ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

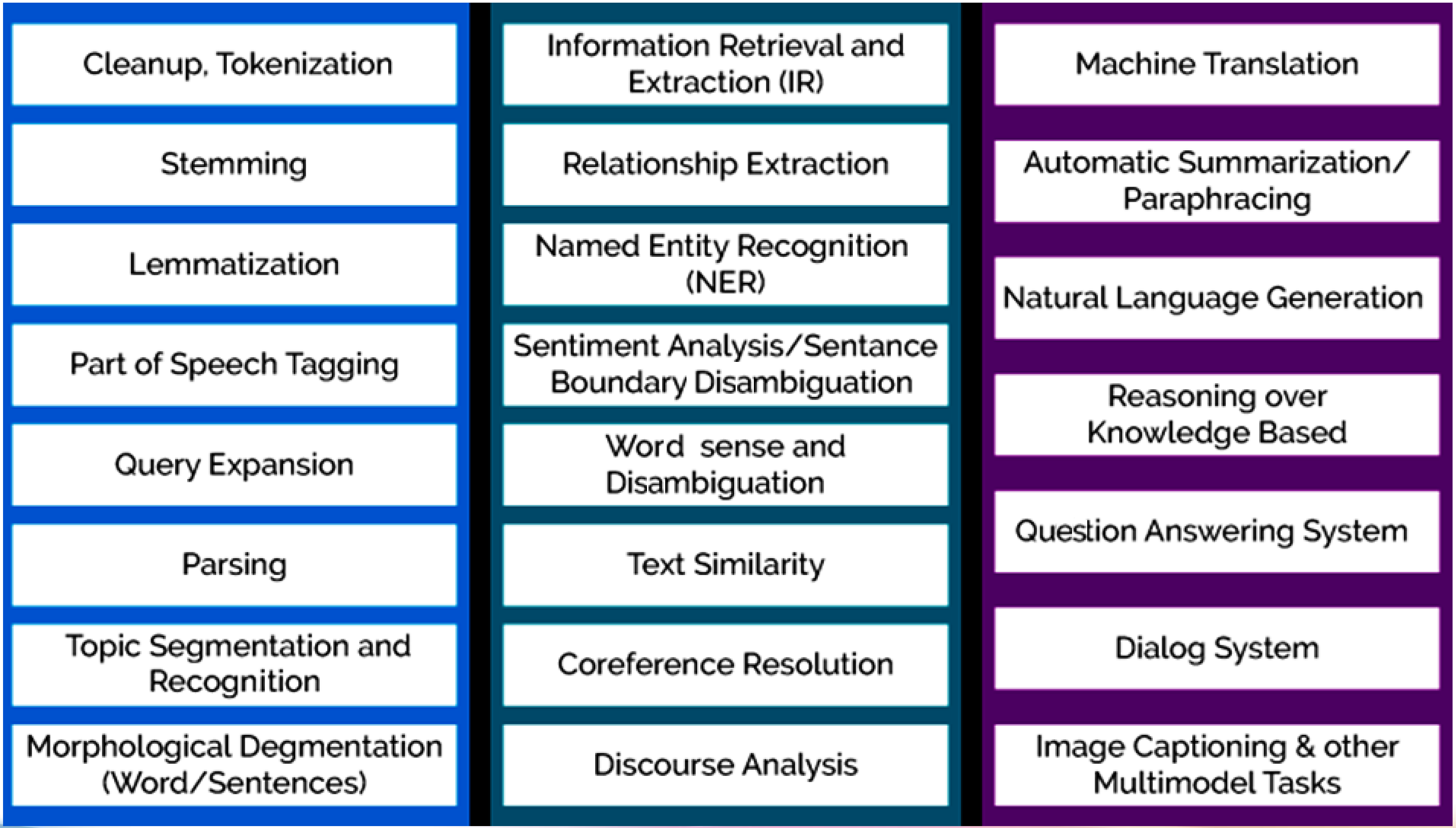
Trường Đại học Tôn Đức Thắng

Machine Learning
Language Modeling

Giảng viên:	Tiến sĩ Bùi Thanh Hùng
	Trưởng Lab Khoa học Phân tích dữ liệu và Trí tuệ nhân tạo
	Giám đốc chương trình Hệ thống thông tin
	Đại học Thủ Dầu Một
Email:	tuhungphe@gmail.com
Website:	https://sites.google.com/site/hungthanhbui1980/

Natural Language Processing

Natural Language Processing



Cleanup, Tokenization	Information Retrieval and Extraction (IR)	Machine Translation
Stemming	Relationship Extraction	Automatic Summarization/ Paraphracing
Lemmatization	Named Entity Recognition (NER)	Natural Language Generation
Part of Speech Tagging	Sentiment Analysis/Sentance Boundary Disambiguation	Reasoning over Knowledge Based
Query Expansion	Word sense and Disambiguation	Question Answering System
Parsing	Text Similarity	Dialog System
Topic Segmentation and Recognition	Coreference Resolution	Image Captioning & other Multimodel Tasks
Morphological Degmentation (Word/Sentences)	Discourse Analysis	

Natural Language Processing

Dependency Parsing

Raw sentence

He reckons the current account deficit will narrow to only 1.8 billion in September.

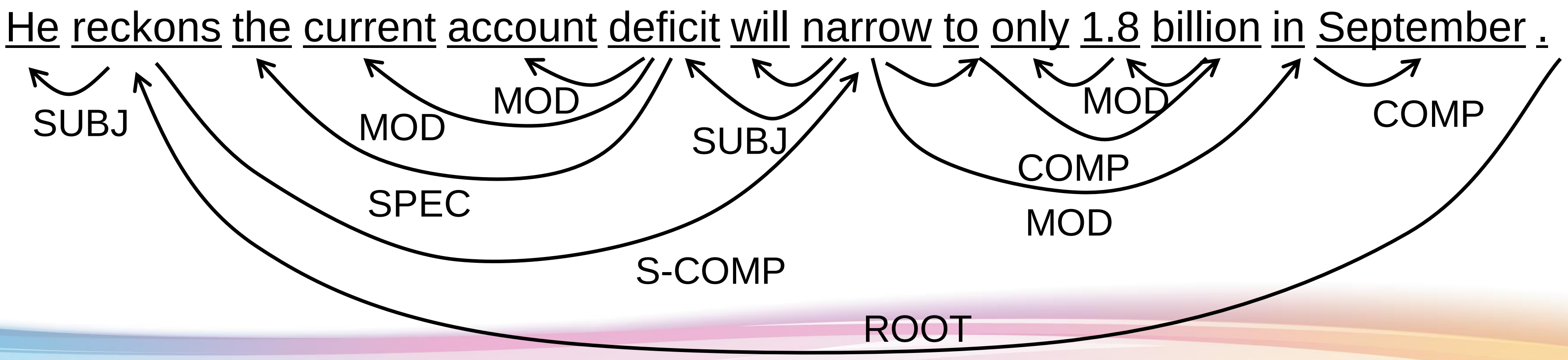
Part-of-speech tagging

POS-tagged sentence

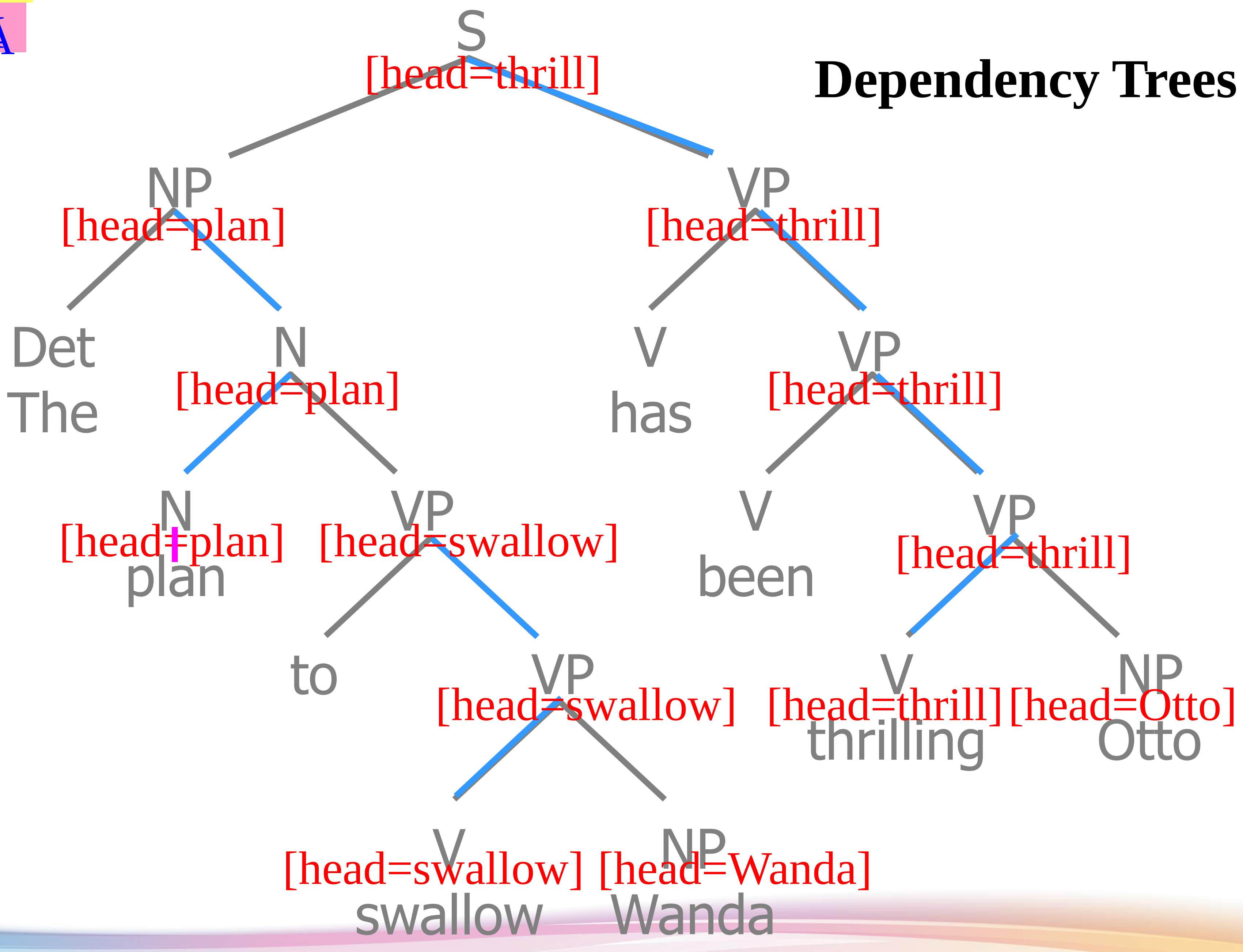
He	reckons	the	current	account	deficit	will	narrow	to	only	1.8	billion	in	September	.
PRP	VBZ	DT	JJ	NN	NN	MD	VB	TO	RB	CD	CD	IN	NNP	.

Word dependency parsing

Word dependency parsed sentence


He reckons the current account deficit will narrow to only 1.8 billion in September .
SUBJ
MOD
MOD
SPEC
SUBJ
S-COMP
MOD
COMP
MOD
COMP
ROOT

Dependency Trees

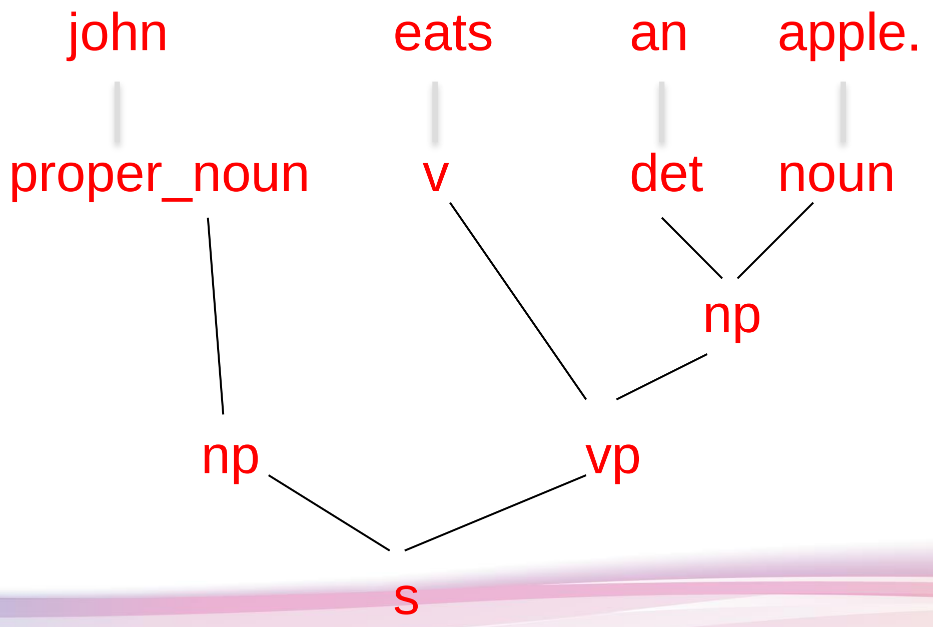


Parsing (in Definite Clause Grammars)

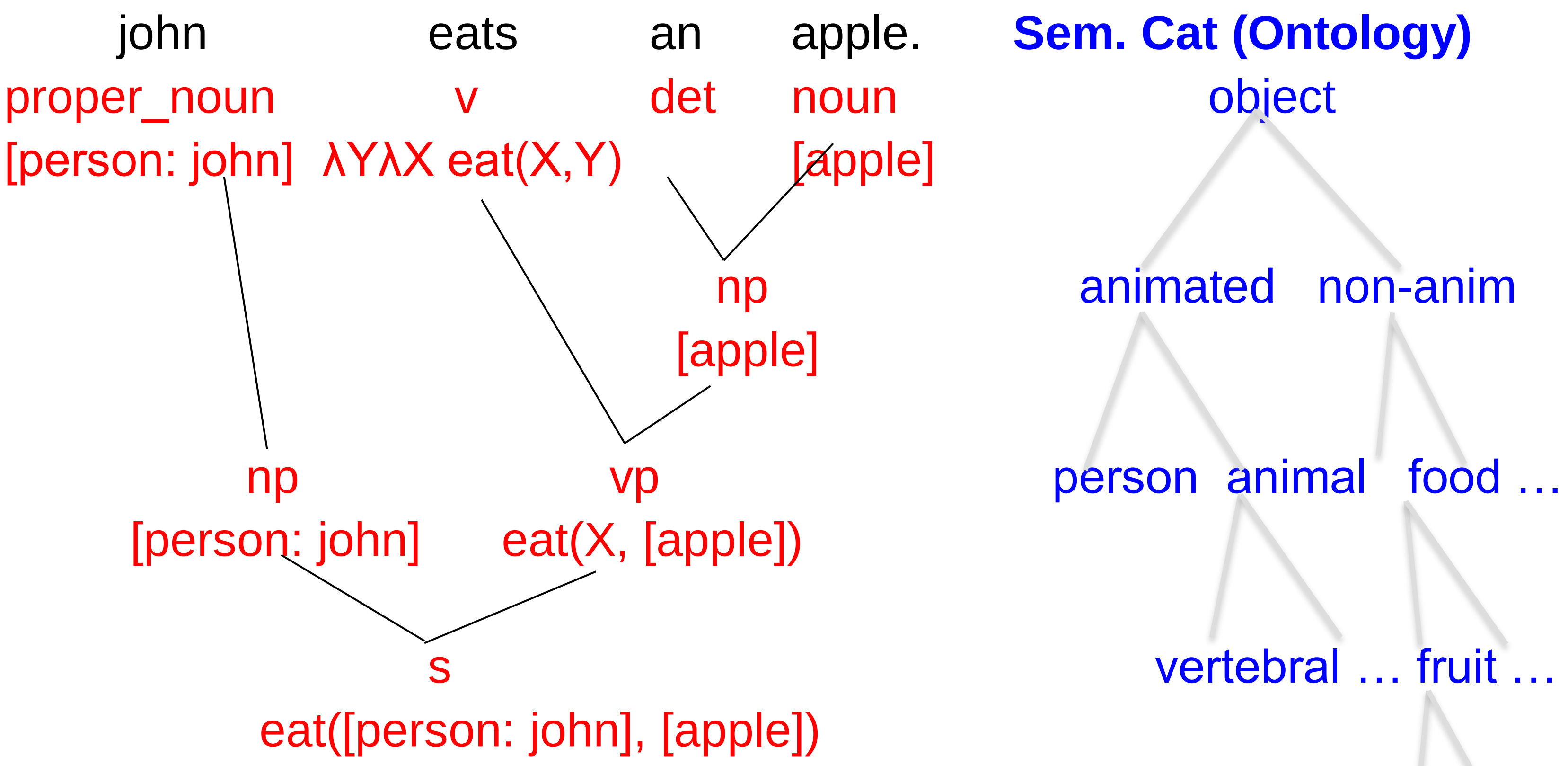
```
s --> np, vp.
np --> det, noun.
np --> proper_noun.
vp --> v, ng.
vp --> v.
```

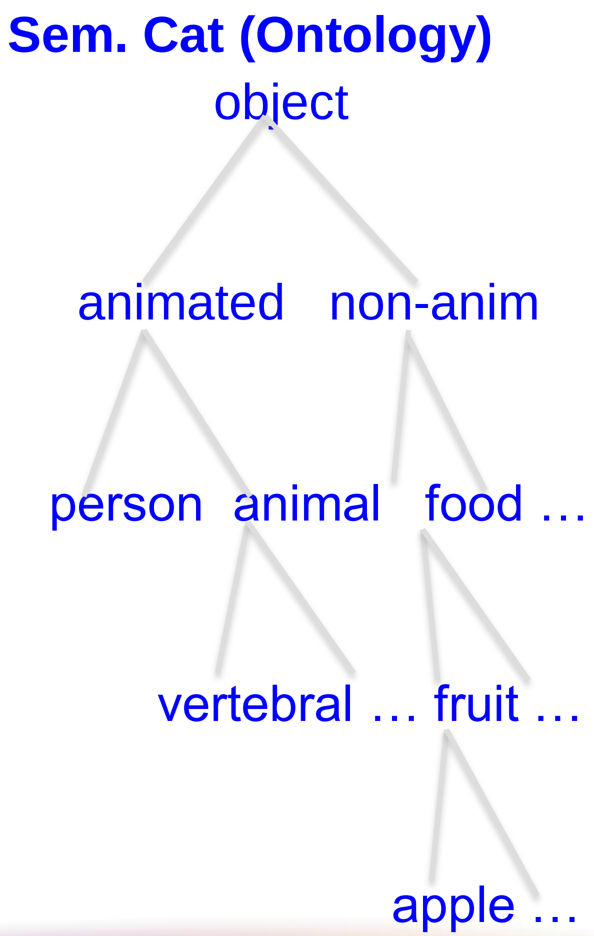
```
det -->[a].        det --> [an].
det --> [the].
noun --> [apple].
noun --> [orange].
proper_noun --> [john].
proper_noun --> [mary].
v --> [eats].
v --> [loves].
```

Eg.



Semantic analysis





Bag of words - One hot vector

Nhà_hàng này ngon, nhân_viên nhiệt_tình

→ [0 , 0 , 0 , 0 , 1 , 1 , 1 , 1 , 1 , 0]

Đồ ăn ngon, nhưng mắc.

→ [1 , 0 , 1 , 1 , 0 , 1 , 0 , 0 , 0 , 1]

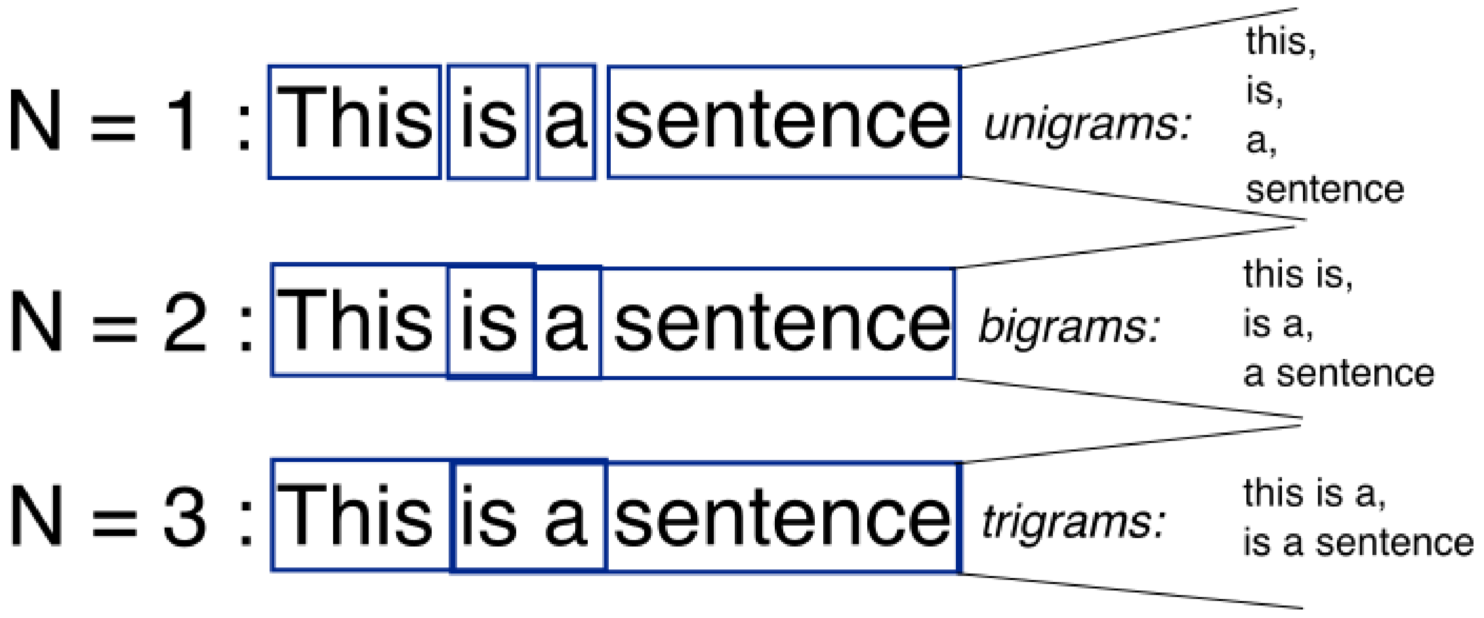
Nhân_viên chuyên_nghiệp

→ [0 , 1 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0]

0	ăn
1	chuyên_nghiệp
2	đồ
3	mắc
4	này
5	ngon
6	nhà_hàng
7	nhân_viên
8	nhiệt_tình
9	nhưng

N-Gram

is a connected string of **N**



Language Modeling

Probabilistic Language Models

- Today's goal: assign a probability to a sentence
  - Machine Translation:
    - » P(**high** winds tonite) > P(**large** winds tonite)
  - Spell Correction
    - » The office is about fifteen **minuets** from my house
      - P(about fifteen **minutes** from) > P(about fifteen **minuets** from)
  - Speech Recognition
    - » P(I saw a van) >> P(eyes awe of an)
  - + Summarization, question-answering, etc., etc.!!

Why?

Probabilistic Language Modeling

- Goal: compute the probability of a sentence or sequence of words:

  $P(W) = P(w_1,w_2,w_3,w_4,w_5\ldots w_n)$

- Related task: probability of an upcoming word:

  $P(w_5|w_1,w_2,w_3,w_4)$

- A model that computes either of these:

  $P(W)$ or $P(w_n|w_1,w_2\ldots w_{n-1})$ is called a **language model**.

- Better: **the grammar** But **language model** or **LM** is standard

How to compute P(W)

- How to compute this joint probability:
  - P(its, water, is, so, transparent, that)
- Intuition: let’s rely on the Chain Rule of Probability

Reminder: The Chain Rule

- Recall the definition of conditional probabilities

  $\mathbf{p(B|A) = P(A,B)/P(A)}$ Rewriting: $\mathbf{P(A,B) = P(A)P(B|A)}$

- More variables:

  $P(A,B,C,D) = P(A)P(B|A)P(C|A,B)P(D|A,B,C)$

- The Chain Rule in General

$$P(x_1,x_2,x_3,\ldots,x_n) = P(x_1)P(x_2|x_1)P(x_3|x_1,x_2)\ldots P(x_n|x_1,\ldots,x_{n-1})$$

The Chain Rule applied to compute joint probability of words in sentence

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$$

P("its water is so transparent") =

P(its) × P(water|its) × P(is|its water)

× P(so|its water is) × P(transparent|its water is so)

How to estimate these probabilities

- Could we just count and divide?

$$P(\text{the} \mid \text{its water is so transparent that}) = \frac{Count(\text{its water is so transparent that the})}{Count(\text{its water is so transparent that})}$$

- No! Too many possible sentences!
- We’ll never see enough data for estimating these

Markov Assumption

Andrei Markov

- Simplifying assumption:

$$P(\text{the} \mid \text{its water is so transparent that}) \text{»} P(\text{the} \mid \text{that})$$

- Or maybe

$$P(\text{the} \mid \text{its water is so transparent that}) \text{»} P(\text{the} \mid \text{transparent that})$$

Markov Assumption

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i \mid w_{i-k} \ldots w_{i-1})$$

- In other words, we approximate each component in the product

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-k} \ldots w_{i-1})$$

Simplest case: Unigram model

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i)$$

Some automatically generated sentences from a unigram model

fifth, an, of, futures, the, an, incorporated, a, a, the, inflation, most, dollars, quarter, in, is, mass

thrift, did, eighty, said, hard, 'm, july, bullish

that, or, limited, the

Bigram model

- Condition on the previous word:

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-1})$$

texaco, rose, one, in, this, issue, is, pursuing, growth, in, a, boiler, house, said, mr., gurria, mexico, 's, motion, control, proposal, without, permission, from, five, hundred, fifty, five, yen

outside, new, car, parking, lot, of, the, agreement, reached

this, would, be, a, record, november

N-gram models

- We can extend to trigrams, 4-grams, 5-grams
- In general this is an insufficient model of language
  - because language has **long-distance dependencies**:

  "The computer which I had just put into the machine room on the fifth floor crashed."

- But we can often get away with N-gram models

Estimating bigram probabilities

- The Maximum Likelihood Estimate

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

$$P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

An example

$$P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

<s> I am Sam </s>

<s> Sam I am </s>

<s> I do not like green eggs and ham </s>

$P(\texttt{I} \mid \texttt{<s>}) = \frac{2}{3} = .67$ $P(\texttt{Sam} \mid \texttt{<s>}) = \frac{1}{3} = .33$ $P(\texttt{am} \mid \texttt{I}) = \frac{2}{3} = .67$

$P(\texttt{</s>} \mid \texttt{Sam}) = \frac{1}{2} = 0.5$ $P(\texttt{Sam} \mid \texttt{am}) = \frac{1}{2} = .5$ $P(\texttt{do} \mid \texttt{I}) = \frac{1}{3} = .33$

More examples:
Berkeley Restaurant Project sentences

- can you tell me about any good cantonese restaurants close by
- mid priced thai food is what i’m looking for
- tell me about chez panisse
- can you give me a listing of the kinds of food that are available
- i’m looking for a good place to eat breakfast
- when is caffe venezia open during the day

Raw bigram counts

- Out of 9222 sentences

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Raw bigram probabilities

- Normalize by unigrams:

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

- Result:

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Bigram estimates of sentence probabilities

P(<s> I want english food </s>) =

P(I|<s>)

× P(want|I)

× P(english|want)

× P(food|english)

× P(</s>|food)

= .000031

What kinds of knowledge?

- P(english|want) = .0011
- P(chinese|want) = .0065
- P(to|want) = .66
- P(eat | to) = .28
- P(food | to) = 0
- P(want | spend) = 0
- P (i | <s>) = .25

Practical Issues

- We do everything in log space
  - Avoid underflow
  - (also adding is faster than multiplying)

$$\log(p_1 ´ \ p_2 ´ \ p_3 ´ \ p_4) = \log p_1 + \log p_2 + \log p_3 + \log p_4$$

Estimating N-gram Probabilities

- Estimating N-gram Probabilities
- Evaluation and Perplexity

Evaluation: How good is our model?

- Does our language model prefer good sentences to bad ones?
  - Assign higher probability to "real" or "frequently observed" sentences
    - Than "ungrammatical" or "rarely observed" sentences?
- We train parameters of our model on a **training set**.
- We test the model's performance on data we haven't seen.
  - A **test set** is an unseen dataset that is different from our training set, totally unused.
  - An **evaluation metric** tells us how well our model does on the test set.

Perplexity

The best language model is one that best predicts an unseen test set

- Gives the highest P(sentence)

Perplexity is the inverse probability of the test set, normalized by the number of words:

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Chain rule:

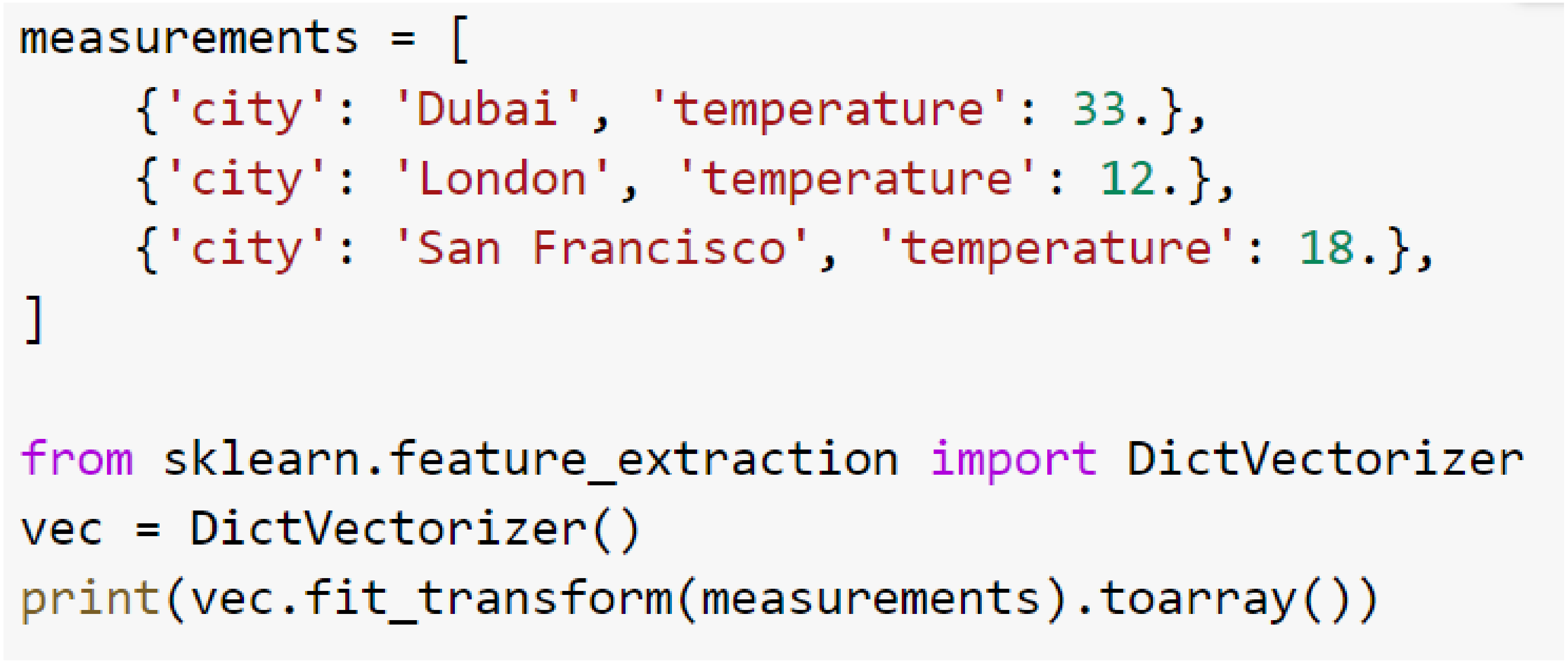
$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}}$$

For bigrams:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

Minimizing perplexity is the same as maximizing probability

Dict to Vec



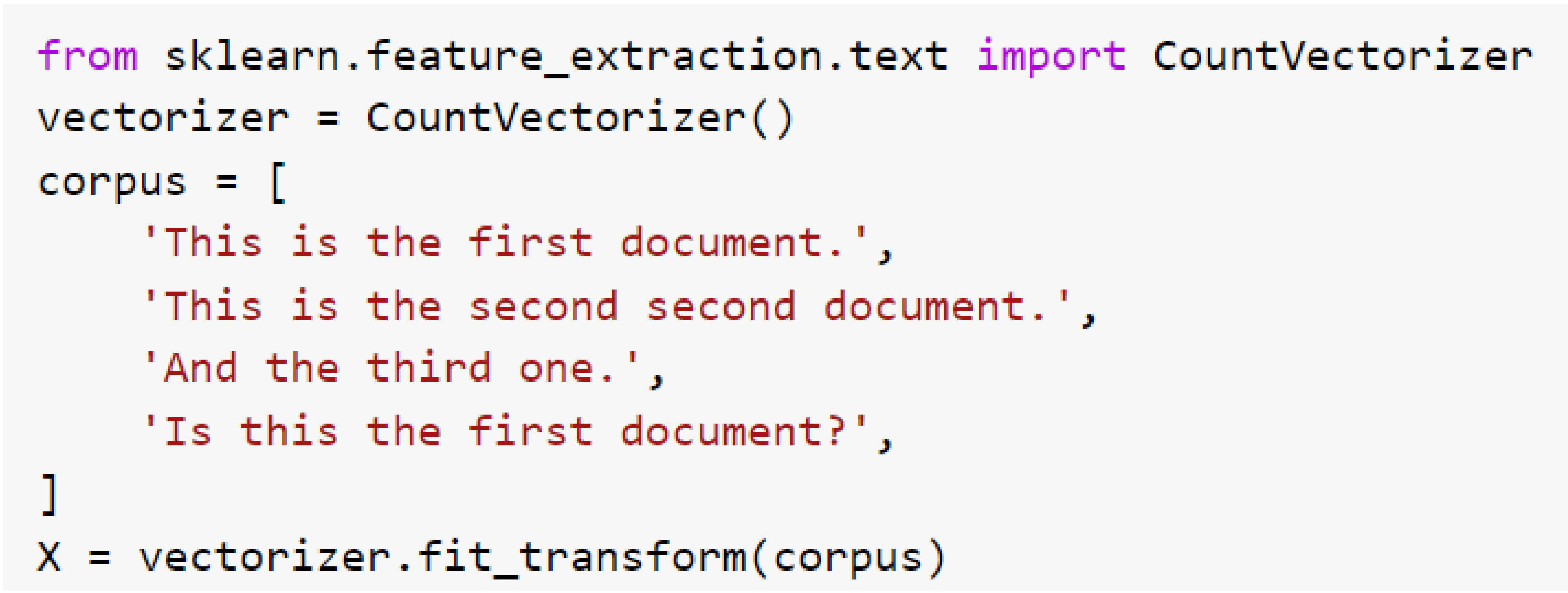
```
measurements = [
    {'city': 'Dubai', 'temperature': 33.},
    {'city': 'London', 'temperature': 12.},
    {'city': 'San Francisco', 'temperature': 18.},
]

from sklearn.feature_extraction import DictVectorizer
vec = DictVectorizer()
print(vec.fit_transform(measurements).toarray())
```

Dict to Vec

```python
measurements = [
    {'city': 'Dubai', 'temperature': 33.},
    {'city': 'London', 'temperature': 12.},
    {'city': 'San Francisco', 'temperature': 18.},
]

from sklearn.feature_extraction import DictVectorizer
vec = DictVectorizer()
print(vec.fit_transform(measurements).toarray())
```

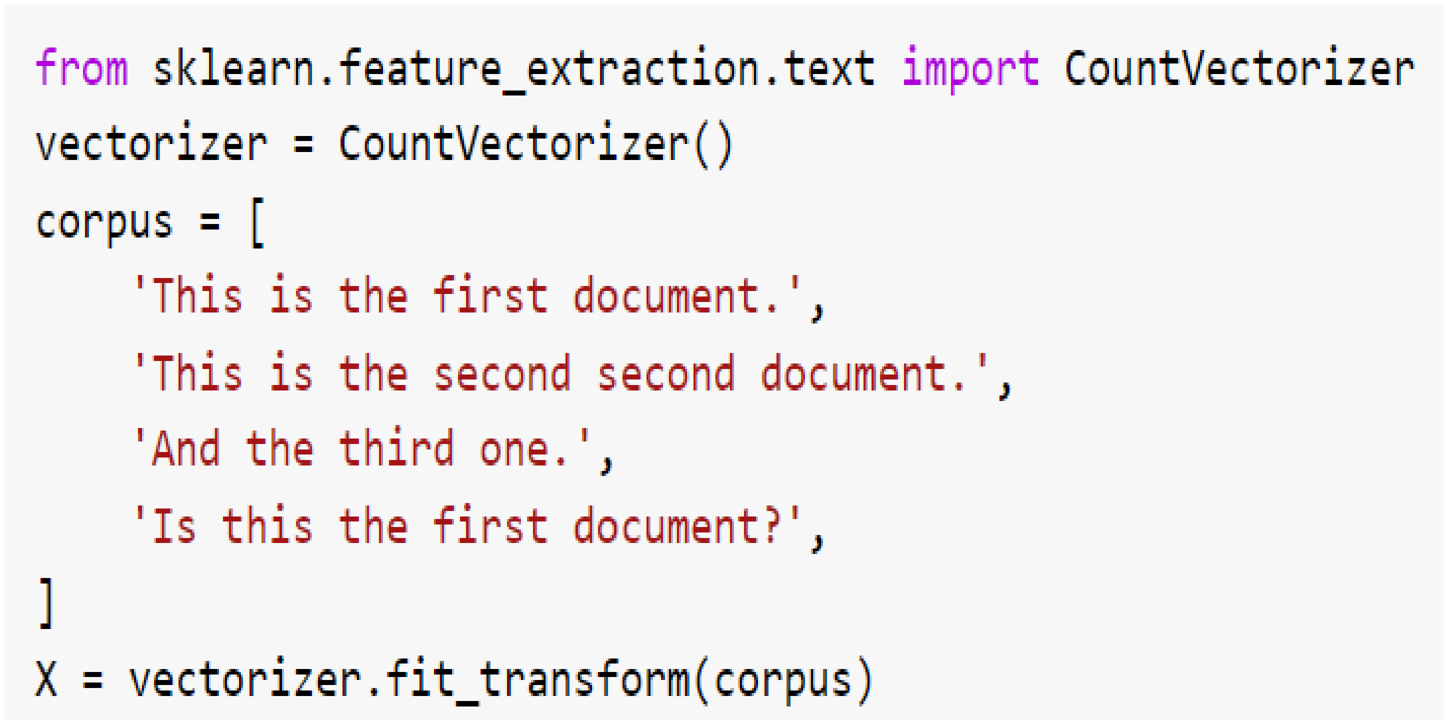
```
[[ 1.   0.   0. 33.]
 [ 0.   1.   0. 12.]
 [ 0.   0.   1. 18.]]
```

Count Vectorize

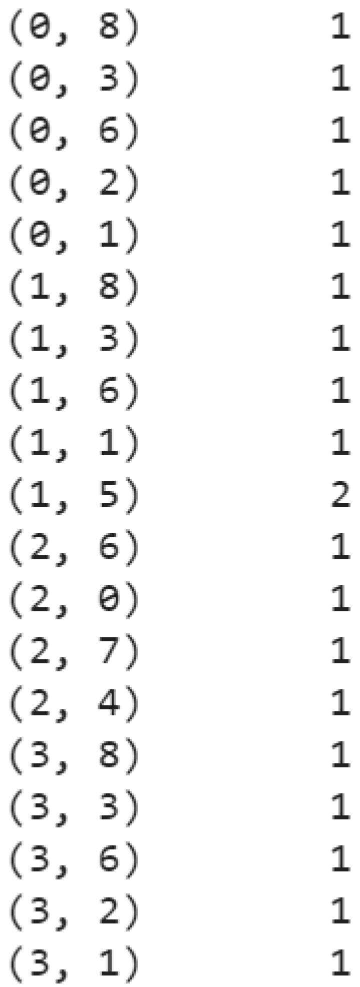


```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
```

Count Vectorize

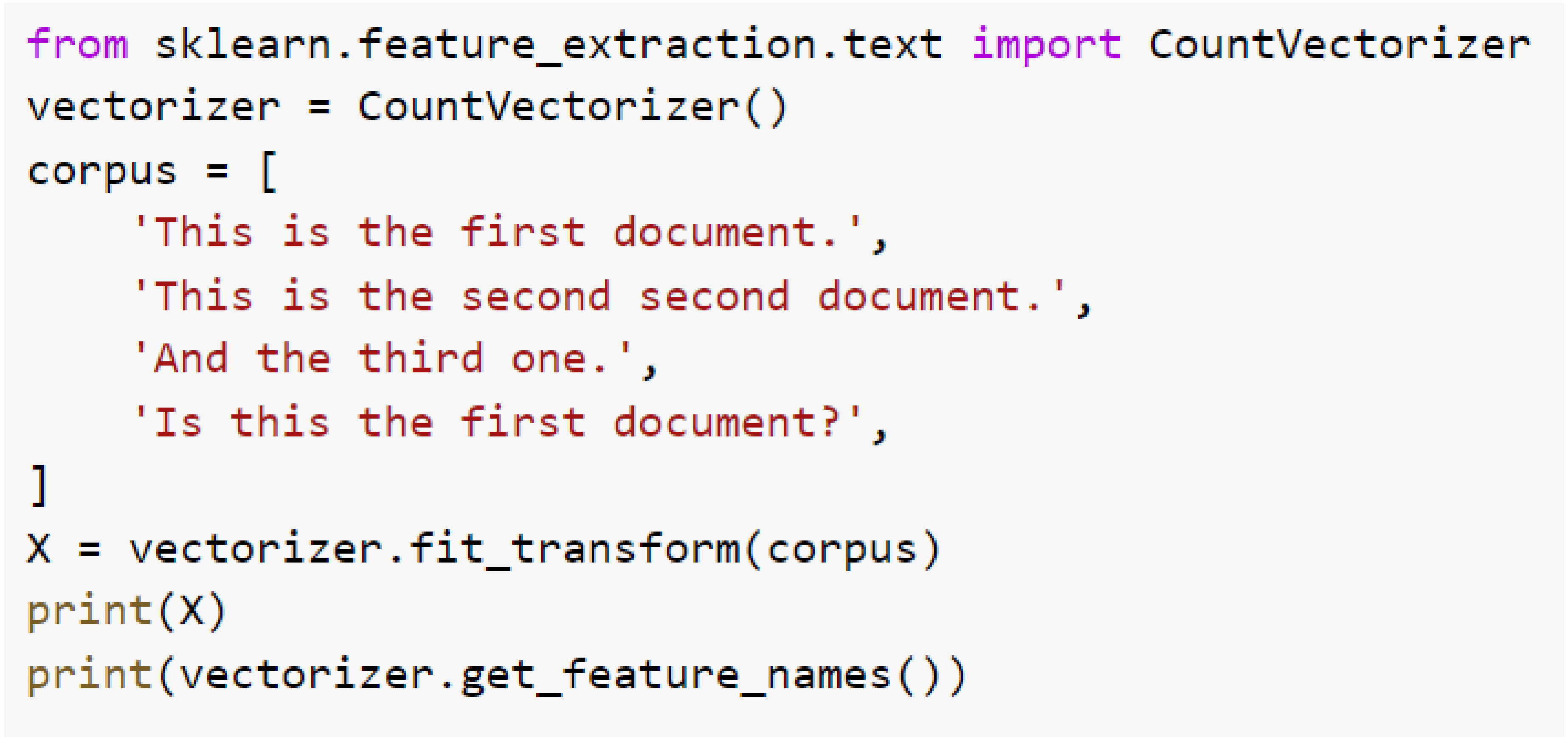


```python
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
```



```
(0, 8)        1
(0, 3)        1
(0, 6)        1
(0, 2)        1
(0, 1)        1
(1, 8)        1
(1, 3)        1
(1, 6)        1
(1, 1)        1
(1, 5)        2
(2, 6)        1
(2, 0)        1
(2, 7)        1
(2, 4)        1
(3, 8)        1
(3, 3)        1
(3, 6)        1
(3, 2)        1
(3, 1)        1
```

Count Vectorize



```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
print(X)
print(vectorizer.get_feature_names())
```

Count Vectorize

```python
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
print(X)
print(vectorizer.get_feature_names())
```

```
(0, 8)        1
(0, 3)        1
(0, 6)        1
(0, 2)        1
(0, 1)        1
(1, 8)        1
(1, 3)        1
(1, 6)        1
(1, 1)        1
(1, 5)        2
(2, 6)        1
(2, 0)        1
(2, 7)        1
(2, 4)        1
(3, 8)        1
(3, 3)        1
(3, 6)        1
(3, 2)        1
(3, 1)        1
```

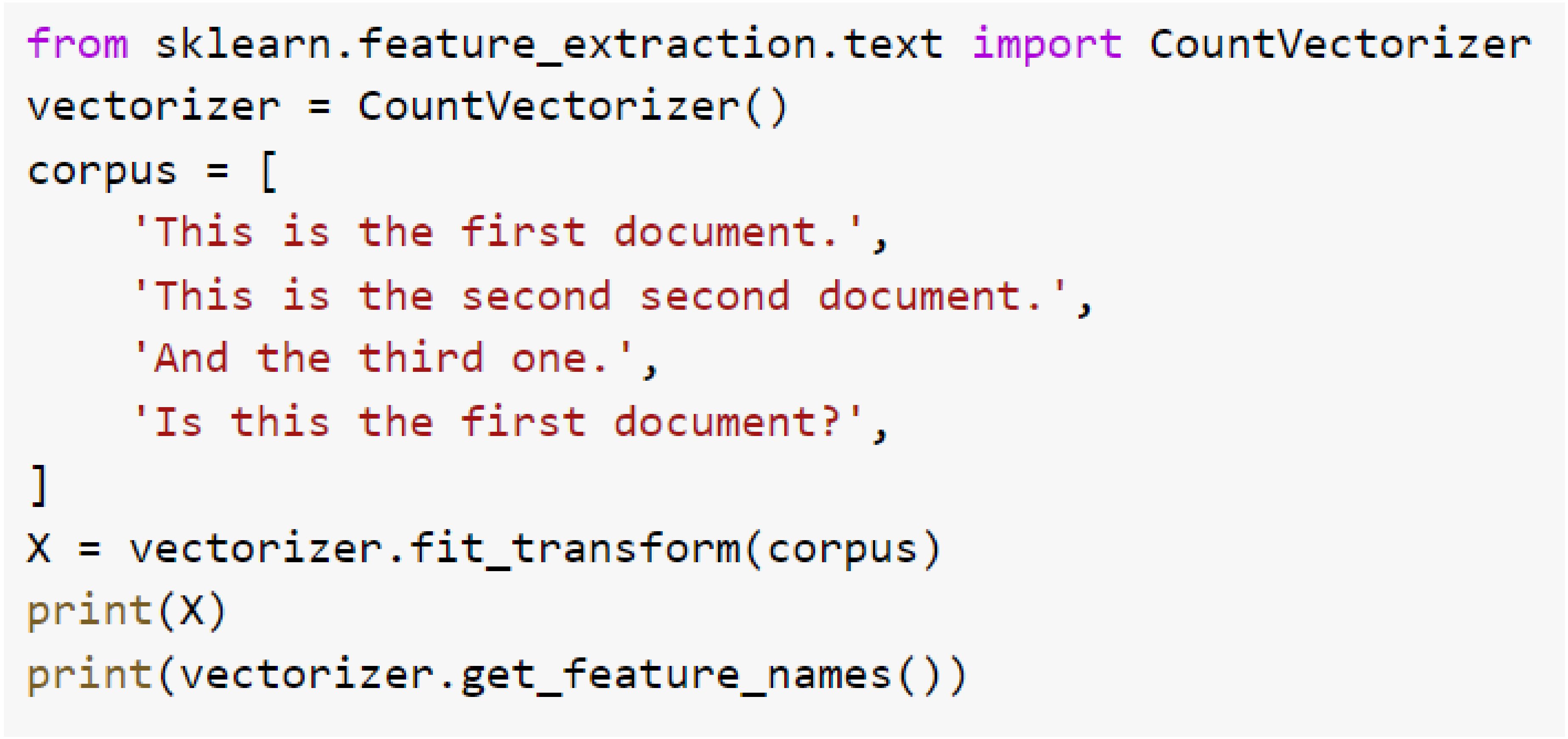
```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

Count Vectorize

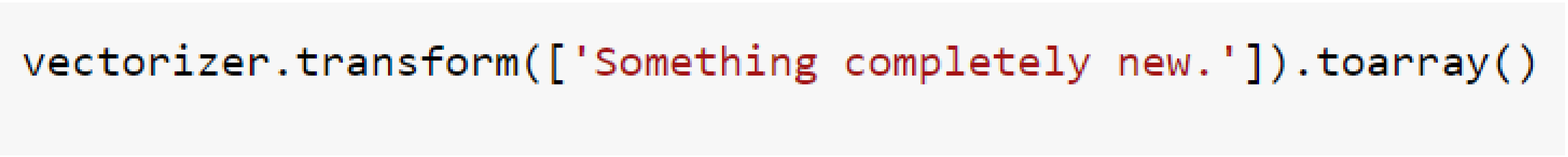
```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
print(X)
print(vectorizer.get_feature_names())
```

```
print(vectorizer.vocabulary_.get('document'))
print(vectorizer.vocabulary_.get('and'))
```

Count Vectorize

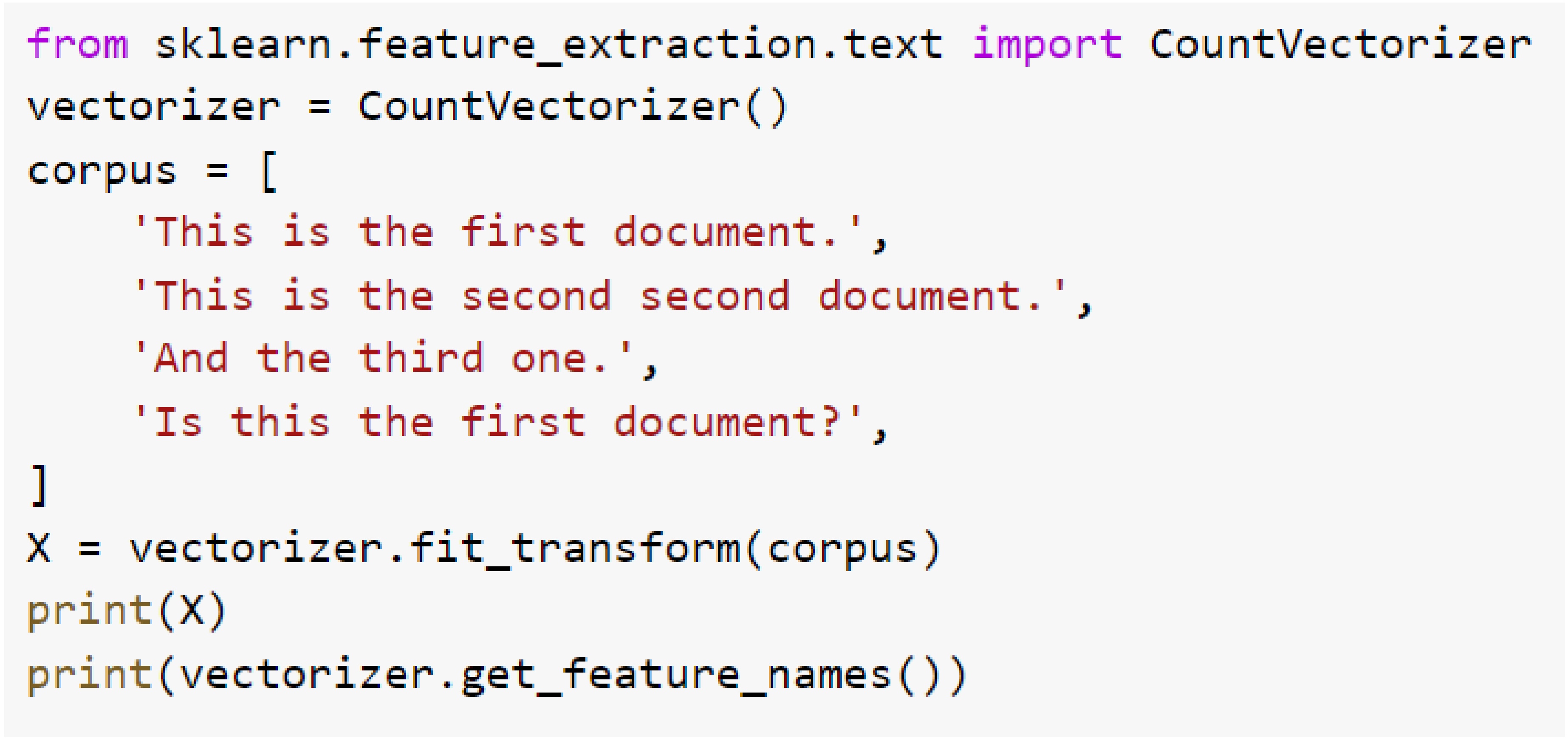


```python
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
print(X)
print(vectorizer.get_feature_names())
```

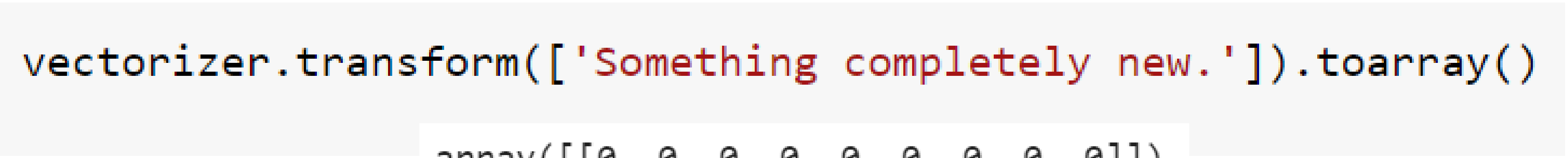


```python
vectorizer.transform(['Something completely new.']).toarray()
```

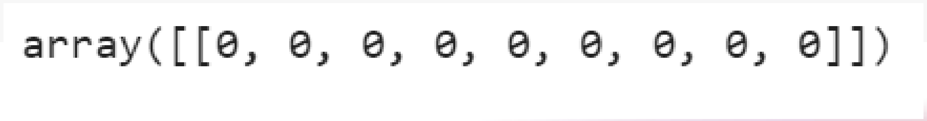
Count Vectorize



```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
print(X)
print(vectorizer.get_feature_names())
```

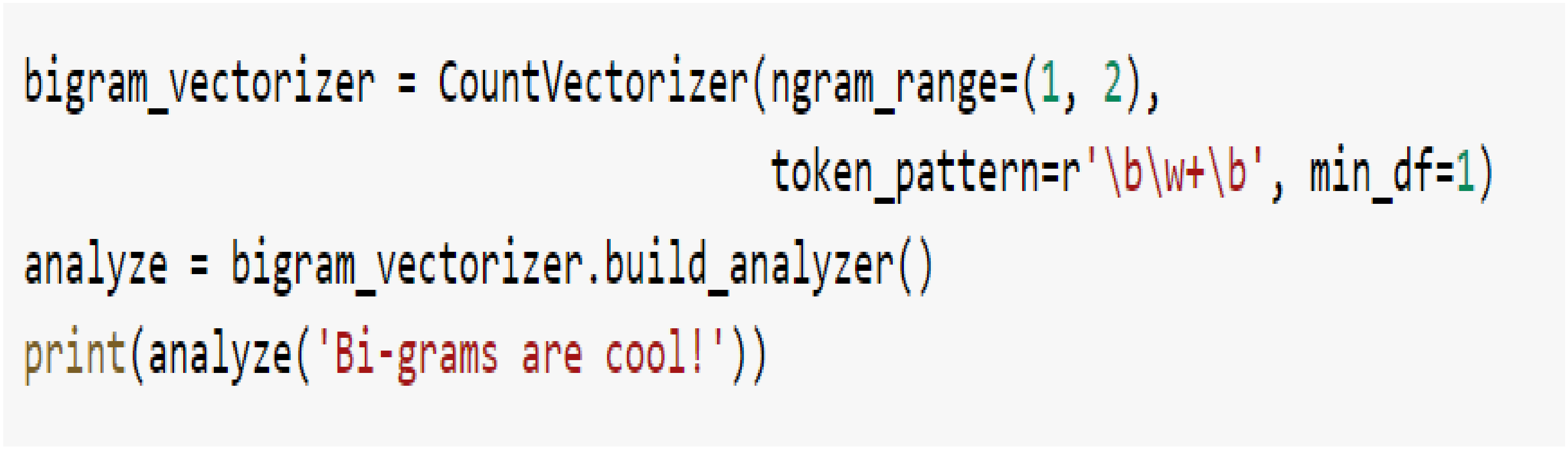


```
vectorizer.transform(['Something completely new.']).toarray()
```



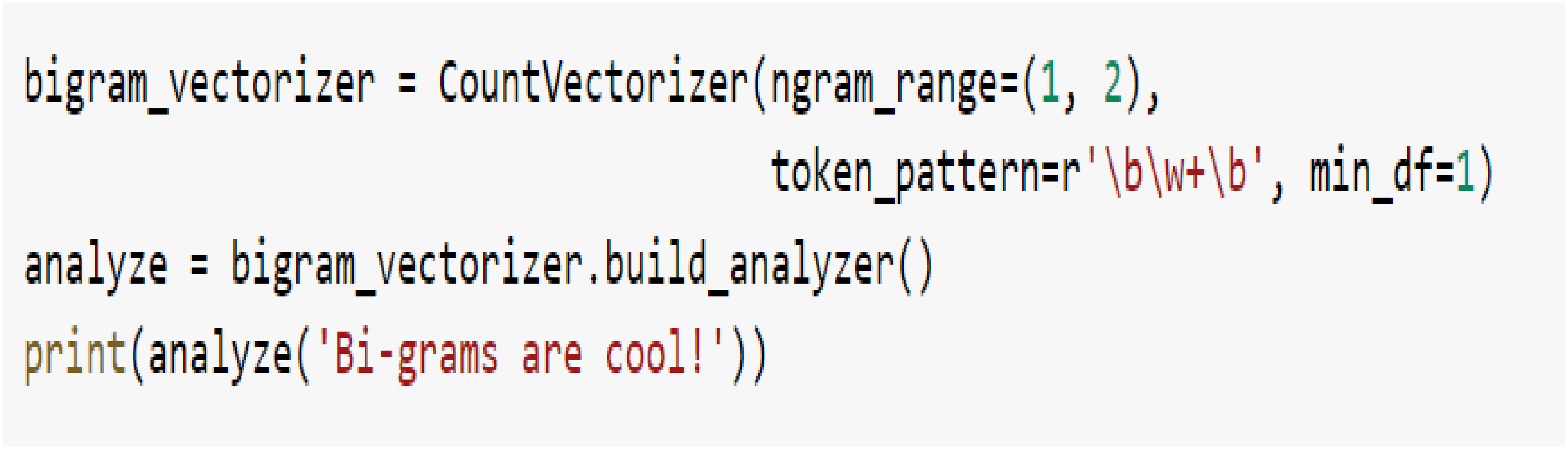
```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Bigram

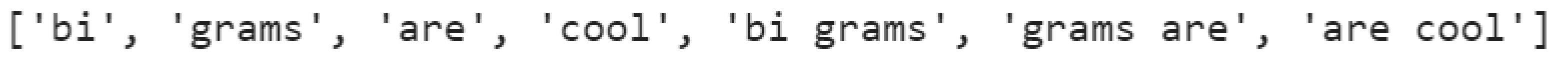


```
bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
                                    token_pattern=r'\b\w+\b', min_df=1)
analyze = bigram_vectorizer.build_analyzer()
print(analyze('Bi-grams are cool!'))
```

Bigram



```python
bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
                                    token_pattern=r'\b\w+\b', min_df=1)
analyze = bigram_vectorizer.build_analyzer()
print(analyze('Bi-grams are cool!'))
```



```
['bi', 'grams', 'are', 'cool', 'bi grams', 'grams are', 'are cool']
```

Bigram

```
from sklearn.feature_extraction.text import CountVectorizer
bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
                                    token_pattern=r'\b\w+\b', min_df=1)
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X_2 = bigram_vectorizer.fit_transform(corpus).toarray()
print(X_2)
```

Bigram

```python
from sklearn.feature_extraction.text import CountVectorizer
bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
                                    token_pattern=r'\b\w+\b', min_df=1)
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
X_2 = bigram_vectorizer.fit_transform(corpus).toarray()
print(X_2)
```

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

TF-IDF

- In a large text corpus, some words will be very present (e.g. “the”, “a”, “is” in English) hence carrying very little meaningful information about the actual contents of the document. If we were to feed the direct count data directly to a classifier those very frequent terms would shadow the frequencies of rarer yet more interesting terms.
- In order to re-weight the count features into floating point values suitable for usage by a classifier it is very common to use the tf–idf transform.

TF-IDF

- t — term (word)
- d — document (set of words)
- N — count of corpus
- corpus — the total document set

TF-IDF

- Tf means **term-frequency** while tf–idf means term-frequency times **inverse document-frequency**:

tf-idf(t, d) = tf(t, d) * log(N/(df + 1))

- *tf(t,d) = count of t in d / number of words in d*

- *df(t) = occurrence of t in documents*

- *idf(t) = log(N/(df + 1))*

TF-IDF

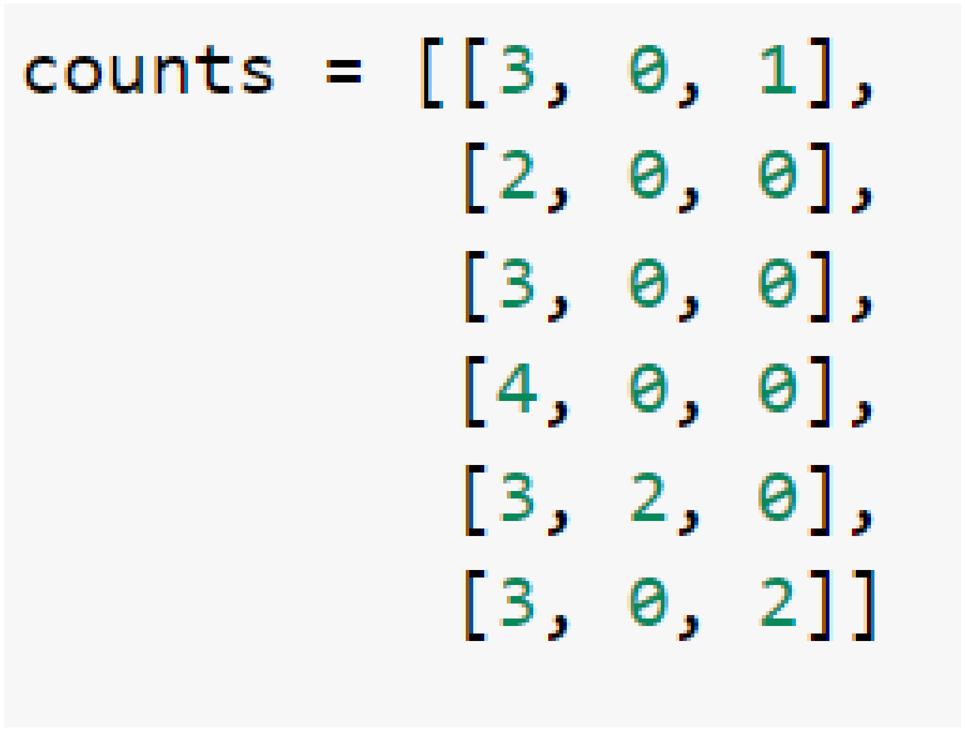
- Tf means **term-frequency** while tf–idf means term-frequency times **inverse document-frequency**:

tf-idf(t, d) = tf(t, d) * log(N/(df + 1))

- The resulting tf-idf vectors are then normalized by the Euclidean norm

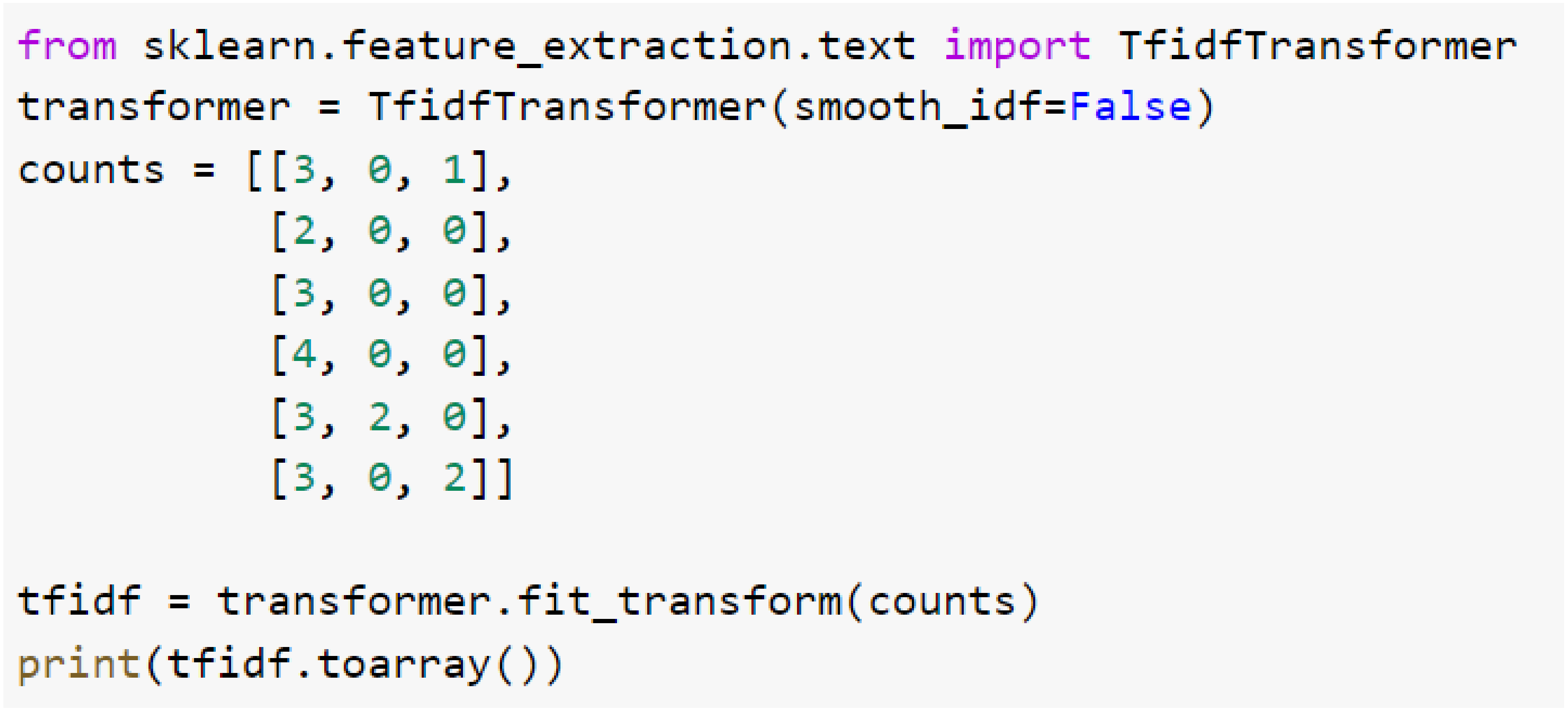
$$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_1{}^2 + v_2{}^2 + \cdots + v_n{}^2}}$$

TF-IDF



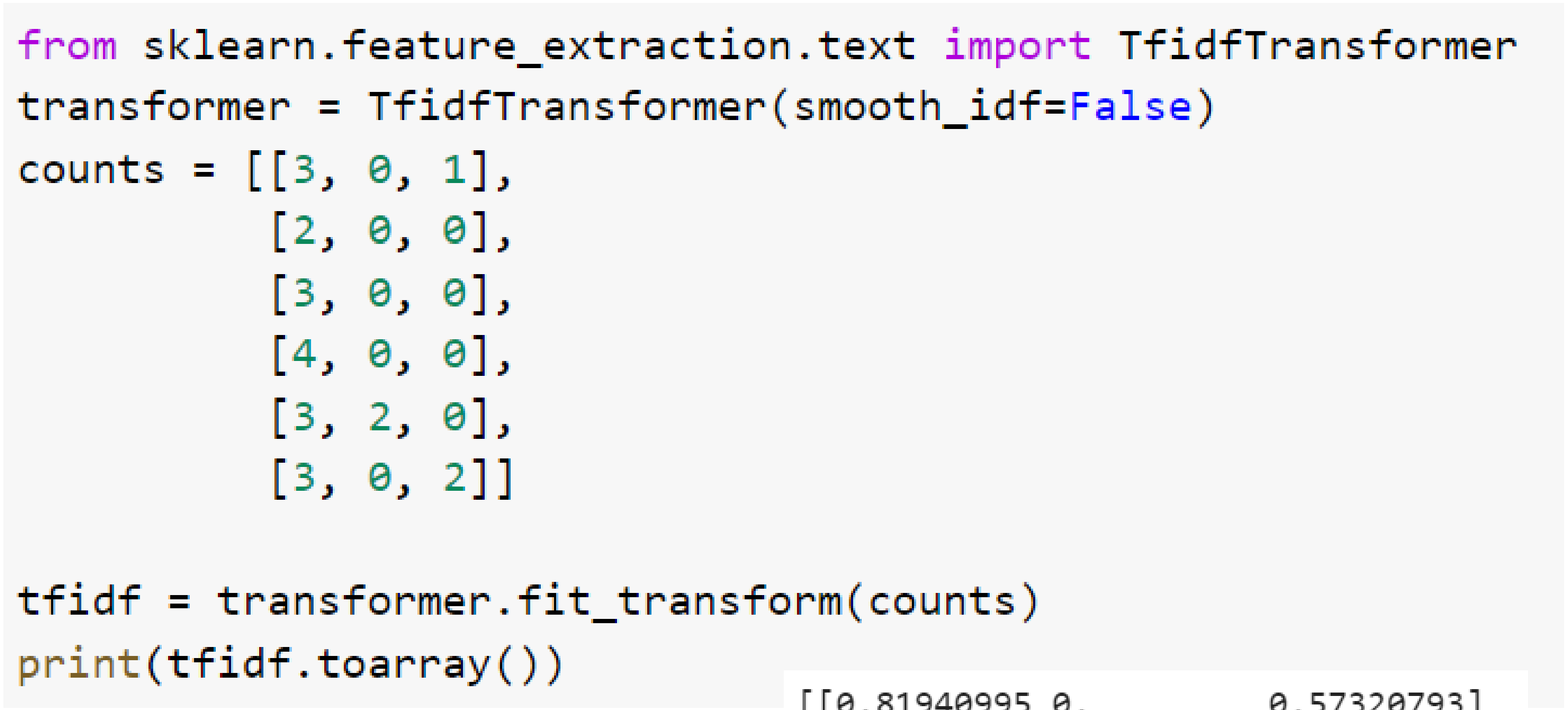
```
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]
```

TF-IDF

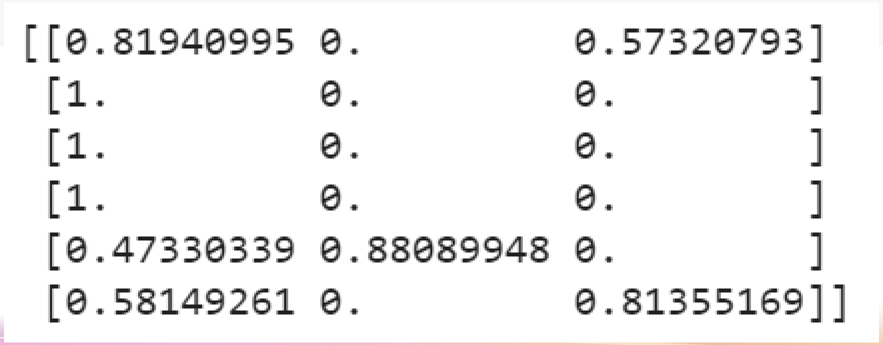


```python
from sklearn.feature_extraction.text import TfidfTransformer
transformer = TfidfTransformer(smooth_idf=False)
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]

tfidf = transformer.fit_transform(counts)
print(tfidf.toarray())
```

TF-IDF

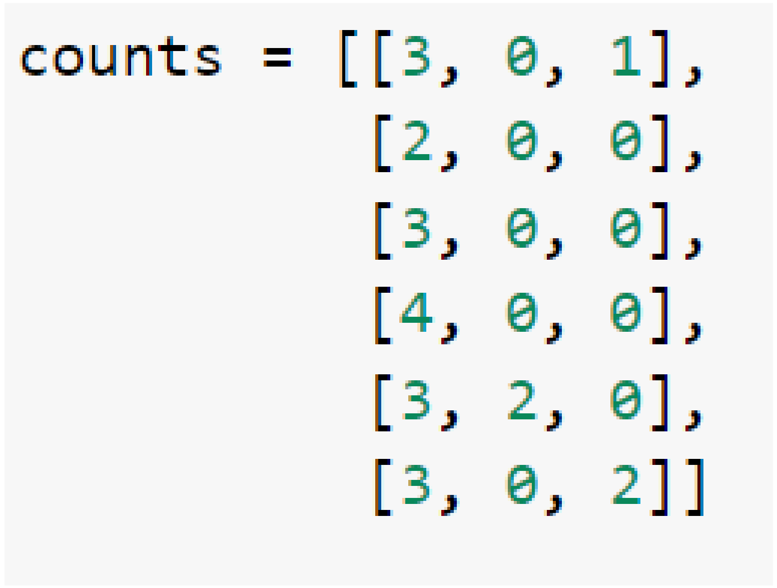


```python
from sklearn.feature_extraction.text import TfidfTransformer
transformer = TfidfTransformer(smooth_idf=False)
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]

tfidf = transformer.fit_transform(counts)
print(tfidf.toarray())
```



```
[[0.81940995 0.         0.57320793]
 [1.         0.         0.        ]
 [1.         0.         0.        ]
 [1.         0.         0.        ]
 [0.47330339 0.88089948 0.        ]
 [0.58149261 0.         0.81355169]]
```

TF-IDF



```
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]
```

Term:
Document:

Df:
Tf:
Tf-idf:

TF-IDF

```
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]
```

Term: 3
Document: 6

tf(t,d) = count of t in d / number of words in d

df(t) = occurrence of t in documents

idf(t) = log(N/(df + 1))

tf-idf(t, d) = tf(t, d) * log(N/(df + 1))

TF-IDF

```
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]
```

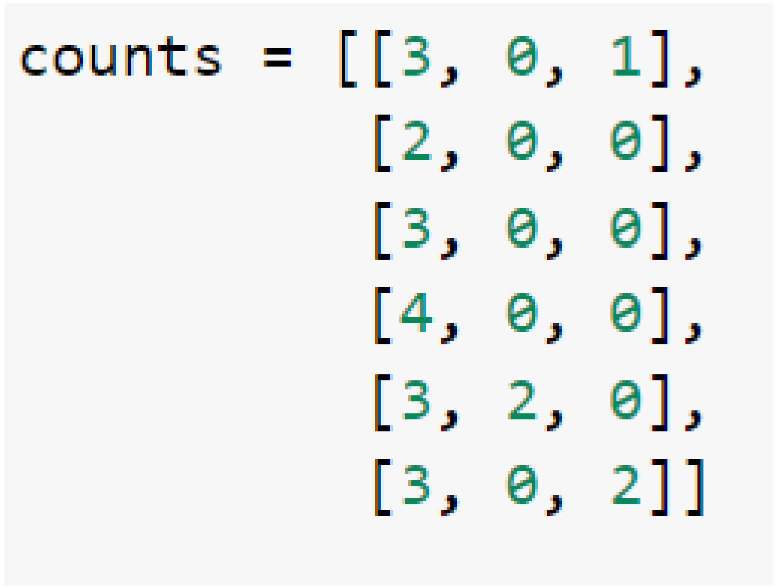
$$n = 6$$

$$\text{df}(t)_{\text{term1}} = 6$$

$$\text{idf}(t)_{\text{term1}} = \log \frac{n}{\text{df}(t)} + 1 = \log(1) + 1 = 1$$

$$\text{tf-idf}_{\text{term1}} = \text{tf} \times \text{idf} = 3 \times 1 = 3$$

TF-IDF



```
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]
```

Term:	Term:	Term:
Document:	Document:	Document:
Df(term 1):	Df(term 2):	Df(term 3):
Tf(term 1):	Tf(term 2):	Tf(term 3:
Tf-idf(term 1):	Tf-idf(term 2):	Tf-idf(term 3):

TF-IDF

```
counts = [[3, 0, 1],
          [2, 0, 0],
          [3, 0, 0],
          [4, 0, 0],
          [3, 2, 0],
          [3, 0, 2]]
```

$$n = 6$$

$$\text{df}(t)_{\text{term1}} = 6$$

$$\text{idf}(t)_{\text{term1}} = \log \frac{n}{\text{df}(t)} + 1 = \log(1) + 1 = 1$$

$$\text{tf-idf}_{\text{term1}} = \text{tf} \times \text{idf} = 3 \times 1 = 3$$

$$\text{tf-idf}_{\text{term2}} = 0 \times (\log(6/1) + 1) = 0$$

$$\text{tf-idf}_{\text{term3}} = 1 \times (\log(6/2) + 1) \approx 2.0986$$

$$\text{tf-idf}_{\text{raw}} = [3, 0, 2.0986]$$

$$\frac{[3,0,2.0986]}{\sqrt{(3^2+0^2+2.0986^2)}} = [0.819, 0, 0.573]$$

TF-IDF

Furthermore, the defaultparameter smooth_idf=True adds “1” to the numerator and denominator as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions:

TF-IDF

TF-IDF

- TF-IDF stands for term frequency-inverse document frequency.
- TF-IDF is used to evaluate how important a word is to a document in a collection or corpus.

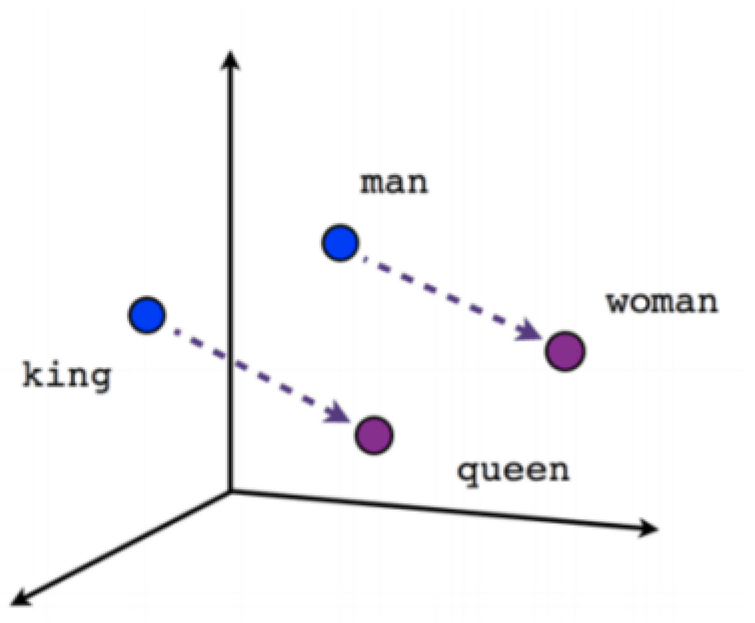
TF: Term Frequency, the number of times a term occurs in a document.

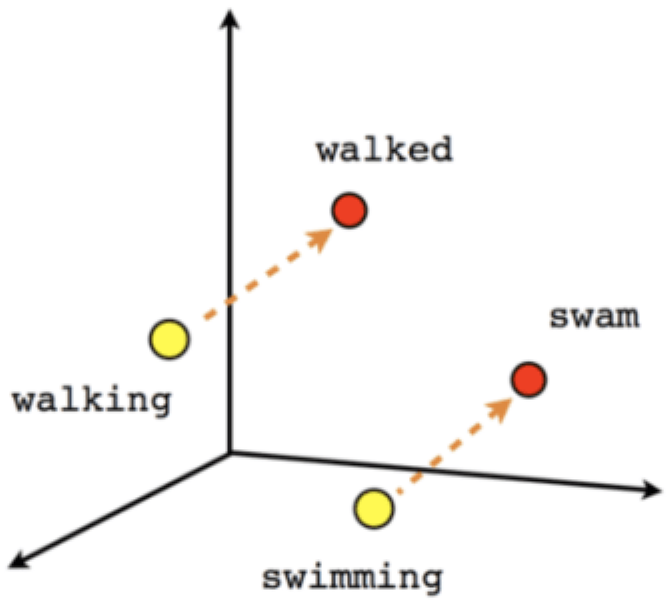
IDF: Inverse Document Frequency, which measures the important of a term.

$$\text{IDF} = \log\frac{1+n}{1+DF} + 1$$

TF-IDF = TF x IDF

Word Embeddings





Male-Female

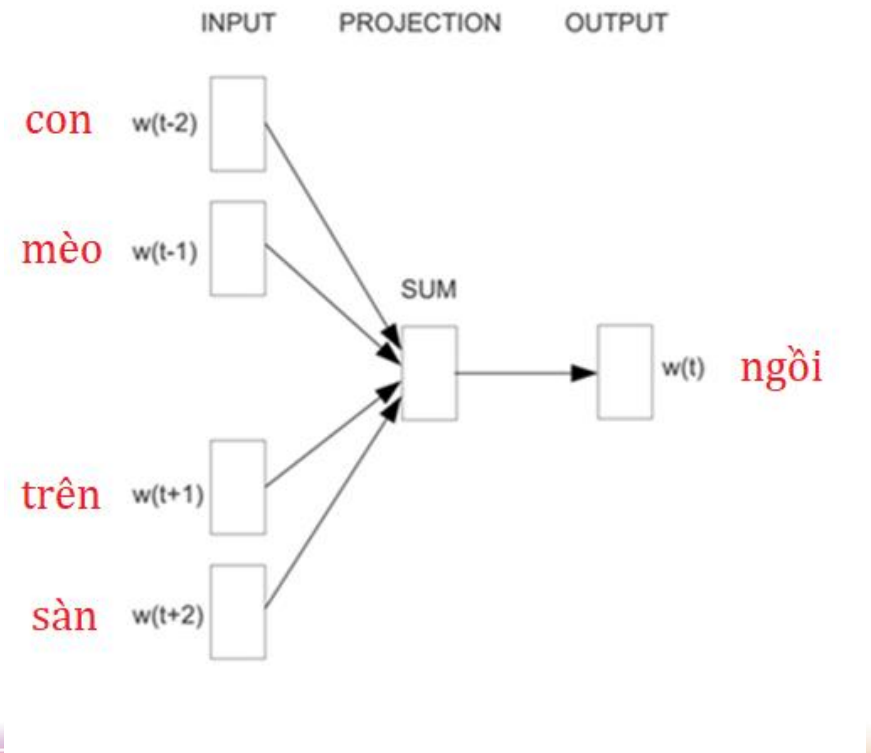
Verb tense

Word Embeddings

CBOW, Skip-gram

- Continuous Bag of Words (CBOW) (Tomas Mikolovet al., 2013)**:**

This model predict from the present context-based in large corpus volumes with a sliding window



Ex: **"Con mèo ngồi trên sàn"**

Sliding window = 2

Natural Language Processing

Word Embeddings

CBOW (Tomas Mikolovet al., 2013, 16840 cited – 15/07/2020)

Skip-gram (Tomas Mikolovet al., 2013, 16840 cited – 15/07/2020)

GloVe (Stanford NLP Research, 2014, 15159 cited – 15/07/2020)

FastText (Facebook AI Research 2016, 5773 cited – 15/07/2020)

BERT (Google AI, 2018: 7529 cited – 15/07/2020)

(Pre-training of Deep Bidirectional Transformers for Language Understanding)

Mid Term

- Crawl data from vnexpress (at least 5 topic, 40 documents/1 topic)
- Output:

- Pre processing (stopword, html,tokenize…)
- Extract TI-IDF feature
- Using SVM for training of document classification
- Evaluate by Accuracy