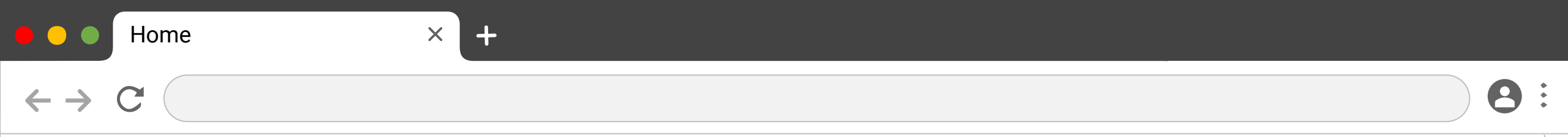


Информационный поиск

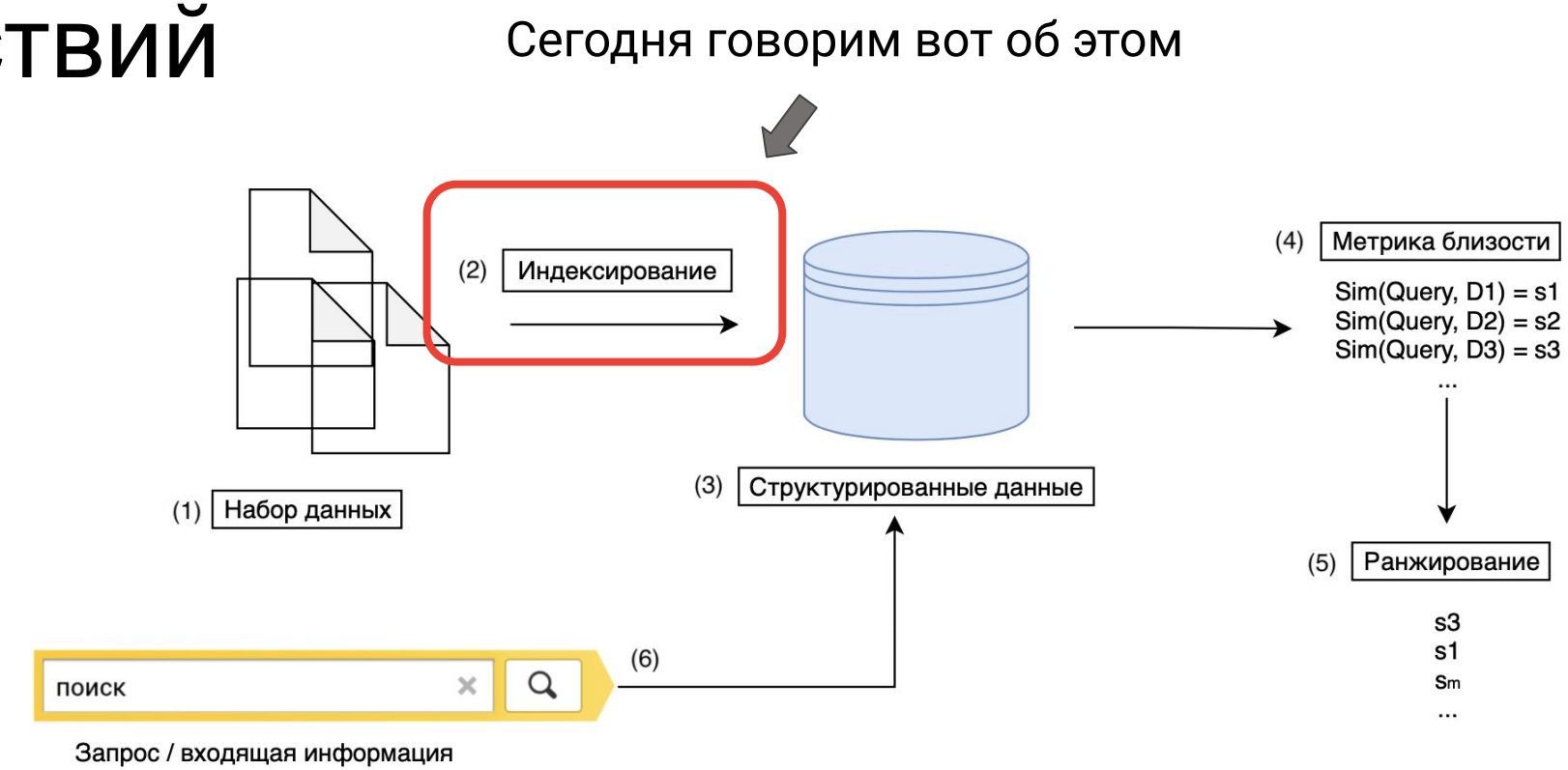
Лекция 2. Векторизация

Ника Зыкова
2024/09/10

Home | Recent Tabs | Current Page

You can type something here...

Последовательность действий



С чего начинаем?

У нас все еще есть корпус, состоящий из нескольких текстов:

doc_1 = Буря мглою небо кроет

doc_2 = Вихри снежные крутя

doc_3 = То, как зверь, она завоет

doc_4 = То заплачет, как дитя

И прямой индекс для него.

Документ	Списко слов
doc_1	буря, кроет, мглою, небо
doc_2	вихри, крутя, снежные
doc_3	завоет, зверь, как, она, то
doc_4	дитя, заплачет, как, то

Обратный индекс

Но с прямым работать не так удобно, поэтому будем использовать обратный. Сначала в виде словаря, потом в виде матрицы. Обычно хранят еще частоту встречаемости, но в нашем случае это не актуально.

```
{
        "буря": [
                "doc_1"
        ],
        "то": [
                "doc_3",
                "doc_4"
        ],
        "как": [
                "doc_3",
                "doc_4"
        ],
  ...
}
```

	буря	мглою	небо	кроет	вихри	снежные	крутя	...
doc_1	1	1	1	1	0	0	0	
doc_2	0	0	0	0	1	1	1	
doc_3	0	0	0	0	0	0	0	
doc_4	0	0	0	0	0	0	0	

Входные данные

Корпус = {

doc_1: Буря мглою небо кроет,

doc_2: Вихри снежные крутя,

doc_3: То, как зверь, она завоет,

doc_4: То заплачет, как дитя

}

Запрос = "буря заплачет над небом, над небом"

Метрика = количество общих слов в запросе и документе

Задача: найти документ, больше всего подходящий под запрос

Индекс по корпусу:

```
{
        "буря": [
                "doc_1"
        ],
        "то": [
                "doc_3",
                "doc_4"
        ],
        "как": [
                "doc_3",
                "doc_4"
        ],
  ...
}
```

Home | Recent Tabs | Current Page

You can type something here...

Порядок поиска

Что нужно получить: для каждого документа нужно подсчитать степень его релевантности запросу, потом отсортировать документу в соответствии с полученными значениями.

Для этого:

1. идем по документам в корпусе
2. считаем метрику для пары (запрос, документ)
3. сохраняем полученное значение
4. сортируем документы по значению метрики

В чем здесь проблема?

Используем преимущества индекса

В предыдущем варианте поиска мы никак не учитывали тот факт, что у нас есть обратный индекс (с которым описанный алгоритм работает очень плохо).

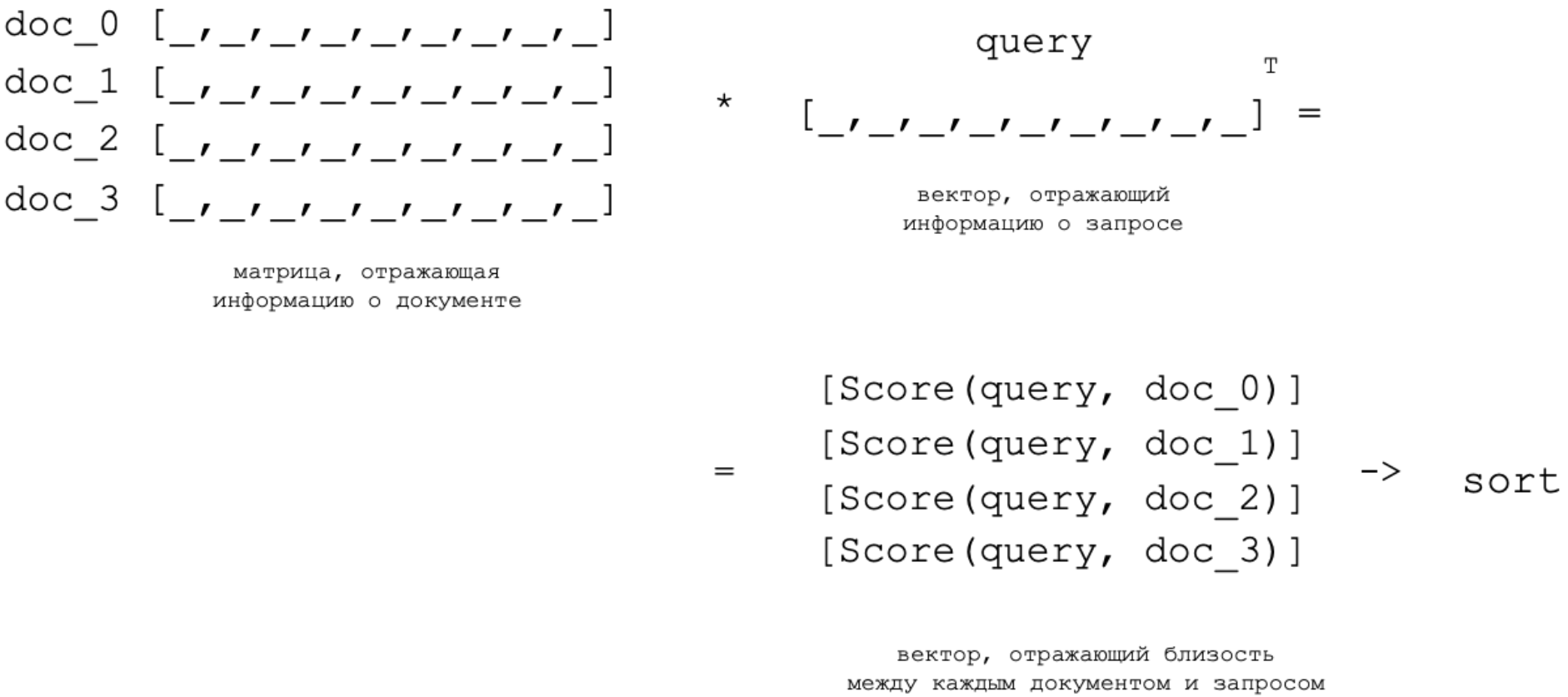
В реальной ситуации у нас тысячи документов и сотни тысяч слов. Хотим ли мы перебирать все?

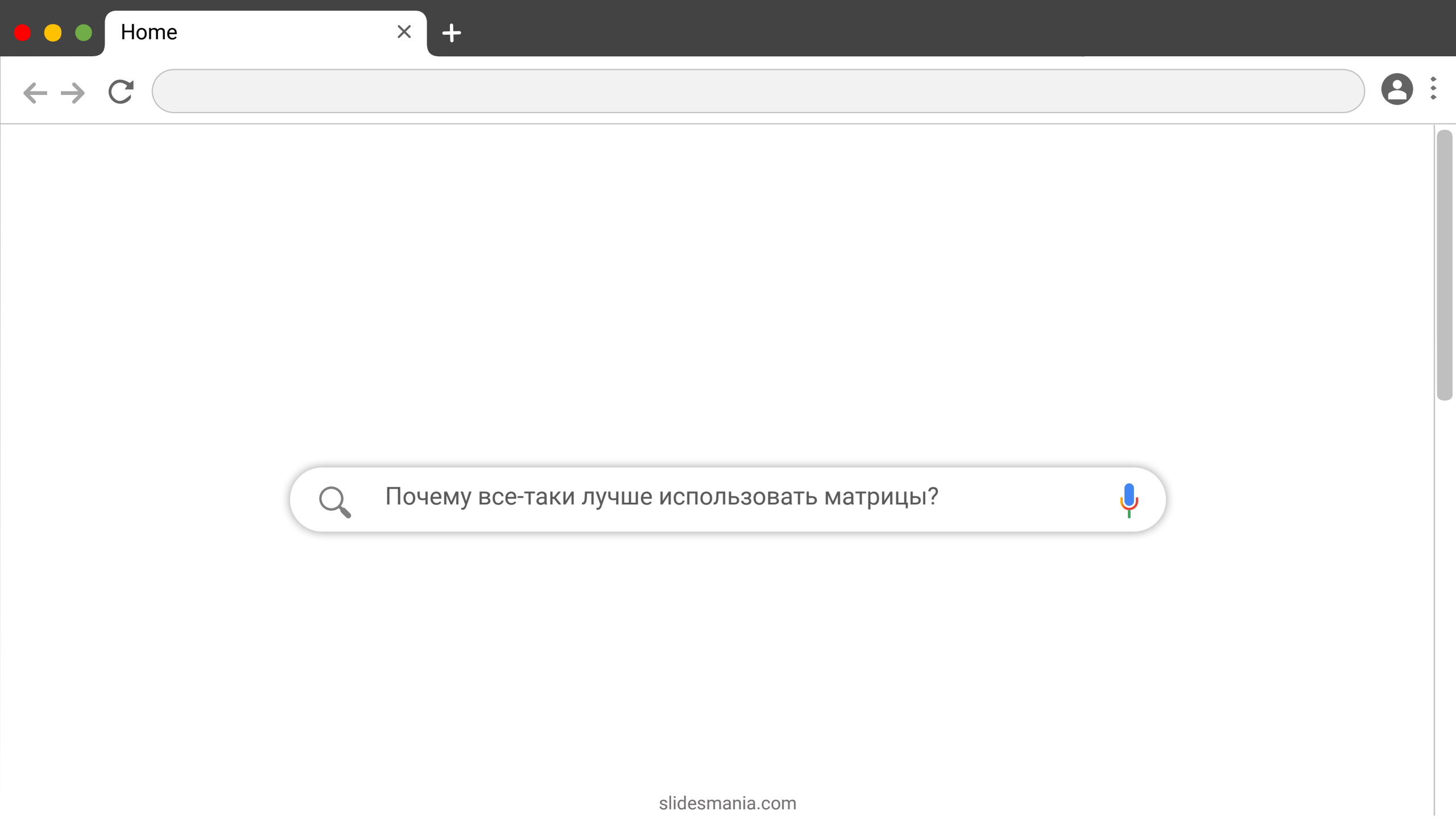
Новый алгоритм:

1. идем по уникальным словам в запросе
2. для документов, где встретилось слово, плюсуем метрику
3. финализируем подсчет метрики (например, нормализуем ее)
4. сохраняем полученное значение
5. сортируем документы по значению метрики

Почему такой алгоритм выгоднее?

Векторная постановка задачи





Почему матрица?

Это даст нам:

- более простое и лаконичное решение за счет перехода от алгоритма к математике (это не всегда так работает, но часто)
- буст по скорости поиска во много раз за счет скорости операций над матрицами

А какой все-таки у нас индекс: прямой или обратный?

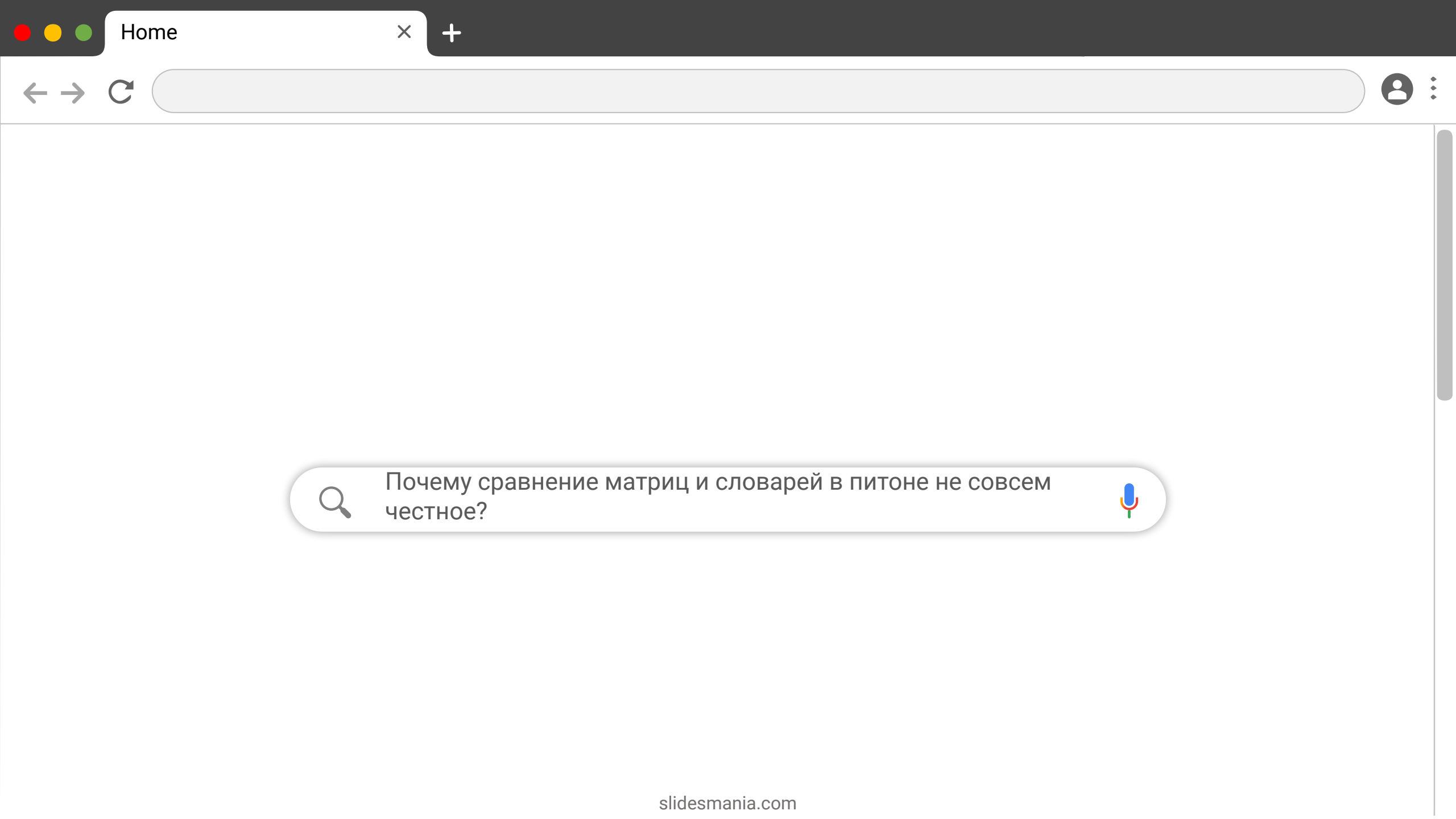
	буря	мглою	небо	кроет	вихри	снежные	крутя	...
doc_1	1	1	1	1	0	0	0	
doc_2	0	0	0	0	1	1	1	
doc_3	0	0	0	0	0	0	0	
doc_4	0	0	0	0	0	0	0	

Легким движением руки...

Прямой индекс превращается в обратный и наоборот. Ведь если мы говорим о матрице, то это изменение - лишь вопрос транспонирования.

Еще часто говорят о document-term (DT) и term-document (TD) матрицах, где первое слово обозначает информацию в строках, а второе - в столбцах.

	буря	мглою	небо	кроет	вихри	снежные	крутя	...
doc_1	1	1	1	1	0	0	0	
doc_2	0	0	0	0	1	1	1	
doc_3	0	0	0	0	0	0	0	
doc_4	0	0	0	0	0	0	0	

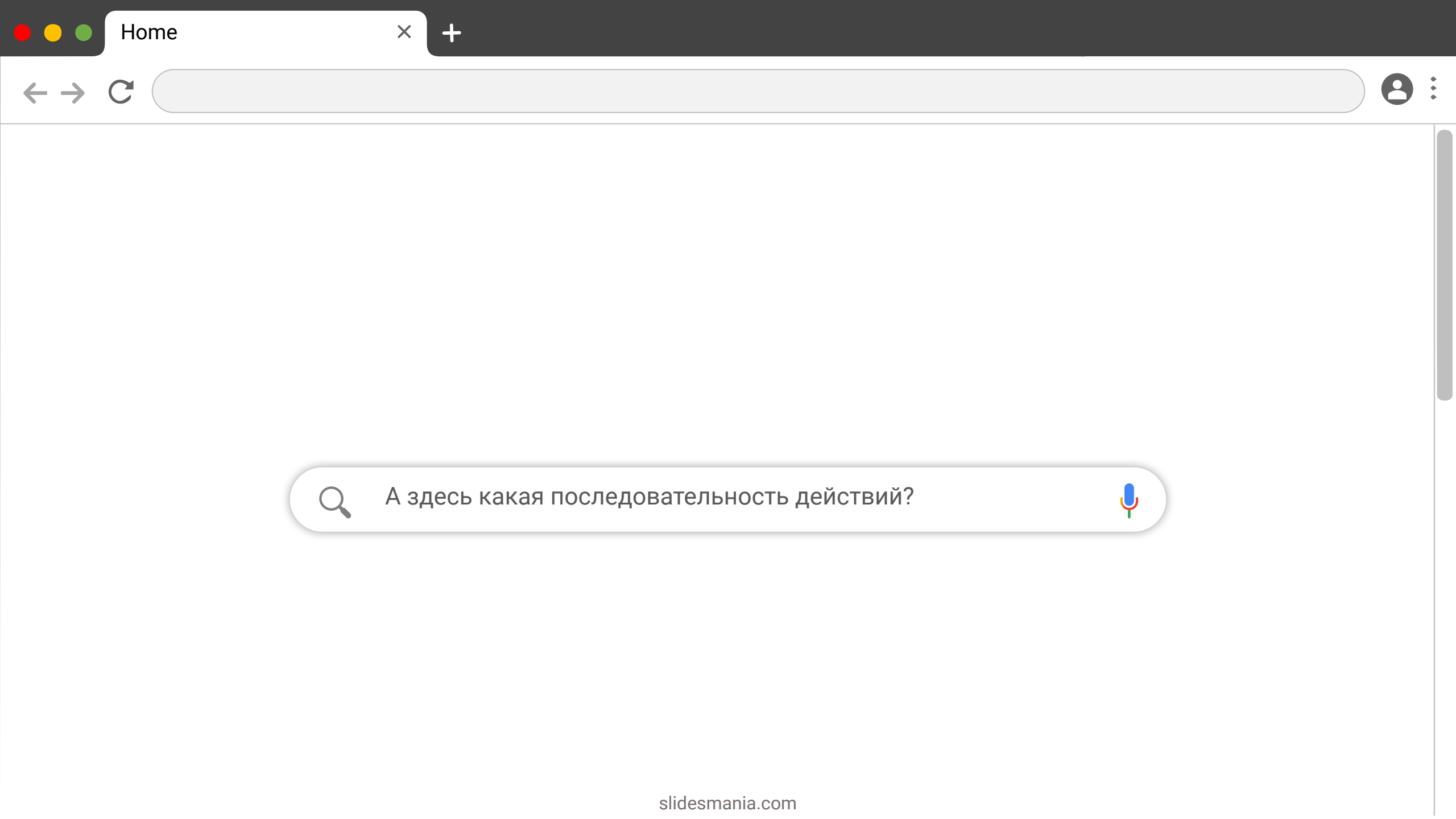


Что может быть в ячейках?

Значения элементов матрицы могут быть различными, например:

- бинарный показатель есть / нет в документе
- какой-то статистический показатель, например, количество в документе
- TF-IDF или BM-25
- и другие показатели значимости слова в документе

	буря	мглою	небо	кроет	вихри	снежные	крутя	...
doc_1	1	1	1	1	0	0	0	
doc_2	0	0	0	0	1	1	1	
doc_3	0	0	0	0	0	0	0	
doc_4	0	0	0	0	0	0	0	



Порядок поиска с матрицей

Нужно все то же самое: оценка релевантности каждого документа запросу.

Для этого:

1. превращаем запрос в вектор тем же способом, что и документы (в нашем случае - считаем частоту слов)
2. умножаем DC матрицу на вектор запроса, получаем метрики сразу для всех документов
3. сохраняем полученное значение
4. сортируем документы по значению метрики

А в чем здесь может быть проблема?

Сравнение

	СЛОВАРЬ	МАТРИЦА
Простота обработки	0	1
Скорость поиска	0	1
Прозрачность логики	0	1

Только это не совсем честная табличка. Почему?

Сравнение (если честно)

	СЛОВАРЬ	МАТРИЦА
Простота обработки	0	1
Скорость поиска	0	1
Прозрачность логики	0	1
Разреженные данные	1	0
Человекочитаемость	1	0

А вот теперь все не так однозначно. Вообще, на практике чаще всего используют вариант матриц, где учтена проблема с разреженными данными: sparse матрицы.

Промежуточное итого

Наша идея поиска – умножение матрицы на вектор для получения близости между запросом и документами из корпуса

Матрица – это проиндексированная коллекция документов

Вектор – это проиндексированный запрос

Результат - ранжированные по убыванию релевантности запросу документы

Векторизация документов

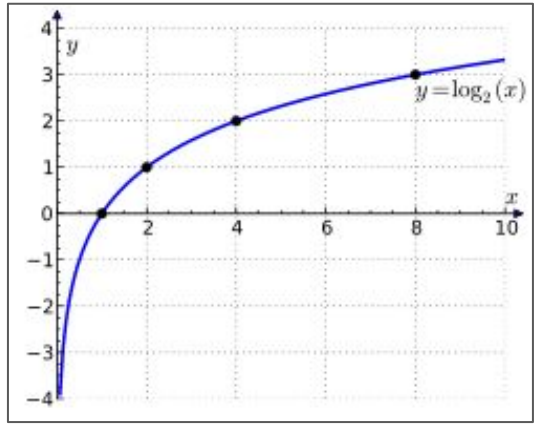
Как уже было сказано, способов множество:

- бинарное кодирование (есть/нет) - OneHotEncoder (sklearn, но так никто не делает)
- кодирование по частоте - CountVectorizer (sklearn)
- TF-IDF - TfidfVectorizer (sklearn)
- BM25 - библиотека rank-bm25
- word2vec - gensim (чаще всего)
- bert (и его родственники) - pytorch или HuggingFace (чаще всего)
- и тд

Вспомним Tf-Idf

$$Metric_{x,y} = tf_{x,y} \times \log(\frac{N}{df_x})$$

где x - слово, y - документ



- В чем общая идея?
- Какие части и что означают?
- Почему логарифм?

Формула BM25

$$bm25(Query, Doc) = \sum_{i=1}^{n} IDF \cdot \frac{TF \cdot (k+1)}{TF + k \cdot (1 - b + b \cdot \frac{l(d)}{avgdl})}$$

Объясним каждую компоненту:

- n - количество слов в запросе
- IDF - обратная документная частота слова q_i из запроса Query
- TF - частота слова q_i в документе Doc
- k - магическая константа
- b - магическая константа
- l(d) - количество слов в документе Doc
- avgdl - константа = средняя длина документа в корпусе

Части формулы

Все, что не зависит от запроса, нужно посчитать заранее. Это даст более оптимальное время исполнения поиска.

Не зависит от запроса:

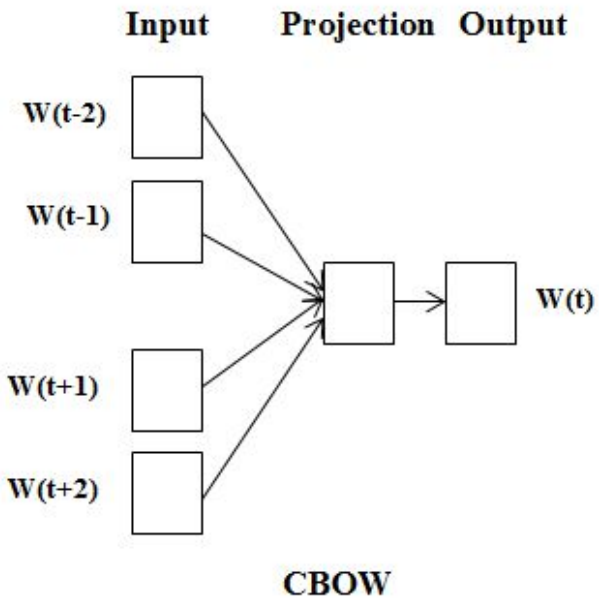
- N – кол-во документов в корпусе
- k = 2
- b = 0.75
- l(d) – кол-во слов в документе Doc
- avgdl – средняя длина документа в корпусе

Зависит от запроса:

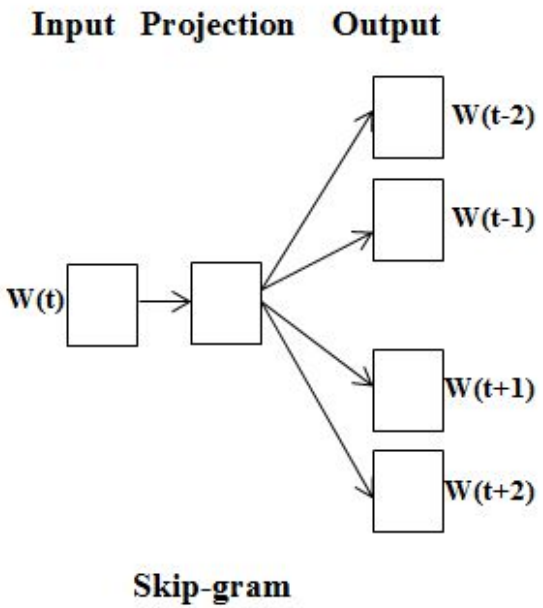
- TF(q_i, Doc) – частота q_i в Doc
- n(q_i) – кол-во доков, где есть q_i

Word2Vec

Учим на две задачи: восстанавливать слово по контексту (CBOW) и контекст по слову (skipgram). Плюс negative sampling.
Какие есть минусы у word2vec?



CBOW



Skip-gram

Чем плох word2vec?

- Плохо понимает разницу между синонимами и антонимами
  - Как мы уже видели, антонимы часто оказываются ближайшими друг для друга, что плохо в общем случае. Хотелось бы решить эту проблему
- Омонимы - это один и тот же вектор
  - Так как у word2vec для каждого слова есть всего один вектор, то для слов зАмок и замОк мы получим одно и то же, что однозначно неверно
- Не учитывает сочетаемость слов
  - Мы не можем учесть смысл слова в контексте, из-за чего модель не понимает идиомы, метафору и т.д.
- Не умеет работать с незнакомыми словами
  - Для слов, которые отсутствуют в словаре модели, у нас просто не будет вектора

Что еще?

- Можно использовать модели, близкие к word2vec:
  - Navec
  - FastText
  - Glove
- RNN и LSTM модели: ELMo and others
- Transformers модели:
  - BERT
  - RoBERTa, ALBERT, DistilBERT
  - ELECTRA

GloVe

- Минимизируем разницу между произведением векторов слов и логарифмом вероятности их совместного появления
- Учитываем частоту совместной встречаемости напрямую
- Не нейросеть (!), но оптимизируется похожими методами (стохастический градиентный спуск)
- Обгоняет word2vec на многих бенчмарках (но меньше open source моделей)

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Navec

- На основе GloVe
- Значительно меньше по занимаемой памяти, чем word2vec от RusVectores
- Не нужны части речи
- По опыту, знает больше слов из современного лексикона
- Вот тут можно почитать: https://natasha.github.io/navec/ (и табличка тоже оттуда)

	Качество	Размер модели, МБ	Размер словаря, ×10³
Navec	0.719	50.6	500
RusVectores	0.638–0.726	220.6–290.7	189–249

Home | Recent Tabs | Current Page

You can type something here...

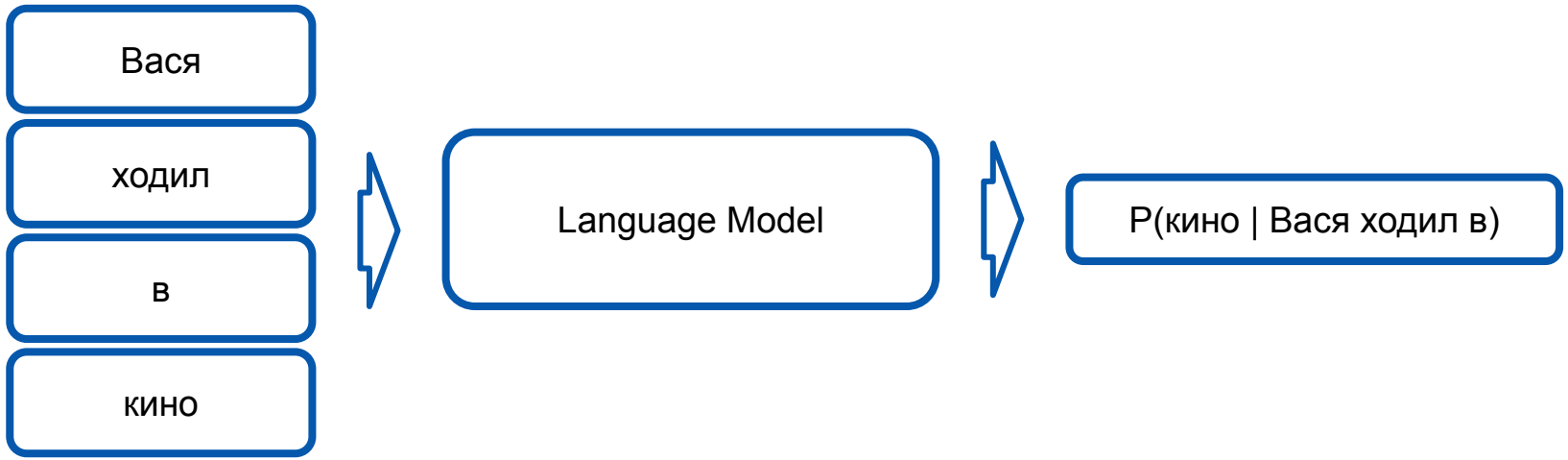
FastText

- Использует все задачи word2vec: CBOW, Skipgram, negative sampling
- К word2vec добавлена модель символьных n-грамм: каждое слово - несколько цепочек символов заданной длины (с наложением), вектор слова – сумма всех его n-грамм.
- Умеет работать с очень редкими и незнакомыми словами
- Более устойчив к опечаткам (как следствие к предыдущему)

Языковая модель

Языковая модель - это модель, способная оценить вероятность конкретной конструкции в языке.

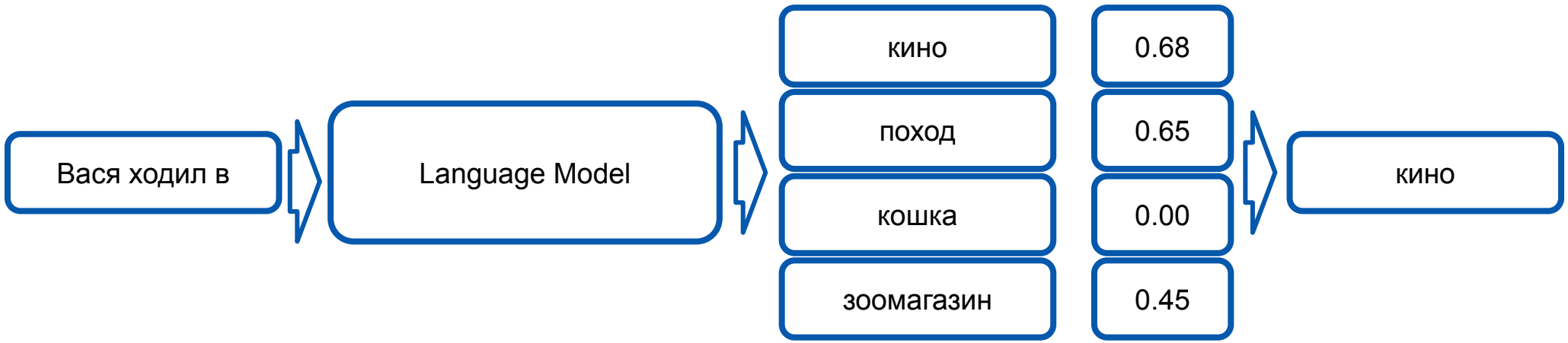
Чаще всего это означает, что по последовательности слов она может выдать вероятность последнего с учетом всех предыдущих.



Языковое моделирование

Модель учат выдавать наибольшую вероятность для последующего слова.

Требует много времени и данных, так как модель имеет только половину контекста (левую), но является единственным вариантом для генеративных моделей.



RNN and etc

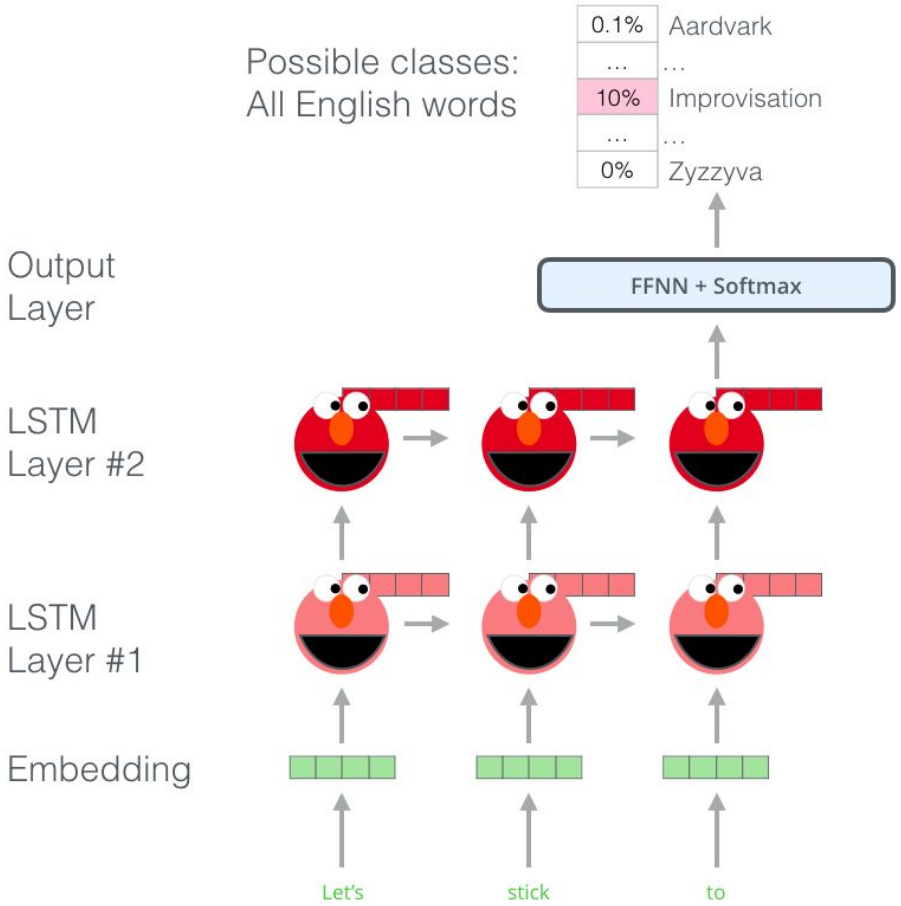
- RNN: помнит только ограниченный предшествующий контекст -> плохо работает с большими текстами (длинными предложениями)
- LSTM: усложненная работа с памятью, модель умеет динамически забывать/помнить -> более длинные тексты
- Attention Models: попытки реализовать внимание до Transfrormers

Общая проблема: таких моделей почти нет в свободном доступе, так как они проиграли BERT & Co. Но если обучать свою, то это хороший компромисс между точностью и сложностью.

Elmo

- Есть статический эмбеддинг (он хранится как часть модели) и есть итоговый, который получается на его основе
- Обучается на задачу **языкового моделирования** (генерации)
- Но Bi-LSTM - учитывается контекст справа и слева

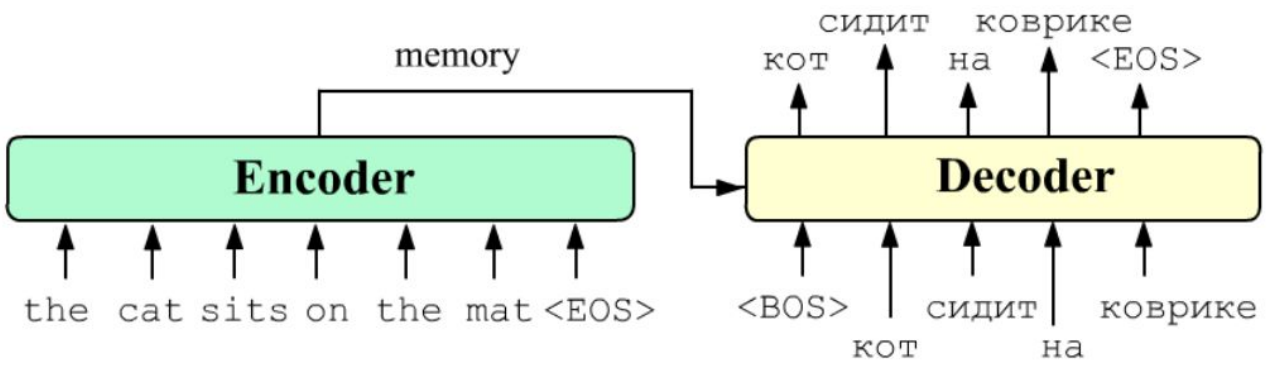
Неплохая модель, но с популярностью трансформеров используется редко.

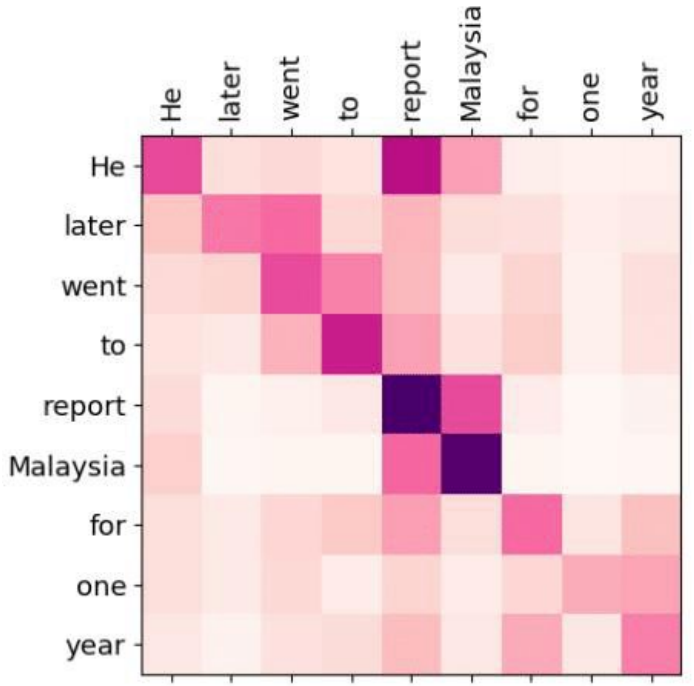


Transformers

В 2017 году Google представил архитектуру Transformers и ее механизм внимания.

Классическая архитектура Трансформера подразумевает наличие энкодера и декодера, но на практике встречаются любые комбинации: только энкодер, только декодер или





Виды моделей

Encoder

- BERT
- RoBERTa
- ALBERT
- ELECTRA

Decoder

- GPT1, 2, 3, 4
- ChatGPT
- PaLM
- LLaMA

Encoder-Decoder

- T5
- Bart
- Pegasus
- ProphetNet

What do BERT, RoBERTa, ALBERT, SpanBERT, DistilBERT, SesameBERT, SemBERT, SciBERT, BioBERT, MobileBERT, TinyBERT and CamemBERT all have in common? And I'm not looking for the answer "BERT" 😉.

Итого по способам векторизации

Критерий сравнения	TF-IDF, BM25	Word2Vec	FastText, GloVe, Navec	Elmo, other RNN, LSTM	BERT & Co.
Качество векторизации					
Скорость работы					
Доступность готовых решений					
Простота обучения					
Возможность дообучения					

Итого по способам векторизации

Критерий сравнения	TF-IDF, BM25	Word2Vec	FastText, GloVe, Navec	Elmo, other RNN, LSTM	BERT & Co.
Качество векторизации				+-	+
Скорость работы	+	+	+-		
Доступность готовых решений		+	+-		+
Простота обучения	+	+			
Возможность дообучения				+	+