SciComp with Py

Text Classification and Retrieval
Part 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

- Review
- Assigning Weights to Terms
- The Search Engine Problem: Finding Texts Relevant to a Text Query

Review

Basic IR Terminology

- **Document** is an indexable and retrievable unit of digital media (text, image, audio, video)
- **Collection** is a set of documents that can be searched/clustered by users
- **Term/Word** is a wordform that occurs in a collection
- **Query** is a set of terms, an image, an audio sample, a video, or a combination thereof

Bags of Words

- In many ML textbooks/papers, texts are commonly referred to as **bags of words**
- **Bag of words** is, for all practical purposes, synonymous with **feature vector**
- In SKLEARN, objects that convert raw texts into bags of words are called **vectorizers**

Displaying Feature Matrix

```
texts = ['How to format my hard disk', ' Hard disk format problems ']
```

feature #

text number: 0 or 1

```
(0, 3)    1
(0, 6)    1
(0, 1)    1
(0, 4)    1
(0, 2)    1
(0, 0)    1
(1, 1)    1
(1, 2)    1
(1, 0)    1
(1, 5)    1
```

Feature 6 (u'to') occurs once in text 0

Features	Feature Numbers
u'disk'	0
u'format'	1
u'hard'	2
u'how'	3
u'my'	4
u'problems'	5
u'to''	6

Creating Feature Matrix from Text Directory

Py

```
from sklearn.feature_extraction.text import CountVectorizer
from utils import DATA_DIR

TOY_DIR = os.path.join(DATA_DIR, 'toy')
posts = [open(os.path.join(TOY_DIR, f)).read() for f in os.listdir(TOY_DIR)]
vectorizer = CountVectorizer(min_df=1)

feat_mat = vectorizer.fit_transform(posts)
num_samples, num_feats = feat_mat.shape
print("num samples: %d, num feats: %d" % (num_samples, num_feats))
print(vectorizer.get_feature_names())
```

Output

```
num samples: 5, num feats: 25
[ u'about', u'actually', u'capabilities', u'contains', u'data', u'databases',
u'images', u'imaging', u'interesting', u'is', u'it', u'learning', u'machine', u'most',
u'much', u'not', u'permanently', u'post', u'provide', u'save', u'storage',
u'store', u'stuff', u'this', u'toy' ]
```

Vector Space

- Suppose that all texts in our universe consist of three words: w1, w2, and w3
- Suppose that there are three texts T1, T2, and T3 such that

  T1 = “w1 w1 w2”

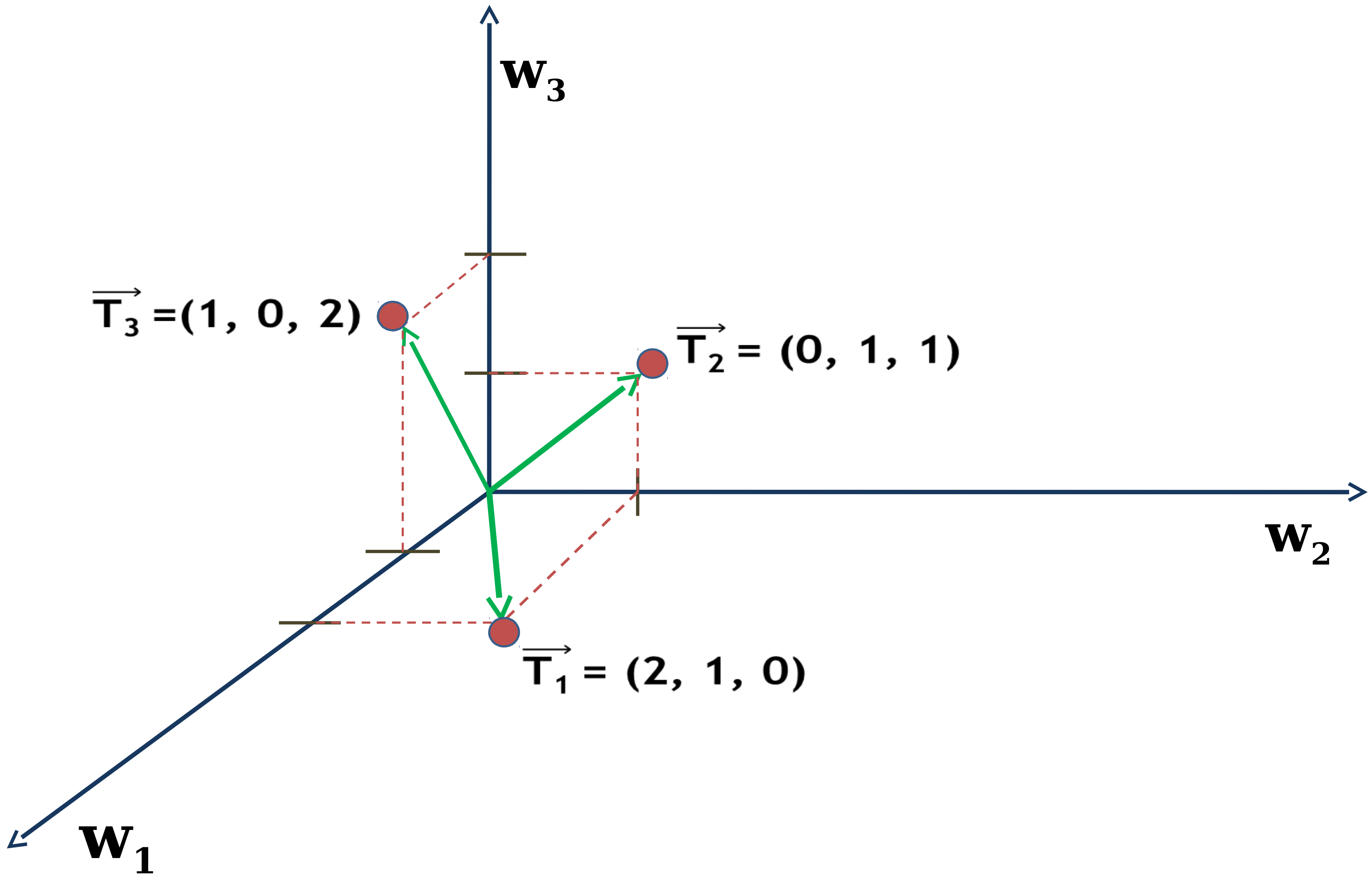
  T2 = “w3 w2”

  T3 = “w3 w3 w1”
- Suppose that we have our feature extractor that finds all words in texts and our feature weight assigner assigns to each found term its frequency in a given text

Example: Term Frequency Table

	W_1	W_2	W_3
T_1	2	1	0
T_2	0	1	1
T_3	1	0	2

Example: 3D Vector Space



Two Principal Tasks for Vector Space Model

- Vocabulary Normalization: how to compute terms in texts
- Term Weighting: how to assign weights to terms in texts

Stemming

- Stemming is a morphological operation that maps each word to its stem
- The basic objective of stemming is to reduce the size of each word bag, i.e., reduce the number of features
- In a typical stemmer, such words as 'image', 'images', 'imaging' are mapped to 'imag'

Stoplisting

- There are two vocabulary normalization operations typically associated with the vector space model of information retrieval: stoplisting and stemming
- Stoplisting discards all common words in texts
- Such common words are placed in the so-called **stoplist**
- Common words typically include articles and propositions: a, an, the, from, to, etc.
- Different systems may use different stoplists

Vectorizing Documents w/ Stoplisting

This is how stoplisting is enabled

Py

```
vectorizer_with_stop_words = CountVectorizer(min_df=1, stop_words='english')
feat_mat = vectorizer_with_stop_words.fit_transform(posts)
num_posts, num_feats = feat_mat.shape
print('features after stoplisting:')
print('#samples: %d, #features: %d' % (num_posts, num_feats))
print(vectorizer_with_stop_words.get_feature_names())
```

Output

```
features after stoplisting:
#samples: 5, #features: 18
[u'actually', u'capabilities', u'contains', u'data', u'databases', u'images',
u'imaging', u'interesting', u'learning', u'machine', u'permanently', u'post',
u'provide', u'save', u'storage', u'store', u'stuff', u'toy']
```

source in vectorizer_with_stopwords.py

Stemming with NLTK.STEM

nltk stands for 'natural language toolkit';
available at http://nltk.org/install.html;
this package has a bunch of stemmers

```
>>> import nltk.stem
>>> snowball_stemmer = nltk.stem.SnowballStemmer('english')
>>> snowball_stemmer.stem('imaging')
u'imag'
>>> snowball_stemmer.stem('image')
u'imag'
>>> snowball_stemmer.stem('images')
u'imag'
```

Assigning Weights to Words/Terms

How Important is a Word?

- OK, now we know how to extract terms from texts with stemming and stoplisting
- Next question: how to estimate the importance of a word in a word bag?
- A word exists in a specific word bag but it also exists in a collection of word bags

Local and Global Weights

- Local weight: the more frequent a word is in a word bag, the more relevant it is to that word bag
- Local weight is called **term frequency** (**TF**)
- Global weight: if a word is in many word bags, it is less important as a distinguishing feature
- Global weight is called **inverse document frequency** (**IDF**)
- Product of local and global weights is referred to as **TFIDF**

Term Frequency (TF)

D is a document

t is a term

S - how many terms occur in D at least once

f_t - how many times t occurs in D

$tf(t, D) = \frac{f_t}{S}$ is the term frequency of t in D

Inverse Document Frequency (IDF)

t is a term

C is a document collection

N is the number of documents in C

N_t is the number of documents in C that contain t at least once

$idf(t, C) = log(\frac{N}{N_t})$ is the inverse document frequency of t in C

TFIDF

t is a term

C is a document collection

D is a document in C

$tfidf(t, D, C) = tf(t, D) \times idf(t, C)$ is the tfidf metric

local importance of term t in document D

global importance of term t in collection C

Vectorizing Docs with TFIDF

The vectorizer that implements tfidf metric

```
from sklearn.feature_extraction.text import TfidfVectorizer
import nltk.stem

english_stemmer = nltk.stem.SnowballStemmer('english')

class StemmedTfidfVectorizer(TfidfVectorizer):
    def build_analyzer(self):
        analyzer = super(TfidfVectorizer, self).build_analyzer()
        return lambda doc: (english_stemmer.stem(w) for w in analyzer(doc))

tfidf_vectorizer = StemmedTfidfVectorizer(min_df=1, stop_words='english')
```

Toy Document Collection

```
## three are documents in our toy collection: A, ABB, ABC
A, ABB, ABC = ['a'], ['a', 'b', 'b'], ['a', 'b', 'c']
## DOCSET is a document collection; we assume that
## DOCSET is immutable
DOCSET = (A, ABB, ABC)
```

source in tfidf.py

Term Frequency

Py

```
# how frequent term t is in document docs
def term_freq(t, doc):
    # sum the occurrences of all terms in doc
    n = sum(doc.count(term) for term in set(doc))
    # compute the occurences of term t in doc
    tf = float(doc.count(t))
    # normalize tf by n
    print('n = %f, tf = %f' % (n, tf))
    return tf/n
```

Output

```
>>> term_freq('a', A)
n = 1.000000, tf = 1.000000
1.0
>>> term_freq('a', ABB)
n = 3.000000, tf = 1.000000
0.3333333333333333
>>> term_freq('b', ABC)
n = 3.000000, tf = 1.000000
0.3333333333333333
```

source in tfidf.py

Inverse Document Frequency

Py

```
def inverse_doc_freq(t, docset):
    num_docs = len(docset)
    num_docs_with_t = len([doc for doc in docset if t in doc])
    return sp.log(float(num_docs)/num_docs_with_t)
```

Output

```
>>> inverse_doc_freq('a', DOCSET)
0.0
>>> inverse_doc_freq('b', DOCSET)
0.4054651081081638
>>> inverse_doc_freq('c', DOCSET)
1.0986122886681098
```

TFIDF

Py

```python
def tfidf(t, doc, docset):
    tf = term_freq(t, doc)
    idf = inverse_doc_freq(t, docset)
    return tf*idf

A, ABB, ABC = ['a'], ['a', 'b', 'b'], ['a', 'b', 'c']
DOCSET = (A, ABB, ABC)
print("tfidf('a', a, D)\t=\t%f" % tfidf('a', A, DOCSET))
print("tfidf('b', abb, D)\t=\t%f" % tfidf('b', ABB, DOCSET))
print("tfidf('a', abc, D)\t=\t%f" % tfidf('a', ABC, DOCSET))
print("tfidf('c', abc, D)\t=\t%f" % tfidf('c', ABC, DOCSET))
```

source in tfidf.py

Output

```
tfidf('a', A, DOCSET)     =     0.000000
tfidf('b', ABB, DOCSET)   =     0.270310
tfidf('a', ABC, DOCSET)   =     0.000000
tfidf('c', ABC, DOCSET)   =     0.366204
```

The Search Engine Problem:
Finding Texts Relevant to a Text Query

Search Engine Problem

- There is a collection of documents (e.g., URLs) each of which is transformed into a bag of words
- The user enters a text (aka query)
- The search engine returns a list of documents related to the user's query

Two Important Questions

- **Indexing question**: How to transform a collection of documents into feature vectors?
- We have answered this question by treating each text as a bag of words
- **Retrieval question**: How to retrieve most relevant feature vectors?
- To answer this question, we need distance metrics to compute how close/similar one bag of words is to another

SCIPY.LINALG.NORM

linarg.norm(x) computes the square root of the sum of the squares of the elements in x

```
>>> v1 = np.array([[0, 3, 4, 5]])
>>> v2 = np.array([[7, 6, 3, -1]])
>>> v1 - v2
array([[-7, -3,  1,  6]])
>>> sp.linalg.norm(v1-v2) ## math.sqrt((-7)**2 + (-3)**2 + 1**2 + 6**2)
9.746794344808963
>>> v1 = np.array([[0, 1]])
>>> v2 = np.array([[1, 0]])
>>> v1 - v2
array([[-1,  1]])
>>> sp.linalg.norm(v1-v2)
1.4142135623730951
```

Two Common Distance Metrics for Bags of Words

linarg.norm(x) computes the square root of the sum of the squares of the elements in x, i.e., the magnitude of the vector

```
## euclidean distance
def dist_raw(v1, v2):
    delta = v1 - v2
    return scipy.linalg.norm(delta)

## normalized euclidean distance
def dist_raw_norm(v1, v2):
    v1_normalized = v1/scipy.linalg.norm(v1)
    v2_normalized = v2/scipy.linalg.norm(v2)
    delta = v1_normalized - v2_normalized
    return scipy.linalg.norm(delta)
```

source in text_retrieval_utils.py

Problem

Write a program that finds posts related to a given post by using the three vectorizers developed in this lecture.

source in find_closest_post_01.py and text_retrieval_utils.py

Solution: Finding Relevant Posts with CountVectorizer

```
vectorizer = CountVectorizer(min_df=1)
post_feat_mat = vectorizer.fit_transform(raw_posts)
num_raw_posts, num_raw_feats = post_feat_mat.shape

# new_raw_post is a user query; dist_fun is a distance metric
def find_closest_post(new_raw_post, dist_fun):
    global vectorizer
    global post_feat_mat
    global num_raw_posts
    global raw_posts
    find_closest_post_aux(vectorizer, new_raw_post,  raw_posts,
                          post_feat_mat, dist_fun, num_raw_posts)
```

Solution: Exploring Features & Feature Matrix

We have 25 features

```
>>> vectorizer.get_feature_names()
[u'about', u'actually', u'capabilities', u'contains', u'data', u'databases', u'images', u'imaging',
u'interesting', u'is', u'it', u'learning', u'machine', u'most', u'much', u'not', u'permanently', u'post',
u'provide', u'save', u'storage', u'store', u'stuff', u'this', u'toy']
```

We have 5 posts each of which is described in terms of 25 features

```
>>> post_feat_mat.shape
(5, 25)
```

These are the actual feature vectors

```
>>> post_feat_mat.toarray()
Array([[0, 0, 0, 0, 3, 3, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0],
      [1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1],
      [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
      [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]])
```

Solution: Exploring Posts and Features

We have 25 features

```
>>> vectorizer.get_feature_names()
[u'about', u'actually', u'capabilities', u'contains', u'data', u'databases', u'images', u'imaging',
u'interesting', u'is', u'it', u'learning', u'machine', u'most', u'much', u'not', u'permanently', u'post',
u'provide', u'save', u'storage', u'store', u'stuff', u'this', u'toy']
```

Raw text of post 0

```
>>> raw_posts[0]
'Imaging databases store data. Imaging databases store data. Imaging databases store data.\n'
```

Feature vector of post 0

```
>>> post_feat_mat.getrow(0).toarray()
array([[0, 0, 0, 0, 3, 3, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0]])

## the other four posts can be explored in the same fashion
```

Solution: Finding Relevant Texts

```
def find_closest_post_aux(v, new_raw_post, raw_posts, post_feat_mat, dist_fun, num_posts):
    # compute feature vector of new_raw_post
    new_post_vec = v.transform([new_raw_post]).getrow(0).toarray()
    assert new_post_vec is not None
    best_post = None
    best_dist = sys.maxint
    best_i = None # number of best post match
    for i in xrange(0, num_posts):
        raw_post = raw_posts[i]
        # skip the post itself
        if raw_post == new_raw_post:
            continue
        # retrieve feature vector of a post in post feature matrix
        post_vec = post_feat_mat.getrow(i).toarray()
        print('new_post_vec: %s' % str(new_post_vec))
        print('post_vec:         %s' % str(post_vec))
        # compute the distance b/w post fv and new post fv
        d = dist_fun(new_post_vec, post_vec)
        print '=== Post %i with dist=%.2f: %s' % (i, d, raw_post)
        if d < best_dist:
            best_dist = d
            best_i = i
    print('Best post is %i with dist = %.2f' % (best_i, best_dist))
```

Test Run with DIST_RAW

```
>>> find_closest_post('imaging databases', dist_raw)
new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 0 0 3 3 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0]]
=== Post 0 with dist=5.10: Imaging databases store data. Imaging databases store data. Imaging databases store data.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[1 1 0 1 0 0 0 0 1 1 1 1 1 0 1 1 0 1 0 0 0 0 1 1 1]]
=== Post 1 with dist=4.00: This is a toy post about machine learning. Actually, it contains not much interesting stuff.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]]
=== Post 2 with dist=1.41: Imaging databases store data.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0]]
=== Post 3 with dist=1.73: Imaging databases provide storage capabilities.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 0 0 0 1 1 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0]]
=== Post 4 with dist=2.00: Most imaging databases save images permanently.

Best post is 2 with dist = 1.41
```

Test Run with DIST_RAW_NORM

```
>>> find_closest_post('imaging databases', dist_raw_norm)
new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 0 0 3 3 0 3 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0]]
=== Post 0 with dist=0.77: Imaging databases store data. Imaging databases store data. Imaging databases store data.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[1 1 0 1 0 0 0 0 1 1 1 1 1 0 1 1 0 1 0 0 0 0 1 1 1]]
=== Post 1 with dist=1.41: This is a toy post about machine learning. Actually, it contains not much interesting stuff.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]]
=== Post 2 with dist=0.77: Imaging databases store data.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0]]
=== Post 3 with dist=0.86: Imaging databases provide storage capabilities.

new_post_vec: [[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
post_vec:     [[0 0 0 0 0 1 1 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0]]
=== Post 4 with dist=0.92: Most imaging databases save images permanently.

Best post is 0 with dist = 0.77
```

Vectorizers with Stoplisting and Stemming

- find_closest_post_02.py contains the counter vectorizer with stoplisting
- find_closest_post_03.py contains the counter vectorizer with stoplisting and stemming

References

W. Richert & L. Coelho. “Building ML Systems with Python”, Ch. 3, Pack, 2013.