SciComp with Py

Text Classification and Retrieval

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

- Information Retrieval (IR)
- Vector Space Model of Information Retrieval (IR)
- Vocabulary Normalization

Information Retrieval

Information Retrieval

- Information Retrieval (IR) is an area of natural language processing (NLP) that deals with storage and retrieval of digital media
- The primary focus of IR has been digital texts (note the rise of web search engines over the past decade)
- Video and audio are rapidly gaining ground

IR & Unsupervised Learning

- Recall that that in supervised learning there are labeled feature vectors: each feature vector is labeled with a class
- In unsupervised learning, there are still feature vectors but no labels
- Many text classification/retrieval problems are examples of unsupervised learning

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**

Compressed Sparse Row (CSR) Matrix

- **scipy.sparse.csr.csr_matrix** is a data structure commonly used in represent texts as bags of words
- Advantages: efficient arithmetic operations, efficient row slicing, efficient matrix vector products
- Disadvantages: slow column slicing operations, expensive changes (scipy has other data structures that handle these operations faster)

Example

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> mat = csr_matrix((4, 5), dtype=np.int8)
>>> mat.toarray()
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=int8)
```

Converting Texts into Bags of Words

```
import scipy as sp
from sklearn.feature_extraction.text import CountVectorizer

# create two simple texts
texts = ['How to format my hard disk', ' Hard disk format problems ']
print('text 0: %s' % texts[0])
print('text 1: %s' % texts[1])

# create a vectorizer
vectorizer = CountVectorizer(min_df=1)
feat_mat = vectorizer.fit_transform(texts)
print type(feat_mat)
print(vectorizer.get_feature_names())
```

min_df – minimum document frequency;
Include words that occur at least once

create feature matrix from the texts

print feature names, i.e., terms

source in explore_count_vectorizer.py

Displaying Feature Numbers

```
<class 'scipy.sparse.csr.csr_matrix'>
[u'disk', u'format', u'hard', u'how', u'my', u'problems', u'to']
```

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

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

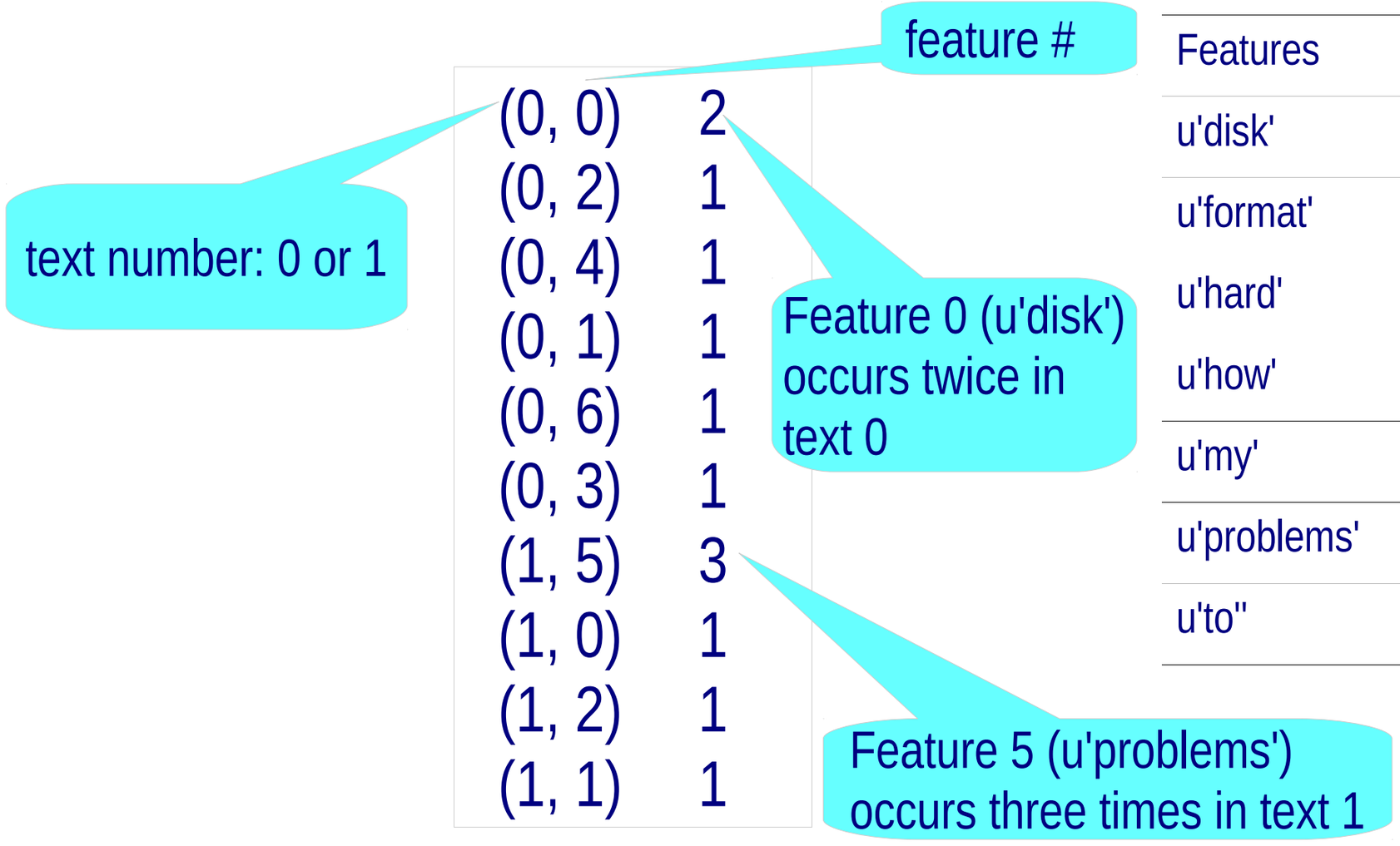
Adding Words to Texts

Let's change the two text documents to see how it affects the frequency counts. We'll add once occurrence of 'disk' to text 0 and two occurrences of 'problems' to text 1.

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

Displaying Feature Matrix

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



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

Displaying Feature Matrix

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

```
> print feat_mat.toarray()

[[1 1 1 1 1 0 1]
 [1 1 1 0 0 1 0]]
```

text 0

text 1

feature 0 is present in text 1

feature 4 is absent in text 1

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

Problem

Write a program to vectorize (i.e., turn into a feature matrix) a directory of text documents.

source in create_feat_mats_from_dir.py

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

Background

- Vector space model of IR was invented by Gerald Salton in the early 1970's
- Document collection is a vector space of document feature vectors
- Terms found in texts are dimensions of that vector space
- Terms are coordinates along specific dimensions

Example: a 3D Feature 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”

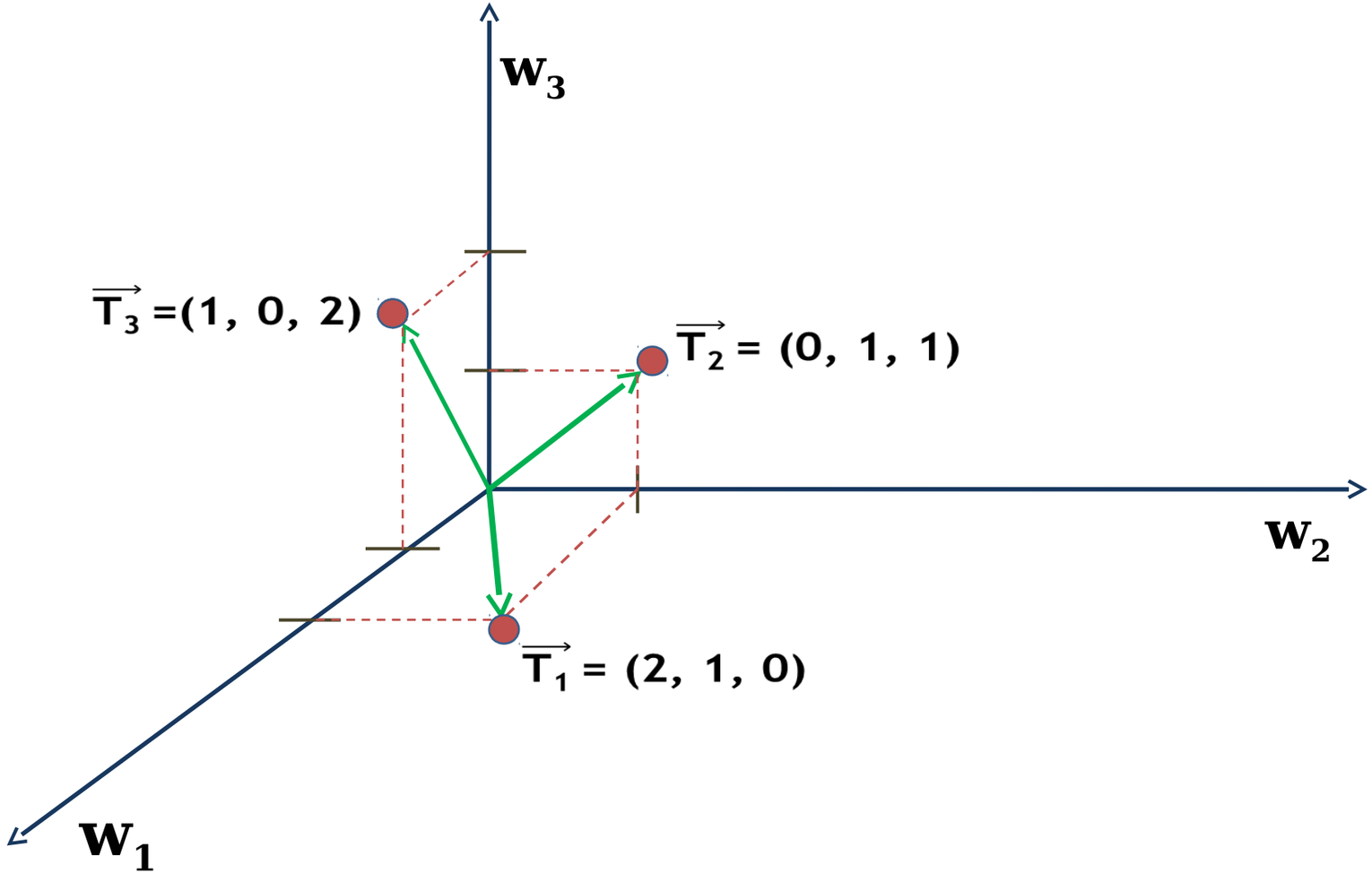
  T2 = “w3 w2”

  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 the 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



Another Example: a Binary Feature 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 1 if it is present in the text and 0 if it is absent

Example: Term Frequency Table

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

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

Vocabulary Normalization

Vocabulary Normalization

- Vocabulary normalization refers to a set of procedures that select which words are placed in bags (i.e., become features) and which are left out
- Vocabulary normalization also refers to a set of procedures that modify words before placing them in bags
- There are two common vocabulary normalization procedures in IR: **stemming** and **stoplisting**

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'

Pros & Cons of Vocabulary Normalization

Vocabulary normalization is done to improve retrieval performance and storage efficiency, because the size of the vocabulary (the set of terms) becomes smaller

Vocabulary normalization is never 100% correct: conflating CONNECTOR and CONNECTION to CONNECT is probably OK; conflating RELATIVE and RELATIVITY to REL is lossy

Porter Algorithm for Suffix Stripping

Martin Porter's original paper is at
http://tartarus.org/martin/PorterStemmer/def.txt

Source code in various languages is at
http://tartarus.org/martin/PorterStemmer/

Porter Algorithm: Stem Conflation

Given a list of stems and a list of suffixes, use a set of rules that match word forms and remove suffixes

Porter Algorithm: Consonants & Vowels

A consonant is a letter different from **A**, **E**, **I**, **O**, **U** and different from **Y** when it is preceded by a consonant

Y is a consonant when it is preceded by **A**, **E**, **I**, **O**, **U**: in **TOY**, **Y** is a consonant but in **BY** it is a vowel

A vowel is not a consonant

Porter Algorithm: Consonants & Vowels

A consonant is denoted as **c** and a vowel as **v**

A sequence of at least one consonant (e.g., **c**, **cc**, **ccc**, **cccc**, etc.) is denoted as **C**

A sequence of at least one vowel (e.g., **v**, **vv**, **vvv**, **vvvv**, etc.) is denoted as **V**

Porter Algorithm: Word Form Representation

Any word form can be represented as one of the sequences

CVCV ... C

CVCV ... V

VCVC ... C

VCVC ... V

These sequences are condensed into **[C]VCVC ... [V]**

and written as **[C](VC)^{m}[V]**, **m >= 0**

Porter Algorithm: Word Form Representation

In the formula **[C](VC)^{m}[V]**, **m >= 0**, **m** is the measure of a word

m = 0: TR, EE, TREE, Y, BY

m = 1: TROUBLE, OATS, TREES, IVY

- TROUBLE: **[C]** = TR; **(VC)^{1}** = OUBL**; [V]** = E

- OATS: **[C]** = NULL; **(VC)^{1}** = OATS**; [V]** = NULL

- TREES: **[C]** = TR; **(VC)^{1}** = EES**; [V]** = NULL

m = 2: TROUBLES, PRIVATE

- TROUBLES: **[C]** = TR; **(VC)^{2}** = (OUBL)(ES)**; [V]** = NULL

- PRIVATE: **[C]** = PR; **(VC)^{2}** = (IV)(AT)**; [V]** = E

Porter Algorithm: Suffix Removal Rules

Rules for removing suffixes are given in the form

(condition) S1 → S2

If a word ends with **S2** and and the stem before **S1** satisfies the optional condition, then **S1** is replaced with **S2**

Example:

(m > 1) EMENT → NULL

If a word ends in **EMENT** and the measure of the stem before **EMENT** is greater than 1, **EMENT** is removed; this rule maps **REPLACEMENT** to **REPLAC**

Porter Algorithm: Condition Specification

Conditions can be specified as follows:

(m > n), where **n** is a natural number

***X** – stem ends with the letter **X**

v - stem contains a vowel

***d** – stem with a double consonant (e.g., -**TT**)

***o** – stem ends in **cvc** where the second **c** is not **W**, **X** or **Y** (e.g., -**WIL**, -**HOP**)

Logical **AND**, **OR**, and **NOT** are allowed: **((m > 1) AND (*S OR *T))**

Porter Algorithm: Length-Based Rule Matching

If several rules match, the one with the longest **S1** wins

Consider this rule set with null conditions:

() **SSES** → **SS**

() **IES** → **I**

() **SS** → **SS**

() **S** → **NULL**

Given this rule set: **CARESSES** → **CARESS** (**S1** = **SS**) and

CARES → **CARE** (**S1** = **S**)

Porter Algorithm: Five Rule Sets

- In the original paper by Porter, there are eight sets of rules: 1A, 1B, 1C, 2, 3, 4, 5A, and 5B
- A word form passes through each rule set consecutively starting from 1A and ending at 5B, in that order
- If no rule is applicable, the word form comes out unmodified

Running Porter Stemmer

```
>>> p = PorterStemmer()
>>> p.stemWordForm('connection')
'connect'
>>> p.stemWordForm('caresses')
'caress'
>>> p.stemWordForm('relative')
'rel'
>>> p.stemWordForm('relativity')
'rel'
>>> p.stemWordForm('windows')
'window'
```

source in porter.py

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

Solution

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'
```

Problem

Write a program to vectorize (i.e., turn into a feature matrix) a directory of text documents with stoplisting and stemming.

source in vectorizer_with_stopwords_and_stemming.py

Solution

Py

```
import nltk.stem
english_stemmer = nltk.stem.SnowballStemmer('english')
class StemmedCountVectorizer(CountVectorizer):
    def build_analyzer(self):
        analyzer = super(StemmedCountVectorizer, self).build_analyzer()
        return lambda doc: (english_stemmer.stem(w) for w in analyzer(doc))

stemmed_vectorizer = StemmedCountVectorizer(min_df=1, stop_words='english')
feat_mat= stemmed_vectorizer.fit_transform(posts)
num_posts, num_feats = feat_mat.shape
print('features after stoplisting and stemming')
print('#samples_3: %d, #features_2: %d' % (num_posts, num_feats))
print(stemmed_vectorizer.get_feature_names())
```

Output

```
features after stemming
#samples_3: 5, #features_2: 17
[u'actual', u'capabl', u'contain', u'data', u'databas', u'imag', u'interest', u'learn',
u'machin', u'perman', u'post', u'provid', u'save', u'storag', u'store', u'stuff', u'toy']
```

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**

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
```

source in tfidf.py

Output

```
>>> term_freq('a', A)
n = 1.000000, tf = 1.000000
1.0
>>> term_freq('a', ABB)
n = 3.000000, tf = 1.000000
0.333333333333333
>>> term_freq('b', ABC)
n = 3.000000, tf = 1.000000
0.333333333333333
```

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)
0.4054651081081643
>>> inverse_doc_freq('c', DOCSET)
1.0986122886681098
```

TFIDF

Py

```
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, D)    =    0.000000
tfidf('b', abb, D)  =    0.270310
tfidf('a', abc, D)  =    0.000000
tfidf('c', abc, D)  =    0.366204
```

References

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