Векторные представления текста

- Зачем нам нужны векторные представления текста?
- Как преобразовать необработанный текст в вектор?

Векторы и векторные пространства

(Vector Space Model)

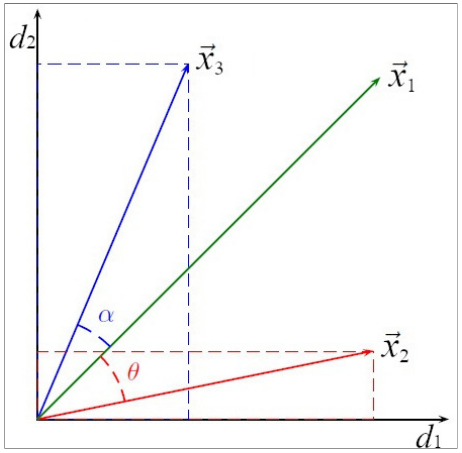
- Вектор $\mathbf{x}$ - это одномерный массив из k элементов (координат), который можно идентифицировать по индексу i.

$$\mathbf{x} = (x_1, x_2, \ldots, x_k)$$

- Набор из n векторов, представляющий собой матрицу X размера $n \times k$, также называется векторным пространством.

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{pmatrix}$$

Пример векторного пространства



$d_j(j \in 1, 2)$ - координаты, $\mathbf{x}_i$ - векторы.

К чему ближе $\mathbf{x}_1$, к $\mathbf{x}_2$ или к $\mathbf{x}_3$? Как это измерить?

Мера сходства

- Скалярное произведение.

$$\mathrm{dot}(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{x}_1 \cdot \mathbf{x}_2 = \mathbf{x}_1 \mathbf{x}_2^T = \sum_{i=1}^{d} x_{1,i} x_{2,i} = x_{1,1} x_{2,1} + \ldots + x_{1,d} x_{2,d}$$

- Косинусное сходство.

$$\mathrm{cosine}(\mathbf{x}_1, \mathbf{x}_2) = \cos\theta = \frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{|\mathbf{x}_1||\mathbf{x}_2|} = \frac{\sum_{i=1}^{d} x_{1,i} x_{2,i}}{\sqrt{\sum_{i=1}^{d} (x_{1,d})^2} \sqrt{\sum_{i=1}^{d} (x_{2,d})^2}}$$

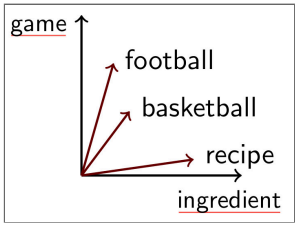
$$|\mathbf{x}| = \sqrt{\mathbf{x} \cdot \mathbf{x}} = \sqrt{x_1^2 + x_2^2 + \ldots + x_d^2}$$

Векторное пространство текста

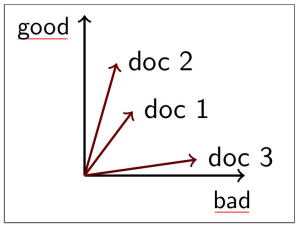
Что будет являться строками и столбцами для текстовых данных?

Векторное пространство текста

терм-терм (term-context)



терм-документ (bag-of-words)



Зачем нужно векторное представление текста?

- Например мы хотим узнать basketball ближе к football или recipe, т.е. вычислить семантическую близость слов (semantic similarity).
- Или хотим получить документы, соответствующие запросу (document retrieval).
- Или хотим сформировать признаковое описание наших текстовых данных, и передать его в алгоритмы машинного обучения.

Текстовые единицы

- токен (слово/term): последовательность из одного или нескольких символов, исключая символы разделители или N-грамма.
- документ (text sequence/snippet): предложение, абзац, раздел, глава, весь документ, поисковый запрос, сообщение в социальной сети и т.д.

Остается вопрос: как можно перейти от необработанного текста к вектору?

Предварительная обработка текста (нормализация)

- Замена чисел.
- Приведение к одному регистру.
- **Токенезация**.
- Удаление стоп-слов.
- **Лемматизация/стемминг**.

In [2]:

```
text = ''
data = []
with open(r'data/input.txt') as f:
    text = f.read()

small_text = text[3709:3709 + 237]
print(small_text)
```

```
Comes now Mr. Charles Goddard to one, Jack London, saying: "The time, the place, and the men a
re met; the moving pictures producers, the newspapers, and the capital, are ready: let us get
together." And we got. Result: "Hearts of Three."
```

Выше показана часть содержимого файла.

Предварительная обработка текста (регулярные выражения, regex)

Регулярные выражения - формальный язык, который используется для для поиска и осуществления манипуляций с подстроками в тексте.
Существует множество различных нотаций, мы будем использовать **POSIX-Extended Regular Expressions**.
Посмотрите состав библиотеки **re**.

Предварительная обработка текста (регулярные выражения, regex)

Простые цепочки

`man`	-	поиск точного соответствия.
`and`	-	аналогично.
`!`	-	и это поиск точного соответствия.

In [2]:

```
import re

# Так
finder = re.compile(r'one')
result = finder.search(small_text)
# Или так
result = re.search(r'one', small_text)

print(result)
```

```
<re.Match object; span=(33, 36), match='one'>
```

Предварительная обработка текста (регулярные выражения, regex)

Множества символов

`[abc]`	-	a, b или c.
`[0123456789]`	-	любая цифра.
`[tT]he`	-	подстроки the или The.

Используем `[` и `]` для объединения символов в **группы**.

In [3]:

```
import re

result = re.findall(r'm[ae]n', small_text)
print(result)
```

```
['men']
```

Предварительная обработка текста (**регулярные выражения**, regex)

Диапазоны

Не очень удобный подход `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]`, можно использовать **диапазоны**.

`[A-Z]`	-	любая прописная буква.
`[a-z]`	-	любая строчная буква.
`[0-9]`	-	любая цифра.

In [4]:

```
import re

result = re.findall(r'[A-Z][a-z][a-z]', small_text)
print(result)
```

```
['Com', 'Cha', 'God', 'Jac', 'Lon', 'The', 'And', 'Res', 'Hea', 'Thr']
```

Предварительная обработка текста (регулярные выражения, regex)

Отрицание

Символ ^ в диапазоне или группе является отрицанием.

`[^A-Z]`	-	любой символ кроме прописной буквы.
`[^Aa]`	-	любой символ кроме `A` и `a`.
`[a^]`	-	`a` или `^`.
`a^b`	-	точное соответствие `a^b`.

In [5]:

```
import re

result = re.findall(r'[^a-zA-Z]', small_text)
print(result)
```

```
[' ', ' ', '.', ' ', ' ', ' ', ' ', ' ', ',', ' ', ' ', ',', ' ', ':', ' ', '"', ' ', ',', ' ', '
', ',', ' ', ' ', ' ', ' ', ' ', ' ', ';', ' ', ' ', ' ', ' ', ' ', ',', ' ', ' ', ',', ' ', ' ',
',', ' ', ' ', ':', ' ', ' ', ' ', ' ', '.', '"', ' ', ' ', ' ', '.', ' ', ':', ' ', '"', ' ',
' ', '.', '"']
```

Предварительная обработка текста (регулярные выражения, regex)

Простые счетчики

`mai?n`	-	`?` после символа делает его необязательным (man или main).
`m.n`	-	`.` это любой символ (man, men, mbn, ...).
`da*`	-	повторение 0 или более раз (d, da, daa, ...).
`da+`	-	повторение 1 или более раз (da, daa, daaa, ...).
`[_a-zA-Z][_a-zA-Z0-9]*`	-	Какие это цепочки?

In [6]:

```
import re
result = re.findall(r'mai?n', small_text)
print(result)
result = re.findall(r'da*', small_text)
print(result)
result = re.findall(r'da+', small_text)
print(result)
```

```
[]
['d', 'da', 'd', 'd', 'd', 'd', 'd', 'd', 'd']
['da']
```

Предварительная обработка текста (регулярные выражения, regex)

Жадность алгоритмов

`".*?"`	-	повторение 0 или более раз lj (d, da, daa, ...).
`".+?"`	-	повторение 1 или более раз (da, daa, daaa, ...).

In [7]:

```
import re

test_text = r'"aaa"aa""aaa"'
result = re.findall(r'".*"', test_text)
print(result)
result = re.findall(r'".*?"', test_text)
print(result)
result = re.findall(r'".+"', test_text)
print(result)
result = re.findall(r'".+?"', test_text)
print(result)
```

```
['"aaa"aa""aaa"']
['"aaa"', '""']
['"aaa"aa""aaa"']
['"aaa"', '""aaa"']
```

Предварительная обработка текста (регулярные выражения, regex)

Якоря

привязывают символ к определенной позиции.

`^The`	-	`^` это начало строки.
`[0-9]$`	-	`$` это конец строки.
`\bthe\b`	-	`\b` граница слова (the, но не then, other и т.д.).
`\Bthe\B`	-	`\B` не является границей слова (other).

In [3]:

```
import re

result = re.findall(r'^[oO]', small_text)
print(result)
result = re.findall(r'\bthe\b', 'then the other')
print(result)
result = re.findall(r'\Bthe\B', 'then the other')
print(result)
```

```
[]
['the']
['the']
```

Предварительная обработка текста (регулярные выражения, regex)

Операторы дизъюнкции и группировки

`dog|cat`	-	`|` это дизъюнкция (dog или cat).
`dog(s|gy)`	-	`(` и `)` группировка элементов.

In [9]:

```
import re

result = re.findall(r'([dD]og(s|gy))', text)
print(result)
```

```
[('dogs', 's'), ('dogs', 's'), ('dogs', 's'), ('dogs', 's'), ('dogs', 's'), ('dogs', 's'), ('d
ogs', 's'), ('dogs', 's')]
```

Предварительная обработка текста (регулярные выражения, regex)

Сокращения

`\d`	`[0-9]`
`\D`	`[^0-9]`
`\w`	`[a-zA-Z0-9_]`
`\W`	`[^a-zA-Z0-9_]`
`\s`	`[ \r\t\n\f]`
`\S`	`[^ \r\t\n\f]`

In [10]:

```
import re

result = re.findall(r'\d\w*', text)
print(result)
```

```
['23', '1916', '27', '000', '50', '50', '50', '100', '300', '4', '9', '6', '9', '7', '1292',
'1292', '1292', '12', '1820']
```

Предварительная обработка текста (регулярные выражения, regex)

Счетчики

`{n}`	`точно n вхождений.`
`{n,m}`	`от n до m вхождений.`
`{n,}`	`как минимум n вхождений.`
`{,m}`	`максимум m вхождений.`

In [11]:

```
import re

result = re.findall(r'a{2,3}', text)
print(result)
```

```
['aa']
```

Предварительная обработка текста (регулярные выражения, regex)

Приоритеты операций (от высокого к низкому)

`()`	-	группировка
`* + ? {}`	-	счетчики
`abc ^ \b \B $`	-	последовательности и якоря.
`|`	-	дизъюнкция.

Предварительная обработка текста (регулярные выражения, regex)

Просмотр без перемещения указателя

`(?= pattern)`	-	истина, если дальше есть совпадение с pattern
`(?! pattern)`	-	истина, если дальше нет совпадения с pattern

In [12]:

```
import re

result = re.findall(r'x+(?=y)', 'xxy xxxy xf')
print(result)
result = re.findall(r'x+(?!y)', 'xxy xxxy xf')
print(result)
```

```
['xx', 'xxx']
['x', 'xx', 'x']
```

Предварительная обработка текста (регулярные выражения, regex)

Экранируемые символы

`\*`	-	символ *
`\.`	-	символ .
`\?`	-	символ ?
`\n`	-	перенос строки
`\t`	-	табуляция

Предварительная обработка текста:

Замена чисел

Необходимо заменить все числа на их текстовое представление.

In [13]:

```
import re
import inflect

small_text += "1, 12, 123."

infl = inflect.engine()
expr = re.compile(r'(\d+)')

small_text = expr.sub(lambda num: infl.number_to_words(num.group()),
                      small_text)

print(small_text)
```

```
Comes now Mr. Charles Goddard to one, Jack London, saying: "The time, the place, and the men a
re met; the moving pictures producers, the newspapers, and the capital, are ready: let us get
together." And we got. Result: "Hearts of Three."one, twelve, one hundred and twenty-three.
```

Предварительная обработка текста:

Токенезация (tokenisation)

Это процесс разбиения потока текстовых данных на слова, термы, предложения, символы или некоторые другие значимые элементы, называемые токенами.
Самый простой способ - использовать метод строк split.

In [14]:

```
tokens = small_text.split()
print(tokens)
```

```
['Comes', 'now', 'Mr.', 'Charles', 'Goddard', 'to', 'one,', 'Jack', 'London,', 'saying:', '"Th
e', 'time,', 'the', 'place,', 'and', 'the', 'men', 'are', 'met;', 'the', 'moving', 'pictures',
'producers,', 'the', 'newspapers,', 'and', 'the', 'capital,', 'are', 'ready:', 'let', 'us', 'g
et', 'together."', 'And', 'we', 'got.', 'Result:', '"Hearts', 'of', 'Three."one,', 'twelve,',
'one', 'hundred', 'and', 'twenty-three.']
```

Проблема наблюдается со знаками препинания.

Предварительная обработка текста:

Токенезация (tokenisation)

Можно использовать более продвинутый метод - конечные автоматы, но писать их не хочеться, поэтому воспользуемся регулярными выражениями.

In [15]:

```
import re

pattern = r'\W+'

tokens = re.split(pattern, small_text)
print(tokens)
```

```
['Comes', 'now', 'Mr', 'Charles', 'Goddard', 'to', 'one', 'Jack', 'London', 'saying', 'The',
'time', 'the', 'place', 'and', 'the', 'men', 'are', 'met', 'the', 'moving', 'pictures', 'produ
cers', 'the', 'newspapers', 'and', 'the', 'capital', 'are', 'ready', 'let', 'us', 'get', 'toge
ther', 'And', 'we', 'got', 'Result', 'Hearts', 'of', 'Three', 'one', 'twelve', 'one', 'hundre
d', 'and', 'twenty', 'three', '']
```

Здесь знаки препинания совсем исчезают.

Предварительная обработка текста:

Токенезация (tokenisation)

Но гораздо лучше пользоваться специализированными библиотеками. Например библиотекой NLTK (Natural Language Toolkit).

In [16]:

```
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r'\w+')
tokens = tokenizer.tokenize(small_text)
print(tokens)
```

```
['Comes', 'now', 'Mr', 'Charles', 'Goddard', 'to', 'one', 'Jack', 'London', 'saying', 'The',
'time', 'the', 'place', 'and', 'the', 'men', 'are', 'met', 'the', 'moving', 'pictures', 'produ
cers', 'the', 'newspapers', 'and', 'the', 'capital', 'are', 'ready', 'let', 'us', 'get', 'toge
ther', 'And', 'we', 'got', 'Result', 'Hearts', 'of', 'Three', 'one', 'twelve', 'one', 'hundre
d', 'and', 'twenty', 'three']
```

Хотя и здесь аналогичная ситуация.

Предварительная обработка текста:

Токенезация (tokenisation)

Попробуем специализированные методы.

In [17]:

```
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt', quiet=True)

tokens = word_tokenize(small_text)
print(tokens)
```

```
['Comes', 'now', 'Mr.', 'Charles', 'Goddard', 'to', 'one', ',', 'Jack', 'London', ',', 'sayin
g', ':', '``', 'The', 'time', ',', 'the', 'place', ',', 'and', 'the', 'men', 'are', 'met',
';', 'the', 'moving', 'pictures', 'producers', ',', 'the', 'newspapers', ',', 'and', 'the', 'c
apital', ',', 'are', 'ready', ':', 'let', 'us', 'get', 'together', '.', "''", 'And', 'we', 'go
t', '.', 'Result', ':', '``', 'Hearts', 'of', 'Three', '.', '``', 'one', ',', 'twelve', ',',
'one', 'hundred', 'and', 'twenty-three', '.']
```

Предварительная обработка текста:

Токенезация (tokenisation)

Можно производить токенезацию по предложениям.

In [18]:

```
import nltk
from nltk.tokenize import sent_tokenize
nltk.download('punkt', quiet=True)

tokens = sent_tokenize(small_text)
print(tokens)
```

```
['Comes now Mr. Charles Goddard to one, Jack London, saying: "The time, the place, and the men
are met; the moving pictures producers, the newspapers, and the capital, are ready: let us get
together."', 'And we got.', 'Result: "Hearts of Three.', '"one, twelve, one hundred and twenty
-three.']
```

Полный список методов токенезации и не только можно посмотреть **здесь**.

Корпус и словарь

- **Корпус** D – это совокупность текстов, собранная в единое целое по определённым, соответствующим конкретной исследовательской задаче, критериям и отражающая ту или иную сферу использования языка. Корпус должен обладать свойством репрезентативности. Чаще всего он содержит морфологическую/синтаксическую/ семантическую разметку (аннотации). Корпусы для русского языка:
    - ГИКРЯ (Генеральный Интернет-корпус Русского Языка) **http://www.webcorpora.ru**
    - OpenCorpora (Открытый корпус) **https://www.opencorpora.org**
    - НКРЯ (Национальный корпус русского языка) **https://ruscorpora.ru**
- **Словарь** $\mathcal{V}$ - это множество, которое содержит все k уникальных слов w_i из D:

$$\mathcal{V} = \{w_1, w_2, \ldots, w_k\}$$

Векторизация слов:

унитарное кодирование (one-hot encoding)

Является самым простым способом преобразования токенов в тензоры. Выполняется следующим образом:

1. каждый токен представляется бинарным вектором;
2. единица соответствует тому компоненту вектора, индекс которого совпадает с индексом заданного слова в словаре $\mathcal{V}$.

Поясним на примере: ***не хочет косой косить косой говорит коса коса***

- $\mathcal{V} = \{\text{не}, \text{хочет}, \text{косой}, \text{косить}, \text{говорит}, \text{коса}\}$.

  Порядок слов с словаре может быть иным.
- Размер словаря $|\mathcal{V}| = 6$

$$\text{косой} = [0, 0, 1, 0, 0, 0]$$
$$\text{косить} = [0, 0, 0, 1, 0, 0]$$

Векторизация слов:

унитарное кодирование (one-hot encoding)

Реализуем алгоритм унитарного кодирования

In [19]:

```
small_text = 'не хочет косой косить косой говорит коса коса'
tokens = small_text.split()

vocabulary = list(set(tokens))

vectors = []
for token in tokens:
    vector = list([int(token == word) for word in vocabulary])
    vectors.append(vector)

for word, vector in zip(tokens, vectors):
    print(word, vector, sep='\t')
```

```
не      [0, 0, 0, 1, 0, 0]
хочет   [0, 0, 0, 0, 1, 0]
косой   [0, 1, 0, 0, 0, 0]
косить  [0, 0, 1, 0, 0, 0]
косой   [0, 1, 0, 0, 0, 0]
говорит [1, 0, 0, 0, 0, 0]
коса    [0, 0, 0, 0, 0, 1]
коса    [0, 0, 0, 0, 0, 1]
```

Векторизация слов:

унитарное кодирование (one-hot encoding)

Унитарное кодирование посредством sklearn

In [20]:

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
import numpy as np

small_text = 'не хочет косой косить косой говорит коса коса'
tokens = small_text.split()
int_encoded = LabelEncoder().fit_transform(tokens)
onehot_encoder = OneHotEncoder(sparse_output=False)
onehot_encoded = onehot_encoder.fit_transform(int_encoded[:,np.newaxis])

for word, vector in zip(tokens, onehot_encoded):
    print(word, vector, sep='\t')
```

```
не      [0. 0. 0. 0. 1. 0.]
хочет   [0. 0. 0. 0. 0. 1.]
косой   [0. 0. 0. 1. 0. 0.]
косить  [0. 0. 1. 0. 0. 0.]
косой   [0. 0. 0. 1. 0. 0.]
говорит [1. 0. 0. 0. 0. 0.]
коса    [0. 1. 0. 0. 0. 0.]
коса    [0. 1. 0. 0. 0. 0.]
```

Векторизация слов:

унитарное кодирование (one-hot encoding)

В чем заключается проблема унитарного кодирования?

- "косой" и "косить" связанные по смыслу слова, но имея:

$$\text{косой} = [0, 0, 1, 0, 0, 0]$$
$$\text{косить} = [0, 0, 0, 1, 0, 0]$$

получаем:

$$\text{dot}(\mathbf{x_1}, \mathbf{x_2}) = 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 0 + 0 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 = 0$$

$$\text{cosine}(\mathbf{x_1}, \mathbf{x_2}) = \frac{\mathbf{x_1} \cdot \mathbf{x_2}}{|\mathbf{x_1}||\mathbf{x_2}|} = \frac{0}{1 \cdot 1} = 0$$

- Как сохранить информацию о контексте?

Экспресс тест

- На столе стоит порция серрадуры.
- Все любят вкусную серрадуру.
- Серрадура сладкая.
- Серрадура делается из печенья, сгущенного молока и взбитых сливок.

Серрадура - это португальский десерт.

Векторизация слов:

дистрибутивная гипотеза

заключается в том, что у слов, встречающихся в одном и том же контексте, есть тенденция иметь схожие значения (Zellig Sabbettai Harris, 1954)

Векторизация слов:

терм-терм матрица (word-word, term-context)

- Матрица X , $n \times m$, где $n = |\mathcal{V}|$ (целевые слова) и $m = |\mathcal{V}_c|$ (контекстные слова).
- Рассчитывается количество употреблений слова $x_i \in \mathcal{V}$ совместно со словом контекста $x_j \in \mathcal{V}_c$.
- Используется контекстное окно из $\pm k$ слов (слева/справа от x_i).
- Частоты вычисляются по огромному массиву документов.
- Обычно $\mathcal{V}$ и $\mathcal{V}_c$ совпадают, что приводит к квадратной матрице.

Векторизация слов: терм-терм матрица

Для примера возьмем Брауновский корпус (создан в 1960-е годы в Университете Брауна), который содержит 1 миллион словоупотреблений.

	aardvark	computer	data	pinch	result	sugar
apricot	0	0	0	1	0	1
pineapple	0	0	0	0	0	1
digital	0	2	0	0	1	0
information	0	0	3	0	2	0

- $\text{cosine}(\text{apricot}, \text{pineapple}) = 1$
- $\text{cosine}(\text{apricot}, \text{digital}) = 0$

Векторизация слов: терм-терм матрица

In [21]:

```
import nltk
from nltk.corpus import brown
nltk.download('brown', quiet=True)

words1 = ['apricot', 'pineapple', 'digital', 'information']
words2 = ['aardvark', 'computer', 'data', 'pinch', 'result', 'sugar']

cfd = nltk.ConditionalFreqDist((word1, word2)
                               for sent in brown.sents()
                               for word1 in words1 if word1 in sent
                               for word2 in words2 if word2 in sent)

cfd.tabulate(conditions=words1, samples=words2)
```

```
             aardvark computer   data    pinch   result   sugar
    apricot         0        0       0        1        0       1
  pineapple         0        0       0        0        0       1
    digital         0        2       0        0        1       0
information         0        0       3        0        2       0
```

Векторизация документов:

терм-документная матрица

(document-word, document-term matrix, bag-of-words)

- Матрица X, $|D| \times |\mathcal{V}|$, где строки - документы из корпуса D, колонки - слова из $|\mathcal{V}|$.
- Для каждого документа подсчитывается количество вхождений слов $w \in \mathcal{V}$.

	computer	data	pinch	result	sugar
doc 1	1	2	0	0	0
doc 2	0	1	0	2	0
doc 3	0	0	1	0	2
doc 4	0	0	4	0	1

- X можно получить путем конкатенации векторов-строк, которые получены путем суммирования всех терм-терм матриц документов по столбцам.

Проблематика векторизации документов и слов

- Наиболее часто в текстах встречаются артикли, местоимения, союзы и т.д., при этом они не информативны.

- Добавим слово **the** в рассмотренный выше пример (терм-терм матрица):

$$\mathcal{V} = [\text{aadvark}, \text{computer}, \text{data}, \text{pinch}, \text{result}, \text{sugar}, \text{the}]$$
$$\text{apricot} = x_1 = [0, 0, 0, 1, 0, 1, 30]$$
$$\text{digital} = x_2 = [0, 2, 1, 0, 1, 0, 45]$$
$$\text{cosine}(x_1, x_2) = \frac{30 \cdot 45}{\sqrt{902} \cdot \sqrt{2031}} \approx 0.999$$

- Как можно решить данную проблему?

Взвешенная терм-терм матрица:

коэффициент удаленности

Вес контекстного слова зависит от расстояния до целевого слова: чем дальше, тем меньше весовой коэффициент. Можно например для окна $k = 3$ использовать весовой вектор следующего вида:

$$\left[\frac{1}{3}, \frac{2}{3}, \frac{3}{3}, \text{apricot}, \frac{3}{3}, \frac{2}{3}, \frac{1}{3}\right]$$

Т.е. рассчитывать весовой коэффициент по следующей формуле: $\frac{k-i}{k}$, где $i \in \{0, 1, \ldots, k-1\}$ - удаленность от целевого слова.

Взвешенная терм-терм матрица:

Pointwise Mutual Information (PMI)

Мера того, как часто два слова w_i и w_j встречаются вместе по отношению к появлению независимо:

$$\mathrm{PMI}(w_i, w_j) = \log_2 \frac{P(w_i, w_j)}{P(w_i)P(w_j)} = \log_2 \frac{C(w_i, w_j)|D|}{C(w_i)C(w_j)}$$

$$P(w_i, w_j) = \frac{C(w_i, w_j)}{|D|}$$

$$P(w) = \frac{C(w)}{|D|}$$

где $C(\beta)$ - возвращает частоту β, $|D|$ - количество слов в корпусе D.

- чем выше PMI, тем информативнее пара w_i, w_j.
- отрицательные значения обычно игнорируются:

$$\mathrm{PPMI}(w_i, w_j) = \max(\mathrm{PMI}(w_i, w_j), 0)$$

Взвешенная терм-терм матрица:

Pointwise Mutual Information (PMI)

Реализуем алгоритм самостоятельно:

In [23]:

```
# Не оптимальный, но понятный пример
import math
from nltk.corpus import brown
nltk.download('brown', quiet=True)

tokens = brown.words(categories='science_fiction')
ngrams = [tuple(tokens[i:i + 2]) for i in range(len(tokens) - 1)]

ngram = ('99.1', 'percent')

pmi = math.log2((ngrams.count(ngram) * len(tokens)) /
                (tokens.count(ngram[0]) * tokens.count(ngram[1])))

print(ngram, pmi)
```

```
('99.1', 'percent') 13.820777301419087
```

Взвешенная терм-терм матрица:

Pointwise Mutual Information (PMI)

Реализуем алгоритм посредством NLTK:

In [24]:

```
import nltk
from nltk.corpus import brown
nltk.download('brown', quiet=True)

tokens = brown.words(categories='science_fiction')

bigram_measures = nltk.BigramAssocMeasures()

finder = nltk.BigramCollocationFinder.from_words(tokens)
finder.score_ngrams(bigram_measures.pmi)[:1]
```

Out[24]:

```
[(('99.1', 'percent'), 13.820777301419087)]
```

Взвешенная терм-документная матрица:

TF-IDF (TF - term frequency, IDF - inverse document frequency)

Это показатель, который равен произведению двух чисел: TF и IDF. Используется для оценки важности слова в контексте документа, являющегося частью коллекции.
TF - равно отношению числа вхождений слова в документ к общему количеству слов в документе.

$$\mathrm{TF}(w,d) = \frac{f_{w,d}}{\sum_{w' \in d} f_{w',d}},$$

где $f_{w,d}$ - число вхождений слова w в документ, а $\sum_{w' \in d} f_{w',d}$ - общее число слов в документе.

Взвешенная терм-документная матрица: TF-IDF

IDF - инверсия частоты, с которой встречается слово в коллекции документов. Чем больше таких документов, тем меньше IDF.

$$\mathrm{IDF}(w, D) = \log \frac{|D|}{|\{\, d_i \in D \mid w \in d_i \,\}|},$$

где $|D|$ - число документов в коллекции; $|\{\, d_i \in D \mid w \in d_i \,\}|$ - число документов из коллекции D, в которых встречается w.

Взвешенная терм-документная матрица: TF-IDF

$$\text{TF-IDF}(w, d, D) = \text{TF}(w, d) \cdot \text{IDF}(w, D)$$

TF-IDF имеет высокое значение для тех слов, которые много раз встречаются в документе, и редко встречаются в остальных документах.

Проблема размерности

- Вышеупомянутые представления (для слов и документов) часто работают хорошо, но:
    - высокая размерность: размер словаря может достигать миллиона слов!
    - матрицы очень разреженные:
        - слова находятся с небольшим количеством слов по соседству;
        - документы содержат небольшую часть словарного запаса.
- Решение: уменьшение размерности!

Сингулярное разложение

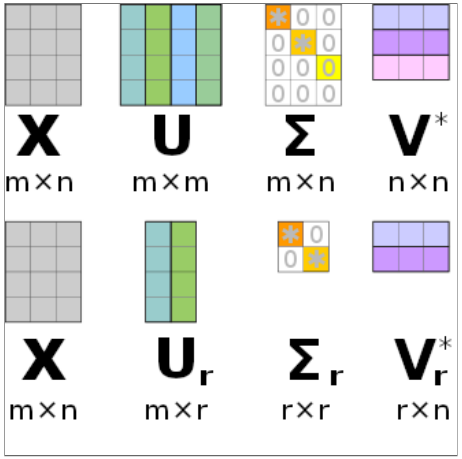
Метод поиска наиболее важных признаков в данных, т.е. тех признаков, по которым данные варьируются больше всего, путем разложения матрицы на скрытые факторы.
В задаче понижения размерности интересно усеченное разложение:

$$X^{m \times n} \approx U_{m \times k} \times \Sigma_{k \times k} \times V^*_{k \times n}$$

Данные метод, используя избыточность, хорошо аппроксимирует данные и удаляет "шумы". Такое приближение является наилучшим низкоранговым приближением с точки зрения средне-квадратичного отклонения.

Сингулярное разложение

терм-терм матрица



Сингулярное разложение:

терм-документная матрица

- Также называется латентно-семантический анализ (Latent Semantic Analysis, LSA).
- $U_{n \times k}$ векторное представление документов.
- $V_{k \times m}$ векторное представление слов.
- Вы можете получить векторное представление u_{new} для нового документа x_{new}, преобразуя его вектор:

$$u_{\text{new}} = x_{\text{new}} v_k^T$$

Терм-терм матрица

- Можно оперировать парами слов, учитывая их смысловое сходство.
- Заменить слово в предложении, не меняя его значения.
- Находить аналогии: Москва для России то же, что Рим для...?
- Использовать для сокращения времени решения задачи (bag of word vectors вместо bag of words)

Терм-документная матрица

- Сходство документов.
- Поиск информации.
- Классификация текста.
- Обнаружение плагиата и т.д.
- Порядок слов игнорируется (но язык последовательный).