



Информационный поиск

Высшая школа цифровой культуры

Университет ИТМО

dc@itmo.ru

Оглавление

Информационный поиск	3
Закон Цифпа	4
Мешок слов	5
Мешок n-грамм	6
Веб поиск	6
Сходство Жаккара	7
Инвертированные индексы	8
Ранжирование	9
VSM	10
TF-IDF	11
Другие задачи в поиске	15
Как сделать свой веб-поиск	16

Информационный поиск

В прошлой лекции мы поговорили о том, что такое обработка естественного языка, окинули взглядом её историю и узнали, как нормализовывать слова, чтобы готовить тексты для дальнейшей более содержательной работы с ними.

В этом занятии мы рассмотрим два приложения обработки естественного языка.

При работе с большим набором текстов так или иначе приходится определять или использовать понятие схожести текстов.

Допустим, мы хотим находить тексты близкой тематики или описывающие одинаковые явления и так далее. Для ЭВМ текст -- ничто иное, как строка, последовательность символов, но мы уже знаем, что текст как строку можно токенизировать и, кроме того, привести все слова к словарным формам с помощью лемматизатора.

Итак, перед нами уже не просто строка, а последовательность токенов. Какие тексты похожи? На этот вопрос для разных задач можно ответить по-разному. Начнём с простого подхода. Ясно, что если тексты более-менее об одном и том же, то множества составляющих их слов должны пересекаться; чем больше общих слов -- тем сильнее похожи.

Всегда ли это так?

Вы, конечно, уже вспомнили, что об одном и том же можно говорить разными словами и никто не отменял наличие синонимов -- слов, которые пишутся по-разному, но имеют одинаковый или похожий смысл. Да, это важная проблема, и с ней работают по-разному в зависимости от задачи. Пока на этом мы останавливаться не будем, но отметим, что если тексты достаточно большие, то -- да -- самые простые подходы работают подчас неплохо.

Рассмотрим три текста, скопированные нами с ресурса “N+1”:

- 1) **Астрофизики** воспользовались оценкой реальных размеров гравитационных линз для уточнения постоянной Хаббла, которая описывает скорость расширения **Вселенной**.
- 2) На выход алгоритм выдает не *самый* вероятный диагноз, а три лучших попадания из 26 возможных, выдавая им свой вес.
- 3) Кроме того, **астрофизикам** удалось обнаружить пульсар с *самой* маленькой орбитой и *самый* яркий пульсар во **Вселенной**.

Ясно, что первый и третий тексты об исследованиях космоса, а второй -- нет. Мы, люди, это поняли по терминам и некоторым опорным словам: “астрофизики”, “гравитационный”, “пульсар”, “диагноз” и так далее. Нужно сделать так, чтобы это было ясно и машине.

Тексты очень маленькие и пример подобран искусственно. Но всё же мы здесь видим, что наш подход, в целом, работает. Первый и третий тексты пересекаются по двум словам, причём это как раз слова, типичные для их тематики. Со вторым текстом первый не пересекается вовсе. А вот второй и третий пересекаются по слову “самый”.

Даже у самых коротких текстов на совсем разные темы могут быть общие слова. Как правило, местоимения, числительные, предлоги, союзы, местоименные наречия, вспомогательные глаголы и так далее. То есть -- служебные части речи и просто очень частотные слова. Когда мы сравниваем тексты как множества, они почти всегда попадают в пересечение, и этим “вредят” нам, так как ничего не говорят о содержательном сходстве. Их называют **стоп-словами** или **шумовыми словами** и часто просто исключают из рассмотрения.

Пришло время остановиться и ознакомиться с одним важным законом.

Закон Цифпа

Закон Ципфа -- это не строгое правило, а эмпирическое наблюдение, популяризованное Джорджем Кингсли Ципфом и несколько ранее подмеченное немецким физиком Феликсом Ауэрбахом и французским стенографистом Жаном-Батистом Эступом.

$$\#word \sim \frac{1}{freq_{word}}$$

Суть закона: если мы возьмём большой набор текстов на любом естественном языке, подсчитаем частоты каждого слова и отсортируем их по этой частоте по убыванию, то окажется, что номер слова в списке обратно пропорционален его частоте. Если мы отложим по вертикальной оси частоты, а слова -- вдоль горизонтальной, то мы почти всегда для любого языка увидим похожую на эту картинку: На этом графике -- первые по частоте слова русской википедии. Если прологарифмировать значение частоты и номер слова в списке, то мы увидим вот такой график: Здесь ломаные были построены для каждого из языков в легенде графика. Частоты получены также по википедии.

Думаю, вы согласитесь, что такой инвариант для разных языков -- интересное наблюдение. Что оно значит для нас?

Во-первых, есть слова, которые есть почти в любом тексте -- они в левой части графика. Вы, конечно, догадались, что именно там “обитают” стоп-слова.

Во-вторых, длинный хвост справа свидетельствует о том, что всегда найдутся слова, которых нет в нашем наборе данных, и отсутствие слова в словаре нужно уметь обрабатывать. И как правило, например, в задачах тематической классификации, тематического моделирования и информационного поиска нас интересуют слова где-то между этими двумя крайностями.

Мешок слов

Вернёмся к сравнению слов. Мы поняли, что шумовые, слишком частотные слова стоит исключать из рассмотрения. Кроме того, полезно сравнивать не только факт наличия слова в тексте, но и сколько раз оно в нём встретилось. То есть рассматривать тексты не как множества слов, а мультимножества. Рассмотрение текста как мультимножества токенов, его составляющих, принято называть “мешком слов”, **bag-of-words**.

Для самых простых задач текстовой классификации, поиска и текстового моделирования этого уже хватает, дальнейшее -- вопрос конкретных представлений и алгоритмов.

Но, наверное, у многих здесь возникает вопрос -- почему используется такая простая, “глупая” модель? Ведь мы же в ней никак не используем порядок слов, а он бывает очень важен. Действительно, как минимум его требуют случаи вроде упоминания города “Нижний Новгород”. Если мы построим тут мешок слов, мы потеряем очень важную информацию: посчитаем составляющие имени города по отдельности, и потом машина никак не узнает, что “нижний” -- это часть имени города, а не, скажем, указание на челюсть, где растёт коренной зуб. Аналогичным с мешком слов образом представить для обработки на ЭВМ все последовательности слов, увы, не выйдет: их огромное число. С последовательностями напрямую можно работать некоторыми видами нейронных сетей, но об этом мы говорить в курсе не будем, чтобы сохранить его объём обозримым.

Мешок n-грамм

Но здесь есть компромиссное решение, которое можно в шутку назвать “мешком слов для бедных”. Мы будем смотреть не на токены -- или не только на токены -- но и на пары токенов, встретившихся в тексте подряд. Или тройки. И так далее. Такие подпоследовательности из N токенов называются n -граммами. Модель, в которой мы рассматриваем текст как мультимножество N -грамм, можно называть “мешком N -грамм”.

Здесь, конечно, важно суметь правильно подобрать N . Для слишком большого N (например, двадцать) даже во всём интернете трудно будет найти другой текст, содержащий эту энграмму. А слишком малое N -- например 1 -- возвращает нас к мешку слов и потере информативности модели.

Веб поиск

Одним из самых известных и успешных приложений обработки естественного языка можно назвать веб-поиск. Веб-поиск бывает разным: поиск по изображениям, поиск по звукам -- например, Shazam. Но мы будем рассматривать только поиск по текстам, потому что он релевантен нашему курсу.

Информационный поиск иногда относят к задачам обработки естественного языка, хотя далеко не все стандартные задачи поиска связаны с языком. Но что бесспорно верно -- это то, что пересечений очень много, и обработка естественного языка обязана информационному поиску как науке многими важными концепциями, методами и научными практиками. Но не будем углубляться в его историю.

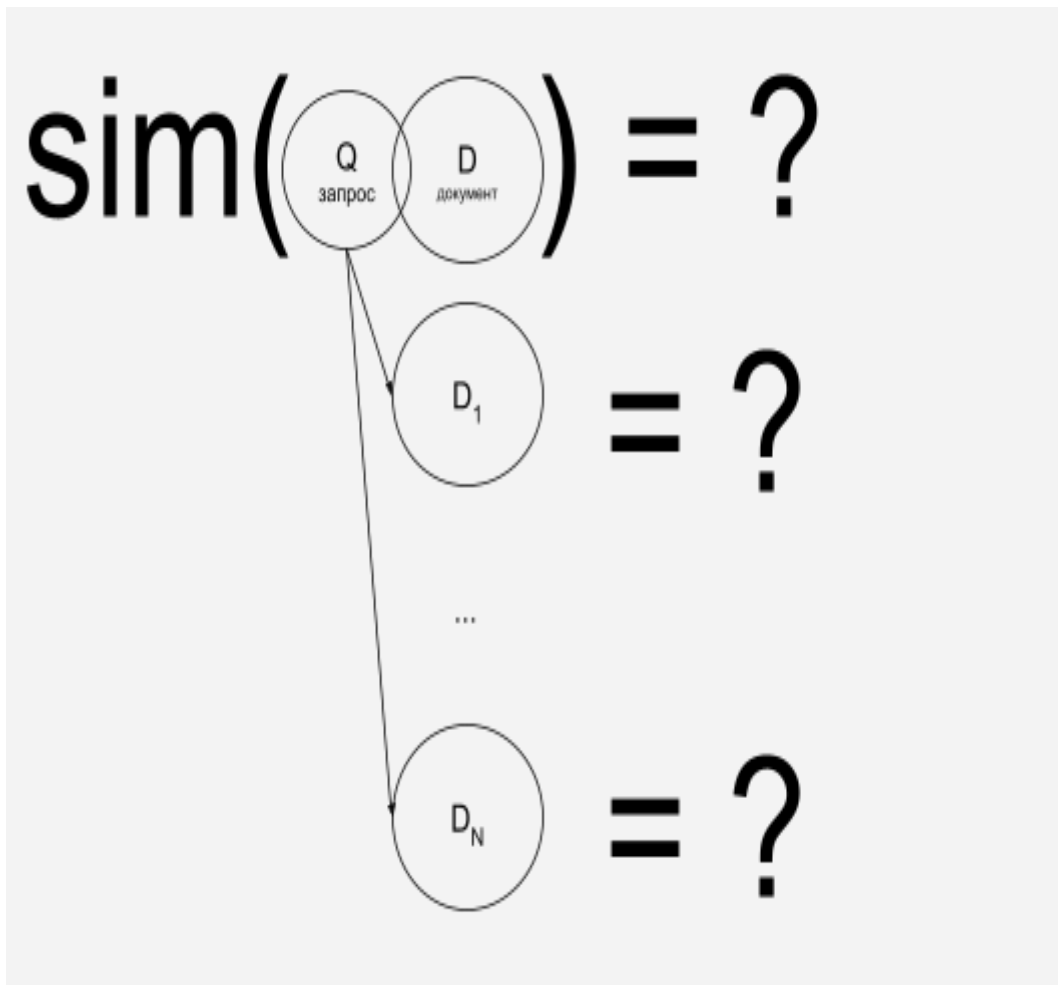
Наиболее полезная с практической точки зрения задача информационного поиска -- это ad-hoc-поиск. То есть необходимость под данный, как правило, краткий запрос, найти наиболее подходящие, то есть релевантные ему, документы. А в нашем случае документы -- это тексты. Это очень понятный всем пример, так как именно так работают все большие поисковики.

Так как сделать поисковик? Мы уже вооружены мешком слов, и, к примеру, можем отнестись к запросу тоже как к тексту или -- вообще говоря -- как к формуле пропозициональной логики, накладывающей ограничения на то, какие термы должны или не должны быть в нужном нам документе. Представить все документы и запрос как множества, оценить похожесть каждого из документов на запрос и выдать самые

подходящие. Такой подход называется булевым поиском.

Сходство Жаккара

Кстати, а как мы можем оценить похожесть двух множеств?



Ведь мы будем сравнивать одно множество попарно с каждым из других. И поэтому было бы удобно, если бы сходство двух множеств можно было выразить числом, чтобы впоследствии по этому числу можно было отсортировать.

Один из стандартных способов это сделать -- **коэффициент Жаккара**. Для двух множеств просто делим мощность, то есть размер, пересечения множеств на мощность объединения множеств.

Иначе говоря, чем больше доля общих элементов у обоих множеств, тем больше коэффициент Жаккара. При точном совпадении множеств объединение будет равно

пересечению, и коэффициент будет равен единице. При отсутствии общих элементов мощность пересечения равна нулю, как и, следовательно, итоговое сходство.

Перед нами матрица термов-документов (term-document-matrix), к которой мы ещё будем обращаться в этом курсе. Каждая строка -- представление некоторого документа как множества, каждый столбец отвечает за какой-то терм из словаря, то есть множества всех возможных термов. В первом документе присутствуют термы “гвоздь” и “теллУр”, а термов никель и копать там нет, поэтому на соответствующих позициях стоят единицы и нули соответственно.

Во-первых, теоретически мы можем пройти по всем строкам с запросом, например ““гвоздь” AND (“теллур” OR “никель”)", и извлечь все подходящие под него документы.

Во-вторых, также заметим, что, чтобы посчитать пересечение множеств термов документов, мы можем просто перемножить по соответствующим координатам строчки нужных документов, и сложить. То есть, можно сказать, вычислить скалярное произведение соответствующих строк матрицы. Ясно, что объединение можно вычислить похожим образом. Значит, расстояние Жаккара между запросом и документом или парами документов у нас в кармане.

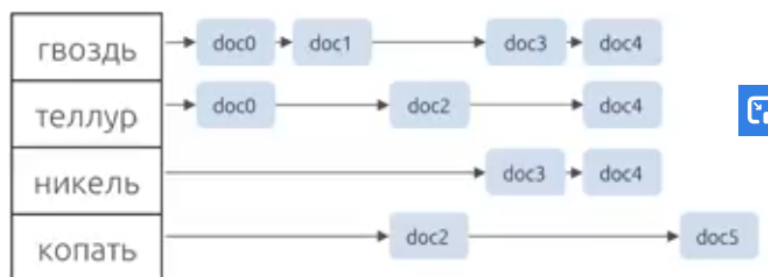
Казалось бы, у нас есть всё для создания простого поисковика: представления документов и запросов, мера сходства запроса и документа, осталось только под каждый запрос вычислить эту меру для каждого документа.

Увы, в реальной жизни сделать это вот так запросто не получится, так как число документов, как правило, большое

На графике -- оценка изменения размера той части Всемирной паутины, по которой ищет Гугл. Под каждый запрос 60 миллиардов раз вычислить коэффициент Жаккара и отсортировать по нему документы -- очень сомнительный план.

Инвертированные индексы

Поэтому в поиске с документами работают совсем иначе. Для каждого терма -- мы храним отсортированный по номерам список документов, в которых этот терм встречается.



Часто вместе с номером документа в этом списке хранится дополнительная информация. Например, положение в тексте (чтобы можно было учитывать близость термов из запроса), фактическая форма слова, сколько раз встречается в тексте и так далее).

Словарь -- то есть список термов -- может храниться в памяти, а списки документов часто хранятся на диске. Операция И в булевой формуле, соответствующей запросу, приводит к пересечению списков документов, а для упорядоченных наборов это делается за линейное время от их длины. Операция ИЛИ приводит к объединению сортированных списков, то есть к алгоритму такой же сложности.

Понятно, что здесь есть много места для улучшения этой структуры, как по быстродействию, так и по памяти, и есть поистине удивительные решения на благо экономии процессорного времени и места на диске. Так, часто списки дополняют дополнительными ссылками на несколько элементов вперёд. Подумайте, как это помогает при вычислении, скажем, пересечения списков.

Ещё один известный способ сэкономить -- так как списки отсортированы -- мы можем хранить номер только первого из документов, а дальше хранить только разности, вместо полных идентификаторов. И так далее.

На благо компактного хранения и быстрой обработки таких структур разработчиками было положено очень много усилий, и если вы когда-нибудь захотите сделать что-то подобное, рекомендуем ознакомиться с уже готовыми решениями. Есть много бесплатных решений с открытым исходным кодом, в которых всё описанное реализовано очень эффективно.

Ранжирование

Итак, допустим, мы собрали данные и уложили их в поисковый индекс, и

делаем к нему очередной запрос. С помощью рассмотренных выше методов извлекаем некоторое количество потенциально релевантных документов.

А мозг поиска, и его самая главная, пожалуй, технология -- это ранжирование, то есть упорядочение, сортировка этих документов. Те, кто застал поисковики начала нулевых, хорошо помнят, почему опираться только на сам факт пересечений по ключевым словам нельзя. Как минимум, этим могут воспользоваться недобросовестные владельцы сайтов и добавить ключевые слова из популярных запросов на свои веб-страницы, и пользователь получит нерелевантный или жульнический контент.

Со всем этим нужно бороться, подбирая, фигурально говоря, заветную формулу ранжирования, которая будет учитывать много факторов.

VSM

Мы уже знаем одну классическую теоретическую модель информационного поиска, булевский поиск. Теперь рассмотрим другую модель -- векторную VECTOR SPACE MODEL. В ней каждому документу и каждому запросу будет сопоставляться некоторый вектор, а близость одних к другим -- по некоторой мере расстояния -- будет считаться отражающей релевантность. То есть в теории по всем нашим документам в базе мы построили по вектору для каждого, затем по каждому приходящему запросу тоже строим вектор и ищем, скажем, десять самых близких по некоторому расстоянию векторы-документы. И их выдаём пользователю.

На деле, как правило, работает всё не совсем так, ведь бездумно запустить поиск ближайших соседей среди нескольких миллиардов векторов не получится. Поиск ближайших соседей -- очень ресурсоёмкая задача. О том, как можно встроить векторную модель поиска в настоящий поиск, поговорим чуть позже. Векторы строить можно по-разному.

Мы уже знакомы с “мешком слов”. Построим длинный вектор, в котором каждой ячейке будет сопоставлено слово, а значение в ячейке -- его частота в данном документе или данном запросе. Или, например, не частота, а единица, если терм встречается, и ноль, если нет. Всё, простейшее представление документов и запросов для векторной модели -- готово.

Размерность такого вектора равна размеру словаря -- числу уникальных термов, с которыми мы работаем. Обычно это десятки или сотни тысяч.

Кстати, ничего страшного для компьютера в том, что векторы такие огромные, нет. В них большая часть значений -- нули, и есть очень эффективные способы представления таких данных. Например, можно хранить список ненулевых значений, как в уже рассмотренных нами инвертированных индексах. Примеры способов представлений разреженных матриц и векторов можно посмотреть, например, в специальном модуле библиотеки `scipy.sparse`.

На слайде пример кусочка вектора, представляющего текст “на берегу пустынных волн” -- с точностью до выбрасывания стоп-слов и лемматизации термов.

абакан	абырвалг	азкабан	аксакал	аксолоть	аксон	берег	волна	заноза	...
0	0	0	0	0	0	1	1	0	...

Запомним это представление. В дальнейшем оно нам понадобится.

TF-IDF

Однако мы уже знаем, что некоторые слова важнее других. Давайте теперь попробуем отойти от обычных счётчиков и частот и придумаем формулу немного похитрее. Таковую, чтобы в ней учитывалась важность данного слова как в контексте всего корпуса, так и в контексте данного документа.

Согласитесь, слово, которое встречается почти в каждом из документов, едва ли представляет большую ценность для поиска. А, в целом, редкое слово, если это не опечатка, ценность имеет высокую. Так, в малом числе документов могут, например, встречаться термины. Отсюда вытекает идея, что можно уравновесить частоту термина в данном документе некоторым “штрафом” за его частотность.

К примеру, поделим общее число документов в нашей базе на количество документов, содержащих данный терм, и логарифмируем эту величину. Чем реже встречается этот терм в корпусе, тем больше будет это значение, и наоборот.

Этот штраф домножим на относительную частоту термина в данном документе, и получим знаменитую формулу `tf-idf`.

Допустим, у нас есть документы:

“... генеральный директор фирмы по производству кракозябрин”

“... финансовый директор ООО “Шорька” ...”

“... кризис затронет все сферы жизни...”

“... заместитель директора по науке НИИ ЧАВО ...”

“... быть директором турфирмы в кризис...”

В реальности коллекции текстов обычно гигантские, многие тысячи документов, но мы рассмотрим игрушечный пример. В новостях науки, промышленности и техники слово “директор” не редкость: в нашем случае этот терм есть в 80% документов, и ценность его для поиска невелика. А вот текстам о кризисе посвящено наверняка меньшее число статей (в конце концов, он не вечен). И в нашем примере слово “кризис” есть в меньшем числе случаев. В реальности 40% это тоже много, но наш пример -- игрушечный.

Представим, что отрывки -- это и есть документы. Давайте вычислим tf-idf терма ДИРЕКТОР в последнем документе. Сначала вычислим IDF -- мы уже знаем, что директор встречается в 80% документов, то есть IDF -- это логарифм от пяти четвёртых. Что касается TF, то количество упоминаний в документе делим на размер документа, то есть одна пятая. Таким образом, если взять двоичный логарифм, tf-idf **равен примерно шести сотым**.

Аналогично вычислим tf-idf для терма “кризис” в том же документе. Доля в документах -- сорок процентов, то есть IDF -- это логарифм от двух и одной пятой. А значение TF такое же, как у директора. Итог -- более **двадцати шести сотых**, что больше, чем tf-idf директора.

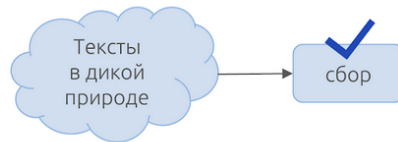
Кроме интуитивного обоснования, у неё есть и вероятностная трактовка¹. Когда мы хотим, чтобы сумма значений tf-idf для всех термов из запроса была побольше для данного документа, это значит, что в рамках некоторой модели мы минимизируем вероятность того, что в этом документе все эти термы оказались случайно.

Итак. Теперь вместо обычных частот важность терма для данного документа будет учитывать и их общее распределение, наказывая слишком частотные. Документ, представленный как вектор с весами tf-idf, обычно лучше справляется с задачей поиска наиболее релевантных документов, чем обычный мешок слов.

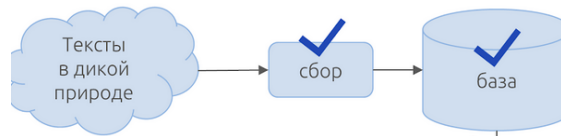
Итак, общая схема поиска часто, хоть и не всегда, выглядит так.

Неподготовленные тексты собираются,

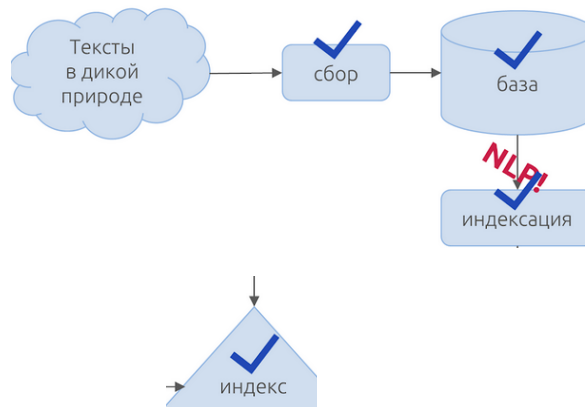
¹(<https://www.youtube.com/watch?v=EioJ902VCmk&t=625>)



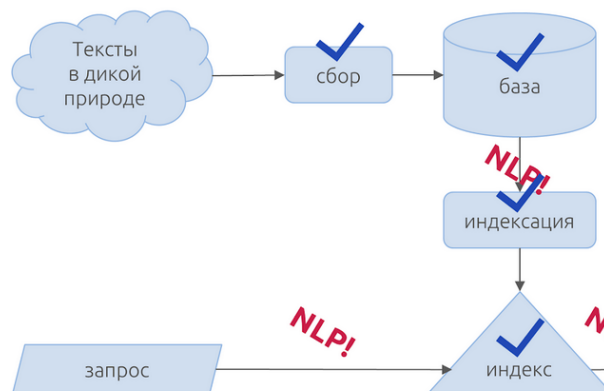
укладываются в базу.



По базе раз в какой-то интервал или непрерывно происходит индексация -- то есть укладка термов и документов, например, в обратные индексы.



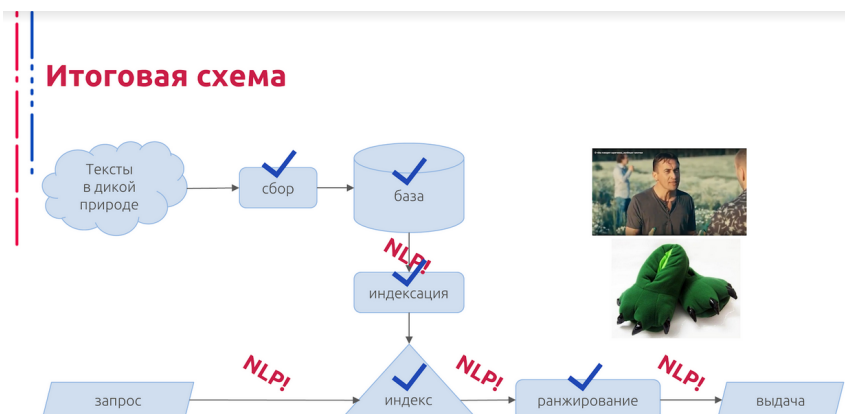
Каждый запрос преобразовывается в запрос в индекс.



К примеру, для запроса “зелёные тапочки” может потребоваться извлечение и пересечение списков документов, прицепленных в инвертированном индексе к термам “зелёный” и “тапок”.



Полученный список документов сортируется алгоритмом ранжирования.



Алгоритмы ранжирования могут использовать не только ти-эф-ай-ди-эф, а машинное обучение для построения функции вычисления релевантности. Можно использовать статистику самых частотных кликов для самых частотных запросов -- и так далее. В данном случае поисковик, к примеру, может с помощью хитрых механизмов предположить, что пользователю надо предложить не только документы, где встречаются слова “зелёный” и “тапочки”, а видеофрагмент фильма “О чём говорят мужчины” или соответствующие предложения магазинов обуви.

Лучшие результаты, специальным образом подготовленные, уже отдаются пользователю.

Почти на всех этапах требуются алгоритмы, которые можно отнести к обработке естественного языка.

Другие задачи в поиске

Если поразмышлять о том, что будет происходить с системой в реальной жизни, когда запросы будут делать живые люди, и если вспомнить, что умеют основные поисковики, то станет ясно, что рассмотренными нами задачами и подходами поиск не ограничивается.

Запросы, вбитые в поиск второпях, часто содержат **ошибки и опечатки**, и нужно уметь их оперативно исправлять.

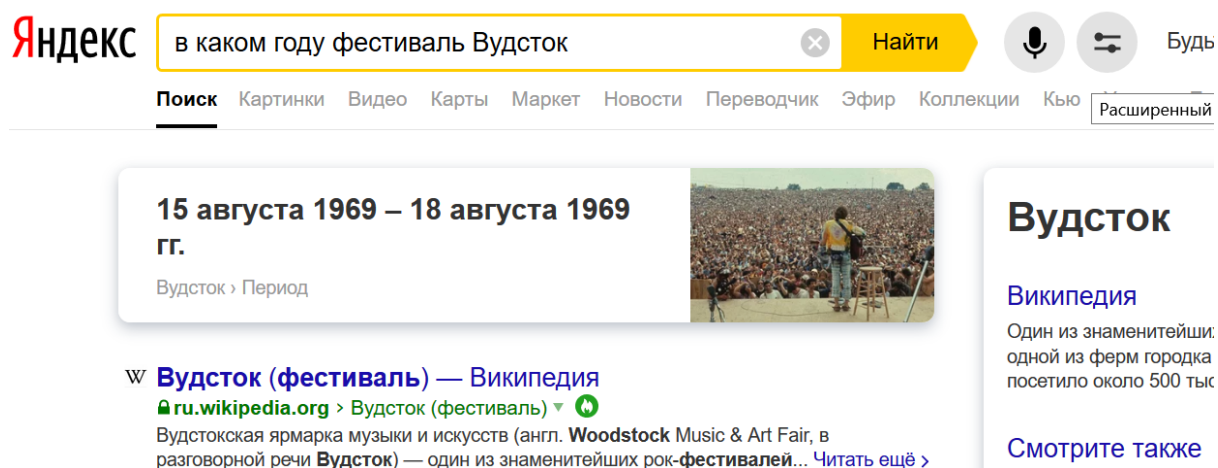
Чтобы увеличить объём выдачи -- то есть множества документов, предоставляемых пользователю в ответ на запрос -- часто запрос **расширяют синонимами** или близкими альтернативными формулировками. Это помогает на этапе выборки документов для последующего ранжирования не отбрасывать потенциально полезные документы, не содержащие исходные термины.

Персонализация. Если, к примеру, вы работаете автодилером и часто ищете в интернете что-то о машинах, то при запросе “Ягуар” в топе у вас должны быть всё же автомобили, а не животные или иные бренды с таким названием.

Вбивая в поиск запрос, мы часто видим вспомогательную панель с догадками системы, что именно мы хотим найти, и клик по нужному варианту, если система угадала, экономит нам время. Особенно это актуально на мобильных устройствах, где хочется минимизировать взаимодействие с виртуальной клавиатурой. Такие **умные подсказки на сленге называются “саджестом”** и разработать хороший алгоритм для неё с использованием статистики запросов и всего, что мы умеем делать с естественным языком, -- сложная задача.

Часто нас интересует не собственно веб-страница, содержащая нужную информацию, а просто конкретный ответ на вопрос. Например, в ответ на запрос “в каком году фестиваль Вудсток” Яндекс выдаёт конкретные даты. Мы видим ответ и источник, и часто можем не сверяться со страницей, чем, опять же, экономим себе время. Для решения подобных задач нужно суметь **определить тип запроса** -- в данном случае требуется некоторый факт. После этого нужно преобразовать запрос в

структурированный в духе “ выдать значение поля “Дата начала” и “Дата конца”, где событие равно “Вудсток” ” -- и обратиться к базе знаний, которую нужно также заранее построить. Всё это относится к задачам обработки естественного языка.



Из-за спама, плагиата, хитрых схем так называемой поисковой оптимизации содержимое некоторых веб-страницы многократно дублируются, поэтому нужно уметь **находить копии**. Это сделает выдачу читаемой и поможет быстройдействию алгоритмов на всех этапах поиска.

Словом, задач в поиске много, и многие из перечисленных уже нами не назвать второстепенными, при этом большую часть задач, над которыми могут трудиться целые отделы, мы и вовсе не упомянули. Работы для специалиста по обработке языка в поиске много.

Как сделать свой веб-поиск

В информационных системах часто требуется обеспечить пользователя интерфейсом поиска на текстовых данных. Вы наверняка много раз видели на разных сайтах текстовые поля ввода с меткой “Поиск по сайту”. Иногда достаточно выполнить поиск точного вхождения -- и здесь подойдёт любая стандартная база данных, но часто требуется поиск с использованием методов обработки естественного языка.

Конечно, в этих случаях не нужно разрабатывать всё то, что мы обсудили ранее, с нуля. Так как сложной задаче ad-hoc-поиска много лет, естественно ожидать, что уже разработаны хорошие и эффективно реализованные алгоритмы поиска. Действительно, есть много готовых бесплатных инструментов для создания собственного поисковика.

Наверное, самые известные из решений с открытым исходным кодом являются частью Apache Software Foundation и написаны на Java: Nutch для сбора страниц, Lucene для построения индекса и поиска, и построенная на последнем поисковая система Solr. Также популярна система Elasticsearch, тоже работающая на основе Lucene.

Есть и решения на питоне, нужно обязательно упомянуть Scrapy для сбора страниц и Xapian для индексации и поиска.