NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS
FACULTY OF SOCIAL SCIENCE

MEGA PAPER

PZDataAnalysis

Political and Physical Science

Student
Anastasia Uspenskaya
Arina Puchkova
Sergey Zakharov
Igor Kolesnikov

Spasibo chto est'
Evgeniy Sokolov

Moscow, 2021

Содержание

1 Опишите две компоненты, из которых состоит perceptual loss (стилевую и контентную). Как можно использовать этот функционал, чтобы делать перенос стиля?

Для контекста:

Мы хотим понять есть ли содержательные отличия между картинками, а не просто ли у нас одинаковые пиксели. Именно поэтому мы не используем Евклидово расстояние, например.

Идея:

используем предобученную сетку VGG для определения различий между изображениями.

1.1 Content

$$L^l_{content}(A, B) = \sum_{i,j}(A^l_{i,j} - B^l_{i,j})^2 \tag{1}$$

В 1 следующие обозначения:

- A^ℓ_{ij} — значение i-го фильтра на j-й позиции в слое ℓ для изображения A
- Чем дальше слой, тем меньше он чувствителен к небольшим изменениям в изображении
- Можем попробовать градиентным спуском из картинки с белым шумом найти изображение, минимизирующее $L^\ell_{\text{content}}(A, B)$

Рис. 1: Нотация для первой компоненты Perceptual loss

Считаем для слоя l, он конкретный. Прогоняю картинку A в VGG беру слой нейросети l, там i канал (свертка i) и смотрю ее значение на позиции j

(позиция здесь просто индекс для всех возможных положений фильтра на картинке).

Чем больше слой, тем более Loss будет отражать содержательные различия, а не какие-то непонятные мелкие признаки.

Возьмем значение этого лосса и будем ее минимизировать по B (пиксель), то есть мы будем менять изображение, чтобы добиться наименьшего лосса.

Лучше брать последние, если нам интересно содержимое. Картинка B меняется попиксельно.

1.2 Style

$$L^l_{style}(A, B) = \sum_{i,j}(G^l_{i,j}(A) - G^l_{i,j}(B))^2 \tag{2}$$

Чтобы посчитать G для A, Мы берем два канала i и j, затем мы рассматриваем k-ю позицию фильтра на указанных каналах. Перемножаем выходы с филтьтров и суммируем по всем k.

$$G^l_{i,j}(A) = \sum_k A^l_{i,k} \times A^l_{j,k} \tag{3}$$

Что по сути происходит в 3: эта штука показывает то, насколько перекликаются (насколько похожи) i и j канал с точки зрения выходов фильтра, проецируемеого на картинку A.

Обобщение всего лосса стилевого: он требует, чтобы корреляции каналов между собой (в нашем случае i,j) совпадали для картинки A и картинки B.

Затем предлагается взять таким образом полученные лосс для отдельных слоев l и просуммировать их с определенными весами. (В этом отличие

от контентного лосса, там мы считали для одного слоя, а тут мы складываем по всем слоям).

$$L_{style}(A, B) = \sum_{l=0}^{L} w_l L^l_{style}(A, B) \tag{4}$$

Как это понимать:

Допустим, что фильтр 1 детектирует человека (то есть там, тем больше выход, тем больше изображение похоже на человеческое), а фильтр 2 - синий цвет детектирует. Если стилевая картинка в синей гамме, то тогда корреляция между фильтрами будет высокая, поскольку тогда скалярное произведение будет большим.

Итоговый лосс будет выглядить следующим образом:

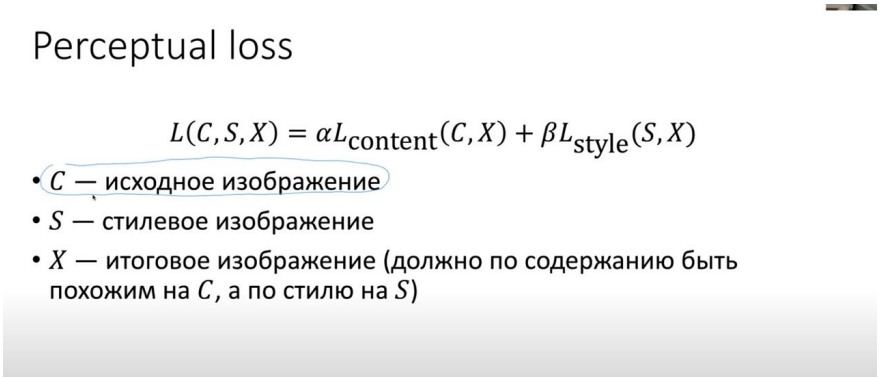

Рис. 2: Полный лосс

Нам дается "исходное"изображение, то есть которое мы пытаемся переделать (до этого это был белый шум). Хотим сделать похожим на S. X-результат. Как и с B, мы постоянно обновляем X для уменьшения лосса. a и β - гиперпараметры модели. Минимизируем по X.

2 Как можно сделать перенос стиля более быстрым, если стилевое изображение известно и не будет меняться?

В чем вообще проблема: минимизация Perceptual Loss напрямую для каких-то адекватных размеров изображений занимает большое время (порядок времени — 4 минуты для изображения 1024x1024). Это с точки зрения пользователя плохая история.

В том случае, если стилевое изображение зафиксировано, мы можем не просто напрямую минимизировать Perceptual Loss, а обучить нейросеть на выборке таким образом, чтобы она принимала на вход изображение x, а ее выход минимизировал Perceptual Loss.

Изначально было: $L(x, S, \hat{y}) \to \min\limits_{\hat{y}}$

Теперь стало: $L(x, S, a_\theta(x)) \to \min\limits_{\theta}$

Здесь x — входное изображение (пытаемся приблизиться к нему по контенту), S — стилевое изображение (пытаемся приблизиться к нему по стилю), которое мы и фиксируем, a_θ — модель с параметрами θ, принимающая на вход входное изображение и выдающее стилизованное.

То есть фактически мы подбираем модель, которая на выходе бы давала картинку, хорошую с точки зрения Perceptual Loss. Вместо прямой оптимизации мы пытаемся обучить какое-то преобразование.

Подобный подход не позволит нам выиграть времени на обучении (все равно придется обучать сетку, а это требует время), но зато затем, с обученной сеткой, создание картинки с нужным стилем будет значительно быстрее.

Однако, если мы поменяем стилевое изображение, то сетку придется обучать новую (что в целом логично).

3 Что в вариационном автокодировщике выдают кодировщик и декодировщик?

Общая идея:

Обычный автокодировщик кодировал нам как-то картинку в вектора. Этот вектор являлся "характеристиками"этого изображения.

Хотим чтобы характеристики были не просто какими-то числовыми значениями, но вероятностными распределениями. Этим мы делаем две вещи: включаем в модель неопределенность (пример: плачущий мальчик и Мона Лиза: насчет первого мы точно знаем, что он не улыбается, его распределение будет сдвинуто, а само распределение прижато к среднему, в тоже самое время насчет Мона Лизы мы до конца не понимаем, улыбается она или нет, и раздвигая границы распределения "улыбки"мы этот момент учитываем).

Задача:

Хотим построить векторное представление картинок $\mathbb{R}^d$, но каждая отдельная картинка соответствует не отдельному числу, но распределению определенному в этом векторном пространстве. Следствие: картинка будет описываться не отдельным числом, но некоторой средней и дисперсией нормального распределения. Получившиееся представление есть набор из d распределений со своими средними и дисперсиями.

Что мы делаем:

Сэмплируем вектор Z из распределения для картинки x, построеного для конкретной картинки. После этого мой декодировщик берет z и превращает его в $\bar{x}$. Добиваемся того, чтобы раскодированная картинка была, как можно ближе к x.

3.1 Кодировщик

Кодировщик представляет картинку в виде набора распределений, которые описываются средним и дисперсией (во время обучения используется логарифм дисперсии).

3.2 Декодировщик

Из построенного распределения сэмплируем значения и декодировщик выдает нам картинку, которая похожа на оригинальную.

4 4

5 В чём состоит идея трюка репараметризации (reparametrization trick)?

Кодировщик выдает нам μ и σ и мы по ним генерируем какой-то вектор z из этого распределения. Проблема: непонятно, как посчитать производную производную выхода по μ и σ, поскольку там происходит вероятностный процесс. Вместо этого сделаем так:

Кодировщик выдает нам μ и σ (это конкретные детерминированные числа), а еще мы будем генерировать стандартное нормальное число ε. И чтобы получить конкретное представление z мы сделаем $z = \mu + \sigma \odot \varepsilon$ (здесь $\odot$ — поэлементное умножение). То есть фактически мы разделяем случайность (запихиваем ее в ε), при этом делая ее без параметров (и никакие градиенты туда не нужны), и выдачу параметров кодировщиком (см. рисунок 3 в качестве иллюстрации).

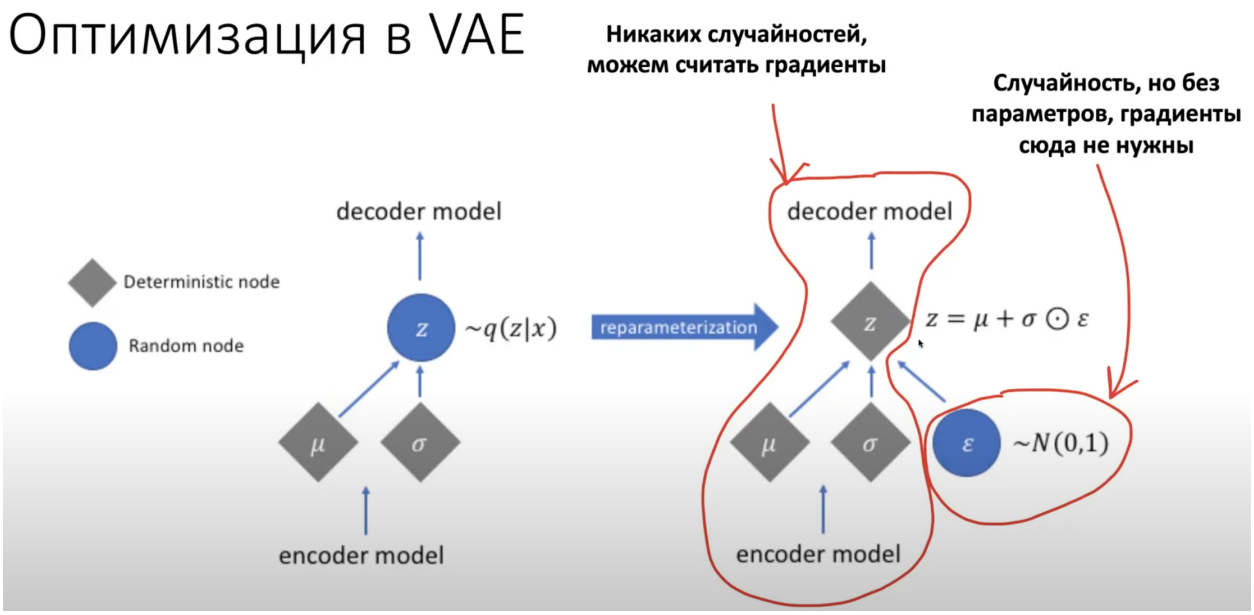

Рис. 3: Трюк репараметризации (слева без него, справа с ним)

6 6

7 Что такое затухание градиентов в GAN? Покажите это математически.

Затухание градиентов — одна из проблем GAN-ов. Предположим, что мы имеем ситуацию, похожую на изображенную на рисунке 4. В таком случае, как видно, дискриминатор может провести линию так, чтобы идеально (или в целом почти идеально, сути дела не меняет) разделить два класса. Это здорово, но не очень.

Если классы идеально разделились дискриминатором, то $D(x_i) = 1$ для любого x_i в X, т.е. для всех объектов первого класса он вернул 1. Для всех же сгенерированных объектов $D(\hat{x}_j) = D(G(z_j)) = 0$ для любого z_j в Z.

Вспомним, как выглядит функция потерь:

$$L(D, G) = -\frac{1}{n} \sum_{x_i \in X} \log D(x_i) - \frac{1}{n} \sum_{z_j \in Z} \log[1 - D(G(z_j))] \tag{5}$$

Подставим получившиеся значения в функцию потерь 5:

$$L(D, G) = -\frac{1}{n} \sum_{x_i \in X} \log 1 - \frac{1}{n} \sum_{z_j \in Z} \log[1 - 0] = 0 = \text{const} \tag{6}$$

Константа в том смысле, что нет зависимости от весов генератора или дискриминатора. Это значит, что градиент функции потерь оказывается равным нулю, т.е. ни дискриминатор, ни генератор более не обучаются, поскольку веса не обновляются. Ну понятно, что картинка эта далека от того, чего мы на самом деле хотим, так что это проблема.

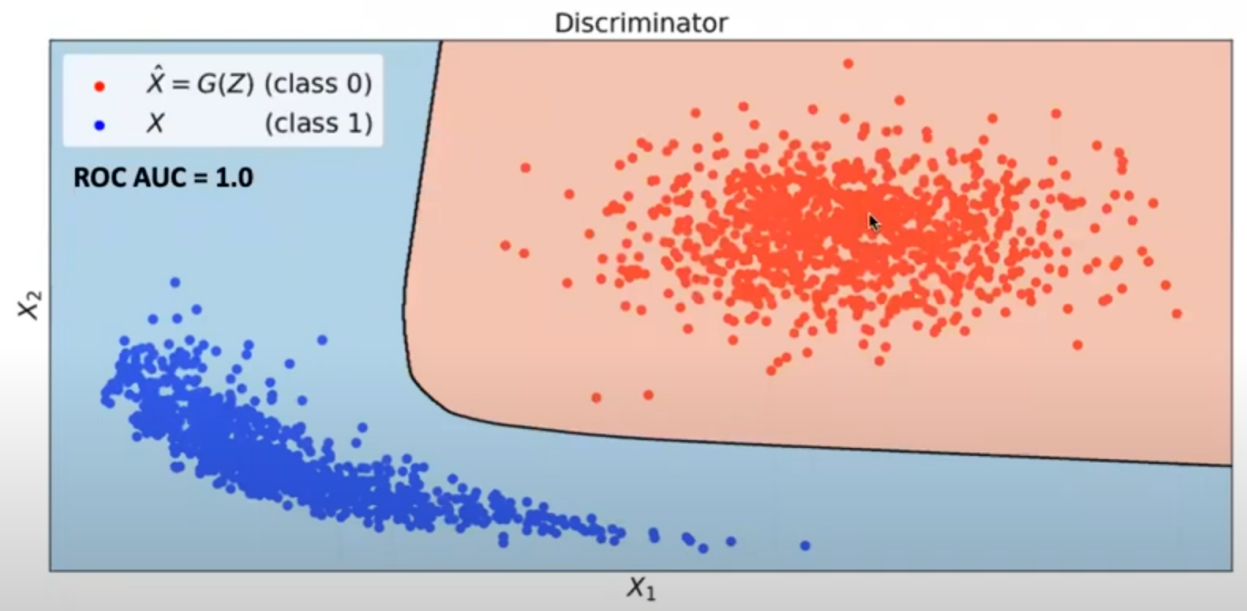

Рис. 4: Ситуация, возникшая на некоторой итерации обучения GAN и ведущая к затуханию градиентов

8 Запишите теорему о замене переменных для функции плотности распределения случайной величины. Что значит каждое обозначение в ней?

Теорема (о замене переменных): *Пусть даны* $p_z(z)$ *и* $z = f(x)$, *тогда* $p_x(x)$ *находится следующим образом:*

$$p_x(x_i) = p_z(f(x_i)) \left| \det \frac{\partial f(x_i)}{\partial x_i} \right| \tag{7}$$

где указан определитель матрицы первых производных, которая записывается как:

$$\frac{\partial f(x_i)}{\partial x_i} = \begin{pmatrix} \frac{\partial f(x_i)_1}{\partial x_{i1}} & \cdots & \frac{\partial f(x_i)_1}{\partial x_{in}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_i)_m}{\partial x_{i1}} & \cdots & \frac{\partial f(x_i)_m}{\partial x_{in}} \end{pmatrix} \tag{8}$$

Здесь $p_z(z)$ — распределение z, $p_x(x)$ — функция распределения x, $z = f(x)$ — функция, переводящая x в z.

Если говорить на пальцах, то этот модуль определителя в 7 — отношение объема ∂z к новому объему ∂x.

Про матрицу первых производных: т.к. x, $z = f(x)$ — некоторые многомерные векторы, то логично, что у них есть различные компоненты, индексы которых и указываются в матрице первых производных (то есть например левая верхняя компонента это производная первой компоненты $f(x_i)$ по первой компоненте x_i).

9 Запишите функцию преобразования для Real-NVP нормализационного потока. Как посчитать ее Якобиан? Как выглядит обратная функция?

Предпосылка: и x, и z имеют размерность D.

9.1 Функция преобразования

$$z = f(x) = \begin{cases} z_{1:d} = x_{1:d} \\ z_{d+1:D} = x_{d+1:D} \odot \exp[s(x_{1:d})] + t(x_{1:d}) \end{cases} \tag{9}$$

Здесь:

- $z_{1:d}$ — первые d компонент вектора z;
- $s(x_{1:d})$ и $t(x_{1:d})$ — нейронные сети с d входами и $(D-d)$ выходами;
- $\odot$ — поэлементное умножение.

9.2 Якобиан

Матрица первых производных нижнетреугольная и выглядит как-то так:

$$\frac{\partial f(x)}{\partial x} = \begin{pmatrix} \mathbb{I}_d & 0 \\ \dfrac{\partial z_{1:d}}{\partial x_{1:d}} & \operatorname{diag}\{\exp[s(x_{1:d})]\} \end{pmatrix} \tag{10}$$

Но нас интересует на сама матрица, а ее определитель, а он для нижнетреугольной матрицы записывается просто как:

$$\left|\det \frac{\partial f(x)}{\partial x}\right| = \exp\left[\sum_{j=d+1}^{d} s(x_{1:d})\right] \tag{11}$$

9.3 Обратная функция

Тут тривиально:

$$x = f^{-1}(z) = \begin{cases} x_{1:d} = z_{1:d} \\ x_{d+1:D} = [z_{d+1:D} - t(z_{1:d})] \odot \exp[-s(z_{1:d})] \end{cases} \tag{12}$$

10 10

11 11

12 Что такое MuLaw-кодирование, для чего оно нужно и как оно вычисляется?

В одном взятом сэмпле обычно находится 16 бит. Это означает, что у нас 2^{16} значений. Нам приходится делать softmax на 2^{16} значений. Очень много получается, хотим сделать поменьше.

Рассматриваемый метод позволяет ужать значения амплитуд, сохранив по максимуму первоначальную информацию.

Интуиция: человеческое ухо слышит в лог шкале: мы хорошо различаем звуки при низких амплитудах и плохо различаем при высоких. Нам надо хранить низкие амплитуды в высоком разрешении, а высокие амплитуды в низком, ибо мы и так их особо не слышим.

Реализация идеи хороша видна на следующем графике 5: здесь показано, как звуки с разными амплитудами преобразуются согласно данному кодированию

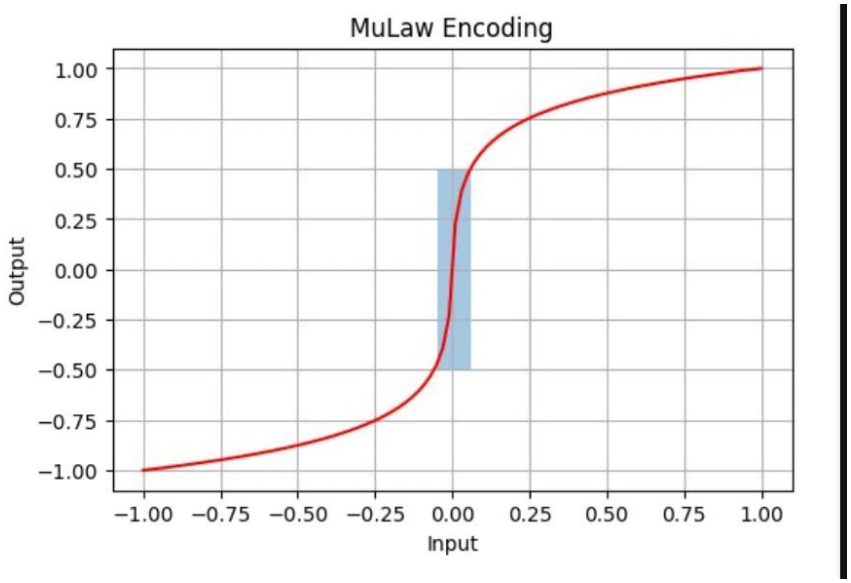

Рис. 5: Визуализация Mu law Encoding

Как можно увидеть, от -0.1 до 0.1 мы очень компактно храним входные частоты, но чем дальше амплитуда, тем больше у нас становится шаг(скорость функции возрастае), и соответственно в тем меньшем разрешении мы храним звуки с указанной амплитудой.

Гиперпараметром считается количество конечных значений, в которые мы хотим перевести все наши амплитуды.

Формула:

$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)} \tag{13}$$

μ - это просто параметр сжатия (обычно он вот такой - 255).

x- нормализованное число, которое подвергается сжатию.

13 13

14 14

15 15

16 Как устроены user-based рекомендации?

Как понять, что пользователю может понравиться товар? Первый вариант — поискать похожих на него пользователей и посмотреть, что нравится им; также можно поискать товары, похожие на те, которые этот пользователь уже покупал. Методы коллаборативной фильтрации строят рекомендации для пользователя на основе похожестей между пользователями и товарами, которые похожие пользователи выбирают.

Существует два подхода к определению сходства между пользователями: Memory-based и Модель со скрытыми переменными

Частью memory based является user-based. Его отличие от item-based и другие подробности про рекомендации смотрите здесь https://github.com/hse-ds/iad-applied-ds/blob/master/2020/lectures/lecture01-recommender.pdf

Два пользователя похожи, если они ставят товарам одинаковые оценки. Рассмотрим двух пользователей u и v Обозначим через I_{uv} множество товаров i, для которых известны оценки обоих пользователей:

$$I_{uv} = \{i \in I \mid \exists r_{ui} \ \& \ \exists r_{vi}\}.$$

Тогда сходство двух данных пользователей можно вычислить через корреляцию Пирсона:

$$w_{uv} = \frac{\sum_{i \in I_{uv}} (r_{ui} - \bar{r}_u)(r_{vi} - \bar{r}_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - \bar{r}_u)^2} \sqrt{\sum_{i \in I_{uv}} (r_{vi} - \bar{r}_v)^2}},$$

где $\bar{r}_u$ и $\bar{r}_v$ — средние рейтинги пользователей по множеству товаров I_{uv}.

Чтобы вычислять сходства между товарами i и j, введём множество пользователей U_{ij}, для которых известны рейтинги этих товаров:

$$U_{ij} = \{u \in U \mid \exists r_{ui} \ \& \ \exists r_{uj}\}.$$

Тогда сходство двух данных товаров можно вычислить через корреляцию Пирсона:

$$w_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)(r_{uj} - \bar{r}_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)^2} \sqrt{\sum_{u \in U_{ij}} (r_{uj} - \bar{r}_j)^2}},$$

где $\bar{r}_i$ и $\bar{r}_j$ — средние рейтинги товаров по множеству пользователей U_{ij}. Отметим, что существуют и другие способы вычисления похожестей — например, можно вычислять скалярные произведения между векторами рейтингов двух товаров.

Мы научились вычислять сходства товаров и пользователей — разберём теперь несколько способов определения товаров, которые стоит рекомендовать пользователю u_0. В подходе на основе сходств пользователей (user-based collaborative filtering) определяется множество $U(u_0)$ пользователей, похожих на данного:

$$U(u_0) = \{v \in U \mid w_{u_0 v} > \alpha\}.$$

После этого для каждого товара вычисляется, как часто он покупался пользователями из $U(u_0)$:

$$p_i = \frac{|\{u \in U(u_0) \mid \exists r_{ui}\}|}{|U(u_0)|}.$$

Пользователю рекомендуются k товаров с наибольшими значениями p_i. Данный подход позволяет строить рекомендации, если для данного пользователя найдутся похожие. Если же пользователь является нетипичным, то подобрать что-либо не получится.

Также существует подход на основе сходств товаров (item-based collaborative filtering). В нём определяется множество товаров, похожих на те, которые интересовали данного пользователя:

$$I(u_0) = \{i \in I \mid \exists r_{u_0 i_0}, w_{i_0 i} > \alpha\}.$$

Затем для каждого товара из этого множества вычисляется его сходство с пользователем:

$$p_i = \max_{i_0 : \exists r_{u_0 i_0}} w_{i_0 i}.$$

Пользователю рекомендуются k товаров с наибольшими значениями p_i. Даже если пользователь нетипичный, то данный подход может найти товары, похожие на интересные ему — и для этого необязательно иметь пользователя со схожими интересами.

Рис. 6: Конспект Соколова прошлый год

17 17

18 18

19 Как обычно устроена рекомендательная система? Опишите шаги отбора кандидатов, ранжирования, переранжирования. Приведите примеры, как каждый из них может быть устроен

В рекомендательной системе может участвовать очень большое количество то- варов. При каждом посещении пользователем веб-страницы, где есть блок рекомендаций, необходимо выдать ему k наиболее подходящих товаров, причём достаточно быстро (пользователь не может ждать минуту, пока загрузится страница). В хоро- шей рекомендательной системе участвуют сотни признаков — их вычисление для каждого товара, а затем ещё и применение ко всем товарам градиентного бустинга или графа вычислений вряд ли получится успеть сделать за 1 секунду. Из-за этого рекомендательные системы работают в несколько этапов: обычно всё начинается с отбора кандидатов, где быстрая модель выбирает небольшое количество (тысячи или десятки тысяч) товаров, а затем только для этих товаров вычисляется полный на- бор признаков и применяется полноценная модель. В качестве быстрой модели может выступать линейная модель на нескольких самых важных признаках или, например, простая коллаборативная модель.

19.1 Отбор кандидатов

Отбор кандидатов это про то, как выбрать подмножество предметов, которые пользователю интересны.

- те же предметы, производители, что и в истории пользователя

- самое популярное сейчас
- Матричное разложение: пользователь - исполнитель. $r_{user,singer} \approx < p_{user}, p_{singer} >$
- на основе сходства: p_{user}, q_{item} - эмбеддинги (например, из LFM). Ищем K items с максимальным значением скалярного произведения $< p_{user}, q_{item} >$. Ищу вещи, которые ближе всего к пользователю.

О ранжировании смотрите здесь https://developers.google.com/machine-learning/recommendation/dnn/re-ranking

и здесь https://github.com/hse-ds/iad-applied-ds/blob/master/2020/lectures/lecture03-recommender.pdf

После отбора кандидатов следует ранжирование

Переранжирование - помимо максимизации вероятности клика, мы хотим учесть бизнес-требования. Например, мы не хотим показать видео от одного и того же автора. Мы пересортировываем наш изначальное ранжирование. Так мы учитываем дополнительные требования к нашей рекомендациии.