KARPOV.COURSES >>>
КОНСПЕКТ

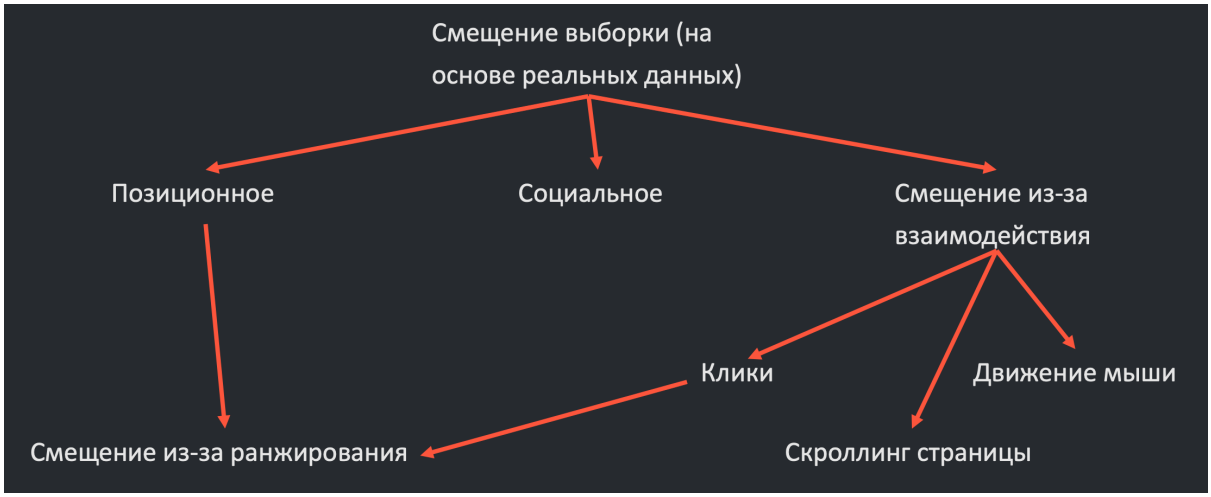
> Конспект > 5 урок > Передовые подходы к ранжированию: обзор ушедшего десятилетия

> Оглавление

> Смещение в задачах ранжирования

Про смещение мы частично говорили на прошлой лекции, когда разбирали алгоритм YetiRank. Обычно оно бывает трёх видов:

- Позиционное
- Социальное
- Из-за взаимодействия с системой



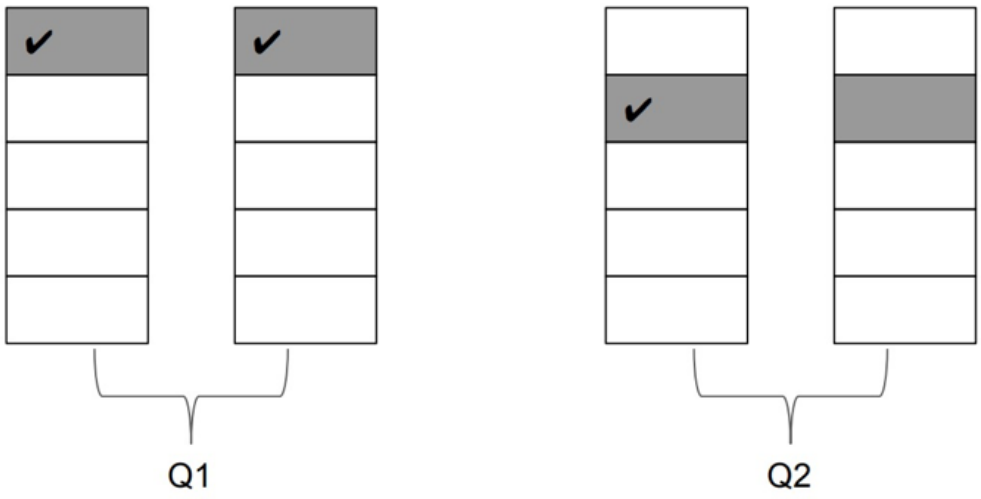
Виды смещений в задачах ранжирования

Социальное смещение может возникать при работе с разными группами людей — такую ситуацию хорошо описывает пример из лекции про пользователей из России и Украины, которые при совершенно ясных инструкциях размечали данные по-разному.

Еще один тип смещения — смещение из-за особенностей взаимодействия с UX. Рассмотрим следующий пример: допустим, у вас есть прототип поисковой системы и вы собираете много дополнительной информации: клики по страницам, поведение курсора мыши, количество времени, проведённого на странице. Это всё прокси-метрики, которые позволяют оценить релевантность документов в поисковой выдаче, т.е. они не являются прямыми показателями релевантности документов, но сильно с ней коррелированы. Однако, чтобы увидеть большую часть выдачи своего поискового запроса, человеку нужно пролистать страницу. Также иногда для нахождения некоторой информации на странице человеку нужно сделать неочевидное действие (довольно синтетический случай, но все же). Если же вы отслеживаете движение курсора и пытаетесь понять поведение пользователя (выяснить, что ему нравится), то результирующая выборка будет смещена из-за специфики вашего пользовательского интерфейса. Всё это примеры смещения

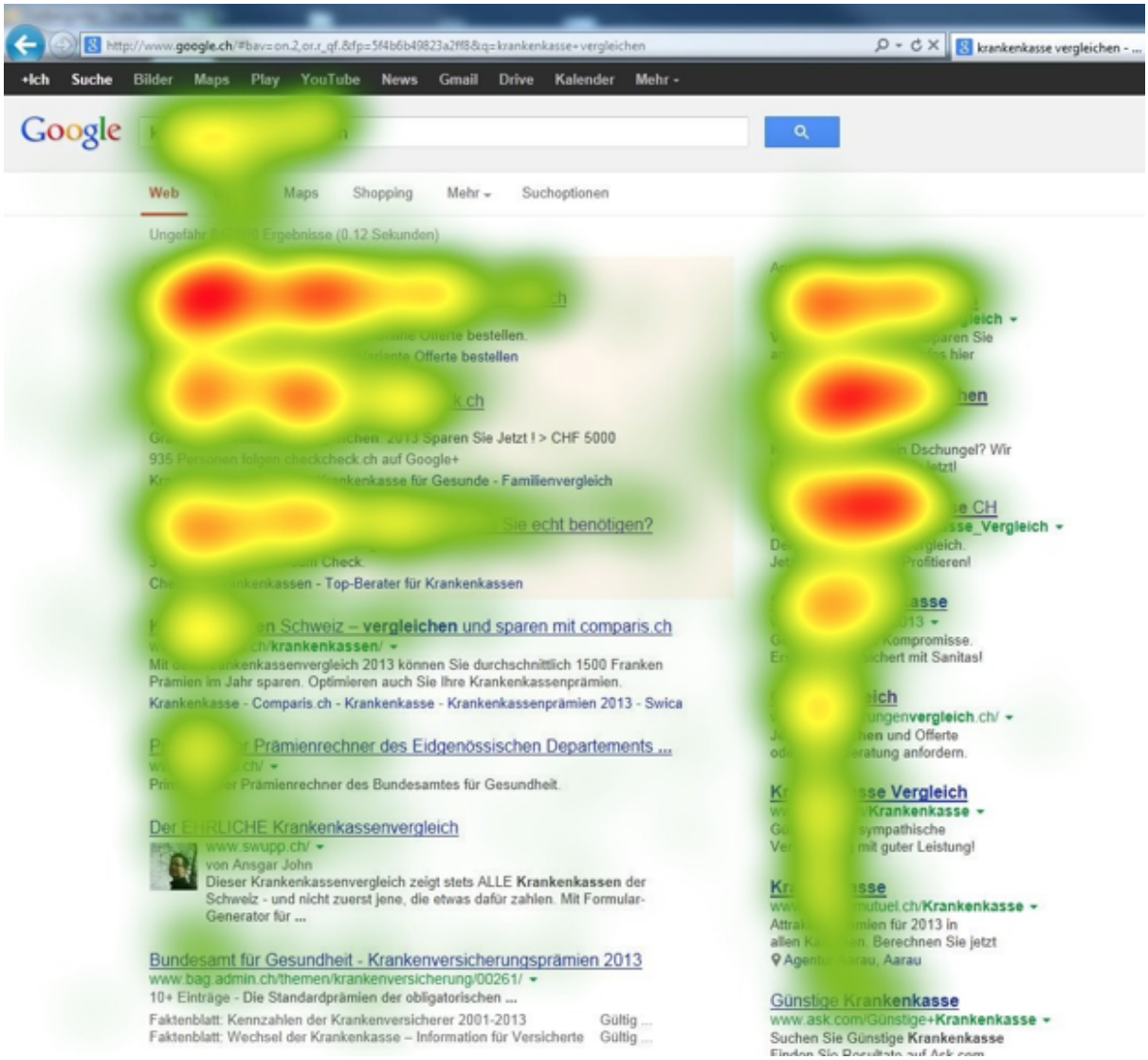
из-за метода взаимодействия и того, как мы оцениваем интерес человека к документам.

И последнее направление, которое частично вытекает из предыдущего, — это позиционное смещение, т.е. некоторая зависимость от того, где в нашей выдаче располагается конкретный документ. Давайте рассмотрим простой пример. Возьмём по 2 выдачи в ответ на 2 запроса. Для каждого запроса выдаётся по 5 документов. Конечно, для конкретного запроса это одинаковые документы, так как модель ранжирования мы не меняли. Серым прямоугольником отметим действительно релевантный документ, а галочкой — клики по ним.



К примеру о позиционном смещении

Для первого запроса документы, расположившиеся на первой позиции, были кликнуты. Это выглядит как нормальная ситуация. Однако если посмотреть на ранжирование для второго запроса, мы увидим, что один и тот же документ (второй), не был кликнут в половине случаев несмотря на то, что он действительно был релевантен второму запросу Q2. Например, это могло произойти потому, что человек просто не обратил на него внимание, так как это не первый документ в выдаче. Если построить статистику кликов на релевантные документы, можно заметить нисходящую тенденцию: чем ниже ссылка в выдаче, тем реже на неё кликают. Это и есть позиционное смещение в данных.



Демонстрация позиционного смещения по кликам

> Глобальная модель смещения в ранжировании

Для учёта смещения в функции потерь воспользуемся уже ставшим классическим методом и будем добавлять веса, зависящие от запроса.

$$\mathrm{L}_{\mathcal{S}}(f) = \frac{1}{|\mathcal{S}|}\sum_{Q\in\mathcal{S}}\frac{P(Q)}{\widehat{P}(Q)}l(Q,f) = \frac{1}{|\mathcal{S}|}\sum_{Q\in\mathcal{S}} w_Q \cdot l(Q,f)$$

$\mathcal{S}$ — наш набор данных (датасет) с парами запросов и документов, $P(Q)$ — вероятность появления такого же запроса Q в генеральной совокупности (важность/частотность запроса), $\widehat{P}(Q)$ — наша оценка этой же вероятности по

имеющемуся набору данных. Эта оценка приблизительная, и не равна в точности истинной величине.

Тогда вес w_Q в функции потерь L есть отношение этих двух величин, т.е. вероятностей встретить запросы в датасете и реальном мире.

Всё дело в том, что мы обучаемся по оценкам релевантности, и в случае кликов нам важны клики, а не отсутствие кликов. Само отсутствие клика не говорит ни о чём.

Поскольку на релевантные документы для второго запроса кликали в 2 раза реже, то $\widehat{P}(Q2)$ для второго запроса будет в 2 раза меньше, чем $\widehat{P}(Q1)$ для первого запроса $Q1$ (иными словами, на каждые две строчки с $Q1$ датасета $\mathcal{S}$ будет одна строчка с $Q2$). При этом мы можем предположить, что в некотором истинном распределении запросов вероятность каждого отдельного запроса крайне мала и все они примерно одинаковы. Это сильное допущение, так как всё очень сильно зависит от темы запроса (все-таки некоторые темы гораздо популярнее других), но допустим, что в данном случае для двух запросов Q1 и Q2 истинные величины P равны, а в наших данных мы наблюдаем смещённую картину. Поэтому наши веса мы привязываем именно к соотношению некоторой истинной и наблюдаемой величины встречаемости запроса, т.е. к полученной дроби. Давайте для этой дроби будем подбирать другую такую дробь, которая будет пропорциональна исходной, и пытаться аппроксимировать её.

$$w_Q = \frac{P(Q)}{\hat{P}(Q)} \propto \frac{1}{b_i}$$

Далее можно отметить, что клики $c^Q_{\mathbf{x}i}$, которые приходятся на i-й документ в выдаче для запроса Q, можно выразить в виде произведения некоторой реальной релевантности документа $r^Q_{\mathbf{x}}$ и некоторого коэффициента b_i, зависящего от позиции i.

$$c^Q_{\mathbf{x}i} = r^Q_{\mathbf{x}} \cdot b_i$$

$$b_i = \frac{\sum_{Q \in \mathcal{R}} \int_{\mathbf{x} \in Q} c_{\mathbf{x}i} \, \mathrm{d}P(\mathbf{x} \mid Q, i)}{\sum_{Q \in \mathcal{R}} \int_{\mathbf{x} \in Q} r_{\mathbf{x}} \mathrm{d}P(\mathbf{x} \mid Q, i)}$$

Можно переписать это произведение в виде вот такого страшного выражения выше, чтобы выразить этот коэффициент b_i. Не будем вдаваться в детали этой формулы. Нам важно понять, что в числителе у нас стоят клики, обозначаемые как $c_{\mathbf{x}i}$, а в знаменателе — релевантность $r_{\mathbf{x}}$. По сути наш позиционный коэффициент b_i показывает, во сколько раз больше пришлось кликов на этот документ, чем должно быть исходя из его релевантности, если мы нивелируем позиционную составляющую.

Тогда по имеющимся данным, по истории кликов мы можем построить модель, оценивающую это позиционное смещение (bias). Такая модель называется Generalized Bias Model, и в простейшем случае это просто N логистических регрессий, где N — количество рассматриваемых позиций. Допустим, мы пытаемся работать с топ-10 результатов, тогда нам нужно обучить 10 моделей, у каждой из которых есть свои коэффициенты в векторах β_i, которые в формуле находятся в знаменателе.

$$b_i^Q = \frac{1}{1 + \exp\left(\beta_i \cdot \mathbf{v}(Q)\right)}$$

Более того, у нас есть некоторая функция $\mathbf{v}(Q)$ перевода запроса Q в вектор. Например, простые статистики по словам или что-то более сложное. Итого для каждого запроса мы имеем возможность оценить сдвиги конкретно для него с учетом его особенностей. Скажем, люди более трепетно относятся к поиску автомобилей на продажу и потому, чтобы не потерять деньги, готовы пролистать и 10 ссылок в поисковой выдаче, а вот запрос по теме “Первая неотложная помощь при удушении” скорее всего сместит внимание пользователей исключительно на первую ссылку.

Итого в знаменателе в скобках мы умножаем вектор весов β_i на вектор признаков запроса $v(Q)$ и получаем скалярную величину, т.е. одно число, обучаемое на имеющихся данных.

Для сбора данных также может быть полезным провести тест со случайной перестановкой документов в топе. То есть получить ранжирование для 10 объектов, а затем перемешать их и показать пользователю. Если мы проделаем такое упражнение много раз, у нас появится возможность оценить реальное влияние позиции на "кликабельность" одних и тех же документов на разных позициях.

> Unbiased LambdaMART

Вообще при обучении модели мы стремимся найти такую функцию f, которая, будучи применённой к признакам x_i, будет давать наименьшее значение некоторой функции потерь L, штрафующей за несоответствие некоторому значению. Проблема в том, что это значение в идеале должно быть релевантностью документа r, и получать мы должны функцию f_{rel}, а при обучении на данных о кликах (или любых других метриках, собранных на реальных данных о пользователях) мы получаем функцию предсказания

"кликабельности". Эти функции не эквиваленты. Более того, при таком подходе у нас получается поточечный метод.

$$P\left(c_i^+|x_i\right) = t_i^+ P\left(r_i^+|x_i\right)$$

Далее мы можем выписать равенство вероятности клика $P\left(c_i^+|x_i\right)$ при условии наличия некоторых признаков x_i между запросом и документом и произведения некоторого позиционного коэффициента t_i^+ на вероятность релевантности $P\left(r_i^+|x_i\right)$. По сути это то же самое произведение, что мы рассматривали на прошлых шагах с коэффициентом b_i. Тогда для получения функции, учитывающей смещения, будем делить значение функции потерь на позиционный коэффициент, и всего коэффициентов будет столько же, сколько рассматриваемых позиций (скажем, k штук). Сам этот коэффициент можно рассматривать как вероятность того, что документ будет кликнут при условии, что документ релевантен. Логично, что он тем выше, чем выше в ранжировании находится документ.

$$t_i^+ = \frac{P\left(c_i^+|x_i\right)}{P\left(r_i^+|x_i\right)} = \frac{P\left(c_i^+, r_i^+|x_i\right)}{P\left(r_i^+|x_i\right)} = P\left(c_i^+|r_i^+, x_i\right)$$

Этот вероятностный подход можно применить и к pairwise методам ранжирования. Мы получим аналогичную формулу для функции, однако функция потерь L зависит от двух документов, у каждого из которых свои истинные релевантности r_i^+, r_j^- и полученные клики c_i^+, c_j^- (литеры c^+ означают значения для положительного документа, более релеватного, находящегося в выдаче выше, а c^- - для того, что ниже). Соответственно получаем всё еще не эквивалентные функции, в которых никак не учтена позиционная составляющая, но зато функции попарны.

$$\hat{f}_{\text{rel}} = \arg\min_f \sum_q \sum_{(d_i,d_j)\in I_q} L\left(f\left(x_i\right), r_i^+, f\left(x_j\right), r_j^-\right)$$

$$\hat{f}_{\text{click}} = \arg\min_f \sum_q \sum_{(d_i,d_j)\in I_q} L\left(f\left(x_i\right), c_i^+, f\left(x_j\right), c_j^-\right)$$

Применим тот же подход с коэффициентами, только теперь их будет вдвое больше — $2k$

Коэффициенты t^+ имеют всю ту же интерпретацию, а вот коэффициенты для негативных примеров, отранжированных ниже, уже меняют смысл. t^- это уже не вероятность, а величина, обратная вероятности того, что некликнутый документ нерелевантен запросу. То есть у нас поменялось условие в условной

вероятности: в одном случае рассматриваем вероятность клика при условии релевантности, в другом — нерелевантность при условии отсутствия клика.

$$t_j^- = \frac{P\left(c_j^-|x_j\right)}{P\left(r_i^-|x_j\right)} = \frac{P\left(c_j^-|x_j\right)}{P\left(r_j^-, c_j^-|x_j\right)} = \frac{1}{P\left(r_j^-|c_j^-, x_j\right)}$$

Иными словами, мы определяем, “насколько вероятно, что тот документ, на который не кликнули, действительно не релевантен запросу”, ведь может быть так, что он вполне способен удовлетворить пользователя, просто до него человек не дошёл, так как встретил i-й документ выше и кликнул на него. Здесь необходимо отметить, по каким причинам могут “не кликнуть” на документ. Во-первых, речь идёт о каждой выдаче на каждый запрос, т.е. в рамках одного примера для обучения модели берём не глобальную статистику по всем кликам, коих могут быть тысячи. Во-вторых, здесь вводится предположение, что человек перестанет кликать на ссылки, когда найдёт релевантный документ, по сути это то же самое предположение, которое делалось в метрике pFound от Яндекса. Но даже если есть две или даже три кликнутые ссылки из десятка, то всё ещё имеется возможность составлять такие пары, что один документ кликнут, а второй нет.

Необходимо отметить, что $t^+ \in [0, 1]$, а $t^- \in [1, +\infty)$. Эту информацию можно использовать для интерпретации взвешивания. Также легко понять, что единственное общее значение между ними — это 1. Поэтому в качестве начального приближения, до обучения моделей, мы инициализируем все веса единицами.

$$\hat{f}_{\text{unbiased}} = \arg\min_f \sum_q \sum_{(d_i, d_j) \in I_q} \frac{L\left(f\left(x_i\right), c_i^+, f\left(x_j\right), c_j^-\right)}{t_i^+ \cdot t_j^-}$$

Эти коэффициенты точно так же можно добавить в функцию потерь L при оптимизации, просто разделив её на произведение коэффициентов t^+ и t^-. Понятно, что эти коэффициенты оптимизируются и меняются во время обучения, подстраиваясь под имеющиеся данные, удаляясь от начального значения, равного единице. Для обучения модели мы должны считать градиент:

$$\frac{\partial \mathcal{L}\left(f, \left(t^+\right)^*, \left(t^-\right)^*\right)}{\partial f} =$$

$$\sum_q \sum_{(d_i, d_j) \in I_q} \frac{1}{\left(t_i^+\right)^* \cdot \left(t_j^-\right)^*} \frac{\partial L\left(f\left(x_i\right), c_i^+, f\left(x_j\right), c_j^-\right)}{\partial f}$$

К этому моменту вы уже могли обратить внимание на сходство этой формулы с теми, которые были у нас на занятиях про LambdaRank и LambdaMART. Действительно:

$$\tilde{\lambda}_i = \sum_{j:(d_i,d_j)\in I_q} \tilde{\lambda}_{ij} - \sum_{j:(d_j,d_i)\in I_q} \tilde{\lambda}_{ji}$$

$$\tilde{\lambda}_{ij} = \frac{\lambda_{ij}}{\left(t_i^+\right)^* \cdot (t_j^-)^*}$$

Выходит, что добавление этих коэффициентов можно совместить с изученным методом. Для этого нужно каждую лямбду нормировать на всё то же произведение позиционных коэффициентов. Тогда модифицированный LambdaMART алгоритм (Unbiased LambdaMART) будет выглядеть следующим образом:

Algorithm 1 Unbiased LambdaMART

Require: click dataset $\mathcal{D} = \{(q, D_q, C_q)\}$; hyper-parameters p, M;
Ensure: unbiased ranker f; position biases (ratios) t^+ and t^-;

```
Initialize all position biases (ratios) as 1;
for m = 1 to M do
    for each query q and each document d_i in D_q do
        Calculate λ̃_i with (t^+)* and (t^-)* using (35) and (36);
    end for
    Re-train ranker f with λ̃ using LambdaMART algorithm
    Re-estimate position biases (ratios) t^+ and t^- with ranker
    f* using (30) and (31)
end for
return f, t^+, and t^-;
```

Псевдокод алгоритма Unbiased LambdaMART (* означает использование фиксированной модели или коэффициента)

Итого на каждой итерации при построении каждого нового дерева мы сначала добавляем новый алгоритм в ансамбль, затем производим переоценку коэффициентов и на следующей итерации считаем λ уже по новым позиционным смещениям. Таким образом, на выходе алгоритма после тренировки у нас есть матрица с коэффициентами, которые можно

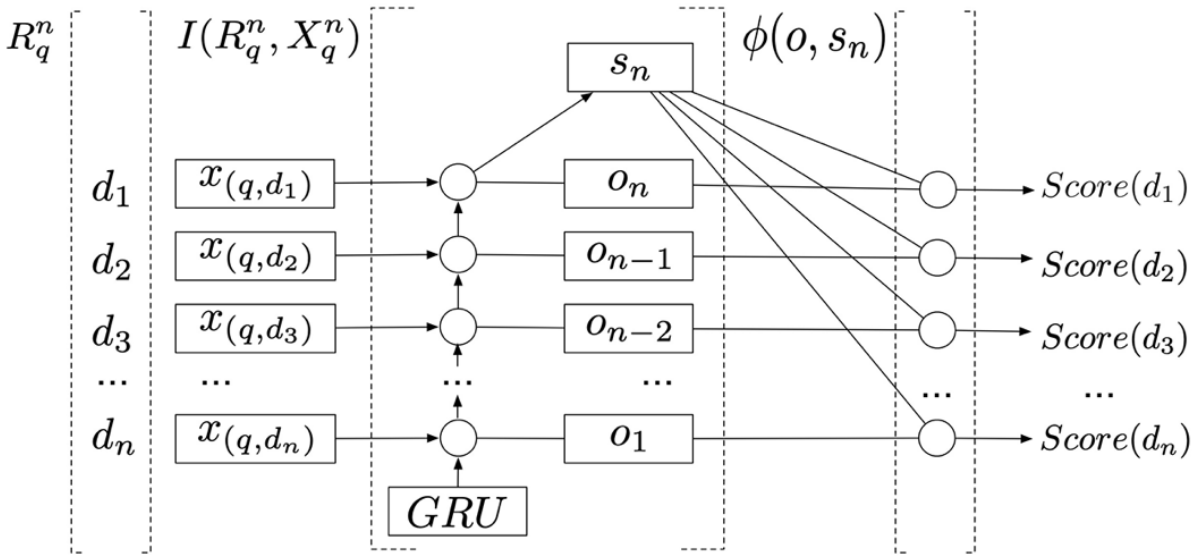
использовать при применении модели, а также для анализа ваших данных — например, чтобы построить гистограммы.

> Listwise Context Model (DLCM)

Данный подход максимально актуален для задачи матчинга.

Основная особенность этого подхода в том, что он поистине listwise. Действительно, базовые listwise подходы при обучении, конечно, используют всю информацию о списке документов, но при применении модели (inference) предсказания делаются для документов независимо друг от друга, и кажется, что тут получается не совсем списочный подход. В идеале же хочется, чтобы объекты в ранжируемом списке “знали" друг о друге и учитывали некоторую мета-информацию о своих соседях при оценке релевантности. Такой подход позволяет делать модель более аккуратную и внимательную к деталям, особенно в случае, когда документы сильно схожи и отличаются, допустим, всего на 1 символ (например, номера моделей iPhone 11 и iPhone 12).

Эту концепцию предлагает воплотить в жизнь модель с названием DLCM или Deep ListWise Context Model (ссылка на статью). Её архитектура выглядит так:



Архитектура DLCM

Имеются некоторые документы $d_1, d_2, \ldots, d_n$, которые ставятся в сопоставление некоторому запросу q из датасета, а некоторая функция $x(q, d_i)$ производит признаки, хранящие в себе совместную информацию из запроса и

документа. Это простые классические признаки, приходящие на вход некоторой модели ранжирования. Документы $d_1, d_2 \ldots, d_n$ уже упорядочены согласно некоторой модели, возможно даже самой простой (какой-то отбор кандидатов и ранжирование уже были произведены).

Берётся рекуррентная нейронная сеть и применяется от последнего документа к первому, записывая скрытые состояния (hidden state/output) рекуррентного блока. В качестве рекуррентного блока предлагается использовать GRU-блок (Gate Recurrent Unit), но это не обязательно. Общая концепция такова, что на каждом этапе принимается на вход предыдущее скрытое состояние блока (или как его ещё называют “hidden state”, признаки пары документ-запрос), а на выходе получается новый hidden state, который, грубо говоря, хранит в себе информацию о том, какие документы уже были отсмотрены, и передает знание дальше, в следующий блок (на схеме это обозначено прямоугольниками с буквой o_i).

Таким образом, когда мы перейдём к предпоследнему документу, т.е. на втором шаге работы рекуррентной сети, так как мы идём снизу вверх, на вход в блок будут поданы преобразованные признаки с информацией о последнем документе, и появится возможность как бы сравнить текущий документ с предыдущими, и так далее. Когда мы дойдём до конца, при обработке, скажем, первого объекта в предварительном ранжировании у нас будет вся информация о тех документах, которые приведены в отранжированном списке, и оценка релевантности может быть скорректированна.

Ещё раз обратите внимание, что по документам мы идём снизу вверх.

Такой порядок обусловлен тем, что у нас есть предварительный шаг ранжирования, и, скорее всего, в самом низу находятся незначительные документы, в которых не так страшно ошибиться. А когда дело доходит до первых позиций, то hidden state o_i уже полон информации о контексте списка ранжирования, ведь это контекстная модель. Если же идти сверху вниз, то контекстная модель насытится знаниями об окружении документов позже, и качественного реранжирования в верхней части (топе) не получится. Более того, у рекуррентных сетей есть проблема забывания давно "увиденной" информации, и потому самые первые рассматриваемые объекты, скорее всего, никак не отразятся на последних итерациях, что опять же говорит в пользу bottom-up подхода, т.е. движения снизу вверх.

На выходе после первого документа, т.е. на последней итерации рекуррентной сети получается некоторый глобальный контекст s_n (он же o_n). После этого просто обучаем некоторую функцию ϕ от этого глобального контекста, который

един для всех документов, и от конкретного скрытого состояния, произведённого рекуррентной сетью на соответствующей для документа итерации. Можно использовать, например, MultiLayer Perceptron (полносвязную сеть) или более сложный алгоритм и на выходе получать те предсказания, по которым будет производиться финальное ранжирование.

$\phi\left(\boldsymbol{o}_{n+1-i}, \boldsymbol{s}_n\right) = \boldsymbol{V}_\phi \cdot \left(\boldsymbol{o}_{n+1-i} \cdot \tanh\left(\boldsymbol{W}_\phi \cdot \boldsymbol{s}_n + \boldsymbol{b}_\phi\right)\right)$ — пример функции ϕ

Такую модель обучают с помощью слегка изменённой функции потерь. Сами предсказания сначала конвертируются в некоторые условные единицы, называемые вниманием (attention). То есть модель учится предсказывать, сколько внимания человека будет уделено конкретному документу. Формула конвертации следующая:

$$a_i^x = \frac{\phi\left(x_{(q,d_i)}\right)}{\sum_{d_k \in R_q^n} \phi\left(x_{(q,d_k)}\right)}$$

По сути это классический SoftMax для приведения величин к единому масштабу (чтобы всё суммировалось в 1), который уже обсуждался ранее. Ниже приведена сама функция потерь, называемая Attention Rank:

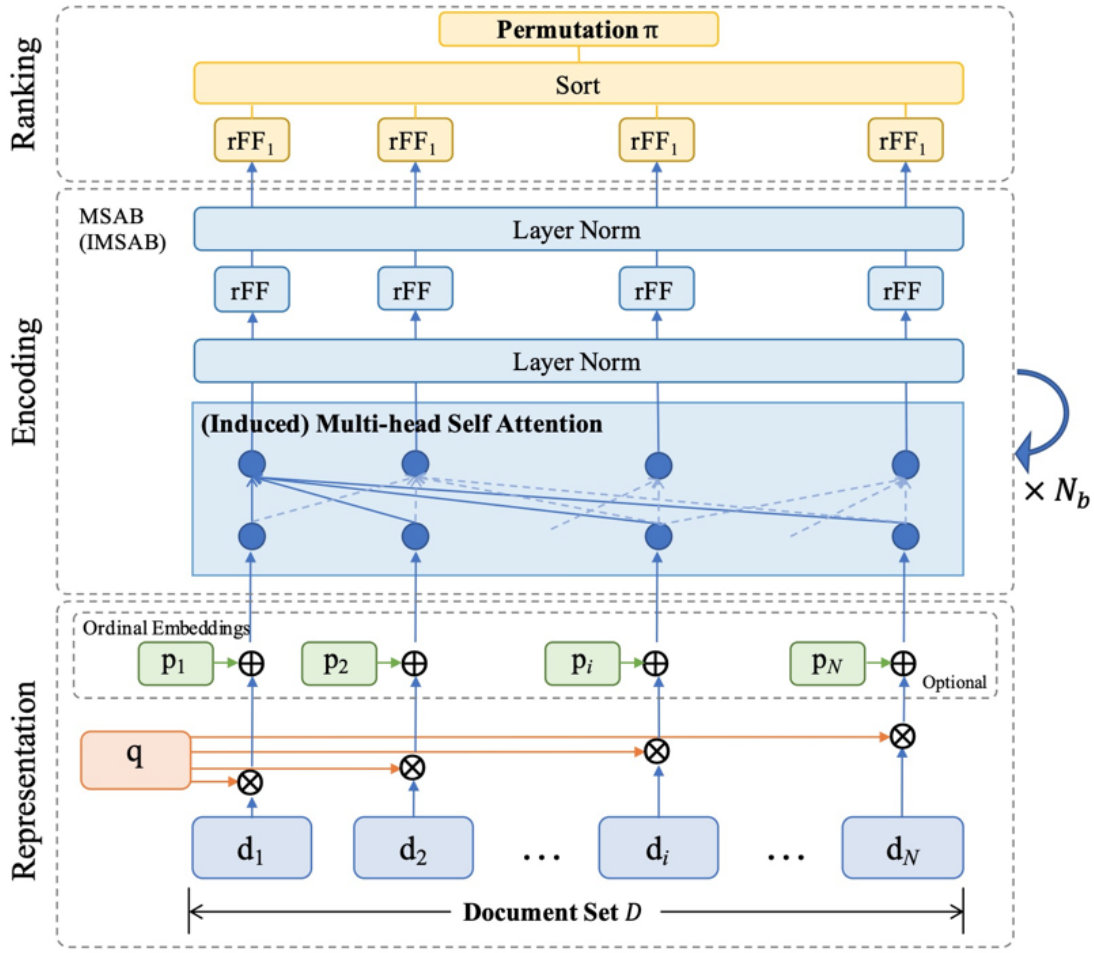
$$\ell\left(R_q^n\right) = -\sum_{d_i \in R_q^n} \left(a_i^x \log\left(a_i^S\right) + \left(1 - a_i^x\right) \log\left(1 - a_i^S\right)\right)$$

Легко заметить, что это по сути Binary CrossEntropy (BCE) или LogLoss.

Подводя итог, можно сказать, что функция потерь фокусируется на относительной важности каждого объекта в ранжируемом списке вместо предсказания релевантности напрямую. То есть мы уделяем больше внимания хорошему документу в списке полностью нерелевантных, нежели блестящему в списке отличных, ведь этот хороший документ бросается в глаза на фоне "горы мусора" в выдаче. Идея учёта контекста при финальном реранжировании была подмечена и во множестве других работ и исследований.

> SetRank

В продолжение темы рассмотрим подход SetRank. Он был вдохновлён появлением архитектуры трансформеров (Transformers) в 2017-2018 гг. и активным расширением областей их применения во всех задачах машинного обучения от работы с картинками до анализа временных рядов.



Архитектура SetRank

Концепция работы трансформеров такова, что в каждом из его блоков, в каждом из слоев этой нейронной сети, мы учитываем взаимодействие каждого входного объекта с каждым другим (в рамках концепции каждый-с-каждым). Это видно в блоке голубого цвета на картинке — здесь к верхнему ряду синих точек, т.е. скрытых представлений документов, присоединены стрелочки от всех входных документов из нижнего ряда. Если идти снизу вверх, то концепция максимально похожа на предыдущий подход. Сначала мы генерируем некоторые вектора с признаковым описанием документов, самый нижний ряд $d_1, d_2 \ldots, d_n$. Затем добавляем информацию о запросе q, к которому мы хотим ранжировать документы. Опционально добавляется позиционное кодирование, которое указывает на результат работы предыдущего метода ранжирования. Например, мы применили индекс BM25 для отсечения кандидатов, по нему отранжировали топ-100 документов и теперь подаём их в трансформер, но при этом хотим учесть и информацию о значениях индекса BM25. Тогда мы можем сделать позиционное кодирование

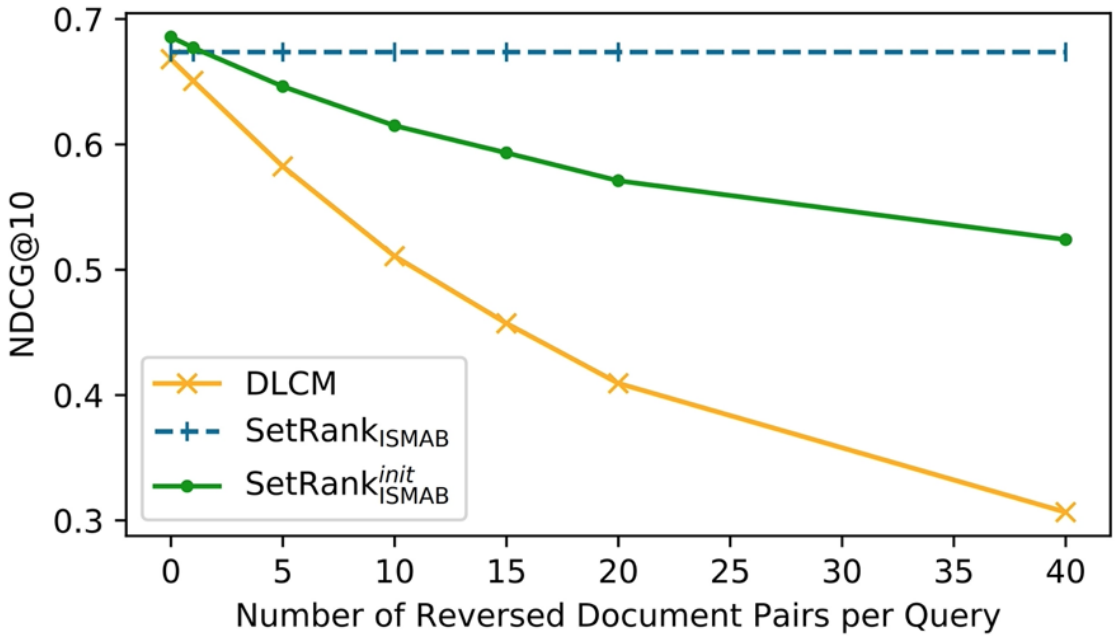
даже самым простым методом — через One-Hot encodding, где для первого, по мнению индекса BM25, документа у нас будет вектор, состоящий из единицы и множества нулей. Для второго документа единица будет на второй позиции (т.е. нуль, единица и далее нули), и так далее.

В случае с BM25 использование подобного приёма является нерациональным, так как сложная и умная сеть в виде трансформера должна и сама выучить такие простые закономерности, однако если у вас есть качественная базовая модель (тот же LambdaMART), то вы можете дополнить её, при этом не теряя имеющиеся наработки, уже обученную и работающую модель.

Здесь в правой части второго блока снизу указано, что позиционное, или порядковое, кодирование опционально, и если его не добавлять, то алгоритм получится инвариантным к порядку входных документов.

На выходе трансформера для каждого документа предсказывается некоторое значение релевантности, и затем по нему происходит финальная сортировка, или упорядочивание, порождающее отранжированный список документов — некоторую перестановку этих документов π, что обозначено на верхней части изображения.

Важной особенностью является инвариантность при отсутствии модели предыдущего уровня. Можно провести следующий эксперимент и его результаты нанести на график. Давайте в правильно отранжированном подмножестве документов намеренно переставлять пары этих самых документов и смотреть на результат работы алгоритма SetRank по сравнению с предыдущим алгоритмом DLCM, который основан на рекуррентной нейронной сети.



Результаты эксперимента по перестановке входных документов

По графику видно, что при росте количества неправильно упорядоченных пар качество SetRank подхода, использующего связь каждого документа с каждым, метрика NDCG никак не меняется, не деградирует (голубая пунктирная линия на графике). В то же время DLCM не может похвастать таким качеством (оранжевая линия), ведь, как вы помните, рекуррентная сеть идёт снизу вверх по предварительно отранжированному списку и начинает "забывать" информацию к моменту подъёма на верхние позиции. Значит, если базовое ранжирование работает плохо, то и модель не сможет показать высоких значений метрики.

Интересно обратить внимание и на зелёную линию — это подход SetRank на трансформерах, однако с позиционным кодированием предсказаний модели предыдущего уровня. Видно, что если у нас очень хорошая модель, в которой количество инверсий документов минимально, то качество предсказаний выходит слегка лучше, однако если модель предыдущего уровня слаба, то и "великие и могучие" трансформеры начинают деградировать. Это стоит учесть при использовании модели на своих данных — нужно обязательно проверить, что лучше именно для вашей прикладной задачи.

Опыт подсказывает, что рескоринг, или реранжирование, документов с большей релевантностью (топа выдачи) очень помогает в задаче матчинга, где всплывает очень много похожих, но всё же неправильных моделей товаров.

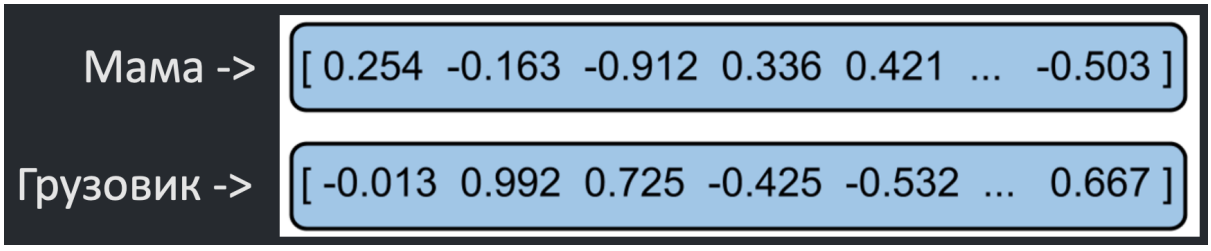
Если нейросеть видит, что есть 15 очень похожих товаров, и по отдельности у каких-то из них атрибуты, например цвет, объём памяти или батареи, совпадают с запросом, но при этом нет единого товара, который бы удовлетворял всем требованиям, то скор предсказания будет занижен, и модель не выдаст ошибочное предсказание. И такой подход уже можно без стеснения назвать ListWise-подходом, который в полной мере учитывает взаимодействие документов в топе ранжирования.

Рассмотренный метод SetRank можно назвать одним из самых современных и продвинутых с точки зрения рескоринга предсказаний в задаче ранжирования и матчинга.

> Embeddings

Далее мы будем говорить об изменении архитектуры самих моделей, работающих с данными для ранжирования. Огромное количество усилий за прошедшее десятилетие было направлено на выявление тех золотых стандартов, которые позволяют максимально эффективно использовать имеющиеся данные при решении задачи ранжирования и матчинга, также как свёрточные нейронные сети в своё время помогли серьёзно продвинуться в задачах компьютерного зрения.

Но сначала давайте введём понятие эмбеддинга. Это модель, которая каждому слову или словосочетанию сопоставляет вектор вещественных чисел. Качественно обученные эмбеддинги выдают осмысленные вектора, которые схожи для схожих понятий. Обычно эмбеддинги выучивают смысл слова по контексту, и потому отлично подходят для поиска схожих текстов (даже если мысли в них выражены разными словами).



Демонстрация эмбеддингов (embeddings)

Так, например, эмбеддинг для слова “мама” может выдать нам некоторый вектор (см. иллюстрацию). Далее с этим вектором можно работать, подавать его в алгоритм как признаки, использовать в нейронных сетях. Это важно,

потому что и для задачи матчинга, и для большинства задач ранжирования, выраженных информационным поиском, самой важной частью данных является текстовая информация, а поэтому важно извлекать из неё как можно больше полезных сигналов, позволяющих на качественном уровне решать задачу.

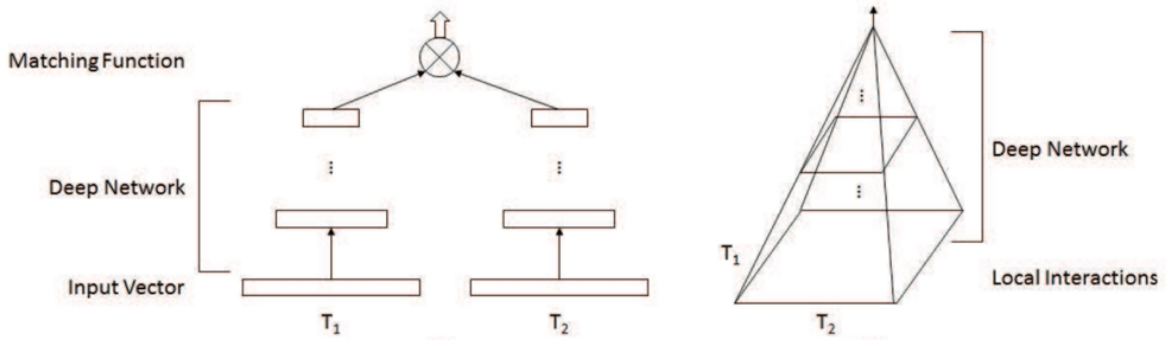
Эра эмбеддингов началась в 2013 году, когда вышла статья, описывающая подход Word2vec, который работает в точности согласно парадигме чёрного ящика: некоторое входное слово будет переводиться в вектор, т.е. word to vector.

> Local & Distributed text representations for web search

Все модели можно разделить на две большие группы:

- Local (локальные) модели, в которых каждое слово соответствует уникальному идентификатору и релевантность представлена функцией от точного соответствия идентификаторов запроса в документе, возможно, с учётом позиций.
- Distributed модели, в которых слова и текст имеют векторное представление в некотором латентном пространстве — те самые эмбеддинги. Релевантность представлена как функция от схожести эмбеддингов запросов и документов.

Эти две группы можно схематично представить в следующем виде:



Local модели

Локальные модели очень эффективны в задаче матчинга. Рассмотрим следующий пример:

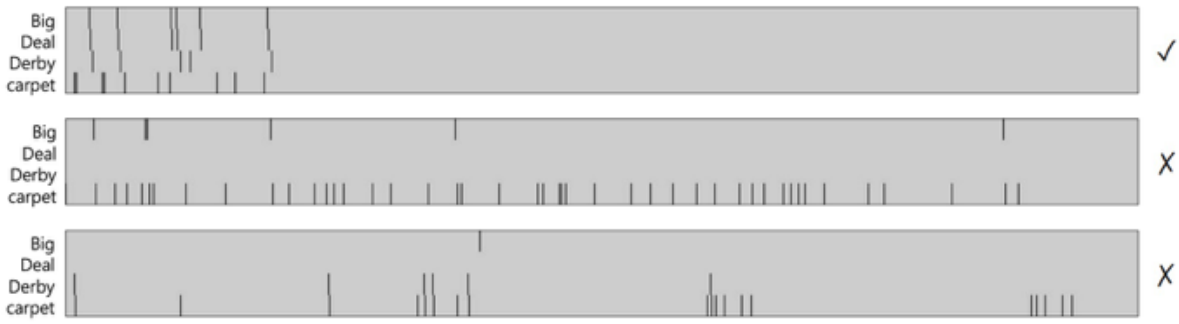
Сайт производителя: "Посудомоечная машина BIO серия Economy qwerty123456"

Сайт продавца: "*Посудомоечная машина Economy (QWERTY123456)*"

При приведении к нижнему регистру можно обратить внимание на невероятно редкий токен, на редкое слово, которое, скорее всего, в остальной части корпуса текстов либо не встречается вообще, либо встречается лишь один раз. Даже простая модель вроде индекса BM25 выведет подобный тайтл в топ (как наиболее релевантный). Принцип очень прост: есть редкий токен, он встретился в документе, а значит, документ релевантен запросу.

Локальная модель похожа на пирамиду (см. иллюстрацию выше), на нижнем этаже которой учитываются индивидуальные локальные взаимодействия, или "интеракции", между запросом и документом, как это было с артикулом у посудомоечной машинки. После этих интеракций происходит некоторая обработка глубокой нейронной сетью, и на выходе получается предсказание релевантности. Такие модели можно назвать Interaction-focused.

В качестве локальной модели можно взять матрицу точных соответствий слов между запросом и документом, причём с учётом позиций. Пример такой матрицы вы можете видеть ниже:



Query: big deal derby carpet

Для каждого слова запроса найдём в каждом из трёх документов упоминания этого слова. По оси OX в каждом прямоугольнике указана позиция в документе, т.е. номер слова: чем правее, тем дальше от начала текста. По оси OY указано само слово из запроса, на перекрестии чёрной чертой отмечается факт наличия слова. По такой матрице можно заметить, что в первом документе, самом верхнем, во-первых, слова из запроса сконцентрированы и идут друг за другом, т.е. образуют словосочетания, а во-вторых, употребляются

в достаточной мере все слова из запроса. Для двух оставшихся документов эти принципы неверны: во втором документе часто фигурирует одно из слов, несколько раз другое слово и вообще отсутствуют ещё два. В третьем примере то же самое. Такую матрицу можно использовать как вход для локальной модели, которая изображена как часть большой системы на левой части большого изображения (см. иллюстрацию ниже). Затем эта входная матрица проходит через несколько разных слоёв, после чего в конце преобразуется в конкретное число — релевантность согласно локальной модели.

Distributed модели

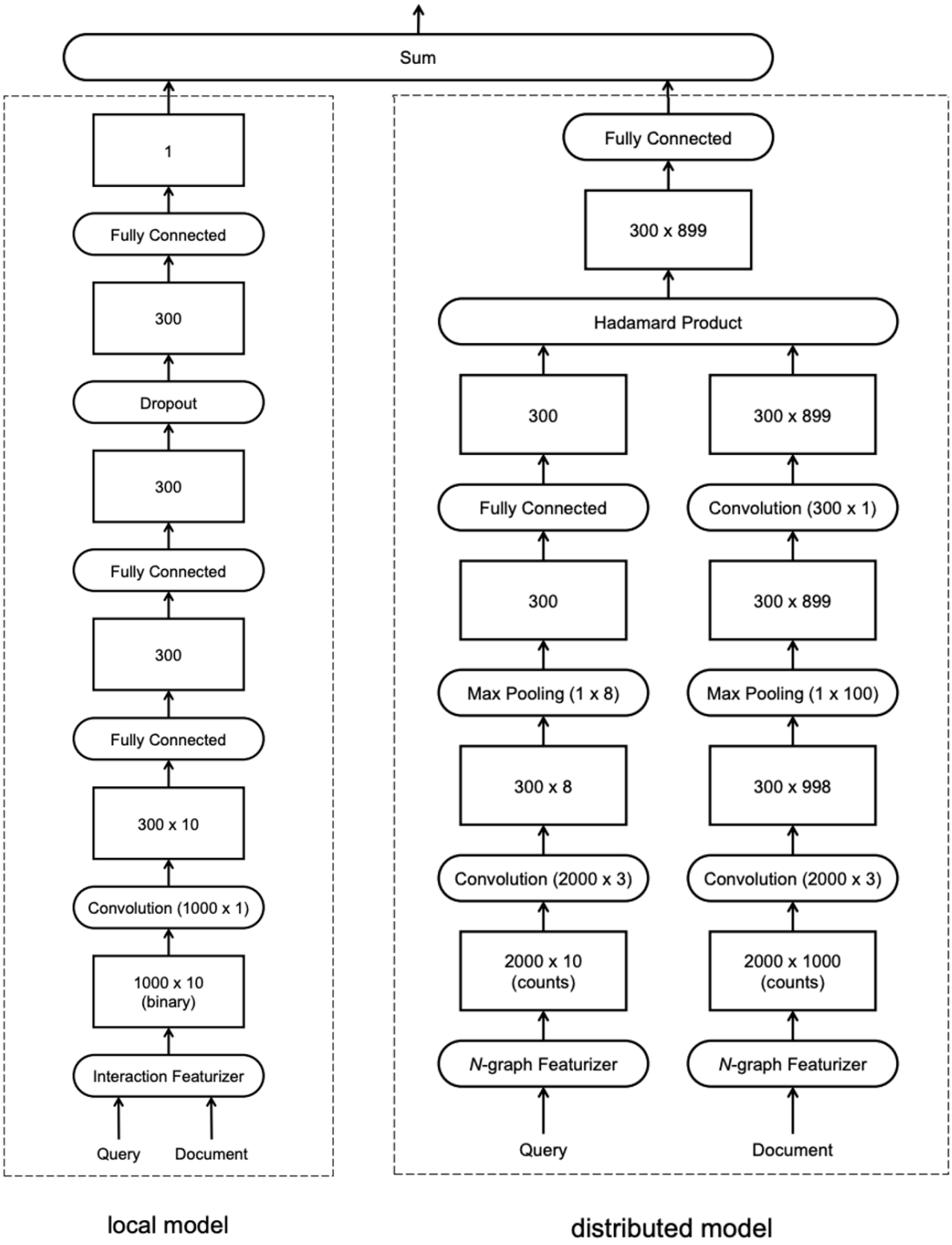
В distributed моделях слова и текст имеют векторное представление в некотором латентном пространстве — эмбеддингах. Релевантность представлена как функция от схожести эмбеддингов запросов и документов. Это может быть крайне полезно, когда хочется учитывать не точные соответствия слов, а соответствие по смыслу. Например, вас вполне устроит статья *“однорукий бандит: вред или польза”* в качестве результата для запроса *“зависимость от игральных автоматов”.* Между этими текстовыми последовательностями нет одинаковых слов, и всё же человек (и хорошо обученные эмбеддинги) способны уловить схожесть, релевантность.

Distributed модель обычно переводит и запрос, и документ в вектор с помощью как бы двух ветвей нейронной сети, которые могут иметь общие веса (вспомните аналогию с сиамскими сетями), и после этого применяется некоторая функция близости, или сопоставления, матчинга, между этими векторами. Альтернативное название такой модели — Representation-focused, т.е. модель, сконцентрированная на представлении смысла подаваемых на вход текстовых данных.

Distributed модель, как мы уже выяснили по схеме, по отдельности векторизует запрос и документ, переводя слова в осмысленные эмбеддинги, и после обработки разными слоями вроде свёрток и пуллингов выдаёт финальные вектора, мера схожести между которыми и служит оценкой релевантности с точки зрения distributed модели. Эта мера схожести может быть не только какой-то простой вроде косинусного расстояния, а обучаемой, т.е. представлять собой набор слоёв, на вход которым подаются признаки векторизованных запросов и документов.

Проблема эмбеддингов в Distributed моделях в том, что как бы хорошо не обучали эти векторные представления слов, вектора для каких-то параметров, артикулов или штрихкодов, отличающихся на 1, 2 или, может, даже 3 символа всё равно будут максимально похожи в векторном пространстве друг на друга, и модель, оперирующая эмбеддингами, будет указывать на их абсолютное соответствие, хоть это и разные модели товара. Так, артикулы `qwerty1234` и `qwerty1235` будут считаться одинаковыми. Таким образом, для решения задачи матчинга товаров в ритейле стоить отдавать предпочтение либо локальным моделям, либо продвинутым альтернативам, о которых и пойдёт речь далее. Отличной идеей является объединение этих методов, чтобы получить хорошо работающую систему.

DUET-NDRM



Архитектура DUET-NDRM

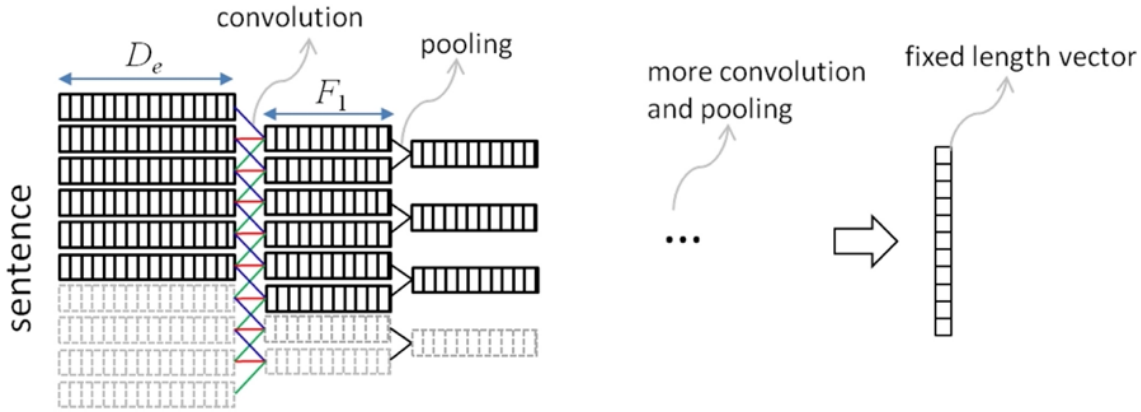
Банально сложив оценки релевантности от двух моделей, получаем финальное предсказание для использования в ранжировании. Такой ансамбль называется DUET, или DUET-NDRM (Neural Document Ranking Model).

Важно отметить, что мы ничего не говорили про метод обучения такой сети. Никаких особенностей тут нет, интерфейс решения остаётся всё тем же: у нас есть нейросеть, которая на вход принимает документ и запрос, а на выходе выдаёт релеватность. Можно применить любой метод, хоть PairWise, хоть ListWise, и добавить к ним учёт позиционного смещения. Мы изменили только метод взаимодействия с данными — то, как формируется информация для вынесения вердикта о релевантности.

> CNNs for matching natural language sentences

Давайте же разберёмся с новым методом и научимся применять его эффективно.

Допустим, что на вход поступает текстовая последовательность некоторой длины. Для каждого слова у нас есть векторное представление в пространстве определённой размерности, все вектора одинаково широкие. Их мы можем упорядочить согласно расположению слов в самом документе и представить в виде некоторой матрицы, ширина которой совпадает с размерностью эмбеддинга, а высота — с некоторой константой, которую мы заранее определим. Если слов не хватает до этой заранее определённой константной длины, то можно записывать нулевые вектора (серые пунктирные ячейки). Если слов больше, чем нужно, то их просто отсекают, так как большая часть информации всё равно содержится в начале текста. Над полученной матрицей можно проделать операцию свёртки по аналогии с моделями, предназначенными для обработки изображений.



Предложенная архитектура сети

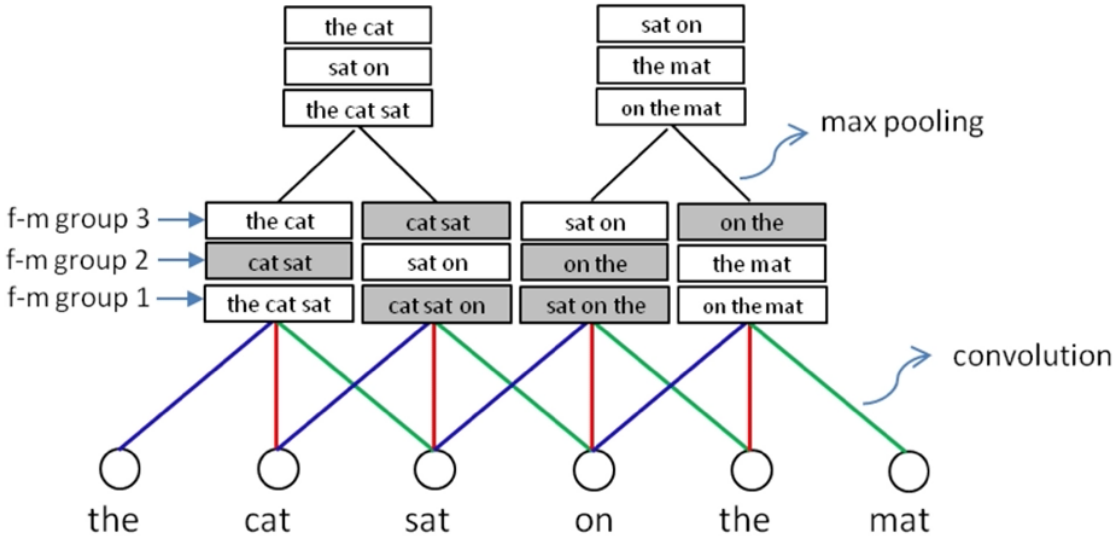
В качестве примера будем использовать свёртку поверх скользящего окна ширины 3. Пусть дано некоторое предложение:

"the cat sat on the mat"

Линии на рисунке отвечают за веса в свёртке. Синяя линия — это вес в выходном значении свёртки для первого объекта в окне, красная — для второго и зелёная — для третьего. Для простоты интерпретации положим, что все веса свёртки для первого фильтра равны $1/3$. Тогда значение первой размерности новополученного эмбеддинга для первых трёх слов, который объединяет весь их смысл, просто равно трём первым словам, с точки зрения смысла полученного вектора. Меняя эти веса, например, зануляя первый и приравнивая второй и третий к $1/2$, можно управлять извлекаемым смыслом с помощью свёртки. Второй так называемый фильтр свёртки с озвученными параметрами проигнорирует первое слово и вберёт в себя смысл второго и третьего. Этот процесс можно повторять множество раз, каждый с разными весами, таким образом формируя новый эмбеддинг, который несёт в себе уже не смысл отдельных слов, но словосочетаний.

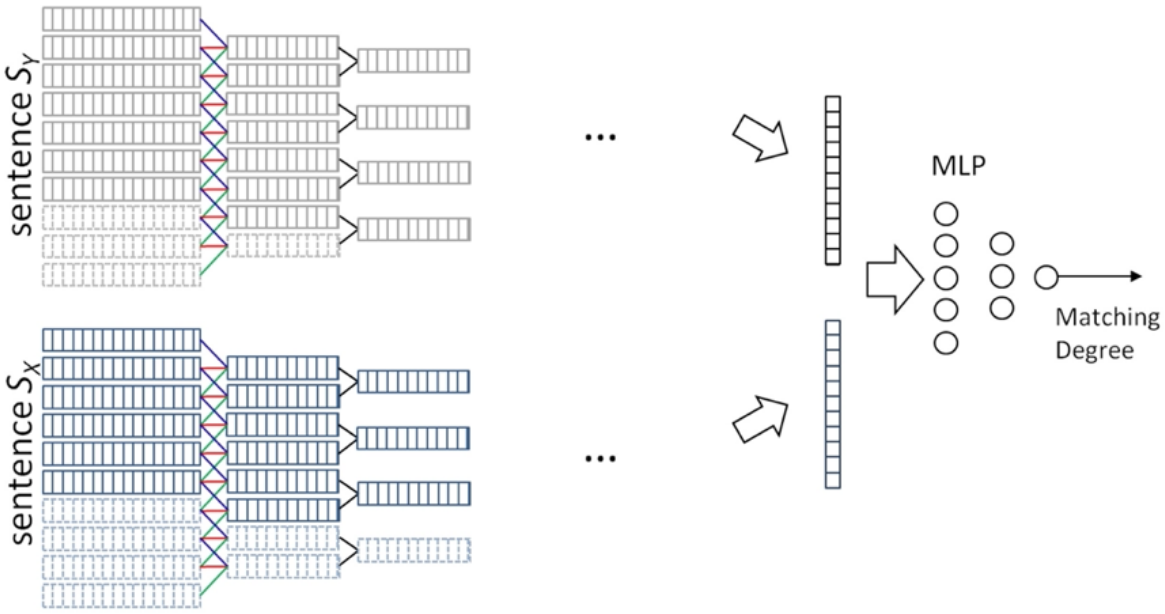
Обратите внимание, что если эту операцию повторять, то будет производиться манипулирование не над отдельными словами, а над смещением смысла словосочетаний, и чем большее число раз строится новое представление, тем абстрактнее смысл, тем он более общий.

Над этими эмбеддингами можно применить операцию пуллинга, в частности MaxPooling. Его смысл прост — полученные с помощью свёрток эмбеддинги делятся на группы последовательно идущих, и из каждой группы берутся признаки с максимальной активацией (значением), а остальные игнорируются. Таким образом, на следующий уровень проходят, если можно провести такую аналогию, только самые сильновыраженные абстракции.



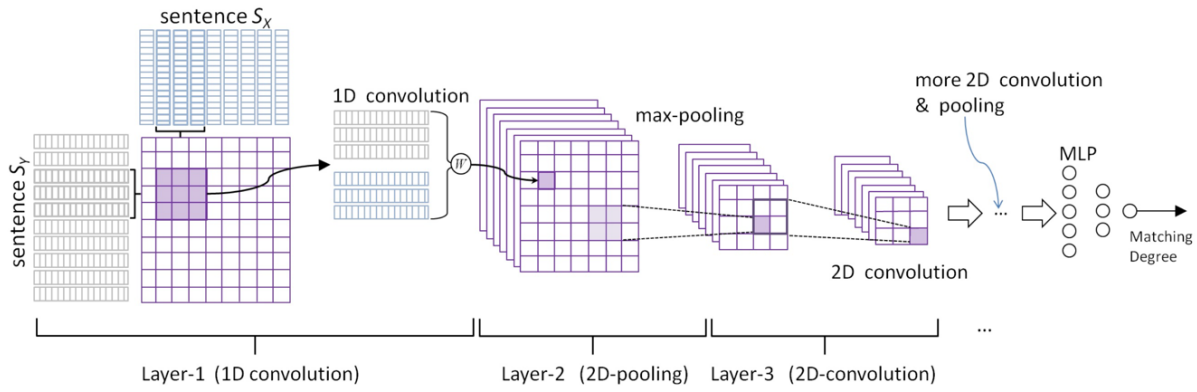
Пример работы свёрток

В конечном итоге после итеративного повторения вышеописанных действий получается единый вектор, характеризующий весь текст. Такую векторизацию можно применить по отдельности ко всем документам в некоторой базе и при поступлении запроса векторизовать и его, после чего осуществлять обычное ранжирование или матчинг поверх векторов. Это классический пример representation-based модели, где признаки извлекаются по отдельности из запроса и документа. Очевидный плюс подхода в том, что векторизацию можно предварительно произвести для огромного корпуса документов и не тратить время во время работы пайплайна. Если для конкретной задачи важна скорость, то пойдёт архитектура с названием ARC-I. Необходимо снова отметить, что мы говорим только про изменение сети и к ней всё ещё применимы абсолютно все разобранные методы обучения, любая функция потерь.



Архитектура ARC-I

Также существует и вторая версия этой архитектуры: ARC-II. Отличительная особенность этой версии модели заключается в том, что признаки извлекаются из всех возможных комбинаций «троек» последовательных слов между запросом и документом, а затем по общей матрице взаимодействия оценивается релевантность и строится ранжирование. Здесь основное отличие заключается в методе формирования эмбеддингов поверх троек слов. Берутся 3 вектора для 3 слов запроса, 3 вектора для документов, затем они просто конкатенируются и умножаются на некоторый вектор весов W конкретной свертки. На выходе получается скаляр, который записывается в матрицу — карту признаков, или, как её называют, feature map. Позиция в этой матрице зависит, как понятно из картинки, от индексов слов, которые обрабатываются в конкретный момент времени. Далее можно применять на этих feature maps свёртки и пуллинги, прямо как в задачах компьютерного зрения, и получать выходной вектор. Затем уже из него можно получать предсказание релевантности или матчинга. Поскольку здесь каждая тройка слов запроса аккуратно сравнивается с каждой тройкой слов документа, то метод работает дольше, и тут ничего нельзя заранее вычислить для оптимизации. Однако учёт связей слов каждый-с-каждым позволяет делать более высококачественное решение с точки зрения метрик.



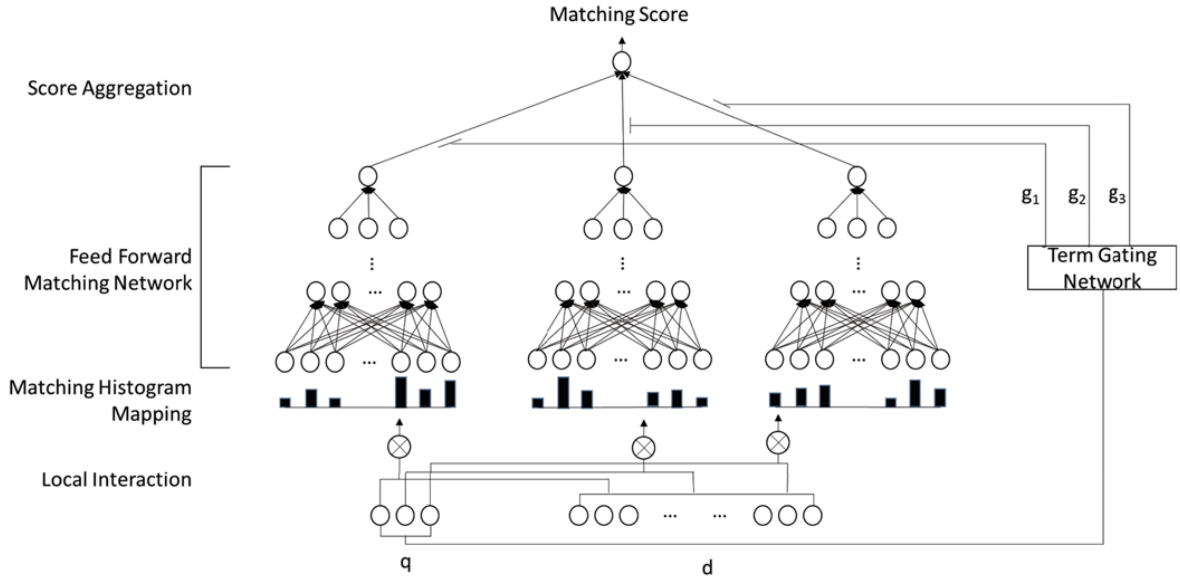
Архитектура ARC-II

Далее мы раскроем идею попарного сравнения слов между запросом и документом.

> Deep relevance matching model for ad-hoc retrieval

Следующий рассматриваемый метод называется Deep Relevance Mathcing Model (DRMM). Это достаточно популярный метод ранжирования, который активно использовался в 2016-2018 гг. чуть ли не повсеместно, чаще с какими-то доработками и дополнениями.

Архитектура представлена ниже, она несколько громоздка, поэтому разберёмся последовательно: снизу вверх. Всё ещё есть запрос и документ, каждый из которых разбит на слова, и словам в соответствие ставятся некоторые эмбеддинги. В примере, как можно видеть напротив надписи Local Interaction, 3 слова у запроса q, и куда больше — у документа d.



Архитектура DRMM

Рассмотрим внимательно следующий уровень, Matching Histogram Mapping. Общая идея следующая: оценивать схожесть каждого слова из запроса со всеми словами из документа.

$$z_i^{(0)} = h\left(w_i^{(q)} \otimes d\right), \quad i = 1, \ldots, M$$

В предложенной формуле d — это вектора слов документа, w_i^q — вектор i-го слова в запросе. Между этими эмбеддингами считается косинусная схожесть (cosine similarity), т.е. угол между этими векторами, и затем строится некоторая функция распределения h. Косинусная схожесть, или cosine similarity, это мера от -1 до 1, где 1 означает, что векторы сонаправлены, а -1 — противоположно направлены. 0 означает, что векторы ортогональны, т.е. перпендикулярны друг другу. Чем больше эта мера, тем более похожи вектора, а значит и смысл слов.

Давайте на примере разберём, что это за функция h и как выглядит значение Matching Histogram. Пусть у нас будет следующий запрос: "автомобиль". А в документе встречаются следующие слова: "автомобиль", "аренда", "грузовик", "удар", "штраф", "дорога". Посчитаем между словами и запросом косинусную схожесть:

слово в документе	cosine similarity
автомобиль	1.0
аренда	0.2
грузовик	0.7

удар	0.3
штраф	-0.1
дорога	0.1

Для полученных косинусных схожестей можно построить гистограмму: в нашем примере взяли 5 бинов, поэтому получим следующее: $[0, 1, 3, 1, 1]$. Таким образом мы строим такую гистограмму для каждого слова в запросе и получаем M гистограмм (M — количество слов в запросе).

В данном примере получается 3 таких распределения, так как было три слова в запросе Q (они изображены в нижней части картинки). Каждая из гистограмм подаётся в нейронную сеть, Matching Network, и все эти сети имеют общие веса и обучаются одновременно. На выходе каждой сети единственное число - релеватность, вернее некоторая составляющая итоговой релевантности. Затем просто применяется взвешенное суммирование этих оценок релевантности, где веса задаются некоторой Term Gating Network. Вес каждого слова рассчитывается по следующей формуле:

$$g_i = \frac{\exp\left(\boldsymbol{w}_g \boldsymbol{x}_i^{(q)}\right)}{\sum_{j=1}^{M} \exp\left(\boldsymbol{w}_g \boldsymbol{x}_j^{(q)}\right)}, \quad i = 1, \ldots, M$$

Это уже хорошо знакомый нам SoftMax, логитом которого является произведение вектора признаков для конкретного слова в запросе x_i на обучаемый вектор весов w_g, т.е. веса gating-части алгоритма. Эти признаки x_i могут быть как исходными эмбеддингами слов, так и простым значением IDF, с которым вы знакомы из TF-IDF. Логика здесь такова, что более редкие слова имеют больший вес для ранжирования, это понятно интуитивно. Может показаться, что IDF крайне примитивен и прост, но в реальных экспериментах было установлено, что на определённых наборах данных метрики выходят выше именно с таким подходом вместо взвешивания на эмбеддинги.

К сожалению, при использовании такого подхода нет возможности обучать эмбеддинги, поскольку шаг с гистограммами и разделением на бины не позволяет производить дифференцирование. Таким образом, подразумевается, что у вас есть какие то вектора, какие-то эмбеддинги, которые получены на словаре естественного языка, на данных с интернета, или, может, вы просто скачали готовые вектора, например word2vec или GloVe, и не меняете эти вектора по ходу решения задачи, не подгоняете их под конкретную проблему. И это хочется исправить, чтобы получать качественные эмбеддинги текстов. Для этого как раз используется модификация — K-NRM.

> End-to-End neural ad-hoc ranking with kernel pooling

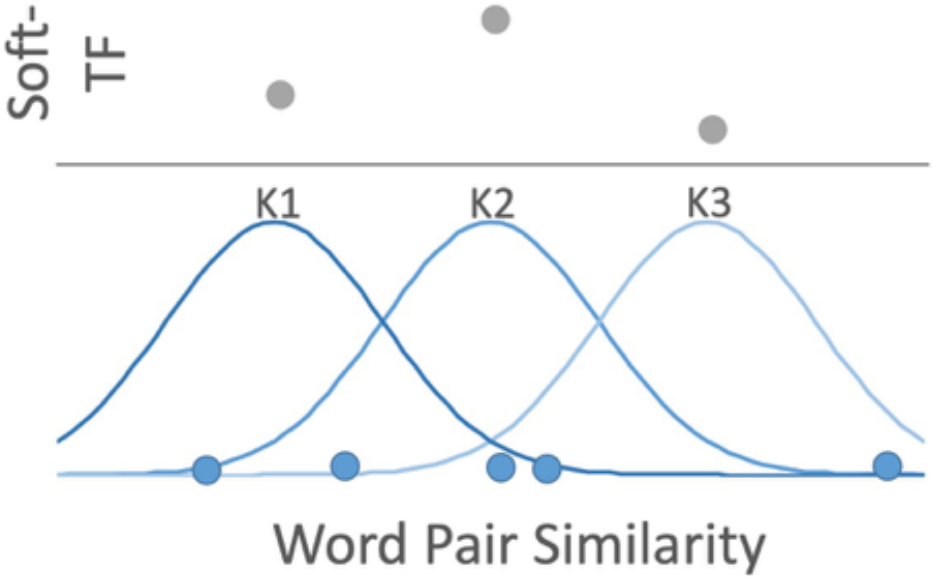
Архитектура K-NRM (Kernel Neural Ranking Model) похожа на DRMM, здесь так же учитываются взаимосвязи каждый-с-каждым относительно слов запроса и документа. В качестве меры близости используется cosine similarity, однако полученная матрица сворачивается с помощью ядер, или кернелов (kernels).

$$M_{ij} = \cos\left(\vec{v}_{t_i^q}, \vec{v}_{t_j^d}\right)$$

Ядро представляет собой RBF-kernel (Radial Basis Function), который вам может быть знаком из машинного обучения. Каждое такое ядро — это функция от целой строки в матрице взаимодействий M_{ij} между запросом и документом, или, иными словами, взаимодействие одного слова из запроса со всеми словами документа за раз. Это ядро имеет два параметра, которые не обучаются, а фиксируются при инициализации: среднее μ_k и параметр дисперсии σ_k, который стоит в знаменателе. Эта сигма обычно одинакова для всех ядер (кернелов).

$$K_k\left(M_i\right) = \sum_j \exp\left(-\frac{\left(M_{ij} - \mu_k\right)^2}{2\sigma_k^2}\right)$$

Давайте разберёмся, как эта формула работает и в чём её смысл. За счёт среднего, параметра μ_k, который уникален для каждого ядра, происходит привязка к конкретному значению в исходной матрице схожести, т.е. к значению similarity, с которым сравнивается каждый элемент матрицы M_{ij} попарных взаимодействий эмбеддингов слов запроса и документа. Каждое ядро можно представить как волну, где по оси OY отмечен вес, с которым объект, или токен в документе, входит в конкретную область.



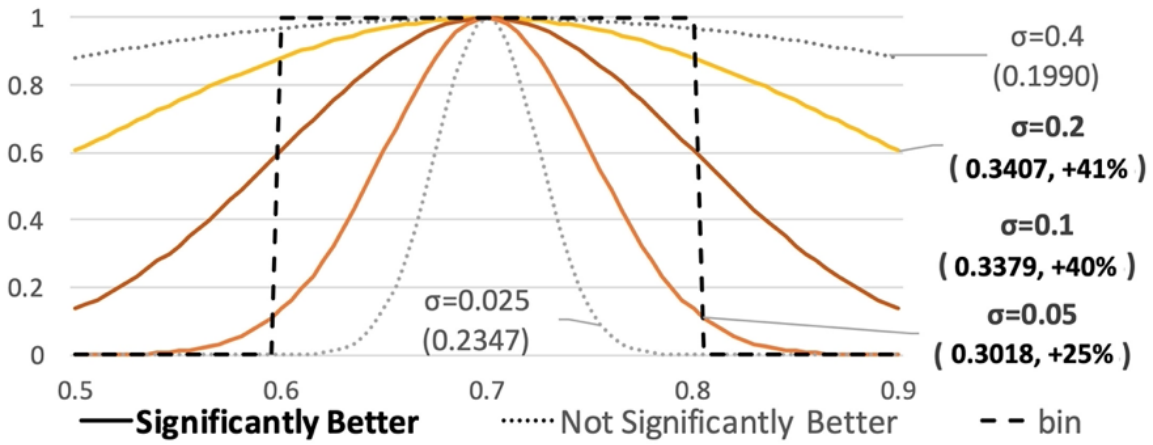
Рассмотрим пример на картинке выше — в нижней части синими точками отмечены документы. Чем левее точка, тем меньше, вплоть до -1, cosine similarity между словом в документе, которое представлено точкой, и конкретным зафиксированным словом в запросе. Серая точка вверху символизирует выходное значение конкретного ядра. Видно, что второй кернел имеет наибольшее значение. Это обусловлено тем, что близко к его центру находится целых два объекта, т.е. они наиболее похожи именно на центр волны, которая описывает второе ядро. Далее идёт кернел с номером один, у которого тоже целых две точки, однако они чуть дальше от центра кернела $K1$, нежели такие точки у кернела $K2$. У третьего кернела совсем всё грустно — точки далеки от его центра, а значит, в это ядро практически ничего не попадает и его значение минимально. По сути это сглаженный и дифференцируемый аналог бинов, на которые разбивали отрезок $[-1, 1]$ — ось cosine similarity при расчёте гистограмм.

Для того чтобы симулировать несколько бинов, т.е. оценивать плотность распределения метрики косинусной схожести слов запроса и документа в разных частях интервала, берётся несколько ядер с некоторым шагом между их средним, т.е. μ_k. Допустим, для отрезка $[-1, 1]$ будет 11 ядер, у первого среднее в 1, далее 0.9, 0.7 и так далее до -0.9. Тогда каждое из ядер даст скалярное числовое значение на выходе, характеризующее схожесть запроса и документа в некоторой окрестности определенной cosine similarity, и объединив все ядра для описания полной картины можно получить вектор. В случае с 11 ядрами, или кернелами, получаем вектор из 11 признаков, который несёт в

себе ту же информацию, что и Matching Histogram в DRMM. Собственно, такой вектор и получится $\vec{K}\left(M_i\right) = \{K_1\left(M_i\right), \ldots, K_K\left(M_i\right)\}$

Ещё раз обращаем внимание на то, что значение ядра тем больше, чем кучнее и ближе к среднему значению μ_k конкретно этого ядра находятся результирующие косинусные схожести всех слов документа и конкретного слова запроса (аналогия с бинами в DRMM).

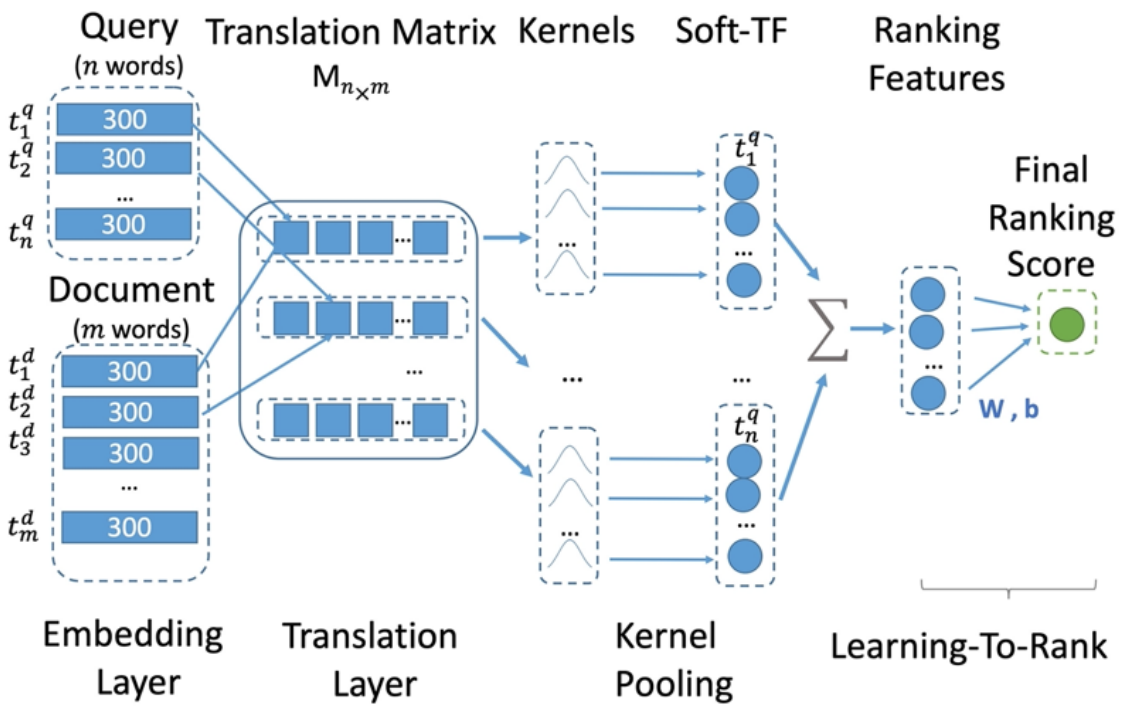
Для полноты картины осталось осознать, на что влияет параметр σ_k, т.е. дисперсия. По сути она задаёт ширину бина, в который попадают слова из документа. На изображении ниже показаны различные параметры σ_k для одного кернела со средним, т.е. параметром μ_k, равным 0.7. Чем больше σ_k, тем шире бин и тем на большую окрестность обращает внимание этот кернел. Под каждой из σ_k справа вынесено изменение метрики MAP (Mean Average Precision) на некотором наборе данных. Это демонстрирует, что подбор σ_k для конкретного набора данных в рамках вашей задачи — важная процедура, так как ширина бина влияет на качество модели. Вообще такие ядра можно назвать сглаженной версией TF (Term Frequency), или SOFT-TF, поскольку логика работы с бинами сохраняется и происходит только нестрогое присваивание конкретного слова из документа определённому бину. При этом само слово учитывается в каждом бине, но в большинстве из них с околонулевым весом. По оси OY на графике отражён вес, с которым объект войдёт в сумму: чем дальше от центра ядра, тем меньше вес. Если все объекты находятся далеко от кернела, то, с одной стороны, можно сказать, что этот бин пустой и в методе DRMM было бы записано 0, а с другой — в soft-версии у нас сумма околонулевых значений, т.е. величина незначительная. И наоборот, если в бине или в центре кернела объектов много, то и значение активации на этом ядре будет высоким.



Демонстрация поведения формы ядра для различных значениях дисперсии

Когда данная операция применена ко всем строчкам таблицы, на выходе получается N векторов размера K, где K — количество ядер. Всё, что осталось сделать — сначала просуммировать по N, т.е. по всем словам из запроса, логарифмы значений, полученных на ядрах $\varphi(M) = \sum_{i=1}^{n} \log \vec{K}\left(M_i\right)$, таким образом собрав их в единый вектор размера K, а затем простым преобразованием вроде $f(q, d) = \tanh\left(w^T \varphi(M) + b\right)$ получить выходное значение релевантности. Тут имеем вектор весов w и смещение b, это обучаемые параметры.

Легко заметить, что если убрать эмбеддинги из модели, то вообще у модели обучаемых параметров всего 12 (это очень мало): 11 в ядрах и 1 для смещения! Вместо многомиллионных моделей обучается всего 12 параметров. Может показаться странным, но в K-NRM из-за замены жестких бинов на сглаженные версии на кернелы появляется возможность дифференцировать весь алгоритм, считать градиенты и прокидывать их на входные эмбеддинги, по которым оценивается косинусная схожесть. Это даёт удивительную возможность дообучать эмбеддинги слов под свою задачу, т.е. в прямом смысле переопределять значения слов и словосочетаний исходя из потребностей. При этом никто не мешает инициализировать эмбеддинги уже готовыми обученными параметрами, скачанными из интернета, и просто дообучать модель под свои нужды. Это позволяет невероятно быстро обучать модель K-NRM под свои данные, так как практически все веса у нас хорошо инициализированы, и нужно их лишь слегка изменить.



Архитектура K-NRM

В практике ML это называется Transfer Learning, когда знания с одной задачи, в данном случае общеязыкового моделирования, переносятся на другую. В применении, т.е. при инференсе эта модель крайне быстра, потому что тут из операций самое сложное — расчёт матрицы cosine similarity от каждого-к-каждому на входных эмбеддингах. Все остальные операции не требуют затрат и очень просты.

> Резюме

Давайте подведём итоги:

- При работе с данными, собранными от пользователей, важно учитывать смещённость(например, позиционную).
- Для улучшения ранжирования наиболее релевантных документов(топа) можно применять продвинутые listwise-подходы, при которых оценка релевантности каждого объекта зависит от всего топа ранжирования (интеракция каждый-с-каждым).
- В рамках конкретной задачи важно учитывать как локальные, так и контекстные особенности в тексте. Поэтому нужны продвинутые архитектурные решения, позволяющие при сравнении запроса и документа учитывать каждое отдельное слово.