_i to each x_i.

```
In [8]: def check_substitution(substituted_model, original_model, substituted_word, original_word, n):
            similar_substituted = set((word[0]) for word in substituted_model.most_similar(substituted_
            similar_original = set((word[0]) for word in original_model.most_similar(original_word, top
            intersection = similar_substituted & similar_original
            return len(intersection)
```

We remind the reader that the this was the original substitution key. This means, for example, that the word `stake` in $\mathbf{x}$ has been replaced by the word `credit` in $\mathbf{y}$.

```
In [9]: print(substitute_key)
```

```
{u'stake': u'credit', u'office': u'california', u'show': u'state', u'september': u'chicago', u'dollar':
```

This means that among the vocabulary in $\mathbf{x}$ and $\mathbf{y}$, the ones that should produce the most similarities are the original substitutions. For example, since `credit` was substituted with `stake`, this pair should have many matches out of 500 words.

Contrast this with `credit` and `california`, which is not an original codeword pair and hence won't have as high a number of matches.

We can exploit this to recover the original codewords.

```
In [10]: check_substitution(substituted_model, original_model, 'credit', 'stake', 500)
```

```
Out[10]: 385
```

```
In [11]: check_substitution(substituted_model, original_model, 'credit', 'california', 500)
```

```
Out[11]: 31
```

We produce a `substitution_recovered` that is the same format as the original `substitute_key`, but recovered without knowledge of the `substitute_key`.

```
In [14]: codeword_count = len(substitute_key)

         substitution_recovered = {}

         for original_word in codewords[:codeword_count]:
             max_count = 0
             max_substitute = ""
             for substitute_word in substitute_key.keys():

                 count = check_substitution(substituted_model, original_model, substitute_word, original
                 if(count > max_count):
                     max_count = count
                     max_substitute = substitute_word

             substitution_recovered[original_word] = max_substitute

         print(substitution_recovered)
```

```
{u'stake': u'credit', u'office': u'california', u'show': u'state', u'september': u'chicago', u'dollar':
```

Turns out we were able to reproduce the matching between the codewords and the original words very successfully.

In [15]:
```
# precision = number of correct pairings out of constructed pairings

precision = 0
for original, substitute in substitution_recovered.iteritems():
    if original in substitute_key and substitute == substitute_key[original]:
        precision += 1

precision
```

Out[15]: 99

In [16]:
```
# recall = number of correct pairings out of original pairings

recall = 0
for original, substitute in substitute_key.iteritems():
    if original in substitution_recovered and substitute == substitution_recovered[original]:
        recall += 1

recall
```

Out[16]: 99