Vector Space Model and Term Weighting

CHIEN CHIN CHEN

Introduction (1/2)

Now, you are able to segment a document into a set of tokens

- Perhaps with stopword removal, stemming, lemmatization, and other normalization techniques.

It is time to turn these tokens (in a document) into numbers.

- And represent a document in a continuous space upon the numbers.

Here, we present **Vector Space Model** (VSM) which has been the mainstay **text representation** for text mining algorithms.

- Also **term weighting** schemes are introduced to help identify words important to a particular document.

Introduction (2/2)

Specifically, we look at the following ways to represent a document.

- **Frequency-based vectors** – vector based word counts.
- **Multi-hot vectors** – vectors based on word absence/presence.
- **TF-IDF vectors** - vectors based on term frequency and inverse document Frequency.

Last, vector operations are showed to measure **the similarity between documents**.

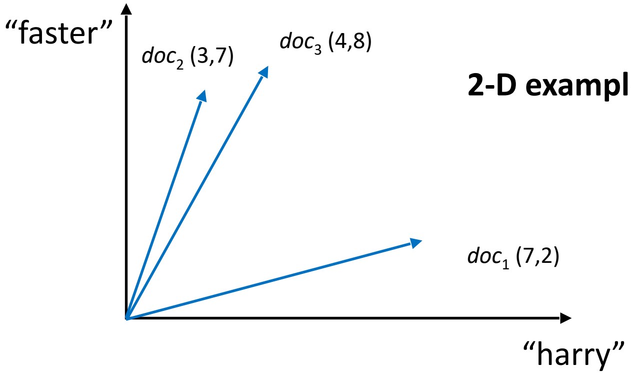
Vector Space Model (1/4)

The **dimensionality** of a vector space is the count of the number of distinct words that appear in the entire corpus.

- The number is just the **vocabulary size** $|V|$.
- Each dimension stands for a unique word.
- Always a **high-dimensional vector space**!!
  - For instance, the IMDB dataset consisting of 50K movie reviews could have more than 20K unique words!!

Each document is represented as a **high-dimensional vector** and the **weights** of words determine its location in a vector space.

2-D example:



Before talking about the weights and vector representation, let's review something we learned in the last chapter.

Vector Space Model (2/4)

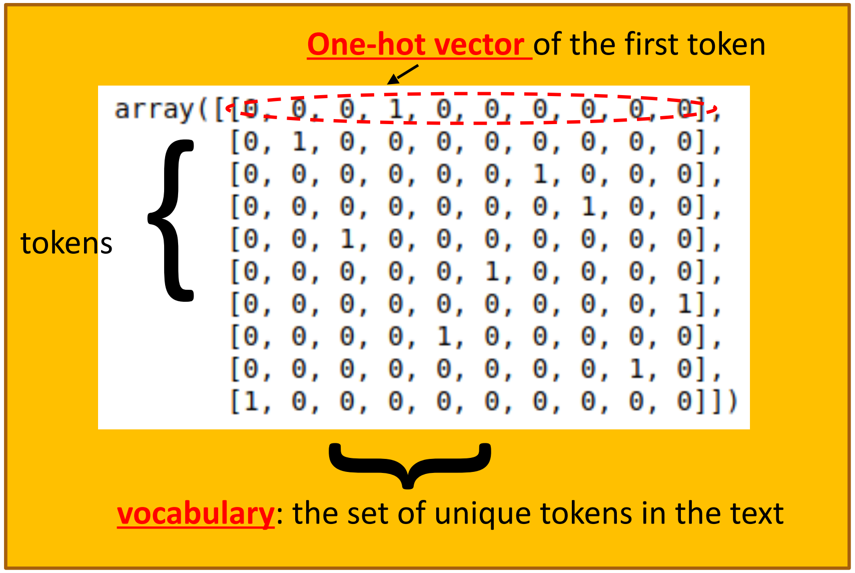
Remember the '**one-hot vectors**' we created in the text-preprocessing chapter?

The one-hot vectors retain all the detail of the original document.

The document vector we are going to show compresses all detail into a single vector!!

- Word order information is lost!!

Why should we do this?



One-hot vector of the first token

tokens

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

vocabulary: the set of unique tokens in the text

Vector Space Model (3/4)

The storage require for one-hot vectors is so high!!

Say that you are going to process 3,000 documents; each of them contains 500 sentences and 15 words per sentence.

- Assuming the vocabulary size is 600,000 words, which is the word number in the Oxford English Dictionary.
- Then, each document needs 500*15=7,500 one-hot vectors.
- Or each document requires a matrix of 7,500 x 600,000 to store the one-hot information.
- That costs 4.2 GB per document (if each matrix element needs one bye) or 536MB per document (one bit per matrix element)

Do not forget you are goinig to process 3,000 documents …

Vector Space Model (4/4)

You certain would like to compress your document down to a single vector rather than a big table.

- That is, a high-dimensional vector in a vector space.
- Easier for text mining algorithms to work with.

Note that summarizing a document as a vector loses the order information of tokens.

- The document vector is just like a set (bag) of words; so sometimes we call vector space model **the bag-of-word model**.

Frequency-based Vectors (1/5)

A simple vector representation of documents is to count the number of times a word in a given document.

- Which is also called **term frequency** (**TF**).

A document *d* is represented as a |*V*|-dimensional vector $\underline{x}_d$.

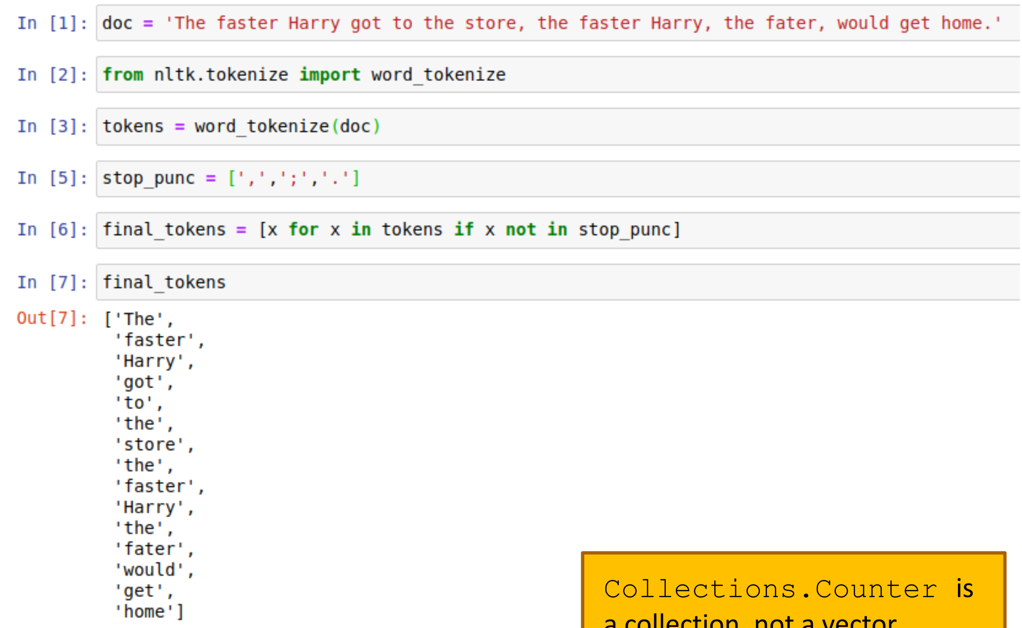
- A vector element $x_{t,d}$ is $\boldsymbol{tf_{t,d}}$ which denotes the term frequency of *t* in *d*.

	‘the’	‘a’	‘cake’	‘tea’	…	‘dog’	‘pay’	‘bad’
doc_1	4	7	0	0	…	5	1	3
doc_2	8	11	7	16	…	1	4	0

Why term frequency?

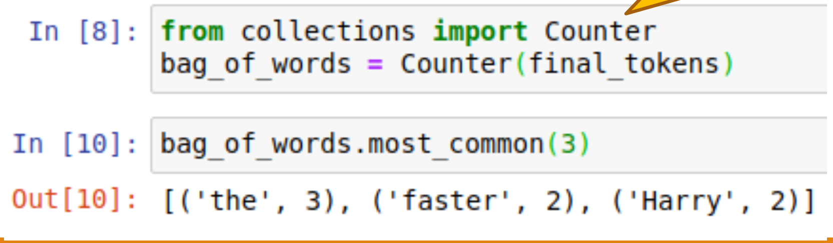
- Intuitively, the more times a word occurs, the more meaning it contributes to that document.
- A document about airplanes should frequently refer “wings” and “flight”

Frequency-based Vectors (2/5)



```
In [1]: doc = 'The faster Harry got to the store, the faster Harry, the fater, would get home.'

In [2]: from nltk.tokenize import word_tokenize

In [3]: tokens = word_tokenize(doc)

In [5]: stop_punc = [',',';','.']

In [6]: final_tokens = [x for x in tokens if x not in stop_punc]

In [7]: final_tokens

Out[7]: ['The',
         'faster',
         'Harry',
         'got',
         'to',
         'the',
         'store',
         'the',
         'faster',
         'Harry',
         'the',
         'fater',
         'would',
         'get',
         'home']
```

`Collections.Counter` is a collection, not a vector…



```
In [8]: from collections import Counter
        bag_of_words = Counter(final_tokens)

In [10]: bag_of_words.most_common(3)

Out[10]: [('the', 3), ('faster', 2), ('Harry', 2)]
```

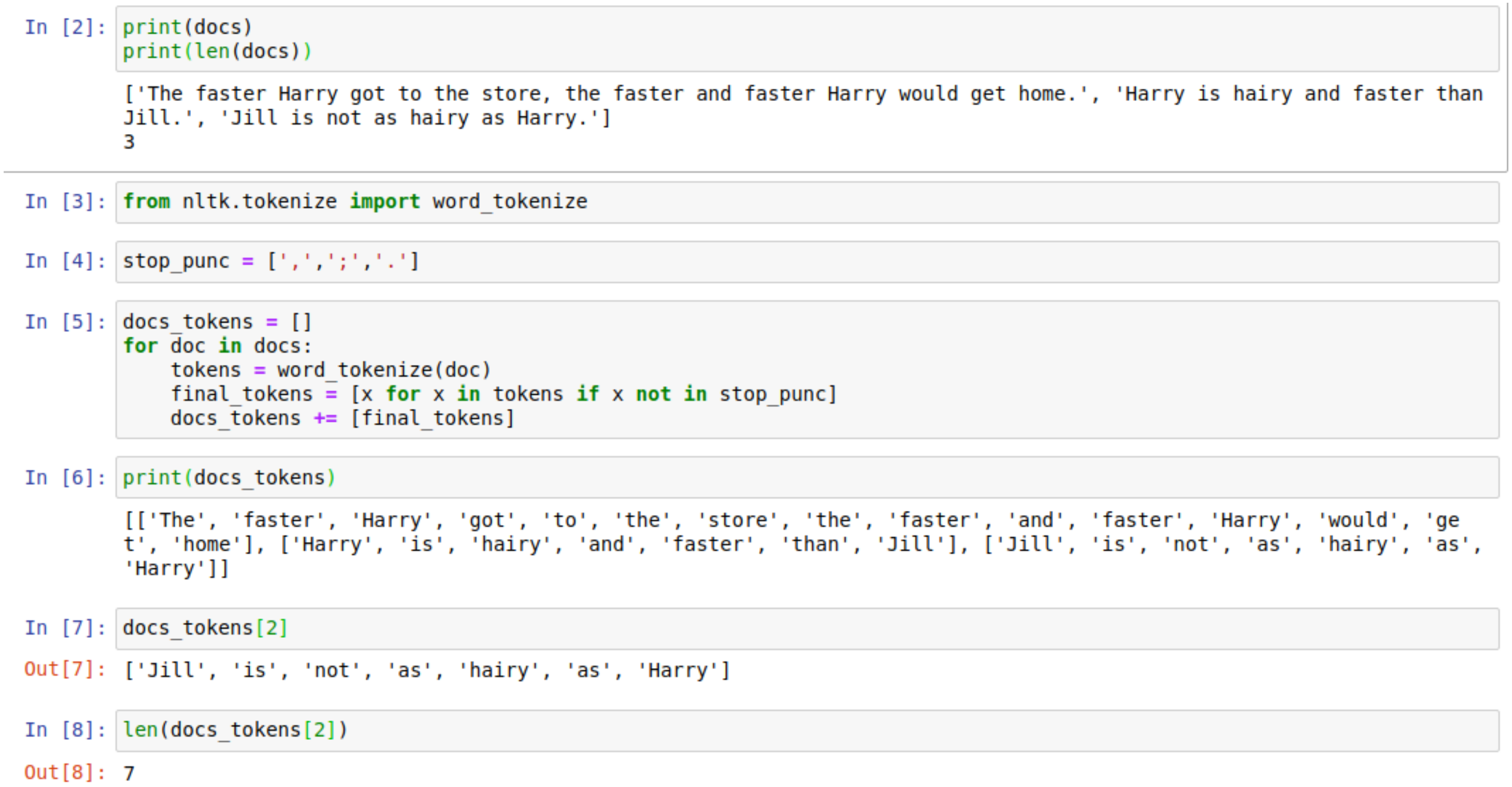
In this document, the most frequent (important) word is '*the*' ☹

- Your can remove it by using a stopword list.
- Or, later, **inverse document frequency** helps reduce its impact (weight).

Generally, term frequencies need to be **normalized**.

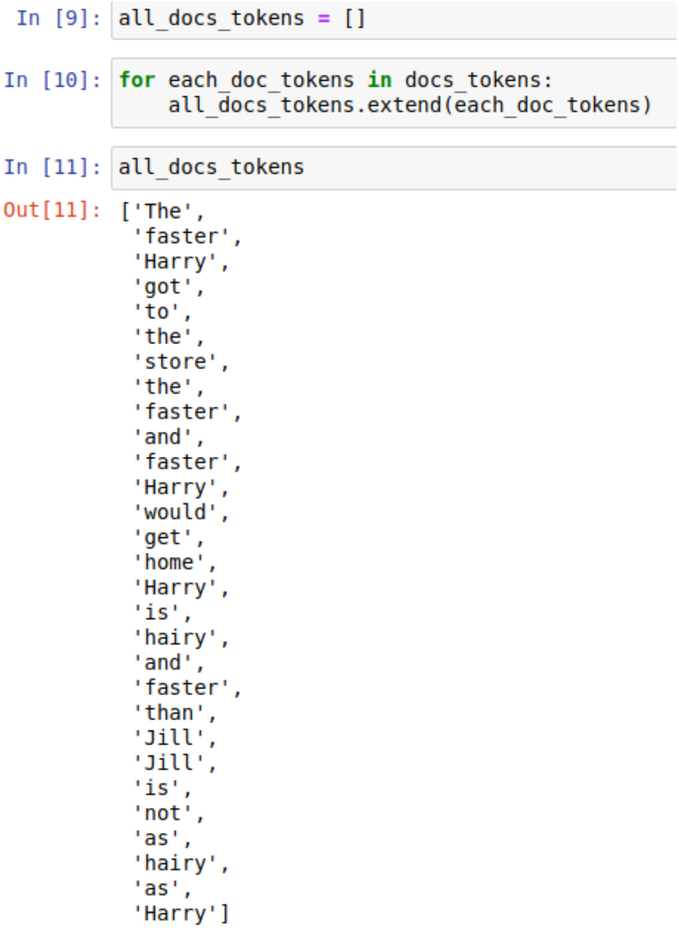
- For two documents *A* and *B*, say the word 'dog' occur 3 times in *A* and 100 times in *B*.
- You would say 'dog' is more important to *B*.
- But what if *A* is a 30-word email and *B* is a 580,000-word novel?
- By normalizing, you get the relative importance to the document of that term.
  - $TF_{A,dog}$ = 3/30 = 0.1 $TF_{B,dog}$ = 100/580000 = 0.001

Frequency-based Vectors (3/5)



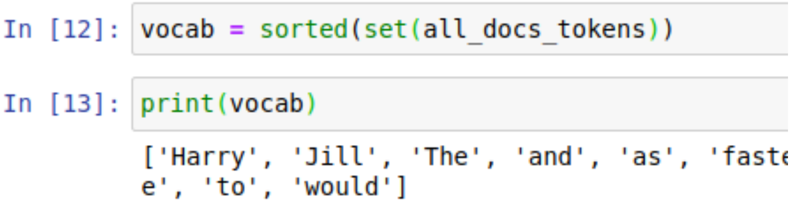
```
In [2]: print(docs)
        print(len(docs))

        ['The faster Harry got to the store, the faster and faster Harry would get home.', 'Harry is hairy and faster than
        Jill.', 'Jill is not as hairy as Harry.']
        3

In [3]: from nltk.tokenize import word_tokenize

In [4]: stop_punc = [',',';','.']

In [5]: docs_tokens = []
        for doc in docs:
            tokens = word_tokenize(doc)
            final_tokens = [x for x in tokens if x not in stop_punc]
            docs_tokens += [final_tokens]

In [6]: print(docs_tokens)

        [['The', 'faster', 'Harry', 'got', 'to', 'the', 'store', 'the', 'faster', 'and', 'faster', 'Harry', 'would', 'ge
        t', 'home'], ['Harry', 'is', 'hairy', 'and', 'faster', 'than', 'Jill'], ['Jill', 'is', 'not', 'as', 'hairy', 'as',
        'Harry']]

In [7]: docs_tokens[2]

Out[7]: ['Jill', 'is', 'not', 'as', 'hairy', 'as', 'Harry']

In [8]: len(docs_tokens[2])

Out[8]: 7
```

Frequency-based Vectors (4/5)



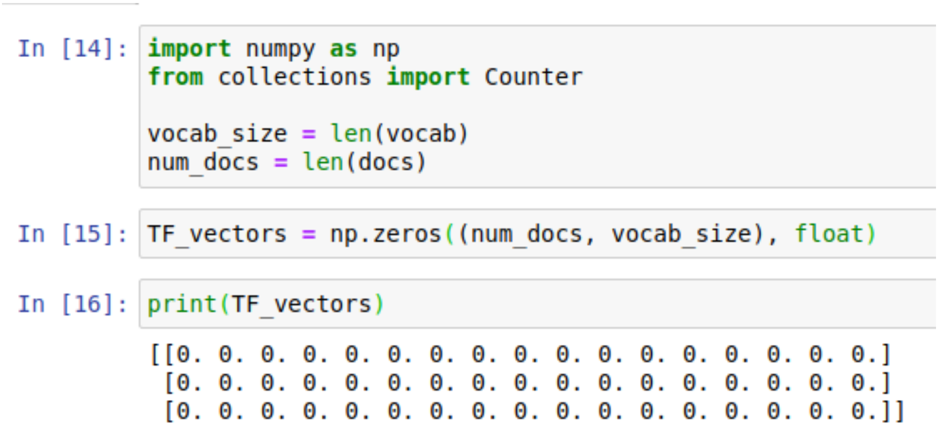
```
In [9]:  all_docs_tokens = []

In [10]: for each_doc_tokens in docs_tokens:
             all_docs_tokens.extend(each_doc_tokens)

In [11]: all_docs_tokens

Out[11]: ['The',
          'faster',
          'Harry',
          'got',
          'to',
          'the',
          'store',
          'the',
          'faster',
          'and',
          'faster',
          'Harry',
          'would',
          'get',
          'home',
          'Harry',
          'is',
          'hairy',
          'and',
          'faster',
          'than',
          'Jill',
          'Jill',
          'is',
          'not',
          'as',
          'hairy',
          'as',
          'Harry']
```

Build the vocabulary



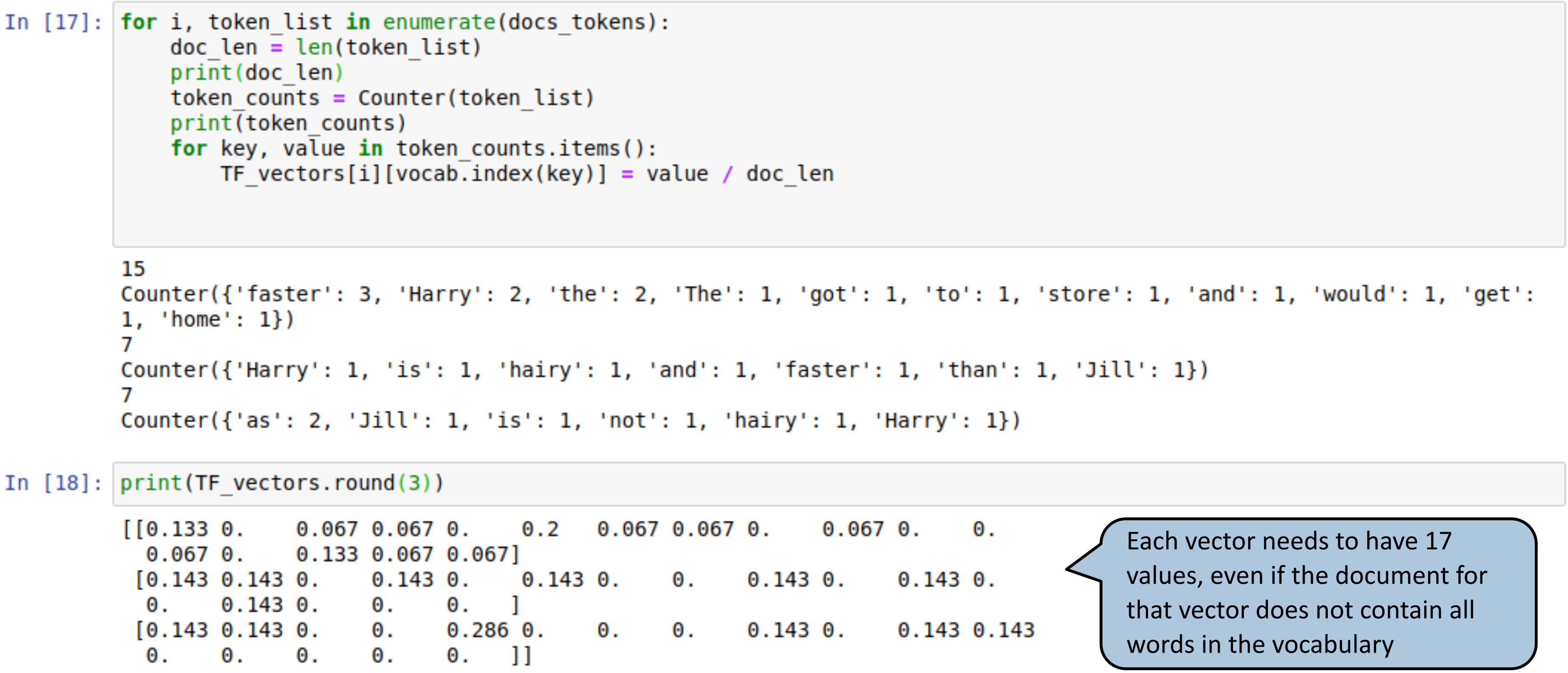
```
In [12]: vocab = sorted(set(all_docs_tokens))

In [13]: print(vocab)

         ['Harry', 'Jill', 'The', 'and', 'as', 'faste
         e', 'to', 'would']
```

Construct TF vectors



```
In [14]: import numpy as np
         from collections import Counter

         vocab_size = len(vocab)
         num_docs = len(docs)

In [15]: TF_vectors = np.zeros((num_docs, vocab_size), float)

In [16]: print(TF_vectors)

         [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
          [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
          [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Frequency-based Vectors (5/5)



```
In [17]: for i, token_list in enumerate(docs_tokens):
             doc_len = len(token_list)
             print(doc_len)
             token_counts = Counter(token_list)
             print(token_counts)
             for key, value in token_counts.items():
                 TF_vectors[i][vocab.index(key)] = value / doc_len

         15
         Counter({'faster': 3, 'Harry': 2, 'the': 2, 'The': 1, 'got': 1, 'to': 1, 'store': 1, 'and': 1, 'would': 1, 'get':
         1, 'home': 1})
         7
         Counter({'Harry': 1, 'is': 1, 'hairy': 1, 'and': 1, 'faster': 1, 'than': 1, 'Jill': 1})
         7
         Counter({'as': 2, 'Jill': 1, 'is': 1, 'not': 1, 'hairy': 1, 'Harry': 1})

In [18]: print(TF_vectors.round(3))

         [[0.133 0.    0.067 0.067 0.    0.2   0.067 0.067 0.    0.067 0.    0.
           0.067 0.    0.133 0.067 0.067]
          [0.143 0.143 0.    0.143 0.    0.143 0.    0.    0.143 0.    0.143 0.
           0.    0.143 0.    0.    0.   ]
          [0.143 0.143 0.    0.    0.286 0.    0.    0.    0.143 0.    0.143 0.143
           0.    0.    0.    0.    0.   ]]
```

Each vector needs to have 17 values, even if the document for that vector does not contain all words in the vocabulary

So complicated … later, we show you an easy way to construct frequency vectors

Multi-Hot Vectors

You can generalize term frequency vectors into **multi-hot vectors**.

- Also called **binary incidence vectors**.
- Is the input of many machine learning or deep learning models (e.g., Bernoulli-based Naïve Bayes Classification)

Again, each document *d* is represented as a |*V*|-dimensional vector $\underline{x}_d$.

- A vector element $x_{t,d}$ is 1 if term *t* is present in the document, otherwise, it is 0.

	'the'	'a'	'cake'	'tea'	…	'dog'	'pay'	'bad'
doc_1	1	1	0	0	…	1	1	1
doc_2	1	1	1	1	…	1	1	0

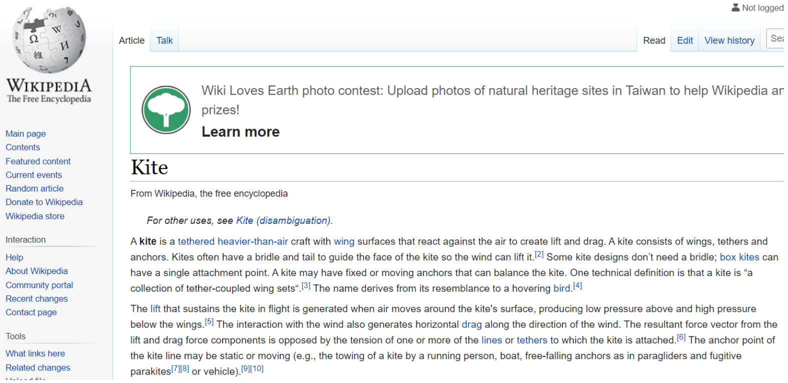
Self-practice: Modify the last term frequency vector program to convert a set of documents into multi-hot vectors!!

TF-IDF Vectors (1/3)

Frequency-based vectors obviously are more informative than multi-hot vectors.

However, pure term frequency may not be able to tell you how important a word is.

- Recall Zipf’s Law – sparse phenomenon the word usage.



Wiki Loves Earth photo contest: Upload photos of natural heritage sites in Taiwan to help Wikipedia ar prizes!

Learn more

Kite

From Wikipedia, the free encyclopedia

For other uses, see Kite (disambiguation).

A **kite** is a tethered heavier-than-air craft with wing surfaces that react against the air to create lift and drag. A kite consists of wings, tethers and anchors. Kites often have a bridle and tail to guide the face of the kite so the wind can lift it.[2] Some kite designs don’t need a bridle; box kites can have a single attachment point. A kite may have fixed or moving anchors that can balance the kite. One technical definition is that a kite is “a collection of tether-coupled wing sets“.[3] The name derives from its resemblance to a hovering bird.[4]

The lift that sustains the kite in flight is generated when air moves around the kite's surface, producing low pressure above and high pressure below the wings.[5] The interaction with the wind also generates horizontal drag along the direction of the wind. The resultant force vector from the lift and drag force components is opposed by the tension of one or more of the lines or tethers to which the kite is attached.[6] The anchor point of the kite line may be static or moving (e.g., the towing of a kite by a running person, boat, free-falling anchors as in paragliders and fugitive parakites[7][8] or vehicle).[9][10]

```
[('the', 26),
 ('a', 20),
 ('kite', 16),
 ('and', 10),
 ('of', 10),
 ('kites', 8),
 ('is', 7),
 ('in', 7),
 ('or', 6),
 ('wing', 5)]
```

The most frequent word is ‘*the*’ again…

Top frequent words are often stop words ☹

TF-IDF Vectors (2/3)

Inverse Document Frequency (IDF): measure how rare a word in a corpus. 全體

$$idf_t = \log\frac{N}{df_t}$$

the number of documents in a collection

the number of documents in a collection that contain term *t*

If a word occur many times in every document, it actually does not provide any information.

- E.g., ‘*term*’ in a set of text-mining-related documents.

Words that are not prevent across the entire corpus, but for the ones where they frequently occurred, are helpful to learn the documents.

- E.g., ‘*Dirichlet*’ often appears in documents about topic mining.

TF-IDF Vectors (3/3)

Finally, the TF-IDF weight of a term *t* in a document *d* is:

- $\mathit{tf\text{-}idf}_{t,d} = \mathit{tf}_{t,d} \times \mathit{idf}_{t}$

The weight of term *t* in document *d* is **high**, when *t* occurs many times in *d* and appears within a small number of documents.

What about **stop words** that occurs in every document?

- The TF-IDF weights of them in the document would be … (almost) **zero**.
- This helps identify the core (keywords) of a document.

Many **vectorizers** are available to
Help us convert text into vectors!!

" Talk is cheap.
Show me the code. "

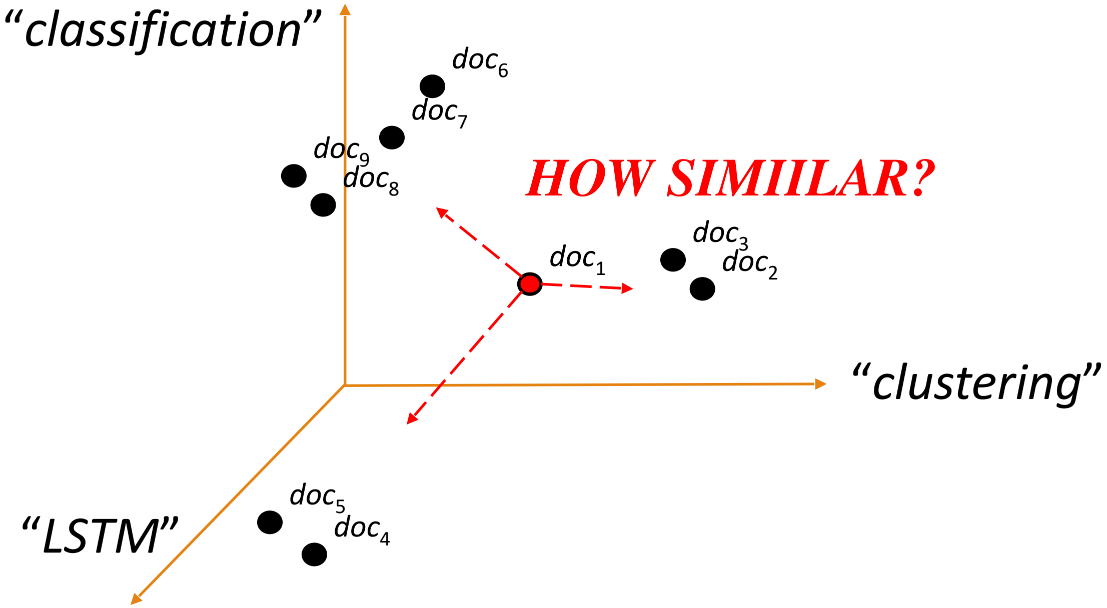
Linus Torvalds (2000)

But let first talk about
SIMILARITY

Comparing Document Vectors (1/3)

You have represented documents as vectors sharing a common vector space.

It is time to compare them!!



Comparing Document Vectors (2/3)

Two popular measures for calculating the **similarity/distance** between vectors:

Euclidean distance between document d_i and d_j:

- $\left| \underline{x}_i - \underline{x}_j \right| = \sqrt{\sum_{t=1}^{M} (x_{t,i} - x_{t,j})^2}$

  $M = |V|$, that is, vocabulary size.

- Note the distance is <u>sensitive to document length</u>!!
  - Normalize vectors if you want to lessen the effect of document length.
  - Using normalized TF, or normalize vectors to their unit vectors.

Comparing Document Vectors (3/3)

Cosine similarity between document *d*$_i$ and *d*$_j$:

$$cosine(\underline{x}_i, \underline{x}_j) = \frac{\underline{x}_i \cdot \underline{x}_j}{|\underline{x}_i||\underline{x}_j|}$$

Inner product of vectors

vector length

$$= \frac{\sum_{t=1}^{M}(x_{t,i} \; * \; x_{t,j})}{\sqrt{\sum_{t=1}^{M}(x_{t,i} \; * \; x_{t,i})^2}\sqrt{\sum_{t=1}^{M}(x_{t,j} \; * \; x_{t,j})^2}}$$

- Cosine automatically normalize document vectors to their unit vectors, then do dot product of the normalized vectors.
- The range of cosine similarity is [-1, 1]
  - But … if your are using frequency-based term weights, the range would always be [0, 1].
  - Close to 1 when two documents are using similar words in similar proportion.
  - 0 – also called orthogonal, if two documents share no words in common; but is possible that they have similar meaning.

Ease Your Life with Packages

Many packages are readily available to help us turn documents into vectors.

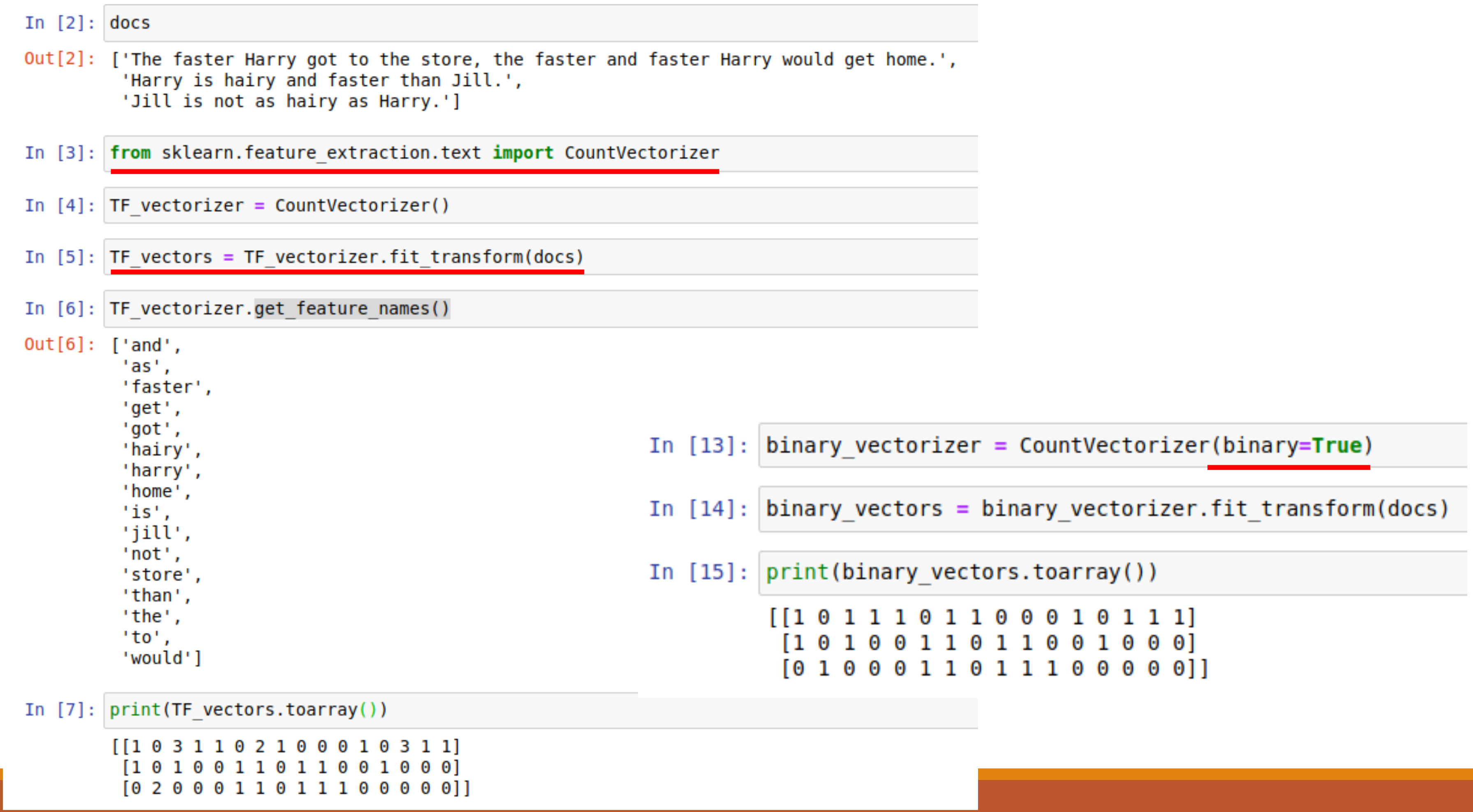
Also, functions of cosine similarity and Euclidean distance are provided to compare the resulting vectors !!

Do it with `sklearn` ☺

scikits
learn
machine learning in Python

CountVectorizer

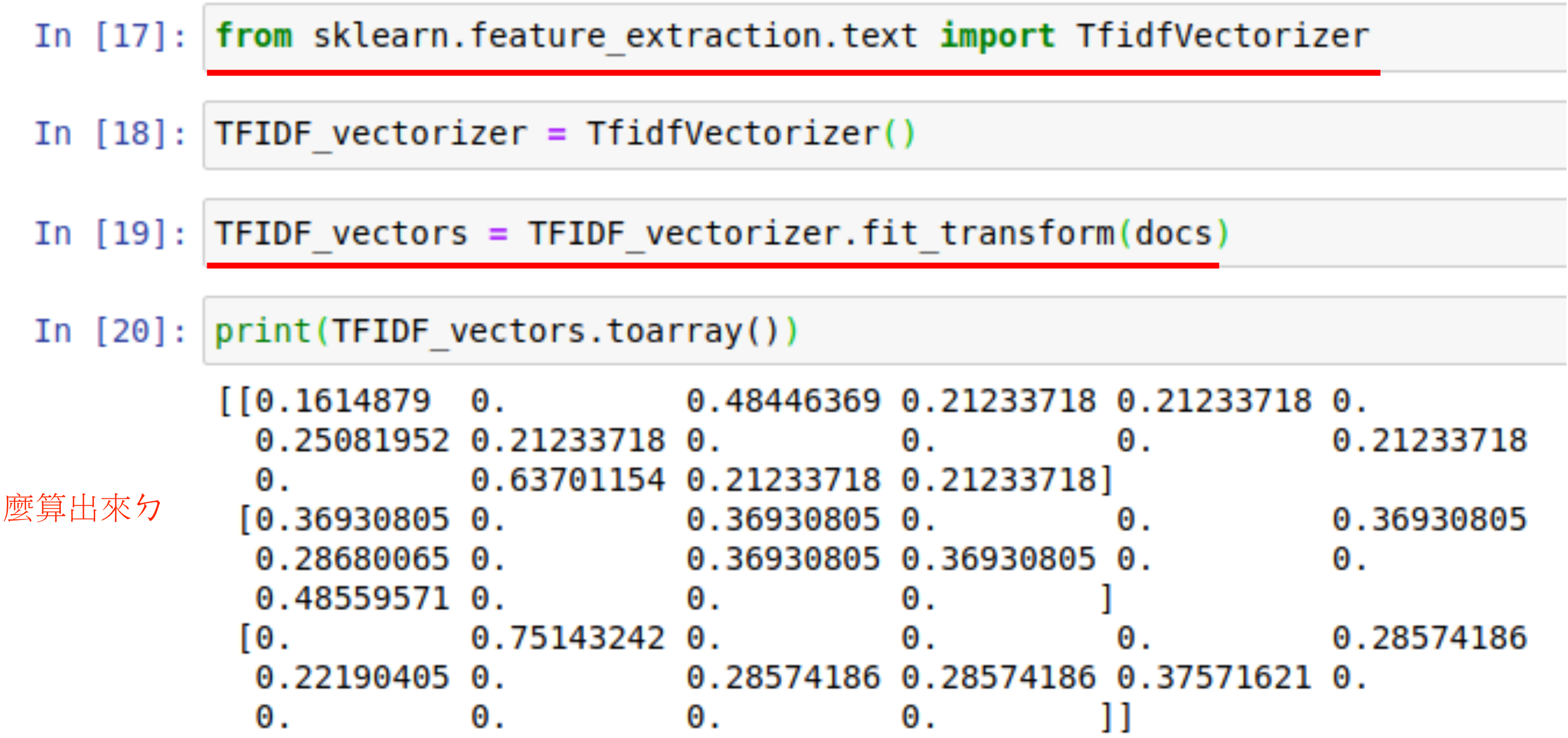
TF/Multi-Hot Vectors



```
In [2]: docs

Out[2]: ['The faster Harry got to the store, the faster and faster Harry would get home.',
         'Harry is hairy and faster than Jill.',
         'Jill is not as hairy as Harry.']

In [3]: from sklearn.feature_extraction.text import CountVectorizer

In [4]: TF_vectorizer = CountVectorizer()

In [5]: TF_vectors = TF_vectorizer.fit_transform(docs)

In [6]: TF_vectorizer.get_feature_names()

Out[6]: ['and',
         'as',
         'faster',
         'get',
         'got',
         'hairy',
         'harry',
         'home',
         'is',
         'jill',
         'not',
         'store',
         'than',
         'the',
         'to',
         'would']

In [7]: print(TF_vectors.toarray())

        [[1 0 3 1 1 0 2 1 0 0 0 1 0 3 1 1]
         [1 0 1 0 0 1 1 0 1 1 0 0 1 0 0 0]
         [0 2 0 0 0 1 1 0 1 1 1 0 0 0 0 0]]
```

```
In [13]: binary_vectorizer = CountVectorizer(binary=True)

In [14]: binary_vectors = binary_vectorizer.fit_transform(docs)

In [15]: print(binary_vectors.toarray())

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

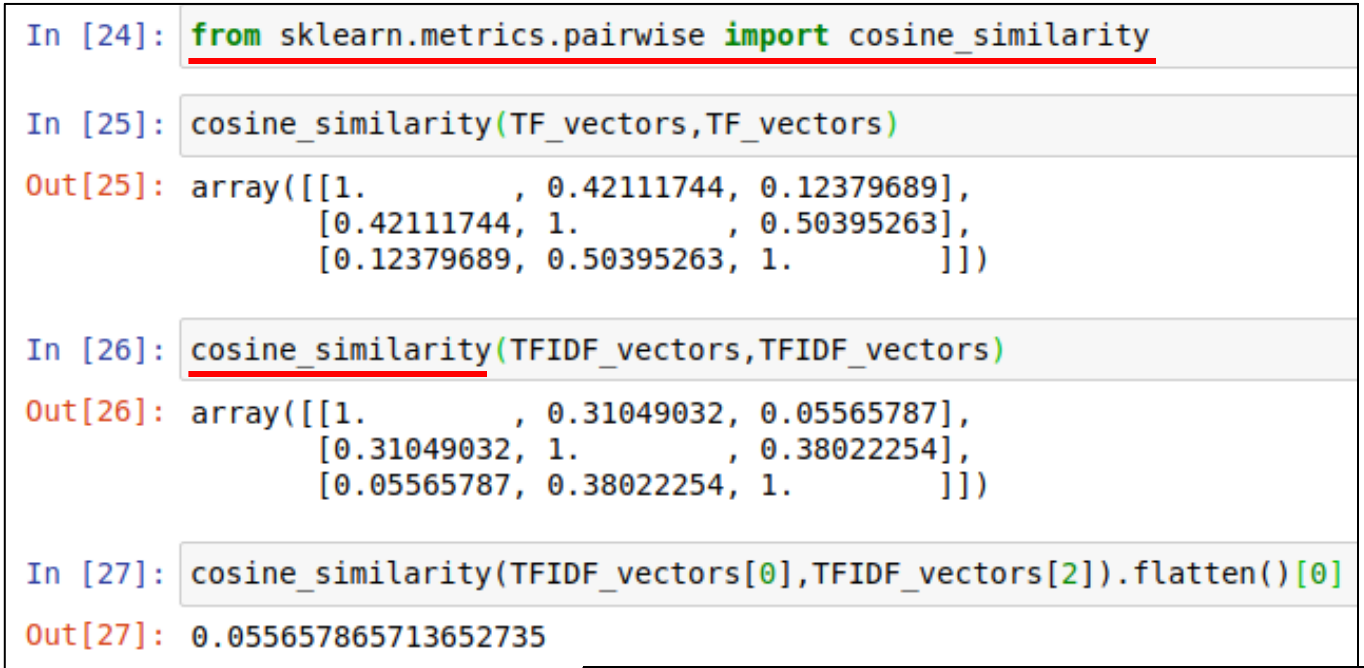
TfidfVectorizer
TF-IDF Vectors



```
In [17]: from sklearn.feature_extraction.text import TfidfVectorizer

In [18]: TFIDF_vectorizer = TfidfVectorizer()

In [19]: TFIDF_vectors = TFIDF_vectorizer.fit_transform(docs)

In [20]: print(TFIDF_vectors.toarray())

         [[0.1614879  0.         0.48446369 0.21233718 0.21233718 0.
           0.25081952 0.21233718 0.         0.         0.         0.21233718
           0.         0.63701154 0.21233718 0.21233718]
          [0.36930805 0.         0.36930805 0.         0.         0.36930805
           0.28680065 0.         0.36930805 0.36930805 0.         0.
           0.48559571 0.         0.         0.        ]
          [0.         0.75143242 0.         0.         0.         0.28574186
           0.22190405 0.         0.28574186 0.28574186 0.37571621 0.
           0.         0.         0.         0.        ]]
```

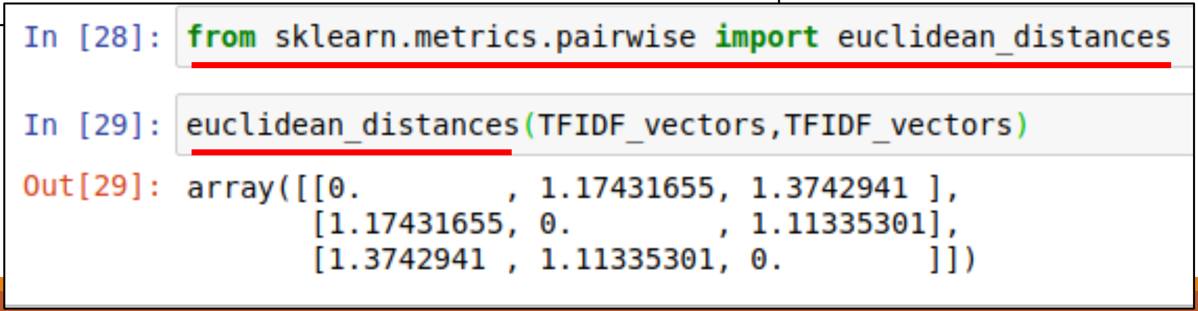
這是怎麼算出來ㄉ

Cosine Similarity & Euclidean Distance



```
In [24]: from sklearn.metrics.pairwise import cosine_similarity

In [25]: cosine_similarity(TF_vectors,TF_vectors)

Out[25]: array([[1.        , 0.42111744, 0.12379689],
                [0.42111744, 1.        , 0.50395263],
                [0.12379689, 0.50395263, 1.        ]])

In [26]: cosine_similarity(TFIDF_vectors,TFIDF_vectors)

Out[26]: array([[1.        , 0.31049032, 0.05565787],
                [0.31049032, 1.        , 0.38022254],
                [0.05565787, 0.38022254, 1.        ]])

In [27]: cosine_similarity(TFIDF_vectors[0],TFIDF_vectors[2]).flatten()[0]

Out[27]: 0.055657865713652735
```

為什麼剩3*3的資料



```
In [28]: from sklearn.metrics.pairwise import euclidean_distances

In [29]: euclidean_distances(TFIDF_vectors,TFIDF_vectors)

Out[29]: array([[0.        , 1.17431655, 1.3742941 ],
                [1.17431655, 0.        , 1.11335301],
                [1.3742941 , 1.11335301, 0.        ]])
```

Parameters When Creating Vectorizer

Useful parameters when creating `CountVectorizer` and `TfidfVectorizer`

- `lowcase`: Convert all characters to lowercase before tokenizing, default is true
- `stop_words`: default = None, or `stop_words = 'English'`
- `max_df`: remove terms whose document frequencies are higher than the given threshold
- `min_df`: remove terms whose document frequencies are lower than the given threshold