NLP

Natural Language Processing

KNN Review

- What is KNN?
- How does it work?
- Distance Metrics?

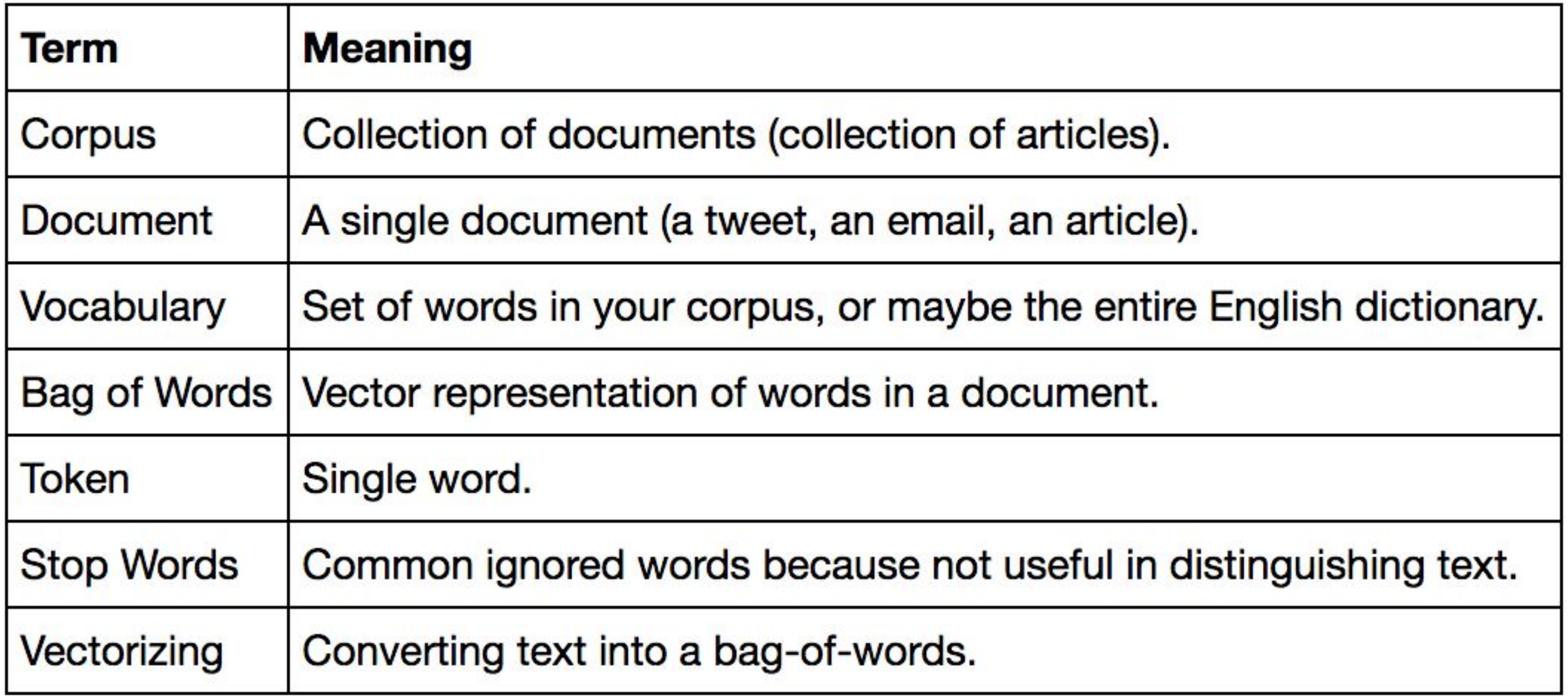
Why NLP?

- To group articles by topic
- To search for related articles
- Email spam detector
- Build Naive Bayes Classifier

Python Prerequisites

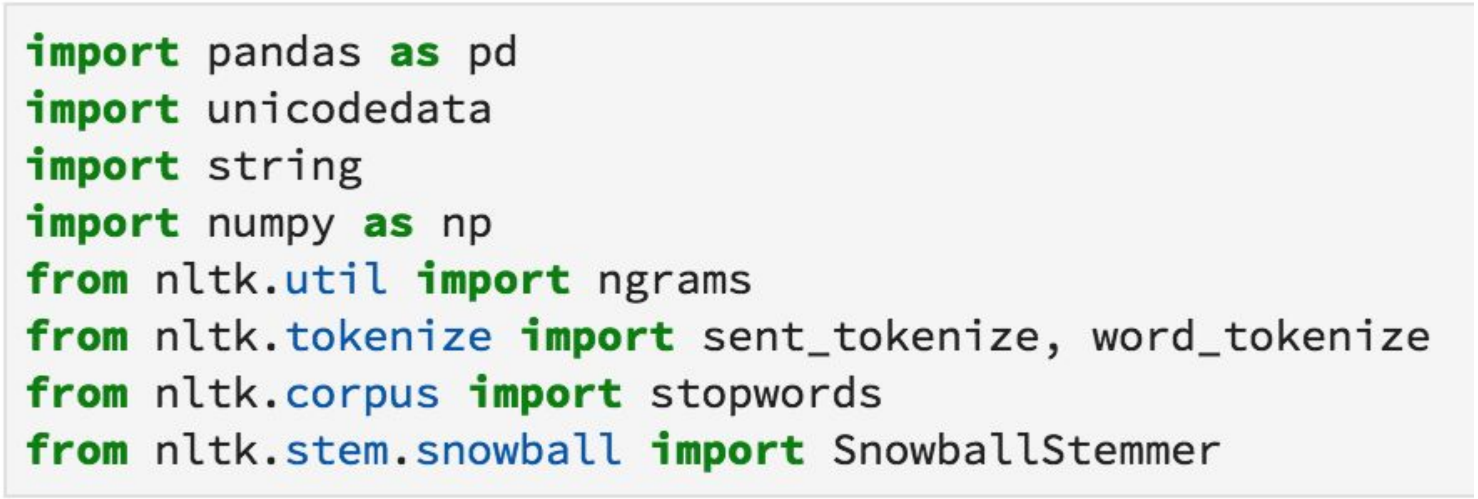
```
$ conda install nltk

$ ipython/jupyter notebook

> nltk.download('all')
```

Terminology



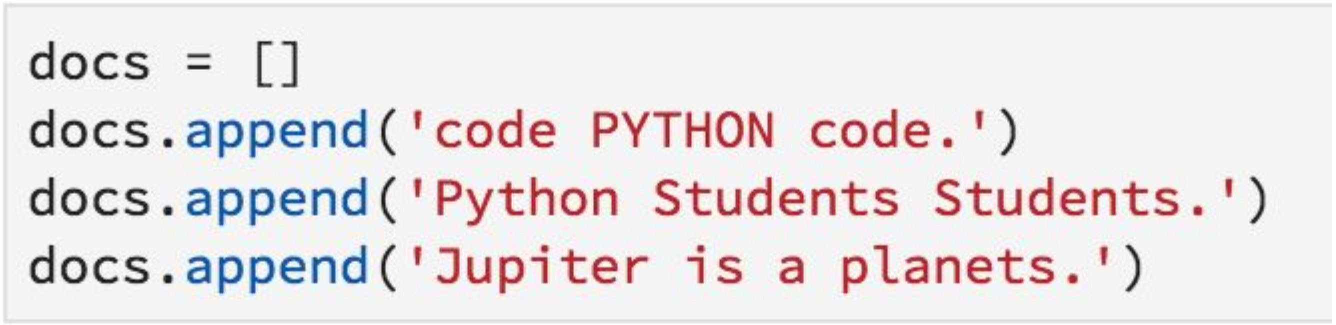
Term	Meaning
Corpus	Collection of documents (collection of articles).
Document	A single document (a tweet, an email, an article).
Vocabulary	Set of words in your corpus, or maybe the entire English dictionary.
Bag of Words	Vector representation of words in a document.
Token	Single word.
Stop Words	Common ignored words because not useful in distinguishing text.
Vectorizing	Converting text into a bag-of-words.

Step 0 : Imports



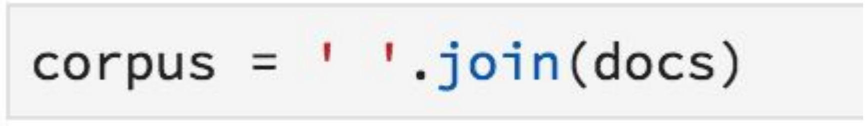
```
import pandas as pd
import unicodedata
import string
import numpy as np
from nltk.util import ngrams
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
```

Step 1 : Get documents



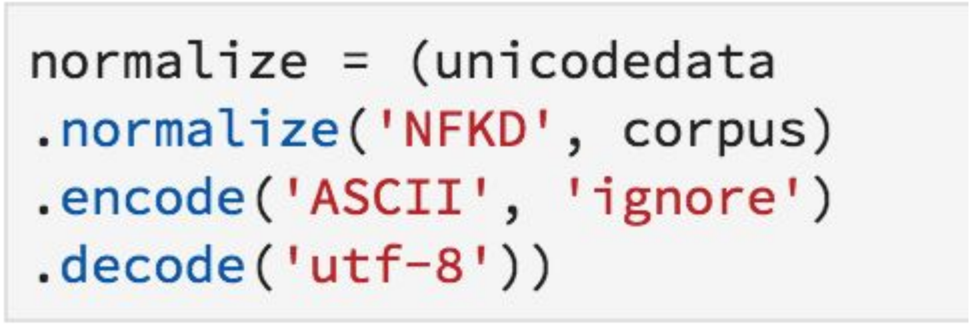
```
docs = []
docs.append('code PYTHON code.')
docs.append('Python Students Students.')
docs.append('Jupiter is a planets.')
```

Step 2 : Join all the documents into the corpus



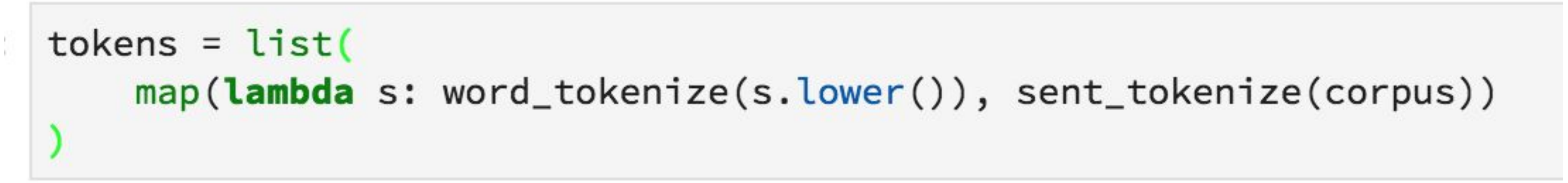
```
corpus = ' '.join(docs)
```

Step 3 : Unicode Normalization



```
normalize = (unicodedata
.normalize('NFKD', corpus)
.encode('ASCII', 'ignore')
.decode('utf-8'))
```

Step 4 : Tokenize

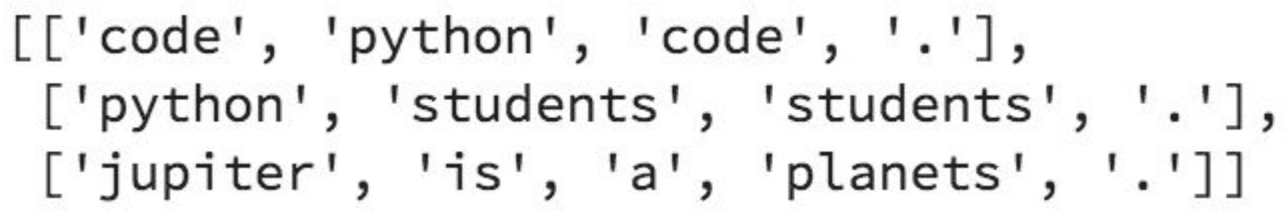


```python
tokens = list(
    map(lambda s: word_tokenize(s.lower()), sent_tokenize(corpus))
)
```

A token is a single word.

1. Convert the corpus to sentences
2. Convert the sentences to lowercase
3. Convert the lowercase sentences to tokens

Tokens

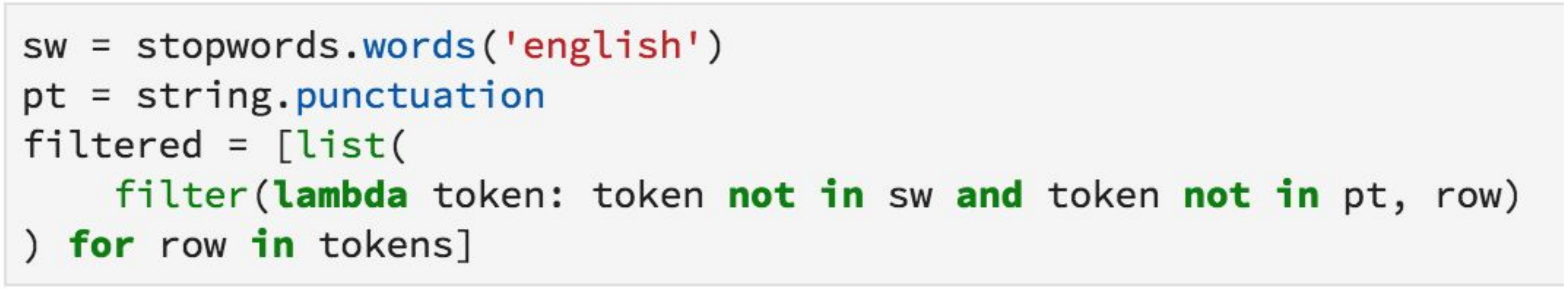


```
[['code', 'python', 'code', '.'],
 ['python', 'students', 'students', '.'],
 ['jupiter', 'is', 'a', 'planets', '.']]
```

Stopwords

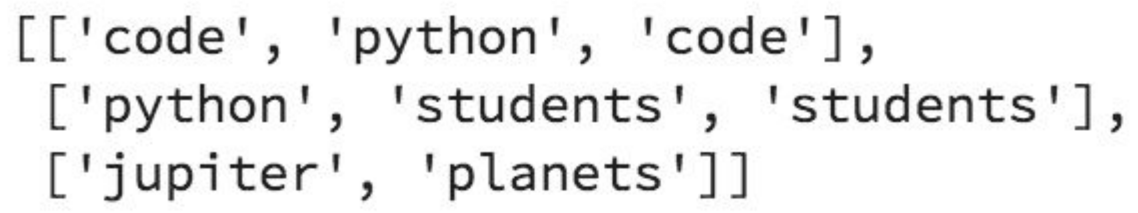
Sometimes, some extremely common words which would appear to be of little value in helping select documents matching a user need are excluded from the vocabulary entirely. These words are called stop words

Step 5 : Remove Stopwords & Punctuation



```
sw = stopwords.words('english')
pt = string.punctuation
filtered = [list(
    filter(lambda token: token not in sw and token not in pt, row)
) for row in tokens]
```

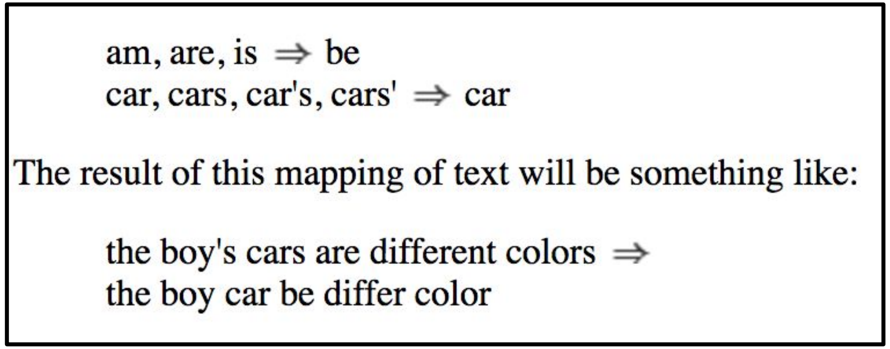
Stopwords and Punctuation removed



```
[['code', 'python', 'code'],
 ['python', 'students', 'students'],
 ['jupiter', 'planets']]
```

Stemming & Lemmatization

Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma.

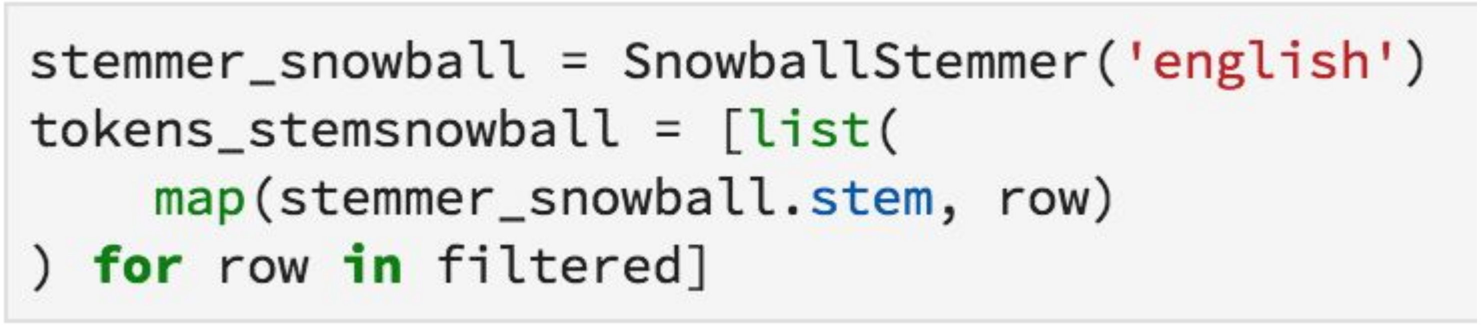


am, are, is $\Rightarrow$ be
car, cars, car's, cars' $\Rightarrow$ car

The result of this mapping of text will be something like:

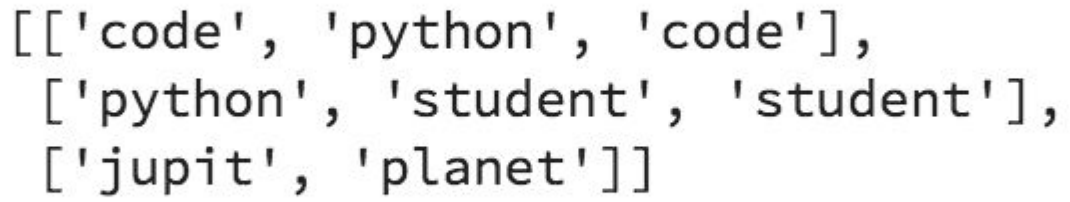
the boy's cars are different colors $\Rightarrow$
the boy car be differ color

Step 6 : Stemming



```
stemmer_snowball = SnowballStemmer('english')
tokens_stemsnowball = [list(
    map(stemmer_snowball.stem, row)
) for row in filtered]
```

After Stemming



```
[['code', 'python', 'code'],
 ['python', 'student', 'student'],
 ['jupit', 'planet']]
```

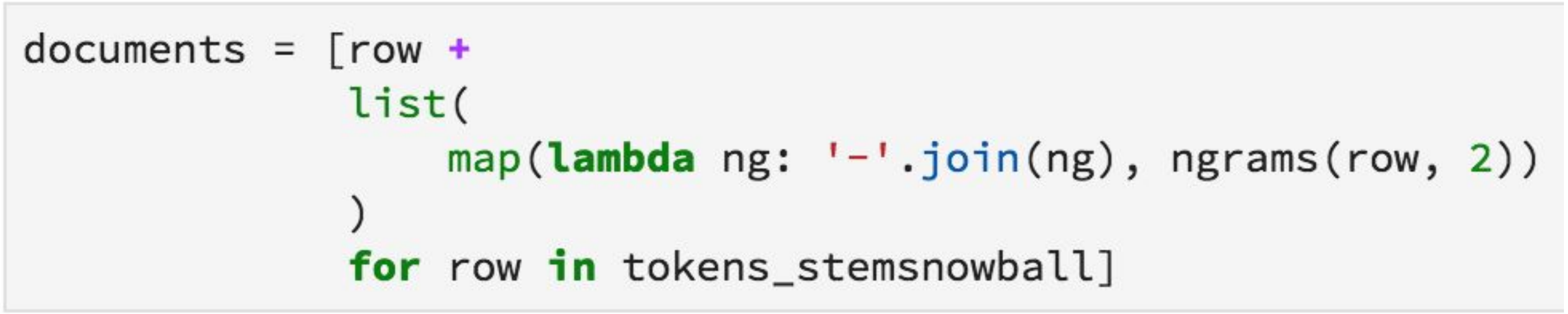
N-grams

N-grams of texts are extensively used in text mining and natural language processing tasks. They are basically a set of co-occurring words within a given window and when computing the n-grams you typically move one word forward.

Example, "The cow jumps over the moon", as a bi-gram

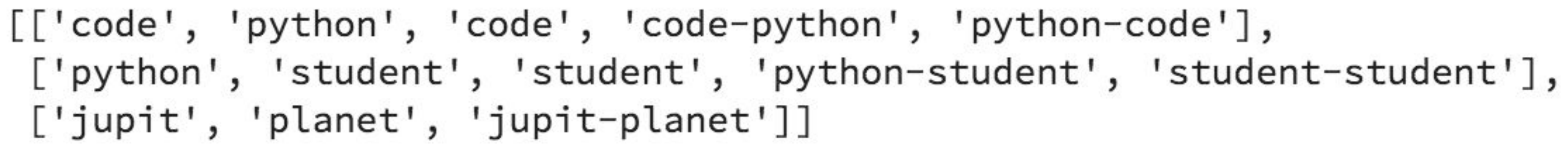
- the cow
- cow jumps
- jumps over
- over the
- the moon

Step 7 : N-grams



```
documents = [row +
             list(
                 map(lambda ng: '-'.join(ng), ngrams(row, 2))
             )
             for row in tokens_stemsnowball]
```

N-grams added



```
[['code', 'python', 'code', 'code-python', 'python-code'],
 ['python', 'student', 'student', 'python-student', 'student-student'],
 ['jupit', 'planet', 'jupit-planet']]
```

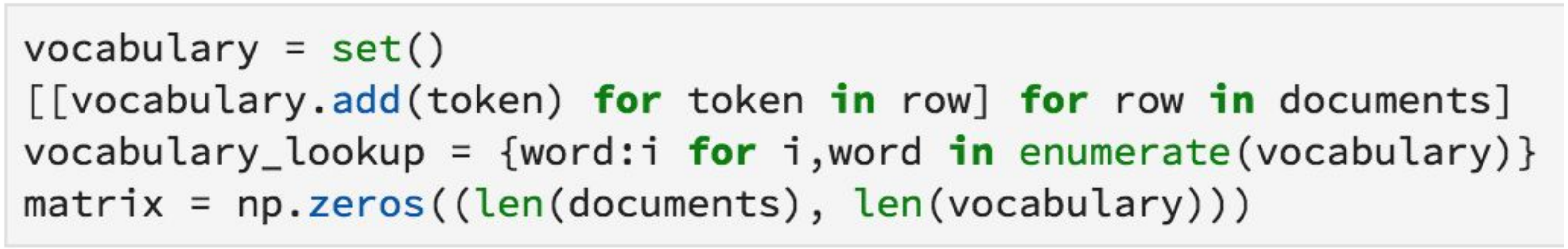
Bag of Words

A document represented as a vector of word counts is called “bag of words”

Vector for our corpus: (galvanize, learn, other, student, teach)

document	galvanize	learn	other	student	teach
Doc 1	0	1	1	2	0
Doc 2	1	0	0	0	1
Doc 3	1	1	0	1	0

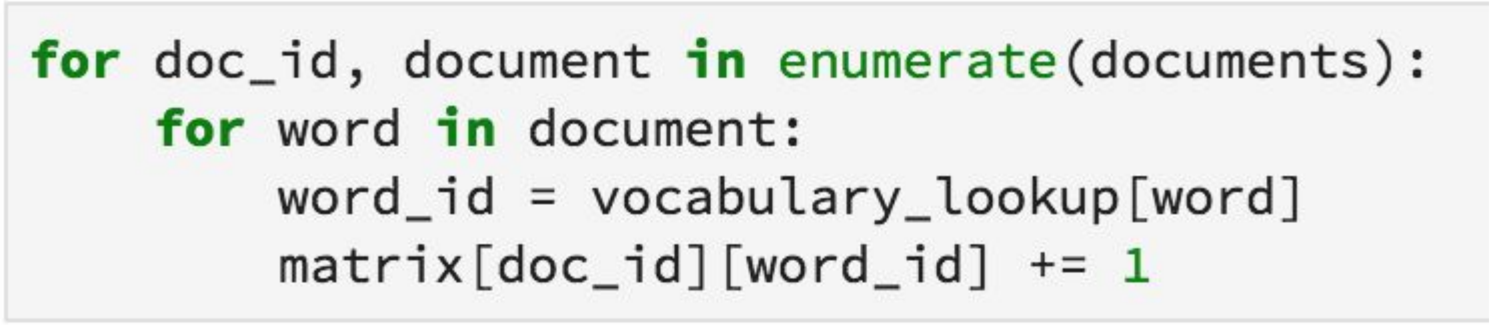
Step 8 : Creating Bag of Words



```
vocabulary = set()
[[vocabulary.add(token) for token in row] for row in documents]
vocabulary_lookup = {word:i for i,word in enumerate(vocabulary)}
matrix = np.zeros((len(documents), len(vocabulary)))
```

1. Create a set of all the words in the corpus
2. Create a lookup from a word to its index or position
3. Initialize an empty numpy matrix which will be used to vectorize the documents

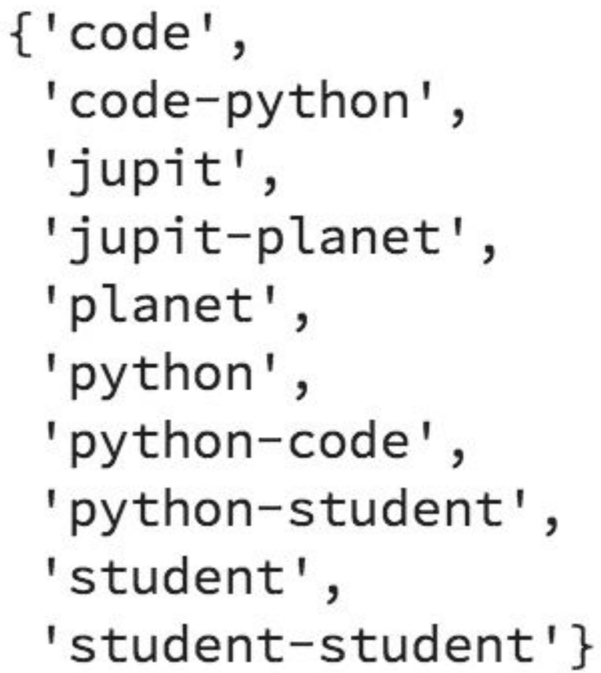
Step 9 : Count words in each document



```
for doc_id, document in enumerate(documents):
    for word in document:
        word_id = vocabulary_lookup[word]
        matrix[doc_id][word_id] += 1
```

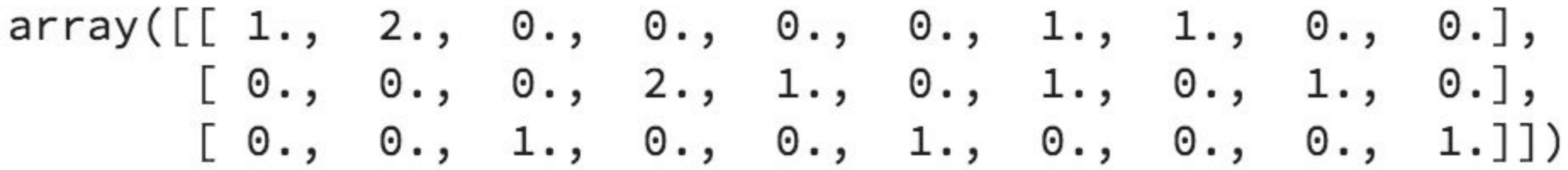
(See "Bag of Words" slide to see an example output)

Vocabulary



```
{'code',
 'code-python',
 'jupit',
 'jupit-planet',
 'planet',
 'python',
 'python-code',
 'python-student',
 'student',
 'student-student'}
```

Word Counts



```
array([[ 1.,  2.,  0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  2.,  1.,  0.,  1.,  0.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  1.]])
```

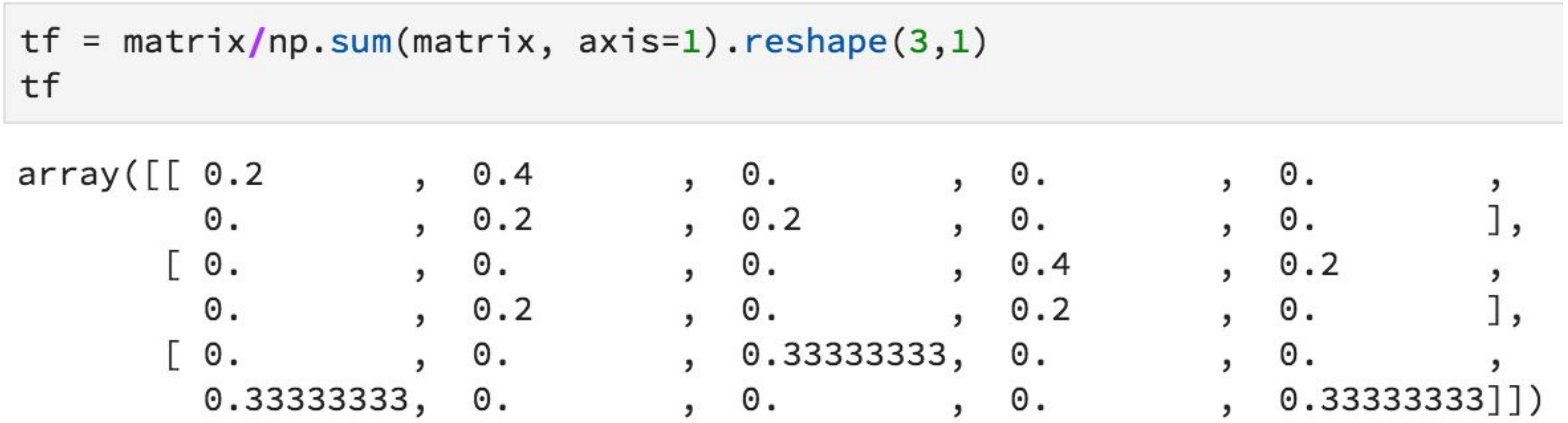
Term Frequency

Normalize counts within a document to frequency

$$tf(t,d) = \frac{total\ count\ of term\ t\ in\ document\ d}{total\ count\ of\ all\ terms\ in\ document\ d}$$

document	galvanize	learn	other	student	teach
Doc 1	0	¼ = 0.25	¼ = 0.25	2/4 = 0.5	0
Doc 2	½ = 0.5	0	0	0	½ = 0.5
Doc 3	⅓ = 0.33	⅓ = 0.33	0	⅓ = 0.33	0

Step 10 : Calculate Term Frequency



```
tf = matrix/np.sum(matrix, axis=1).reshape(3,1)
tf
```

```
array([[ 0.2       ,  0.4       ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.2       ,  0.2       ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.4       ,  0.2       ,
         0.        ,  0.2       ,  0.        ,  0.2       ,  0.        ],
       [ 0.        ,  0.        ,  0.33333333,  0.        ,  0.        ,
         0.33333333,  0.        ,  0.        ,  0.        ,  0.33333333]])
```

Inverse Document Frequency

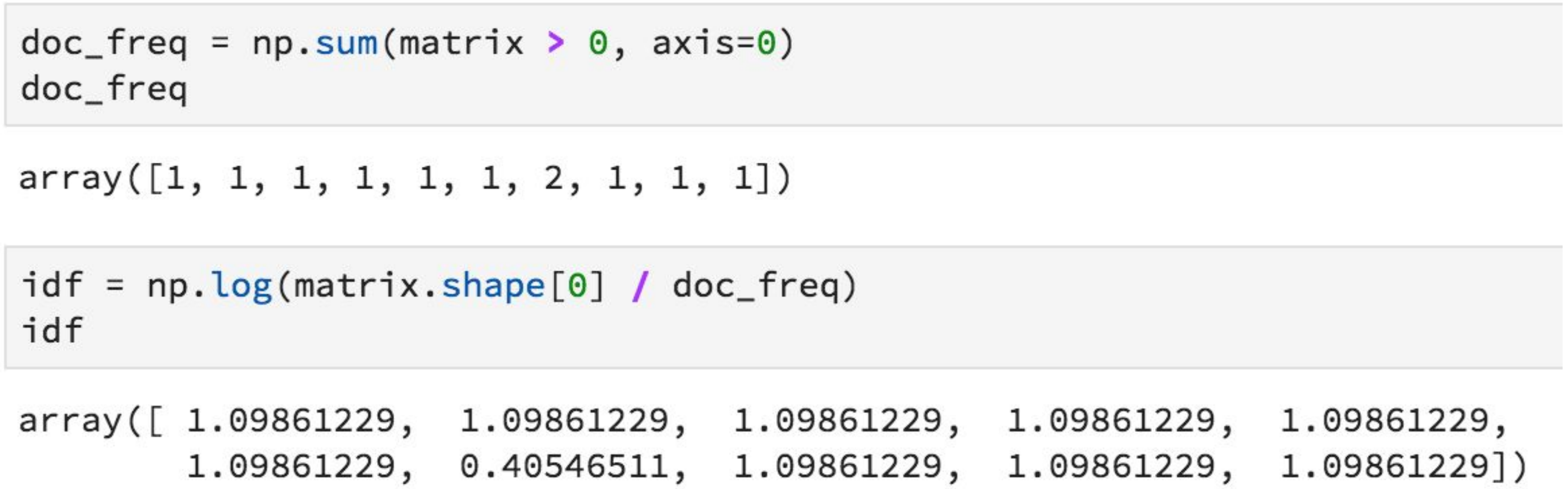
- IDF gives high weights to rarely seen words
- IDF gives low weights to common words

Inverse Document Frequency

$$idf(t, D) = log\frac{total\ number\ of\ document\ incorpus\ D}{count\ of\ document\ containing\ term\ t}$$

document	galvanize	learn	other	student	teach
Doc 1		X	X	X	
Doc 2	X				X
Doc 3	X	X		X	
idf(t,D)	log(3/2)	log(3/2)	log(3/1)	log(3/2)	log(3/1)

Step 11 : Calculate Inverse Document Frequency



```python
doc_freq = np.sum(matrix > 0, axis=0)
doc_freq
```

```
array([1, 1, 1, 1, 1, 1, 2, 1, 1, 1])
```

```python
idf = np.log(matrix.shape[0] / doc_freq)
idf
```

```
array([ 1.09861229,  1.09861229,  1.09861229,  1.09861229,  1.09861229,
        1.09861229,  0.40546511,  1.09861229,  1.09861229,  1.09861229])
```

TF-IDF

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$$

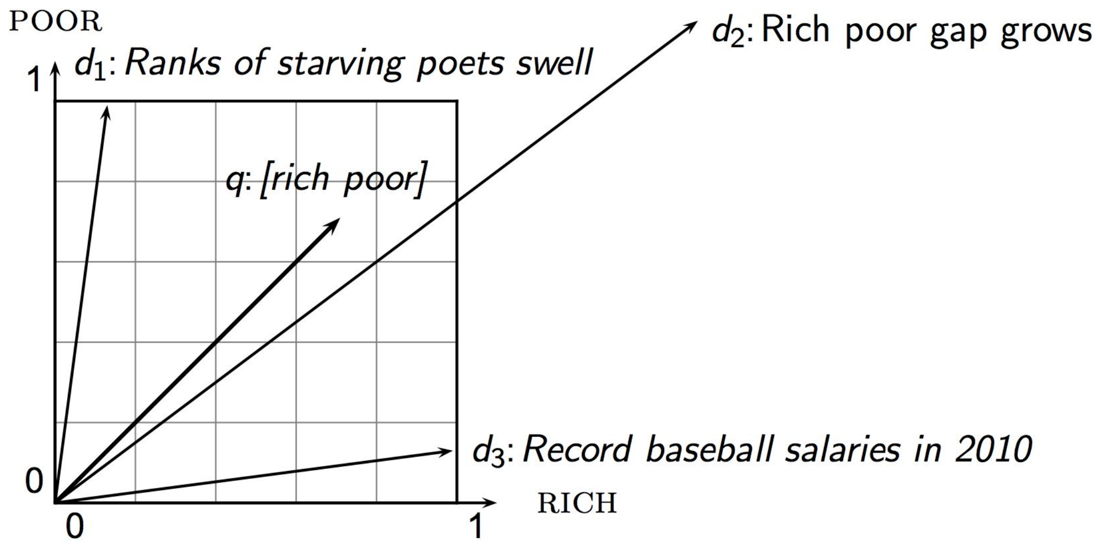
document	galvanize	learn	other	student	teach
Doc 1	0	0.25xlog(3/2) = 0.101	0.25xlog(3/1) = 0.275	0.5xlog(3/2) = 0.203	0
Doc 2	0.5xlog(3/2) = 0.203	0	0	0	0.5xlog(3/1) = .549
Doc 3	0.33xlog(3/2) = 0.135	0.33xlog(3/2) = 0.135	0	0.33xlog(3/2) = 0.135	0

Step 12 : Calculate TF-IDF

```
tfidf = tf * idf
tfidf
```

```
array([[ 0.21972246,  0.43944492,  0.        ,  0.        ,  0.        ,
         0.        ,  0.08109302,  0.21972246,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.43944492,  0.21972246,
         0.        ,  0.08109302,  0.        ,  0.21972246,  0.        ],
       [ 0.        ,  0.        ,  0.3662041 ,  0.        ,  0.        ,
         0.3662041 ,  0.        ,  0.        ,  0.        ,  0.3662041 ]])
```

Using distance to compare documents


POOR
d_1: *Ranks of starving poets swell*
d_2: Rich poor gap grows
q: *[rich poor]*
d_3: *Record baseball salaries in 2010*
RICH
1
0
0
1

The Euclidean distance of $\vec{q}$ and $\vec{d_2}$ is large although the distribution of terms in the query q and the distribution of terms in the document d_2 are very similar. □

Comparing Documents

Cosine similarity: $$similarity = cos\theta = \frac{A \cdot B}{\|A\| \, \|B\|}$$

- Doc 1 vs. Doc 2:
  - ["student", "learn", "other", "student"] vs ["teach", "galvanize"]

$$(0, 0.101, 0.275, 0.203, 0) \; vs. \; (0.203, 0, 0, 0, 0.275)$$

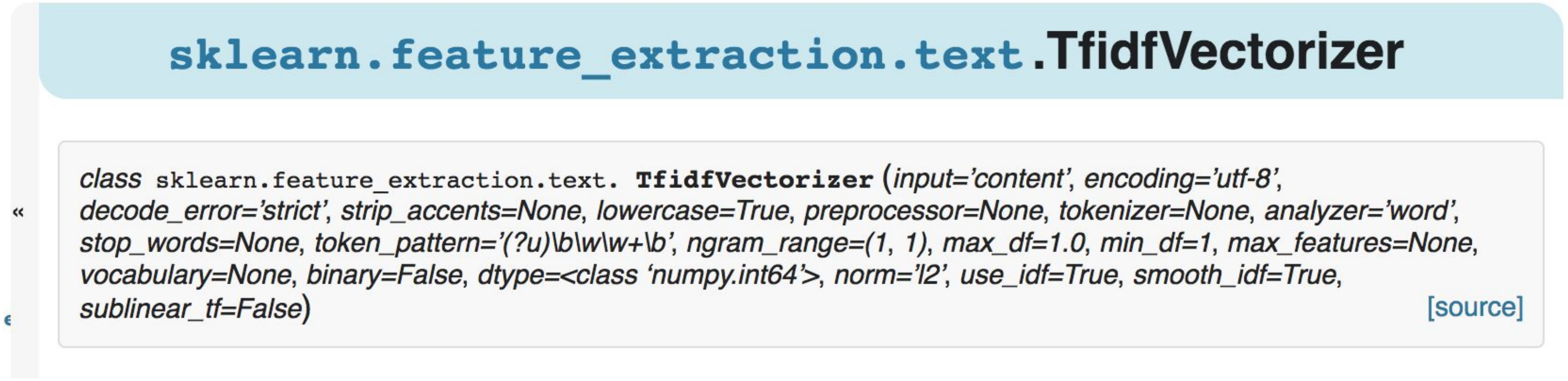
$$similarity = \frac{0}{0.36 \times 0.34} = 0$$

- Doc 1 vs. Doc 3:
  - ["student", "learn", "other", "student"] vs ["student", "learn", "galvanize"]

$$(0, 0.101, 0.275, 0.203, 0) \; vs. \; (0.135, 0.135, 0, 0.135, 0)$$

$$similarity = \frac{0.041}{0.36 \times 0.23} = 0.34$$

NLP, Automatic!



sklearn.feature_extraction.text.TfidfVectorizer

class `sklearn.feature_extraction.text.` **`TfidfVectorizer`** (*input='content', encoding='utf-8', decode_error='strict', strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, analyzer='word', stop_words=None, token_pattern='(?u)\b\w\w+\b', ngram_range=(1, 1), max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False, dtype=<class 'numpy.int64'>, norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False*) [source]