Санкт-Петербургский государственный университет

Математика и компьютерные науки

Отчёт по научно-исследовательской работе (6 семестр)

Алгоритмы для реализации рекомендательных систем

Выполнила:

Майстер Анастасия Владимировна 20.Б13-мм

Научный Руководитель:

кандидат физико-математических наук

доцент кафедры статистического моделирования

Шпилёв Пётр Валерьевич

Кафедра статистического моделирования

Санкт-Петербург

2023

Постановка цели и задач

Данная работа является прямым продолжением работы, начатой в предыдущем семестре. Её **целью** является изучение и анализ алгоритмов для рекомендательных систем, в связи с чем на текущий семестр были поставлены следующие **задачи**:

1. изучение и реализация новых алгоритмов с применением их на практике;
2. анализ полученных результатов;
3. изучение методов матричной факторизации и их использования для проектирования рекомендательных систем.

В работе используются те же наборы данных, а именно **transactions.csv** и **ratings.csv**. Для удобства некоторые файлы с реализациями были изменены или перемещены относительно прошлого семестра.
Каждая глава отчёта посвящена отдельному алгоритму.

1 Ассоциативные правила

1.1 Охват

Алгоритм построения рекомендаций при помощи ассоциативных правил был реализован в прошлом семестре. Он использует данные о транзакциях и идею «покупательской корзины», предлагая пользователю те фильмы, которые часто встречались в корзинах вместе с уже просмотренными им.

Данными для алгоритма являются два столбца — 'element_uid' и 'user_uid', то есть универсальные идентификаторы элемента и пользователя и сам факт взаимодействия пользователя с элементом (будь то покупка, аренда или просмотр по подписке).

	element_uid	user_uid	consumption_mode	ts	watched_time	device_type	device_manufacturer
0	3336	5177	S	4.430518e+07	4282	0	50
1	481	593316	S	4.430518e+07	2989	0	11
2	4128	262355	S	4.430518e+07	833	0	50
3	6272	74296	S	4.430518e+07	2530	0	99
4	5543	340623	P	4.430518e+07	6282	0	50
5	236	332814	S	4.430518e+07	3109	0	50

Рис. 1: Первые 6 строк **transactions.csv**

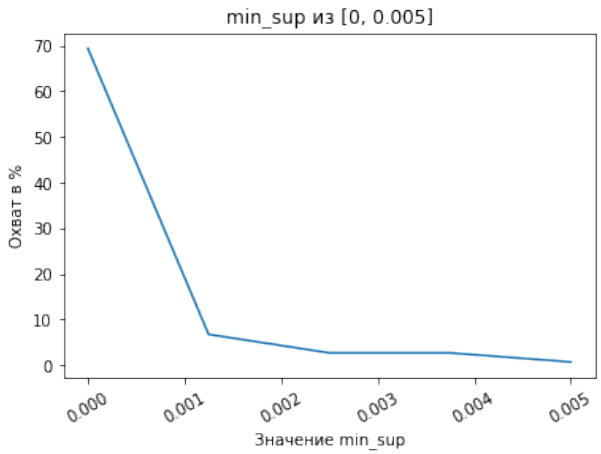
Ключевой параметр алгоритма — минимальная поддержка (min_sup), который отвечает за то, какие правила мы считаем релевантными: говорят, что корзина поддерживает правило $A \Longrightarrow B$ (здесь A, B — наши элементы из корзины), если в ней встречаются оба этих элемента. В этой ситуации корзина соответствует конкретному пользователю, параметр min_sup является вещественным, а правило принимается, если его поддерживает число корзин, большее минимального необходимого (которое равно $min\_sup \cdot N$, где N — число всех корзин, то есть число всех пользователей в датасете).

Таким образом, увеличивая параметр min_sup, мы уменьшаем количество правил, и следовательно, возможных элементов для рекомендации. Это может быть crucial(существенно, критично), если мы пытаемся рекомендовать что-то пользователю со специфичными вкусами ('black ships' в теории рекомендательных систем): например, тому, чьим аккаунтом пользуется несколько людей. Причём такая ситуация возникает нередко, и в корзине могут соседствовать мультфильмы, которые включают детям утром выходного дня, и боевики, которые родители смотрят вечером.

Уменьшая значение min_sup, мы ослабляем требования на поддержку, что может привести к появлению нерелевантных рекомендаций, когда любителям боевиков, например, предложат посмотреть мультфильм.

Есть, конечно, один важный момент: мы, как пользователи, хотим, чтобы рекомендации нас иногда удивляли и открывали нам что-то новое, потому что наши вкусы меняются с течением времени. Но это новое должно не слишком сильно отличаться от наших текущих предпочтений. Поэтому вопрос выбора значения min_sup — вопрос баланса между качеством и количеством.

Мерой оценки количества является **охват**, он показывает, какой процент фильмов попадает в рекомендации. Ниже можно наблюдать график зависимости охвата от значения min_sup. Охват рассчитывался для 6-ти значений порога минимальной поддержки на промежутке $[0, 0.005]$ путём составления рекомендаций для каждого пользователя и добавления их в общий список рекомендованных элементов (**5** лучших элементов от каждого). При этом использовались только первые 30000 строк датасета для экономии времени.

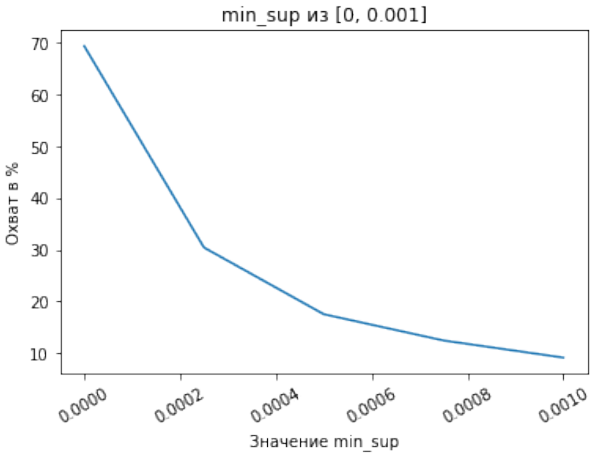


По графику видно, что охват заметно убывает при увеличении порога поддержки. При этом, избавляясь от условия необходимости наличия определённой поддержки (иначе говоря, полагая порог равным нулю), мы получаем охват, равный почти 70%, то есть 70% от всех фильмов попадают в рекомендации, что можно считать хорошим результатом, потому что это свидетельствует о разнообразии рекомендаций.

Рассмотрим подробнее промежуток $[0, 0.001]$ (график строим аналогично), так как именно на нём происходит резкое падение значения.

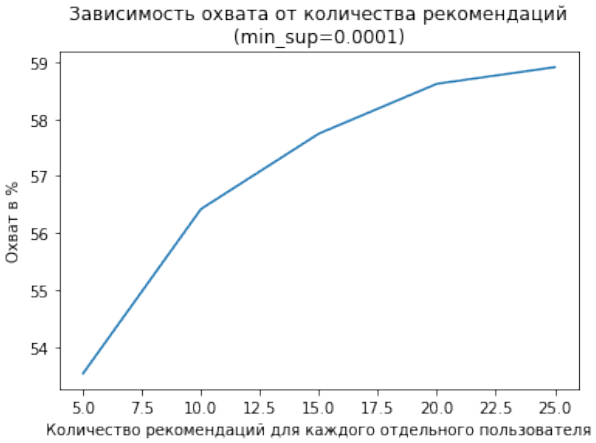
Опираясь на этот график, можно считать $min\_sup = 0.0001$ оптимальным значением в нашей ситуации, так как в этом случае хотя бы половина фильмов попадает в рекомендации. А так как в реальной задаче для составления рекомендаций используется несколько алгоритмов (например, те, о которых будет сказано далее), охват получается даже большим.

Интересно также узнать, повлияет ли на охват увеличение количества элементов, которые от каждого пользователя мы добавляем в список. То есть увеличится ли охват, если мы будем брать не 5, а 10 лучших элементов, например. График ниже показывает, что есть увеличение, но небольшое, поэтому вполне можно считать 10 элементов оптимальным значением. Таким образом, при $min\_sup = 0.0001$ и 10-ти рекомендациях получаем охват 56%.



1.2 Частота попадания в рекомендации

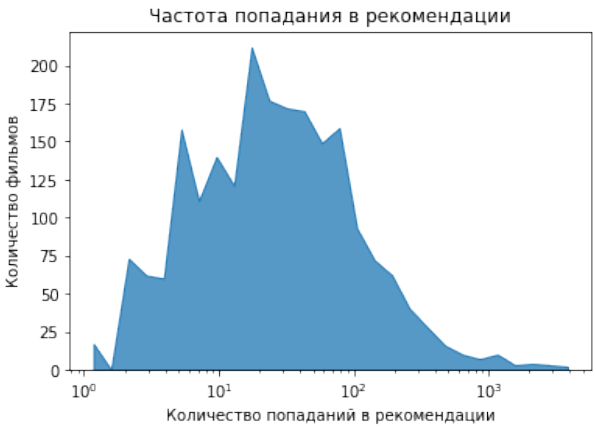
Как уже было упомянуто, охват позволяет нам что-то говорить говорить о разнообразии рекомендуемых элементов, но не определяет его полностью. В частности, он не учитывает, как часто рекомендуется тот или иной элемент. А эта информация могла бы дать новые сведения о пользователях (у похожих пользователей похожие рекомендации) и самих элементах (можно выделить наиболее и наименее популярные фильмы и попытаться соотнести это с жанром, средней оценкой фильма и прочим).



Приведённый ниже график отражает зависимость количества фильмов от количества попаданий в рекомендации. Изучая его, можно заметить, что среди всех рекомендаций пользователям большинство фильмов (более 200) встречается около 20-ти раз — иначе говоря, попадает в рекомендации 20-ти пользователям. В среднем каждый фильм рекомендуется около 75-ти раз, но есть и экстремальные значения — есть как фильмы, которые рекомендуются всего нескольким из 18-ти с лишним тысяч пользователей, так и те, что рекомендуются каждому 20-му.

1.3 Время работы

Не менее важным показателем рекомендательного алгоритма является время, затрачиваемое им на составление рекомендаций. В силу того что рекомендации для каждого отдельного пользователя (при фиксированном их числе) на выходе определяются как срез списка уже отсортированных рекомендаций, время, необходимое для 5-ти или 10-ти рекомендаций, почти не будет отличаться.



Экспериментальным путём (с выбранными выше параметрами и данными) было обнаружено, что 10 рекомендаций для одного пользователя этот алгоритм генерирует за примерно 10^{-4} секунды, а все предварительные вычисления выполняет за секунду (сразу для всех пользователей). Поэтому при подобном порядке размерности входных данных (30000 строк) и небольшом потоке посетителей он может работать в онлайн-режиме, составляя рекомендации в тот момент, когда пользователь заходит на сайт, и выполняя общие вычисления по мере поступления новых данных с учётом текущей нагрузки на систему.

2 Коллаборативная фильтрация

Коллаборативная фильтрация — один из методов построения персонализированных рекомендаций для пользователя путём прогнозирования оценок для ещё не оценённых им элементов. Она делится на три типа: коллаборативная фильтрация на основе соседства (или коллаборативная фильтрация в окрестности), на основе модели и гибридная, которая совмещает результаты первых двух.

Коллаборативная фильтрация в окрестности использует два **подхода** (Рис. 2):
(a) по пользователям (user-based),
(b) по элементам (item-based).

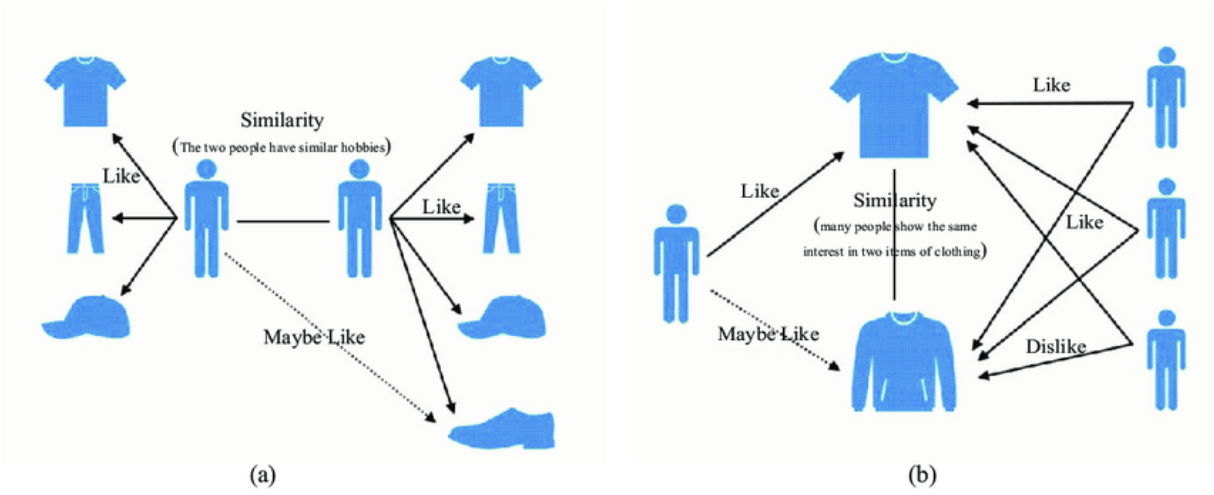


Рис. 2: Два подхода к фильтрации в окрестности [1]

В первом случае мы находим пользователей, схожих с активным, то есть тем, для которого хотим построить рекомендации, а затем рекомендуем ему элементы, которые понравились схожим пользователям. Во втором находим элементы, схожие с теми, которые пользователю понравились ранее.

Работа алгоритма состоит их нескольких **этапов**:

1. В качестве входных данных он принимает матрицу оценок, строки которой соответствуют пользователям, а столбцы — элементам.
2. Затем, в зависимости от выбранного подхода, вычисляются коэффициенты сходства между активным пользователем и всеми остальными пользователями (user-based подход) или между элементом, для которого выполняется прогноз, и всеми остальными элементами (item-based подход).
3. После этого список пользователей или список элементов, соответственно, сортируется по степени сходства и выбирается окрестность, то есть множество пользователей/элементов, степень сходства которых с нашим пользователем/элементом превышает некоторый порог.
4. И, наконец, вычисляется оценка.

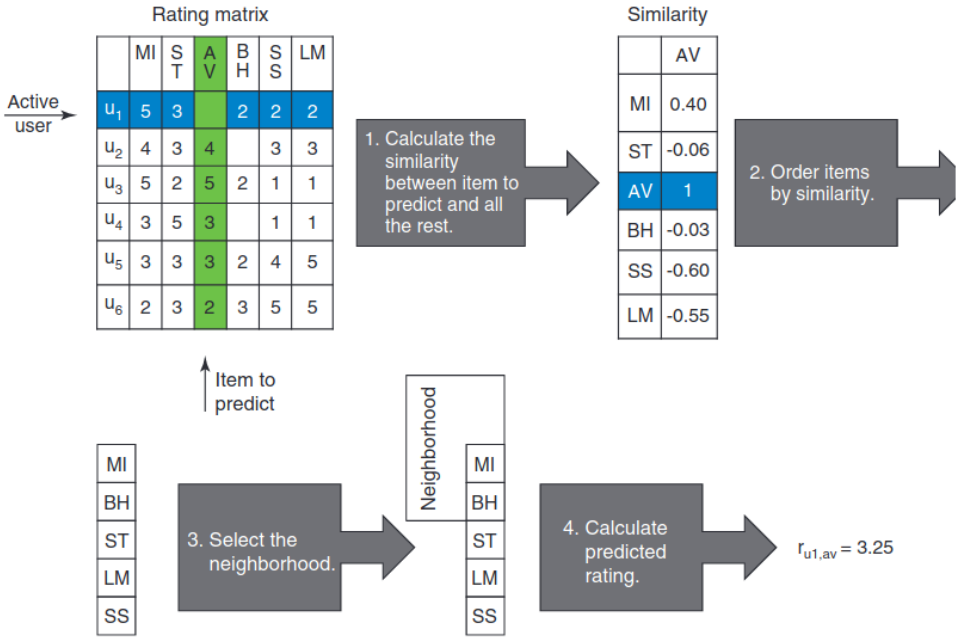


Рис. 3: Этапы фильтрации по элементам [2]

Выбор подхода определяется соотношением размерностей входных данных. В данной работе был использован именно item-based подход, так как количество уникальных пользователей в датасете почти в 14 раз превышает количество уникальных элементов, и следовательно, на каждый фильм приходится в среднем больше оценок, чем ставит среднестатистический пользователь. Поэтому схожесть элементов определяется «точнее».

В качестве входных данных использовался датасет **ratings.csv**, а точнее, первые три его столбца (без временной метки 'ts').

В процессе его анализа обнаружилось, что есть фильмы с одной единственной оценкой на весь датасет. Их пришлось удалить из общего списка фильмов, так как оценки от одного пользователя недостаточно, чтобы как-то охарактеризовать фильм (мнение одного человека очень субъективно) и спрогнозировать оценки других пользователей, которые ему, предположительно, могли бы быть даны. Также были исключены данные о нулевой оценке (явно ошибочные, так как используется шкала от 1 до 10). Опять же для экономии времени с помощью стратифицированной по элементам выборки были отобраны 30000 строк для обучения алгоритма, так как из-за большой разреженности матрицы оценок при случайном выборе строк рекомендации получатся неинформативными.

	user_uid	element_uid	rating	ts
0	571252	1364	10	4.430517e+07
1	63140	3037	10	4.430514e+07
2	443817	4363	8	4.430514e+07
3	359870	1364	10	4.430506e+07
4	359870	3578	9	4.430506e+07
5	557663	1918	10	4.430505e+07

Рис. 4: Первые 6 строк **ratings.csv**

В итоговом варианте датасета оказалось около 21-й тысячи пользователей и 4-х тысяч элементов, то есть заполненность матрицы составила меньше 36%.

Сходство между элементами определялось через косинусную меру:

$$sim(i,j) = \frac{\sum\limits_{u \in U} nr_{u,i} \cdot nr_{u,j}}{\sqrt{\sum\limits_{u \in U} (nr_{u,i})^2} \cdot \sqrt{\sum\limits_{u \in U} (nr_{u,j})^2}},$$

где $nr_{u,i} = r_{u,i} - \overline{r}_u$ — нормализованная оценка пользователем u элемента i,
$\overline{r}_u = \frac{\sum\limits_{i \in P} r_{u,i}}{|P|}$ — средняя из всех оценок пользователя u,
$r_{u,i}$ — оценка, которую пользователь u поставил элементу i,
P — множество всех элементов, оценённых пользователем u,
U — множество всех пользователей

Пустые ячейки матрицы были заполнены нулями.

Затем была вычислена матрица перекрытия — квадратная матрица размерности $M \times M$ (M — количество всех фильмов в датасете), значение ячейки (i,j) которой соответствует количеству пользователей, оценивших оба фильма i и j ($\forall i,j$). И задан один из двух параметров алгоритма — минимальное необходимое перекрытие для включения элемента в окрестность (min_overlap), — для которого было выбрано значение 3. Этот параметр отвечает за то, насколько мы доверяем значению коэффициента сходства между элементами: вполне могла произойти ситуация, что двое пользователей оценили только два некоторых фильма. Тогда, как нетрудно заметить, коэффициент сходства будет равен -1, какие бы оценки эти пользователи ни поставили. Если же перекрывающихся пользователей слишком много, итоговые рекомендации получатся слишком общими.

Окрестность прогнозируемого элемента определялась при помощи матрицы корреляции элементов, порогом было выбрано значение сходства 0.4. То есть все элементы, отобранные на предыдущем шаге, сходство которых с нашим оказывалось меньше, не учитывались при расчёте оценки. Большее значение порога давало бы более «надёжные» прогнозы, но в некоторых ситуациях могло привести к тому, что в окрестность не попали бы никакие элементы вообще.

После определения окрестности оценка расчитывалась по следующей формуле:

$$Pred(u,i) = \overline{r}_u + \frac{\sum\limits_{j \in S_i} sim(i,j) \cdot r_{u,j}}{\sum\limits_{j \in S_i} sim(i,j)},$$

где $\overline{r}_u = \frac{\sum\limits_{j \in P} r_{u,j}}{|P|}$ — средняя из всех оценок пользователя u,
$r_{u,j}$ — оценка пользователем u элемента j,
S_i — набор элементов в окрестности оценённых пользователем u

Элементы для рекомендации получались выбором топ-10 по прогнозируемым оценкам среди ещё не оценённых пользователем.

2.1 Важные наблюдения и комментарии

- Выбор окрестности можно было производить и другим способом. Например, для каждого элемента выбирать топ-N элементов по значению коэффициента сходства. Это решило бы проблему с пустой окрестностью, но в таком случае в неё могли бы попасть непохожие объекты (если фильм нельзя отнести к одному жанру, к примеру). Поэтому способ выбирается в зависимости от наших

потребностей: с чем можно смириться — с не очень точными рекомендациями или их отсутствием (в случае пустой окрестности оценка получается равной нулю).

- Прогнозы рассчитывались от средней оценки пользователя, так как разные люди по-разному оценивают свои впечатления и могут иметь привычку завышать или занижать оценку.

- На практике большинство прогнозов получились нулевыми из-за большой разреженности матрицы оценок. Также в процессе анализа тренировочного датасета (на 30000 строк) было выявлено, что в среднем каждый фильм имеет по 7 оценок, но у 2355 фильмов (то есть у более чем половины) оценок меньше трёх, что подтверждает график распределения фильмов по оценкам.

- Что касается **времени**, затрачиваемого данным алгоритмом, на предварительные действия с датасетом в среднем уходит меньше секунды, на непосредственную генерацию рекомендаций для пользователя — около 15-20-ти секунд.

- Понятно, что модель, выдающую нули в качестве прогнозов, нельзя назвать точной. Но интересно оценить её работу с точки зрения значений какой-нибудь **метрики**. Самыми популярными метриками качества регрессионных моделей являются среднеквадратичное отклонение (RMSE) или её квадрат (MSE) и средняя абсолютная ошибка (MAE). В данной работе этот и все последующие алгоритмы оценивались при помощи RMSE, так как оптимизация алгоритмов с точки зрения RMSE позволит всем пользователям получать хорошие результаты (большие погрешности в прогнозах даже нескольких оценок приведут к большому значению метрики).

  Ввиду разреженности матрицы оценок, было решено считать RMSE только для тех пользователей, которые поставили хотя бы 30 оценок (из более чем 4000-х тысяч), их оказалось 10 человек. Среднее значение метрики для них составило 6.97, то есть модель каждый раз ошибалась почти на 7 единиц.

- Таким образом, в подобной (базовой) реализации на выбранных данных алгоритм использовать нецелесообразно. Но его можно оптимизировать. Например,

не рассчитывать оценки для всех не оценённых пользователем фильмов, а выбрать 30 рекомендаций при помощи ассоциативных правил, рассчитать оценки для них и затем уже выбрать десять лучших. В этом случае, даже если спрогнозированные оценки окажутся нулевыми, мы сможем включить элементы в рекомендации.

3 SVD

Сингулярное разложение (SVD) — один из широко известных методов матричной факторизации, применяемый в таких задачах, как сжатие изображений, обработка сигналов и автоматизации производства.

SVD позволяет представить любую матрицу R размера $m \times n$ в виде произведения трёх матриц

$$R = U \cdot S \cdot V^T,$$

где U и V — две ортогональные матрицы размера $m \times r$ и $r \times n$, соответственно,
r — ранг матрицы R,
S — диагональная матрица размера $r \times r$, у которой на диагонали стоят сингулярные значения матрицы R.

Все диагональные элементы S неотрицательны и расположены по убыванию.

Основное свойство SVD-разложения, благодаря которому оно находит такое широкое применение, заключается в том, что с помощью SVD можно получить лучшую низкоранговую аппроксимацию исходной матрицы R — матрицу R_k $(k < r)$, такую что

$$R_k = U_k \cdot S_k \cdot V_k^T,$$

где $U_k = U[:, : k], \quad S_k = S[: k, : k], \quad V_k^T = V^T[: k, :]$.

В рекомендательных системах SVD используется следующим образом [3]:

1. Сначала разреженная матрица оценок R заполняется так, чтобы в ней не осталось пустых ячеек.
2. Далее она раскладывается в произведение матриц U, S и V.
3. Затем матрица S сокращается до нужной размерности k и из неё извлекается корень.
4. После чего вычисляются матрицы $U_k \cdot S_k^{1/2}$ и $S_k^{1/2} \cdot V_k^T$.

Прогнозируемая оценка пользователем элемента получается взятием соответствующей ячейки в матрице, являющейся произведением двух матриц с последнего шага (то есть в матрице R_k).

Для этих двух матриц в теории рекомендательных систем существует хорошая интерпретация. Говорят, что первая матрица показывает вкусы пользователей, а вторая — профили элементов. С точки зрения линейной алгебры, получается, что каждый пользователь и каждый элемент представляются вектором факторов, и можно считать, что вектор факторов для пользователя показывает, насколько пользователю нравится или не нравится тот или иной фактор, а вектор факторов элемента — насколько тот или иной фактор в элементе выражен [4].

3.1 О реализации

В качестве входных данных для этого алгоритма использовались те же данные, что и для алгоритма коллаборативной фильтрации. Аналогично были удалены све-

дения о нулевых оценках и не учитывались фильмы, оценённые менее чем тремя пользователями. Размер итогового датасета составил 30000 строк.

В данной работе пустые ячейки заполнялись усреднённой по всему датасету оценкой (которая составила 8.21 из 10-ти), так как заполнение нулём приводило к нулевым прогнозам. Благодаря этому удалось получить более разнообразные оценки, и, следовательно, появилась возможность отсортировать непросмотренные фильмы по предполагаемой оценке, что непосредственно использовалось для составления рекомендаций. Однако такой вариант не является единственным подходящим: перспективным может быть также заполнение средней оценкой по пользователю или средней оценкой по элементу.

В качестве k выбиралось минимальное такое значение, при котором сумма элементов на диагонали матрицы S_k составляла более 0.9 суммы элементов на диагонали матрицы S. В этом случае при приближении матрицы оценок R матрицей R_k сохранялось как минимум 90% исходной информации. Благодаря такому выбору k размерность матрицы S сократилась более чем в 9 раз.

Что касается **времени**, на предварительные вычисления, общие для всех пользователей, в среднем уходило около 10.5 минут (причём почти всё время — на SVD-разложение), на непосредственное построение рекомендаций для пользователя – менее секунды.

Также было посчитано **значение метрики** RMSE для пользователей, оценивших более 30-ти фильмов. На тренировочных данных она составила 0.38, на тестовых — 1.58. Что уже является неплохим результатом, по сравнению с коллаборативной фильтрацией. Но для многих пользователей прогнозы будут менее точными ввиду недостатка информации. В частности, для 10-ти случайных пользователей ошибка на тренировочных данных составила 0.84, а на тестовых — 1.93.

4 ALS

ALS (Alternating Least Squares) — ещё один метод матричной факторизации, который позволяет представить матрицу R в виде произведения двух матриц U и V, таких что

$$R \approx U^T V$$

Количество строк в матрицах U и V является параметром и называется числом латентных факторов (или рангом).

В контексте рекомендательных систем матрицу U называют матрицей пользователей, а матрицу V — матрицей элементов.

Они находятся как решение следующей задачи:

$$\arg\min_{U,V} \sum_{\{i,j|r_{i,j}\neq 0\}} (r_{i,j} - u_i^T v_j)^2 + \lambda \left(\sum_i n_{u_i} ||u_i||^2 + \sum_j n_{v_j} ||v_j||^2 \right) \boxed{=}$$

где $(R)_{i,j} = r_{i,j}$,
u_i — i-й столбец матрицы U,
v_i — i-й столбец матрицы V,
λ — параметр регуляризации,
n_{u_i} — число элементов, оценённых пользователем i,
n_{v_j} — число пользователей, оценивших элемент j.

$$\boxed{=} \arg\min_{U,V} f(U,V)$$

Общий алгоритм выглядит следующим образом [5]:

1. Сначала заполняется матрица V: первая строка — средней оценкой для соответствующего элемента, остальные — небольшими случайными числами.
2. Затем при фиксированной матрице V находится матрица U как решение задачи минимизации функции $f(U, V)$.
3. После чего аналогично находится матрица V при фиксированной матрице U.
4. Шаги 2 и 3 повторяются, пока не будет выполнен критерий останова.

Критерием останова может служить максимальное число итераций или условие, что значение RMSE на тестовом наборе уменьшилось менее чем на ε для некоторого заданного $\varepsilon > 0$.

Рекомендации для пользователя строятся аналогично SVD.

4.1 О реализации

В данной работе использовалась версия алгоритма из библиотеки MLlib PySpark и тот же набор данных с оценками пользователей. После аналогичной предобработки с помощью стратифицированной по элементам выборки было отобрано 130000 строк для обучения и 35000 строк для тестирования.

Далее перебирались значения параметров алгоритма для поиска оптимальных с точки зрения минимизации RMSE на тестовом наборе: $[2, 10, 50, 100]$ для ранга и $[0.01, 0.1, 1, 10]$ для параметра регуляризации (с фиксированным числом итераций, равным 10). Результаты можно наблюдать на графике 5.

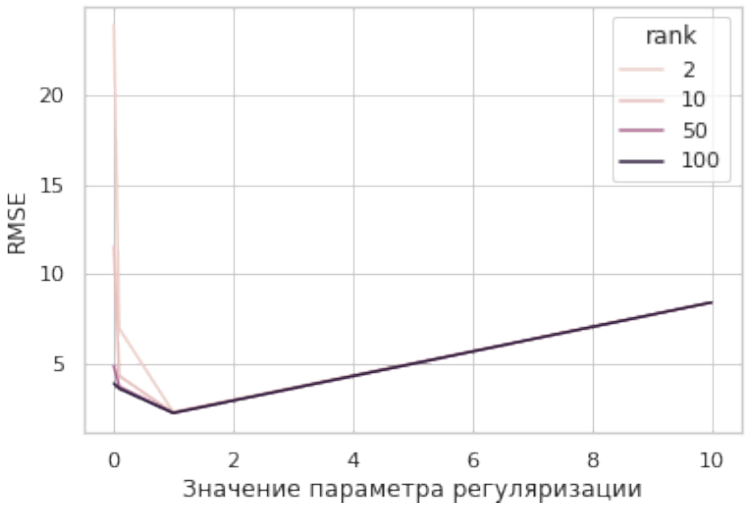


Рис. 5: График зависимости RMSE от параметра регуляризации

Можно заметить, что для всех значений ранга ошибка убывает при значении параметра регуляризации $\in [0, 1]$. Кроме того, больший ранг даёт меньшую ошибку. Поэтому рассмотрим подробнее при ранге $\in [50, 100]$.

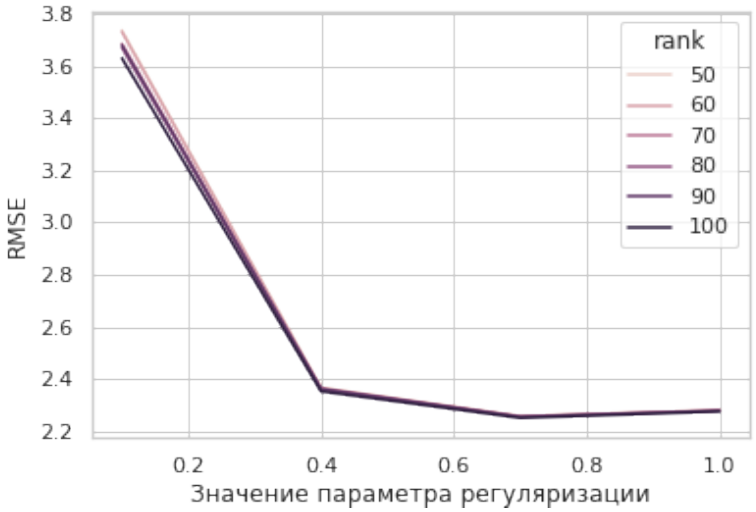


Рис. 6: График зависимости RMSE от параметра регуляризации

Получаем, что минимальное значение RMSE на тестовом наборе, равное 2.25, достигается при ранге 100 и параметре регуляризации 0.7. При этом ошибка на тренировочном наборе оказывается равной 1.17.

Также интересно, повлияет ли увеличение датасета на качество модели. Запуская модель с теми же параметрами на датасете удвоенного размера, получаем, что ошибка на тестовом наборе уменьшилась на 0.2, а на тренировочном, как ни странно, выросла на те же 0.2. При этом на обучение модели было затрачено лишь на 30% времени больше.

Перебирая параметры для второго датасета, заключаем, что оптимальными для него являются значения 90 и 0.7, соответственно. Таким образом, оптимальное значение количества латентных факторов для этого датасета отличается, но значение метрики лучше лишь на 0.002. Вероятно, причина кроется в способе генерации выборки. Она осуществлялась стратифицированно, поэтому новой информации при увеличении датасета не добавилось.

5 Дальнейшее развитие

Возможно, кажется, что нет большого смысла в реализации подобных (простых) алгоритмов на практике, когда появились нейронные сети, способные решать те же задачи даже с большей точностью (например, GNN - graph neural networks). Но это не так: в отличие от нейронных сетей, приведённые методы легко интерпретировать и видоизменять для достижения лучших показателей. Кроме того, они не требуют наличия больших вычислительных мощностей, так что вполне могут использоваться в рекомендательных системах с небольшой нагрузкой (например, на сайте специализированного магазина).

Поэтому данную работу можно продолжать: можно оптимизировать уже имеющиеся алгоритмы с точки зрения времени их работы, использовать неявные оценки (**transactions.csv**), использовать временные метки (при разделении на тренировочную и тестовую выборку — обучаться на старых данных и прогнозировать более свежие, а также при расчёте оценок — брать оценки с весовыми коэффициентами, зависящимии от того, насколько давно была выставлена оценка).

Результаты

В рамках данной работы

1. были изучены и реализованы алгоритмы построения рекомендаций при помощи коллаборативной фильтрации в окрестности и методов матричной факторизации SVD и ALS;

2. был проведён анализ указанных выше алгоритмов, а также алгоритма, построенного на ассоциативных правилах,

3. были подобраны оптимальные параметры для алгоритмов с параметрами.

Список литературы

[1] Ni J. Collaborative Filtering Recommendation Algorithm Based on TF-IDF and User Characteristics // Applied Sciences. — 2021. — Last accessed 28 May 2023. Access mode: https://www.researchgate.net/publication/355218515_Collaborative_Filtering_Recommendation_Algorithm_Based_on_TF-IDF_and_User_Characteristics.

[2] Falk K. Practical Recommender Systems. — Manning Shelter Island.

[3] Sarwar B. M. Application of Dimensionality Reduction in Recommender System – A Case Study. — 2000. — Access mode: https://www.researchgate.net/publication/2824548_Application_of_Dimensionality_Reduction_in_Recommender_System_--_A_Case_Study.

[4] Nikolenko S. Recommender Systems: SVD, Chapter I. — Access mode: https://habr.com/ru/companies/surfingbird/articles/139863/.

[5] Zhou Y. Large-Scale Parallel Collaborative Filtering for the Netflix Prize / ed. by Fleischer R., Xu J. — AAIM. — P. 337–348.

Исследование алгоритма, построенного на ассоциативных правилах

В этом файле изучается зависимость охвата по элементам (то есть процент фильмов, попадающих в рекомендации хотя бы одному человеку) от разных параметров, частота попадания фильмов в рекомендации и время работы алгоритма.

Часть 1. Охват

In [1]:
```
import pandas as pd
import os
import numpy as np
from collections import defaultdict
from itertools import combinations

%matplotlib inline
import matplotlib.pyplot as plt
```

In [2]:
```
DATA_PATH = '../data/'
```

In [3]:
```
transactions_main = pd.read_csv(
    os.path.join(DATA_PATH, 'transactions.csv'),
    dtype={
        'element_uid': np.uint16,
        'user_uid': np.uint32,
        'consumption_mode': 'category',
        'ts': np.float64,
        'watched_time': np.uint64,
        'device_type': np.uint8,
        'device_manufacturer': np.uint8
    }
)
```

In [4]:
```
transactions_main.head(6)
```

Out[4]:

	element_uid	user_uid	consumption_mode	ts	watched_time	device_type	device_manufacturer
0	3336	5177	S	4.430518e+07	4282	0	50
1	481	593316	S	4.430518e+07	2989	0	11
2	4128	262355	S	4.430518e+07	833	0	50
3	6272	74296	S	4.430518e+07	2530	0	99
4	5543	340623	P	4.430518e+07	6282	0	50
5	236	332814	S	4.430518e+07	3109	0	50

In [5]:
```
def get_rec_list(trans_dict: dict, user, rules: list): # функция-часть main() из файла 'as

    recs_list = []

    for rule in rules:
        for element_id in trans_dict[user]:
            if rule[0] == element_id:
                recs_list.append(rule)
```

```
    recs_list = sorted(recs_list, key = lambda l : l[2], reverse = True)

    # удаляем дубликаты, пересчитывая поддержку как среднее арифметическое
    # поддержок всех рекомендаций соответствующего элемента
    recs_dict = {}

    for rec in recs_list:
        if rec[1] not in recs_dict:
            recs_dict[rec[1]] = [rec[2], 1]
        else:
            (recs_dict[rec[1]])[0] += rec[2]
            (recs_dict[rec[1]])[1] += 1

    recs_list_final = []

    for key, item in recs_dict.items():
        recs_list_final.append((key, item[0]/item[1]))

    recs_list_final = sorted(recs_list_final, key = lambda t : t[1], reverse = True)

    return [t[0] for t in recs_list_final]
```

In [6]:
```
import sys
sys.path.append('../code/')
from associative_rules import generate_transactions, calculate_itemsets_one
from associative_rules import calculate_itemsets_two, calculate_association_rules, get_rec
```

In [7]:
```
def main(data):
    transactions_dict = generate_transactions(data)
    users = transactions_dict.keys()

    coverage_list = []
    movies_rec_sets = [set() for i in range(5)]

    n = len(data['element_uid'].unique())

    for t in np.linspace(0, 0.005, 5):
        k = int(t // 0.00125)
        one_itemsets = calculate_itemsets_one(transactions_dict, t)
        two_itemsets = calculate_itemsets_two(transactions_dict, one_itemsets)
        rules = calculate_association_rules(one_itemsets, two_itemsets, len(transactions_d

        # составляю рекомендации для каждого пользователя с определённным порогом поддержк
        # и добавляю 5 лучших во множество всех рекомендованных фильмов movies_rec_sets
        for user in users:
            l = get_rec_list(transactions_dict, user, rules)
            movies_rec_sets[k] |= set(l[:5])

        c = (len(movies_rec_sets[k]) / n)
        coverage_list.append(c)

    return coverage_list
```

In [8]:
```
transactions = transactions_main.iloc[:30000][['element_uid', 'user_uid']]
```

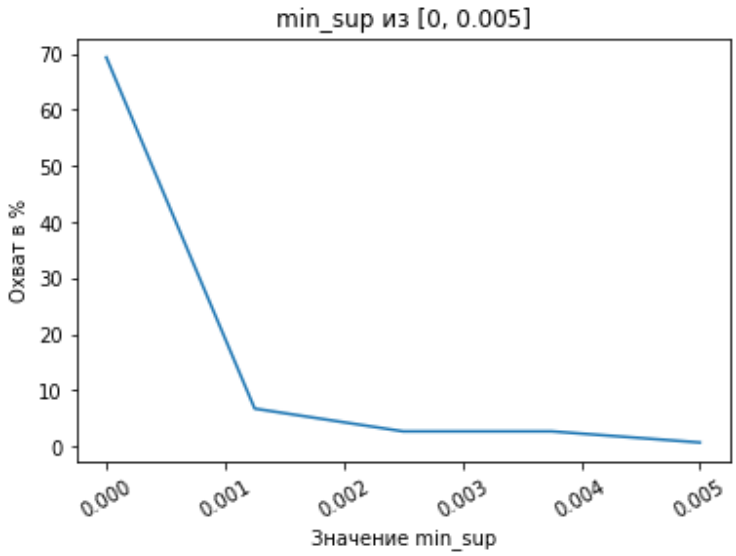
In [9]:
```
coverage_list = main(transactions)
```

In [10]:
```
coverage_list = [i*100 for i in coverage_list] # перевожу в проценты
```

```
coverage_list
```

Out[10]:
```
[69.36722266348954,
 6.777866031241726,
 2.7270320360074134,
 2.7270320360074134,
 0.7413290971670639]
```

In [11]:
```
plt.plot(np.linspace(0, 0.005, 5),
         coverage_list)
plt.xticks(rotation=30)
plt.xlabel('Значение min_sup')
plt.ylabel('Охват в %')
plt.title('min_sup из [0, 0.005]')
plt.savefig('ar_coverage1.png',
            transparent=True,
            bbox_inches='tight')
plt.show()
```



Наблюдение: \ Охват заметно убывает при увеличении порога поддержки (min_sup). \ Вопрос выбора порога --- это вопрос баланса между качеством и количеством, так как, уменьшая порог, мы уменьшаем нашу уверенность в том, что наши рекомендации релевантны и понравятся пользователю. При этом, избавляясь от условия необходимости наличия определенной поддержки (иначе говоря, полагая порог равным нулю), мы получаем охват, равный почти 70%, то есть 70% от всех фильмов попадают в рекомендации. А при пороге поддержки, равном 0.0025, более 70% остаются неохваченными, то есть не могут обнаружены благодаря рекомендациям --- см.далее.

На графике выше видно, что резкое падение значения происходит на промежутке $[0, 0.001]$ - рассмотрим подробнее:

In [12]:
```
def main2(data):
    transactions_dict = generate_transactions(data)
    users = transactions_dict.keys()

    coverage_list = []
    movies_rec_sets = [set() for i in range(5)]

    n = len(data['element_uid'].unique())

    for t in np.linspace(0, 0.001, 5): # аналогично main(), но для значений min_sup из [0,
        k = int(t // 0.0002)
        one_itemsets = calculate_itemsets_one(transactions_dict, t)
```

```
        two_itemsets = calculate_itemsets_two(transactions_dict, one_itemsets)
        rules = calculate_association_rules(one_itemsets, two_itemsets, len(transactions_d

        for user in users:
            l = get_rec_list(transactions_dict, user, rules)
            movies_rec_sets[k] |= set(l[:5])

        c = (len(movies_rec_sets[k]) / n)
        coverage_list.append(c)

    return coverage_list
```

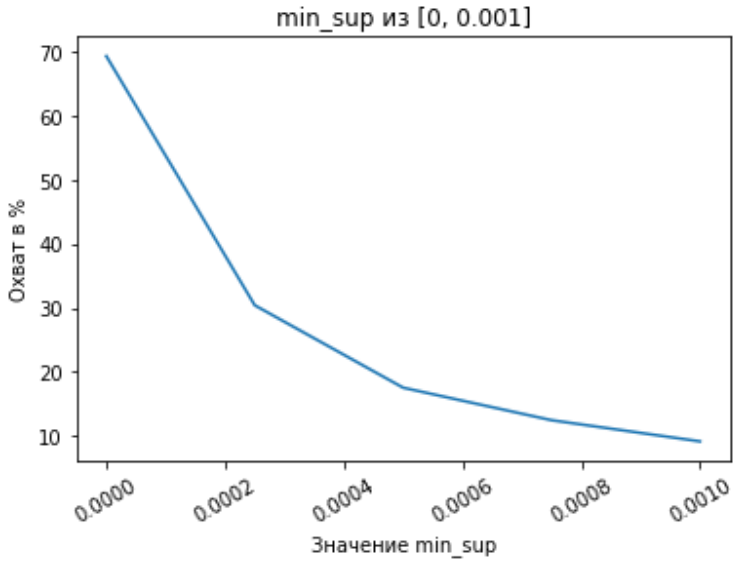
In [13]:
```
transactions = transactions_main.iloc[:30000][['element_uid', 'user_uid']]
```

In [14]:
```
coverage_list_2 = main2(transactions)
```

In [15]:
```
coverage_list_2 = [i*100 for i in coverage_list_2] # перевожу в проценты
coverage_list_2
```

Out[15]:
```
[69.36722266348954,
 30.394492983849613,
 17.500661900979615,
 12.417262377548319,
 9.10775747948107]
```

In [16]:
```
plt.plot(np.linspace(0, 0.001, 5),
         coverage_list_2)
plt.xticks(rotation=30)
plt.xlabel('Значение min_sup')
plt.ylabel('Охват в %')
plt.title('min_sup из [0, 0.001]')
plt.savefig('ar_coverage2.png',
            transparent=True,
            bbox_inches='tight')
plt.show()
```



Выбираю значение min_sup = 0.0001 как оптимальное - в этом случае хотя бы половина фильмов попадёт в рекомендации пользователям.

In [17]:
```
def main3(data):
```

```
    transactions_dict = generate_transactions(data)
    users = transactions_dict.keys()

    coverage_list = []
    movies_rec_sets = [set() for i in range(5)]

    n = len(data['element_uid'].unique())

    one_itemsets = calculate_itemsets_one(transactions_dict, 0.0001)
    two_itemsets = calculate_itemsets_two(transactions_dict, one_itemsets)
    rules = calculate_association_rules(one_itemsets, two_itemsets, len(transactions_dict))

    for user in users:
        l = get_rec_list(transactions_dict, user, rules)

        for i in range(5, 30, 5):
            if i < len(l):
                movies_rec_sets[i // 5 - 1] |= set(l[:i])
            else:
                movies_rec_sets[i // 5 - 1] |= set(l)

    for j in range(5):
        c = (len(movies_rec_sets[j]) / n)
        coverage_list.append(c)

    return coverage_list
```

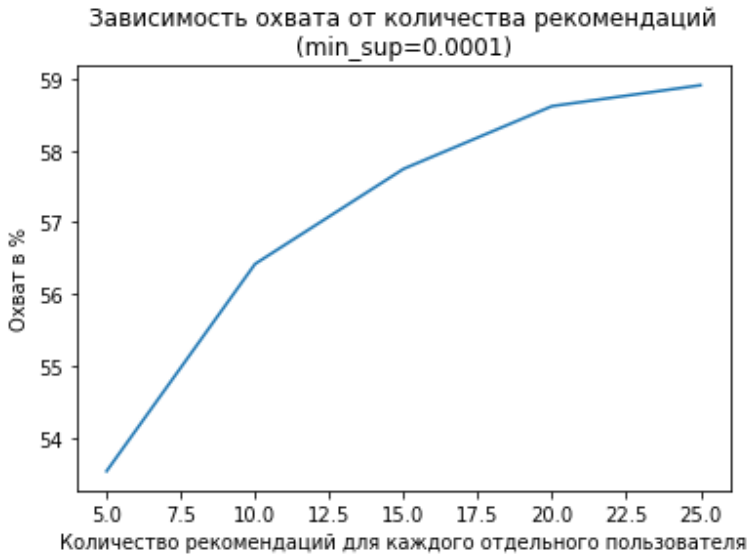
In [18]:
```
transactions = transactions_main.iloc[:30000][['element_uid', 'user_uid']]
```

In [19]:
```
coverage_list_3 = main3(transactions)
```

In [20]:
```
coverage_list_3 = [i*100 for i in coverage_list_3] # перевожу в проценты
coverage_list_3
```

Out[20]:
```
[53.53455123113582,
 56.42043950225046,
 57.744241461477365,
 58.61795075456712,
 58.909187185597034]
```

In [21]:
```
plt.plot(np.linspace(5, 25, 5),
         coverage_list_3)
plt.xlabel('Количество рекомендаций для каждого отдельного пользователя')
plt.ylabel('Охват в %')
plt.title('Зависимость охвата от количества рекомендаций\n'
          '(min_sup=0.0001)')
plt.savefig('ar_coverage3.png',
            transparent=True,
            bbox_inches='tight')
plt.show()
```



Наблюдение: \ Значение охвата не сильно меняется при увеличении количества рекомендаций для каждого пользователя (рассматриваю сетку от 5 до 25 включительно с шагом 5). Однако, как логично предположить, чем больше рекомендаций для каждого конкретного пользователя мы выделяем, тем больше разнообразие рекомендуемых фильмов. Учитывая, что в рекомендательной системе алгоритм, построенный при помощи ассоциативных правил, является не единственным, 10-ти рекомендаций будет вполне достаточно.

Таким образом, при min_sup = 0.0001 и 10-ти рекомендациях получаем охват 56%.

Часть 2. Частота попадания в рекомендации

Посмотрим, как часто фильмы попадают в рекомендации. Для этого будем отбирать лучшие 10 рекомендаций для каждого пользователя.

In [22]:
```
from collections import Counter
import seaborn as sns
```

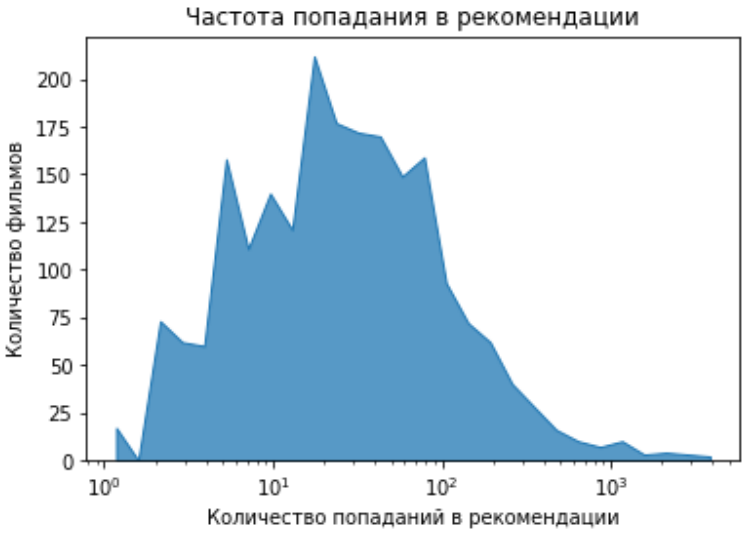
In [23]:
```
def main4(data):
    transactions_dict = generate_transactions(data)
    users = transactions_dict.keys()

    movies_rec_list = np.array([])

    n = len(data['element_uid'].unique())

    one_itemsets = calculate_itemsets_one(transactions_dict, 0.0001)
    two_itemsets = calculate_itemsets_two(transactions_dict, one_itemsets)
    rules = calculate_association_rules(one_itemsets, two_itemsets, len(transactions_dict))

    for user in users:
        l = get_rec_list(transactions_dict, user, rules)[:10]
        movies_rec_list = np.concatenate((movies_rec_list, np.array(l)))

    return movies_rec_list
```

In [24]:
```
transactions = transactions_main.iloc[:30000][['element_uid', 'user_uid']]
```

In [25]:
```
movies_rec_list = main4(transactions)
```

In [26]:
```
freq = list(dict(Counter(movies_rec_list)).values())
```

In [27]:
```
sns.histplot(x=freq, element="poly", log_scale=True)
plt.xlabel('Количество попаданий в рекомендации')
plt.ylabel('Количество фильмов')
plt.title('Частота попадания в рекомендации')
plt.savefig('ar_coverage4.png',
            transparent=True,
            bbox_inches='tight')
plt.show()
```



In [28]:
```
print('Среднее число попаданий в рекомендации:', round(np.mean(freq), 1))
```

```
Среднее число попаданий в рекомендации: 74.7
```

In [29]:
```
print('Число уникальных пользователей:', len(transactions['user_uid'].unique()))
```

```
Число уникальных пользователей: 18652
```

Приведённый выше график отражает зависимость количества фильмов от количества попаданий в рекомендации. Изучая его, можно заметить, что среди всех рекомендаций пользователям большинство фильмов (более 200) встречается около 20-ти раз - иначе говоря, попадает в рекомендации 20-ти пользователям. В среднем каждый фильм рекомендуется около 75-ти раз, но есть и экстремальные значения --- есть как фильмы, которые рекомендуются всего нескольким из 18-ти с лишним тысяч пользователей, так и те, что рекомендуются каждому 20-му.

Часть 3. Время работы

В силу того что рекомендации для каждого отдельного пользователя (при фиксированном их числе) на выходе определяются как срез списка уже отсортированных рекомендаций, время, затрачиваемое на генерацию 5-ти или 10-ти рекомендаций, почти не будет отличаться.

In [30]:
```
import time
```

In [31]:
```
transactions = transactions_main.iloc[:30000]
```

In [37]:
```
def main5(user, transactions):
    start1_time = time.time()
```

```
    transactions_dict = generate_transactions(transactions)
    one_itemsets = calculate_itemsets_one(transactions_dict)
    two_itemsets = calculate_itemsets_two(transactions_dict, one_itemsets)
    rules = calculate_association_rules(one_itemsets, two_itemsets, len(transactions_dict)

    end1_time = time.time()
    start2_time = time.time_ns()
    recs_list = get_rec_list(transactions_dict, user, rules)[:10]

    end2_time = time.time_ns()

    return (recs_list, round(end1_time - start1_time, 2), round(end2_time - start2_time, 2
```

In [38]:
```
user = 240316
recs_list, t1, t2 = main5(user, transactions)
```

In [39]:
```
print(f'Рекомендации для пользователя {user}:', recs_list)
print(f'Время на ассоциативные правила: {t1}s')
print(f'Время на рекомендации: {t2}ns')
```

```
Рекомендации для пользователя 240316: [9341, 9661, 2694, 8739, 4548, 4366, 6955, 9311, 60
3, 9817]
Время на ассоциативные правила: 1.0s
Время на рекомендации: 98691ns
```

10 рекомендаций этот алгоритм генерирует за примерно 10^{-4} секунды. А все предварительные вычисления выполняет за секунду (сразу для всех пользователей). Поэтому при подобном порядке размерности входных данных (30_000 строк) и небольшом потоке посетителей рекомендательный алгоритм может работать в онлайн-режиме, составляя рекомендации в тот момент, когда пользователь заходит на сайт, и выполняя общие вычисления по мере поступления новых данных с учётом текущей нагрузки на систему.

In [ ]:
```
```

Реализация алгоритма коллаборативной фильтрации

In [103…
```
import pandas as pd
import os
import numpy as np

DATA_PATH = '../data/'

ratings_main = pd.read_csv(
    os.path.join(DATA_PATH, 'ratings.csv'),
    dtype={
        'user_uid': np.uint32,
        'element_uid': np.uint16,
        'rating': np.uint8,
        'ts': np.float64,
    }
)
```

In [104…
```
ratings_main.head(6)
```

Out[104…

	user_uid	element_uid	rating	ts
0	571252	1364	10	4.430517e+07
1	63140	3037	10	4.430514e+07
2	443817	4363	8	4.430514e+07
3	359870	1364	10	4.430506e+07
4	359870	3578	9	4.430506e+07
5	557663	1918	10	4.430505e+07

In [105…
```
print('Number of unique users =', len(ratings_main['user_uid'].unique()))
print('Number of unique elements =', len(ratings_main['element_uid'].unique()))
```

```
Number of unique users = 104563
Number of unique elements = 7519
```

Уникальных элементов (то есть фильмов, которым выставлена хотя бы одна оценка) в датасете почти в 14 раз меньше, чем уникальных пользователей, поэтому имеет смысл производить фильтрацию по элементам.

Кроме того, стоит удалить из списка фильмов фильмы с одной единственной оценкой на весь датасет, так как оценки от одного пользователя недостаточно, чтобы как-то охарактеризовать фильм (мнение одного человека очень субъективно) и спрогнозировать оценки других пользователей, которые ему, предположительно, могли бы быть даны.

In [106…
```
data = ratings_main.groupby('element_uid').agg({'element_uid': ['count']})
ind = set(data[data['element_uid']['count'] > 2].index)
```

In [107…
```
grouped = ratings_main.groupby('element_uid')

ratings_main = grouped.filter(lambda x: set(x['element_uid']).issubset(ind))
```

In [108…
```
ratings_main.shape
```

Out[108…
```
(436281, 4)
```

Далее отбираю 30_000 строк для обучения, так как обучение на всём датасете займёт большое время. Делаю это с помощью стратифицированной по элементам выборки, иначе из-за большой разреженности матрицы рекомендации получатся неинформативными.

In [109…
```
from sklearn.model_selection import StratifiedShuffleSplit
```

In [110…
```
sss = StratifiedShuffleSplit(n_splits=1, train_size=30_000, random_state=0)
```

In [111…
```
for _, (train_index, _) in enumerate(sss.split(ratings_main, ratings_main['element_uid'])):
    ratings_main = ratings_main.iloc[train_index]
```

In [112…
```
ratings_main
```

Out[112…

	user_uid	element_uid	rating	ts
265211	249124	125	10	4.275606e+07
164880	269488	8344	10	4.329471e+07
296680	518023	793	10	4.254842e+07
47493	427046	5777	10	4.393893e+07
433322	529368	3916	8	4.175389e+07
...	...	...	...	...
387991	538047	9966	6	4.199625e+07
267002	453865	3757	8	4.274535e+07
409758	482593	6594	9	4.187928e+07
423263	429960	3478	10	4.180310e+07
204980	18744	5926	7	4.309187e+07

30000 rows × 4 columns

In [113…
```
print('Number of unique users =', len(ratings_main['user_uid'].unique()))
print('Number of unique elements =', len(ratings_main['element_uid'].unique()))
```

```
Number of unique users = 21367
Number of unique elements = 4173
```

Удаляю строки с оценкой фильма 0 (используется шкала от 1 до 10, поэтому это мусор)

In [114…
```
ratings_main = ratings_main.drop(labels=ratings_main.groupby('rating').get_group(0).index)
```

In [115…
```
from scipy.sparse import coo_matrix
from sklearn.metrics.pairwise import cosine_similarity
```

In [116…
```
def normalize(x):
    x = x.astype(float)
    x_sum = x.sum()
```

```
    x_num = x.astype(bool).sum()
    if x_num == 0:
        print(x)
    x_mean = x_sum / x_num
    x_range = x.max() - x.min()

    if x_range == 0:
        return 0.0

    return (x - x_mean) / x.std()
```

In [117…

```
def get_coo_matrix(ratings_main):

    # составляю разреженную матрицу оценок пользователями просмотренных фильмов
    # по датасету ratings_main
    # строки соответствуют пользователям, столбцы --- фильмам

    users = ratings_main['user_uid'].astype('category')
    elements = ratings_main['element_uid'].astype('category')
    ratings = ratings_main.groupby('user_uid')['rating'].transform(lambda x: normalize(x))

    coo = coo_matrix((ratings.astype(float),
                      (users.cat.codes.copy(), elements.cat.codes.copy())))

    return coo
```

In [118…

```
def get_cor_matrix(ratings_main):

    coo = get_coo_matrix(ratings_main)

    # считаю матрицу перекрытия, которая показывает,
    # сколько пользователей оценили оба фильма i и j

    overlap_matrix = coo.T.astype(bool).astype(int) @ coo.astype(bool).astype(int)

    # задаю минимальное допустимое число пользователей, оценивших оба фильма

    min_overlap = 3

    # вычисляю матрицу сходства фильмов

    cor = cosine_similarity(coo.T, dense_output=False)
    cor = cor.multiply(cor > 0.4) # удаляю слишком низкие значения сходств

    # удаляю сходства с недостаточным перекрытием

    cor = cor.multiply(overlap_matrix > min_overlap)

    return cor
```

In [119…

```
def predict_rating(user, elem, ratings_main, coo, cor):

    # рассчитываю прогнозы, отталкиваясь от средней оценки пользователя,
    # так как разные пользователи по-разному оценивают свои впечатления
    # и могут иметь привычку завышать/занижать оценку
    mean_r = ratings_main.groupby('user_uid')['rating'].mean()[user]
    i = ratings_main[ratings_main['user_uid']==user].index[0]
    j = ratings_main[ratings_main['element_uid']==elem].index[0]
    ind1 = ratings_main['user_uid'].astype('category').cat.codes.to_frame().loc[i]
    ind2 = ratings_main['element_uid'].astype('category').cat.codes.to_frame().loc[j]
    numerator = coo.toarray()[ind1[0]] @ cor.toarray()[ind2[0]]
```

```
    denominator = cor.toarray()[ind2[0]].sum()

    if denominator == 0:
        return 0.0

    return mean_r + numerator / denominator
```

In [120…

```
# пример составления рекомендаций для пользователя 24124

import random

user = 24124

ratings_without_user = ratings_main.drop(labels=ratings_main.groupby('user_uid').get_group
                                         inplace=False)
elements_list = list(ratings_without_user['element_uid'].unique())
elements_sample = random.sample(elements_list, k=10)
predicted_ratings = {}

coo = get_coo_matrix(ratings_main)
cor = get_cor_matrix(ratings_main)

for elem in elements_sample:
    r = predict_rating(user, elem, ratings_main, coo, cor)
    predicted_ratings[elem] = round(r, 2)

predicted_ratings
```

Out[120…

```
{30: 7.76,
 5821: 0.0,
 7279: 0.0,
 5180: 0.0,
 6330: 0.0,
 2625: 0.0,
 9920: 0.0,
 9898: 0.0,
 4639: 7.76,
 6909: 0.0}
```

Многие прогнозы получаются нулевыми из-за большой разреженности матрицы - ниже можно наблюдать таблицу с количеством оценок по каждому фильму (в исходном датасете). У многих фильмов количество оценок несоизмеримо мало по сравнению с количеством пользователей (пара десятков оценок на более сотни тысяч пользователей), поэтому имеющиеся оценки вносят большой вклад.

In [121…

```
data
```

Out[121…

	element_uid
	count
element_uid	
3	29
4	2
6	12
7	14
9	1
...	...
10187	2

	element_uid
	count
element_uid	
10194	2
10196	4
10197	1
10199	3

7519 rows × 1 columns

In [ ]:

Этот файл посвящён изучению алгоритма коллаборативной фильтрации

In [1]:
```
import sys
sys.path.append('../code/')
from collaborative_filtering import normalize, get_coo_matrix, get_cor_matrix
from collaborative_filtering import predict_rating, predict
```

In [2]:
```
import pandas as pd
import os
import numpy as np

DATA_PATH = '../data/'

ratings_main = pd.read_csv(
    os.path.join(DATA_PATH, 'ratings.csv'),
    dtype={
        'user_uid': np.uint32,
        'element_uid': np.uint16,
        'rating': np.uint8,
        'ts': np.float64,
    }
)
```

In [3]:
```
ratings_main.head(6)
```

Out[3]:

	user_uid	element_uid	rating	ts
0	571252	1364	10	4.430517e+07
1	63140	3037	10	4.430514e+07
2	443817	4363	8	4.430514e+07
3	359870	1364	10	4.430506e+07
4	359870	3578	9	4.430506e+07
5	557663	1918	10	4.430505e+07

In [4]:
```
from sklearn.model_selection import StratifiedShuffleSplit
```

In [5]:
```
%%time

# удаляю из списка фильмов фильмы с одной единственной оценкой на весь датасет
data = ratings_main.groupby('element_uid').agg({'element_uid': ['count']})
ind = set(data[data['element_uid']['count'] > 2].index)
grouped = ratings_main.groupby('element_uid')
ratings_main = grouped.filter(lambda x: set(x['element_uid']).issubset(ind))

# удаляю строки с оценкой фильма 0 (используется шкала от 1 до 10, поэтому это мусор)
ratings_main.drop(labels=ratings_main.groupby('rating').get_group(0).index,
                  inplace=True)

# отбираю 30_000 строк для обучения
# с помощью стратифицированной по элементам выборки
sss = StratifiedShuffleSplit(n_splits=1, train_size=30_000, random_state=0)
for _, (train_index, _) in enumerate(sss.split(ratings_main, ratings_main['element_uid'])):
    ratings_ds = ratings_main.iloc[train_index].copy()
```

```
CPU times: user 866 ms, sys: 20.2 ms, total: 886 ms
Wall time: 883 ms
```

In [6]:
```
%%time

# пример составления рекомендаций для пользователя
user = 24124
predicted_dict = predict(user, ratings_ds)
```

```
CPU times: user 19.3 s, sys: 1.18 s, total: 20.4 s
Wall time: 20.3 s
```

In [7]:
```
print(sorted(predicted_dict, reverse=True)[:10])
```

```
[9775, 8895, 8523, 8268, 8233, 6290, 6272, 5982, 5933, 5885]
```

In [31]:
```
predicted_dict
```

Out[31]:
```
{4939: 0.0,
 5885: 0.0,
 8268: 0.0,
 245: 0.0,
 1422: 0.0,
 4760: 0.0,
 8895: 0.0,
 113: 0.0,
 5208: 0.0,
 9775: 0.0,
 1842: 0.0,
 8233: 0.0,
 6290: 0.0,
 4732: 0.0,
 5982: 0.0,
 5215: 0.0,
 8523: 0.0,
 944: 7.65,
 5933: 0.0,
 6272: 7.65}
```

На предварительные действия с датасетом в среднем уходит меньше секунды, на непосредственную генерацию рекомендаций для пользователя --- около 20-ти секунд.

Понятно, что модель, выдающую нули в качестве прогнозов, нельзя назвать точной. Однако всё же интересно, насколько хорошо наша модель справляется с задачей в терминах значений метрики. Для этого построим прогнозы для элементов, уже оценённых пользователями, но не попавших в наши выборку, и посчитаем RMSE:

In [71]:
```
from sklearn.metrics import mean_squared_error as mse
```

In [54]:
```
a = ratings_ds.groupby(by='user_uid').count()
users = list(a[a['rating']>30].index) # выбираем пользователей, которые поставили больше
users
```

Out[54]:
```
[24124, 83691, 107891, 110138, 124821, 247177, 278352, 334052, 412991, 453355]
```

In [60]:
```
def calc_rmse(user, ratings_ds, ratings_main, coo, cor):
    elem_train = set(ratings_ds[ratings_ds['user_uid']==user]['element_uid'])
    elem_test = set(ratings_main[ratings_main['user_uid']==user]['element_uid'])
    elem_test -= elem_train # убираем элементы из тренировочной выборки
```

```
    elem_test &= set(ratings_ds['element_uid']) # убираем не попавшие в датасет на 30_000

    elem_train = list(elem_train)
    elem_test = list(elem_test)

    predicted_ratings = {}

    for elem in elem_test:
        r = predict_rating(user, elem, ratings_ds, coo, cor)
        predicted_ratings[elem] = round(r, 2)

    ratings_true = ratings_main.iloc[elem_test]['rating']
    ratings_pred = list(predicted_ratings.values())

    return round(mse(ratings_true, ratings_pred, squared=False), 2)
```

In [61]:
```
coo = get_coo_matrix(ratings_ds)
cor = get_cor_matrix(ratings_ds)
```

In [62]:
```
arr_rmse=[]
```

In [63]:
```
for user in users:
    arr_rmse.append(calc_rmse(user, ratings_ds, ratings_main, coo, cor))
```

In [70]:
```
print('Среднее значение RMSE для пользователей, поставивших более 30-ти оценок:', round(r
```

```
Среднее значение RMSE для пользователей, поставивших более 30-ти оценок: 6.97
```

In [16]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
```

In [38]:
```
data1 = ratings_ds.groupby('element_uid').agg({'element_uid': ['count']})
```

In [45]:
```
sns.histplot(x=list(data1['element_uid']['count']), element="poly", log_scale=True)
plt.xlabel('Количество оценок')
plt.ylabel('Количество фильмов')
plt.title('Распределение фильмов по количеству оценок')
plt.savefig('cf.png',
            transparent=True,
            bbox_inches='tight')
plt.show()
```

In [40]:
```
data1
```

Out[40]:

	element_uid
	count
element_uid	
3	2
6	1
7	1
15	4
18	1
...	...
10173	6
10178	2
10180	1
10184	2
10185	11

4163 rows × 1 columns

In [49]:
```
sum(data1['element_uid']['count']<3)
```

Out[49]:
```
2355
```

In [43]:
```
data1['element_uid']['count'].mean()
```

Out[43]:
```
7.20634158059092
```

In [44]:
```
len(ratings_ds['element_uid'].unique())
```

Out[44]:
```
4163
```

В среднем каждый фильм имеет по 7 оценок, но более чем у 2355 фильмов оценок меньше трёх, а это

почти больше половины всех фильмов в тренировчном датасете!

SVD

In [1]:
```
import pandas as pd
import os
import numpy as np

DATA_PATH = '../data/'

ratings_main = pd.read_csv(
    os.path.join(DATA_PATH, 'ratings.csv'),
    dtype={
        'user_uid': np.uint32,
        'element_uid': np.uint16,
        'rating': np.uint8,
        'ts': np.float64,
    }
)
```

In [2]:
```
ratings_main.head(6)
```

Out[2]:

	user_uid	element_uid	rating	ts
0	571252	1364	10	4.430517e+07
1	63140	3037	10	4.430514e+07
2	443817	4363	8	4.430514e+07
3	359870	1364	10	4.430506e+07
4	359870	3578	9	4.430506e+07
5	557663	1918	10	4.430505e+07

In [3]:
```
from sklearn.model_selection import StratifiedShuffleSplit
from numpy import linalg
from scipy.sparse import coo_matrix
```

In [4]:
```
%%time

# удаляю из списка фильмов фильмы с менее чем тремя оценками на весь датасет
data = ratings_main.groupby('element_uid').agg({'element_uid': ['count']})
ind = set(data[data['element_uid']['count'] > 2].index)
grouped = ratings_main.groupby('element_uid')
ratings_main = grouped.filter(lambda x: set(x['element_uid']).issubset(ind))

ratings_ds = ratings_main.copy()

# отбираю 30_000 строк для обучения
# с помощью стратифицированной по элементам выборки
sss = StratifiedShuffleSplit(n_splits=1, train_size=30_000, random_state=0)
for _, (train_index, _) in enumerate(sss.split(ratings_main, ratings_main['element_uid'])):
    ratings_main = ratings_main.iloc[train_index]


# удаляю строки с оценкой фильма 0 (используется шкала от 1 до 10, поэтому это мусор)
ratings_main.drop(labels=ratings_main.groupby('rating').get_group(0).index,
                  inplace=True)
```

```
CPU times: user 966 ms, sys: 43 ms, total: 1.01 s
```

```
Wall time: 1.01 s
```

In [5]:
```
%%time

users = ratings_main['user_uid'].astype('category')
elements = ratings_main['element_uid'].astype('category')
ratings = ratings_main['rating']

coo = coo_matrix((ratings.astype(float), # coo --- матрица оценок
                  (users.cat.codes.copy(), elements.cat.codes.copy()))).toarray()
r_average = round(coo[coo > 0.0].mean(), 2) # заполняем нули (неизвестные значения) средне
coo[coo == 0] = np.NaN
coo = np.nan_to_num(coo, nan=r_average, copy=False)
```

```
CPU times: user 728 ms, sys: 608 ms, total: 1.34 s
Wall time: 1.35 s
```

In [6]:
```
%%time
U, Sigma, Vt = linalg.svd(coo, full_matrices=False)
```

```
CPU times: user 10min 21s, sys: 2min 4s, total: 12min 25s
Wall time: 1min 44s
```

In [7]:
```
def compute_k(Sigma):
    target_sum = 0.9 * np.sum(Sigma)
    cur_sum = 0
    k = 0

    while cur_sum < target_sum:
        cur_sum += Sigma[k]
        k += 1

    return k
```

In [8]:
```
%%time
k = compute_k(Sigma)
```

```
CPU times: user 1.55 ms, sys: 317 µs, total: 1.87 ms
Wall time: 4.13 ms
```

In [9]:
```
round(len(Sigma)/k, 2)
```

Out[9]:
```
9.26
```

В 9 с лишним раз сократили размерность Sigma, сохранив при этом 90% информации!

In [10]:
```
# сокращаем размерности разложенных матриц

def rank_k(k):
    U_reduced= np.array(U[:,:k])
    Vt_reduced = np.array(Vt[:k,:])
    Sigma_reduced = np.diag(Sigma[:k])
    Sigma_sqrt = np.sqrt(Sigma_reduced)

    return U_reduced @ Sigma_sqrt, Sigma_sqrt @ Vt_reduced
```

In [11]:
```
%%time
U_reduced, Vt_reduced = rank_k(k)
ratings_matrix = (U_reduced @ Vt_reduced).round(2)
```

```
CPU times: user 9.07 s, sys: 7.91 s, total: 17 s
Wall time: 2.86 s
```

In [12]:
```
%%time

# датасет спрогнозированных оценок
# строки соответствуют пользователям, столбцы --- фильмам

ratings_df = pd.DataFrame(ratings_matrix, columns=elements.cat.categories, index=users.cat
```

```
CPU times: user 2.03 ms, sys: 63 µs, total: 2.1 ms
Wall time: 5.27 ms
```

In [13]:
```
ratings_df.head(6)
```

Out[13]:

	3	6	7	15	18	26	28	29	30	31	...	10166	10168	10169	10170	10171	10173	10
1	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	...	8.21	8.21	8.21	8.21	8.21	8.21	8
17	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	...	8.21	8.21	8.21	8.21	8.21	8.21	8
20	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	...	8.21	8.21	8.21	8.21	8.21	8.21	8
25	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	...	8.21	8.21	8.21	8.21	8.21	8.21	8
72	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	...	8.21	8.21	8.21	8.21	8.21	8.21	8
105	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	8.21	...	8.21	8.21	8.21	8.21	8.21	8.21	8

6 rows × 4169 columns

In [14]:
```
%%time

# пример составления рекомендаций для пользователя 231944
user = 231944

# убираем из рекомендаций уже оценённые пользователем элементы
rated_elems = set(ratings_ds[ratings_ds['user_uid']==user]['element_uid'])
columns = set(ratings_df.columns) - rated_elems

pairs = [[ratings_df.loc[user, col], col] for col in columns]
pairs.sort(key=lambda p: p[0], reverse=True)

recs = np.array(pairs).T[1][:10]
recs = recs.astype('I')
```

```
CPU times: user 247 ms, sys: 35.8 ms, total: 283 ms
Wall time: 399 ms
```

In [15]:
```
recs
```

Out[15]:
```
array([3349, 3861, 7279, 8569, 3903, 6373,  622, 1003, 2746, 4173],
      dtype=uint32)
```

In [16]:
```
ratings_df.loc[user, recs]
```

Out[16]:
```
3349    8.28
3861    8.25
7279    8.24
8569    8.24
3903    8.23
6373    8.23
```

```
622     8.22
1003    8.22
2746    8.22
4173    8.22
Name: 231944, dtype: float64
```

Посчитаем среднюю квадратичную ошибку на тренировочных (из rating_ds) и тестовых (из ratings_main) данных для пользователей, оценивших более 30-ти элементов:

In [17]:
```
a = ratings_main.groupby(by='user_uid').count()
users = list(a[a['rating']>30].index) # выбираем пользователей, которые поставили больше
users
```

Out[17]:
```
[24124, 83691, 107891, 110138, 278352, 334052, 412991, 453355]
```

In [19]:
```
from sklearn.metrics import mean_squared_error as mse
```

In [20]:
```
def calc_rmse(user, ratings_ds, ratings_main):
    elem_train = set(ratings_main[ratings_main['user_uid']==user]['element_uid'])
    elem_test = set(ratings_ds[ratings_ds['user_uid']==user]['element_uid'])
    elem_test &= set(ratings_main['element_uid'].unique()) # рассматриваем только попавшие
    elem_test -= elem_train # убираем элементы из тренировочной выборки

    elem_train = list(elem_train)
    elem_test = list(elem_test)

    user_ds = ratings_ds[ratings_ds['user_uid']==user]

    r_train_pred = ratings_df.loc[user, list(elem_train)]
    r_train_true = [user_ds[user_ds['element_uid']==elem]['rating'] for elem in elem_train
    r_test_pred = ratings_df.loc[user, list(elem_test)]
    r_test_true = [user_ds[user_ds['element_uid']==elem]['rating'] for elem in elem_test]

    rmse_train = round(mse(r_train_true, r_train_pred, squared=False), 2)
    rmse_test = round(mse(r_test_true, r_test_pred, squared=False), 2)
    return [rmse_train, rmse_test]
```

In [21]:
```
rmse_arr = []
```

In [22]:
```
for user in users:
    rmse_arr.append(calc_rmse(user, ratings_ds, ratings_main))
```

In [25]:
```
rmse_arr = np.array(rmse_arr).T
rmse_arr
```

Out[25]:
```
array([[0.25, 0.39, 0.4 , 0.17, 0.62, 0.13, 0.14, 0.92],
       [1.71, 1.41, 0.65, 2.56, 1.25, 1.56, 2.42, 1.05]])
```

In [26]:
```
print('Среднее значение RMSE на тренировочной выборке:',np.mean(rmse_arr[0]))
print('Среднее значение RMSE на тестовой выборке:', np.mean(rmse_arr[1]))
```

```
Среднее значение RMSE на тренировочной выборке: 0.3775
Среднее значение RMSE на тестовой выборке: 1.57625
```

Видно, что ошибка на тренировочной выборке оказалась небольшой, а на тестовой составила около 1.5. То есть для наших пользователей прогноз рекомендательного алгоритма в среднем отличался от реальной оценки на 1.5 единицы из 10.

Аналогичные вычисления для случайных 10-ти пользователей:

In [28]:
```
import random
```

In [33]:
```
a = ratings_main.groupby(by='user_uid').count()
users = list(a[a['rating']>2].index) # выбираем оценивших хотя бы два фильма
users = random.sample(users, 10) # выбираем 10 случайных пользователей
users
```

Out[33]:
```
[535086,
 398861,
 109987,
 434600,
 549799,
 133267,
 378841,
 166311,
 255046,
 417917]
```

In [34]:
```
rmse_arr = []
```

In [35]:
```
for user in users:
    rmse_arr.append(calc_rmse(user, ratings_ds, ratings_main))
```

In [36]:
```
rmse_arr = np.array(rmse_arr).T
rmse_arr
```

Out[36]:
```
array([[0.04, 0.64, 1.05, 1.53, 1.04, 1.02, 1.02, 0.46, 0.02, 1.59],
       [1.76, 1.84, 1.68, 1.55, 2.3 , 2.72, 1.82, 1.96, 1.49, 2.19]])
```

In [37]:
```
print('Среднее значение RMSE на тренировочной выборке:',np.mean(rmse_arr[0]))
print('Среднее значение RMSE на тестовой выборке:', np.mean(rmse_arr[1]))
```

```
Среднее значение RMSE на тренировочной выборке: 0.841
Среднее значение RMSE на тестовой выборке: 1.9310000000000003
```

Этот файл посвящён изучению алгоритма ALS и построению рекомендаций с его помощью

In [1]:
```
import pandas as pd
import os
import numpy as np

DATA_PATH = '../data/'

ratings_main = pd.read_csv(
    os.path.join(DATA_PATH, 'ratings.csv'),
    dtype={
        'user_uid': np.uint32,
        'element_uid': np.uint16,
        'rating': np.uint8,
        'ts': np.float64,
    }
)
```

In [2]:
```
ratings_main.head(6)
```

Out[2]:

	user_uid	element_uid	rating	ts
0	571252	1364	10	4.430517e+07
1	63140	3037	10	4.430514e+07
2	443817	4363	8	4.430514e+07
3	359870	1364	10	4.430506e+07
4	359870	3578	9	4.430506e+07
5	557663	1918	10	4.430505e+07

In [3]:
```
from sklearn.model_selection import StratifiedShuffleSplit
```

In [4]:
```
%%time

# удаляю из списка фильмов фильмы с менее чем тремя оценками на весь датасет
data = ratings_main.groupby('element_uid').agg({'element_uid': ['count']})
ind = set(data[data['element_uid']['count'] > 2].index)
grouped = ratings_main.groupby('element_uid')
ratings_main = grouped.filter(lambda x: set(x['element_uid']).issubset(ind))

# удаляю строки с оценкой фильма 0 (используется шкала от 1 до 10, поэтому это мусор)
ratings_main.drop(labels=ratings_main.groupby('rating').get_group(0).index,
                  inplace=True)

# отбираю 130_000 строк для обучения и 35_000 строк для тестирования
# с помощью стратифицированной по элементам выборки
sss = StratifiedShuffleSplit(n_splits=1, train_size=130_000,
                             test_size=35_000, random_state=0)
for _, (train_index, test_index) in enumerate(sss.split(ratings_main, ratings_main['elemer
    ratings_train = ratings_main.iloc[train_index].copy()
    ratings_test = ratings_main.iloc[test_index].copy()
```

```
CPU times: user 775 ms, sys: 0 ns, total: 775 ms
Wall time: 783 ms
```

In [5]:
```
ratings_train.head(10)
```

Out[5]:

	user_uid	element_uid	rating	ts
383899	537756	7179	8	4.201520e+07
133946	53885	9657	9	4.346024e+07
432486	359190	9894	10	4.175567e+07
355227	352028	2807	8	4.218910e+07
115338	6239	1435	8	4.356288e+07
180645	103177	3113	10	4.321203e+07
214552	342591	503	7	4.305285e+07
385205	9097	3336	10	4.200697e+07
430887	164994	3336	9	4.176695e+07
395255	8504	2923	8	4.196499e+07

In [6]:
```
ratings_train = ratings_train[['user_uid', 'element_uid', 'rating']] # убираю временные ме
ratings_test = ratings_test[['user_uid', 'element_uid', 'rating']]
```

In [7]:
```
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator
```

In [8]:
```
# подбираю лучшую модель с помощью перебора разных параметров
# модель оценивается при помощи RMSE
# результаты записываются в Pandas DataFrame

def tune_ALS(train_data, validation_data, regParams, ranks, maxIter=10):
    df = pd.DataFrame(data={'rank':[0], 'regParam':[0.0], 'rmse':[0.0]})

    for rank in ranks:
        for reg in regParams:
            # создание модели
            als = ALS(rank=rank, maxIter=maxIter, regParam=reg, seed=0)
            als.setUserCol('user_uid')
            als.setItemCol('element_uid')
            # обучение модели
            model = als.fit(train_data)
            model.setColdStartStrategy('drop')
            # расчет RMSE на валидационных данных
            predictions = model.transform(validation_data)
            evaluator = RegressionEvaluator(metricName='rmse',
                                            labelCol='rating',
                                            predictionCol='prediction')
            rmse = evaluator.evaluate(predictions)
            df1 = pd.DataFrame(data={'rank':[rank], 'regParam':[reg], 'rmse':[rmse]})
            df = pd.concat([df, df1], ignore_index=True)

    return df.copy()
```

In [9]:
```
from pyspark.sql import SparkSession

# создаю SparkSession под названием 'pandas to spark'
spark = SparkSession.builder.appName('pandas to spark').getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
```

```
# Apache Arrow используется для конвертации
# Pandas DataFrame в pySpark DataFrame
spark.conf.set('spark.sql.execution.arrow.pyspark.enabled', 'true')

# создаю DataFrame
spark_train = spark.createDataFrame(ratings_train)
spark_test = spark.createDataFrame(ratings_test)
spark_train.printSchema()
```

```
23/05/30 05:39:43 WARN Utils: Your hostname, mypc resolves to a loopback address: 127.0.1.
1; using 192.168.31.90 instead (on interface wlo1)
23/05/30 05:39:43 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLeve
l).
23/05/30 05:39:43 WARN NativeCodeLoader: Unable to load native-hadoop library for your pla
tform... using builtin-java classes where applicable
/home/aya/.local/lib/python3.8/site-packages/pyspark/sql/pandas/conversion.py:428: UserWar
ning: createDataFrame attempted Arrow optimization because 'spark.sql.execution.arrow.pysp
ark.enabled' is set to true; however, failed by the reason below:
  Unsupported type in conversion from Arrow: uint32
Attempting non-optimization as 'spark.sql.execution.arrow.pyspark.fallback.enabled' is set
to true.
  warn(msg)
root
 |-- user_uid: long (nullable = true)
 |-- element_uid: long (nullable = true)
 |-- rating: long (nullable = true)
```

In [10]:

```
# перебор параметров
df = tune_ALS(spark_train, spark_test,
              [0.01, 0.1, 1, 10],
              [2, 10, 50, 100])
```

In [11]:

```
df = df.iloc[1:] # DataFrame поможет визуализировать результаты
df
```

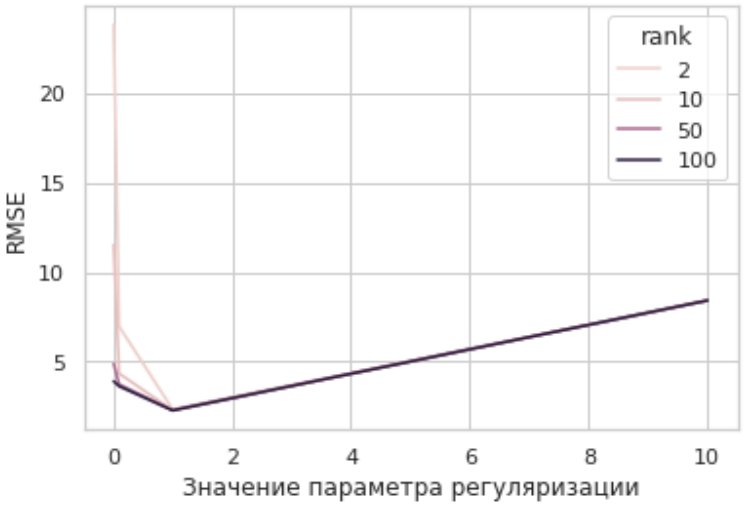
Out[11]:

	rank	regParam	rmse
1	2	0.01	23.858659
2	2	0.10	6.973863
3	2	1.00	2.337052
4	2	10.00	8.428251
5	10	0.01	11.531024
6	10	0.10	4.341076
7	10	1.00	2.283780
8	10	10.00	8.428251
9	50	0.01	4.883948
10	50	0.10	3.736698
11	50	1.00	2.279337
12	50	10.00	8.428251
13	100	0.01	3.909147

	rank	regParam	rmse
14	100	0.10	3.629209
15	100	1.00	2.274526
16	100	10.00	8.428251

In [12]:
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='whitegrid')

import warnings
warnings.filterwarnings('ignore')
```

In [13]:
```
sns.lineplot(x='regParam', y='rmse', data=df,
             hue='rank')
plt.xlabel('Значение параметра регуляризации')
plt.ylabel('RMSE')
plt.savefig('ALS1.png', transparent=True)
plt.show()
```



Так как мы хотим минимизировать ошибку, рассмотрим подробнее при значениях параметра регуляризации, меньших единицы, и ранге от 50 до 100:

In [14]:
```
df1 = tune_ALS(spark_train, spark_test,
               [0.1, 0.4, 0.7, 1],
               [50, 60, 70, 80, 90, 100])
```

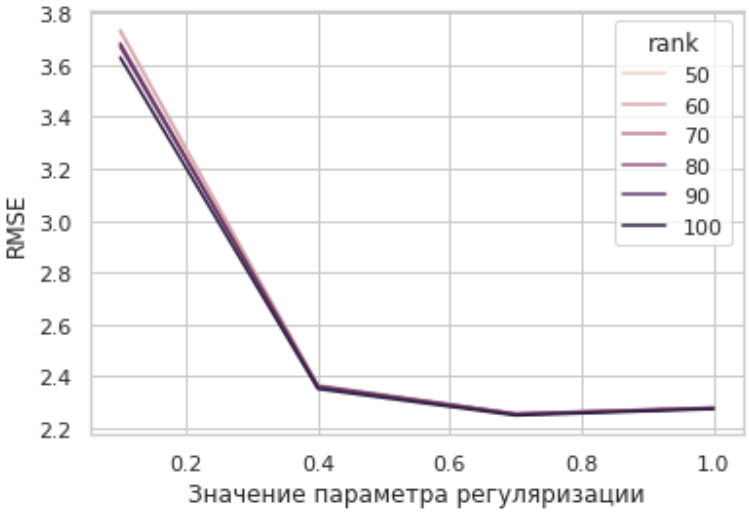
In [15]:
```
df1 = df1.iloc[1:]
df1
```

Out[15]:

	rank	regParam	rmse
1	50	0.1	3.736698
2	50	0.4	2.366010
3	50	0.7	2.256177
4	50	1.0	2.279337
5	60	0.1	3.730703

	rank	regParam	rmse
6	60	0.4	2.364981
7	60	0.7	2.256699
8	60	1.0	2.279998
9	70	0.1	3.685005
10	70	0.4	2.362341
11	70	0.7	2.255947
12	70	1.0	2.278721
13	80	0.1	3.670364
14	80	0.4	2.361774
15	80	0.7	2.255093
16	80	1.0	2.278363
17	90	0.1	3.679578
18	90	0.4	2.360294
19	90	0.7	2.253692
20	90	1.0	2.276997
21	100	0.1	3.629209
22	100	0.4	2.351626
23	100	0.7	2.249565
24	100	1.0	2.274526

In [16]:
```
sns.lineplot(x='regParam', y='rmse', data=df1,
             hue='rank')
plt.xlabel('Значение параметра регуляризации')
plt.ylabel('RMSE')
plt.savefig('ALS2.png', transparent=True)
plt.show()
```



Посмотрим, при каких значениях параметра достигается минимум RMSE:

In [17]:
```
df1[df1['rmse']==df1['rmse'].min()]
```

Out[17]:

	rank	regParam	rmse
23	100	0.7	2.249565

In [18]:
```
# оптимальные параметры для выбранного датасета
rank = 100
regParam = 0.7
```

In [19]:
```
%%time

als = ALS(rank=rank, regParam=regParam, maxIter=10, seed=0)
als.setUserCol('user_uid')
als.setItemCol('element_uid')
# обучение модели
model = als.fit(spark_train)
model.setColdStartStrategy('drop')
predictions_test = model.transform(spark_test) # прогнозы на тестовых данных
```

```
CPU times: user 71.6 ms, sys: 4.37 ms, total: 76 ms
Wall time: 47.5 s
```

In [20]:
```
predictions_train = model.transform(spark_train) # прогнозы на тренировочных данных
evaluator = RegressionEvaluator(metricName='rmse',
                                labelCol='rating',
                                predictionCol='prediction')
rmse_test = evaluator.evaluate(predictions_test)
print(f'{rank} latent factors and regularization = {regParam}: '
      f'test RMSE is {rmse_test}')
rmse_train = evaluator.evaluate(predictions_train)
print(f'{rank} latent factors and regularization = {regParam}: '
      f'train RMSE is {rmse_train}')
```

```
100 latent factors and regularization = 0.7: test RMSE is 2.2495654170098622
[Stage 4899:>                                                       (0 + 8) / 8]
100 latent factors and regularization = 0.7: train RMSE is 1.1164846733350684
```

In [21]:
```
predictions_test.show(10)
```

```
+--------+-----------+------+----------+
|user_uid|element_uid|rating|prediction|
+--------+-----------+------+----------+
|  114910|       1238|    10|  8.426525|
|  167974|       4519|    10| 5.1506243|
|  201566|       2659|     7| 7.2862873|
|  207623|       4519|    10| 6.2007885|
|  216377|       8592|     7| 7.6536365|
|  252099|       2866|     9| 3.6358652|
|  254353|       1238|    10|  7.707731|
|  256337|       9900|    10|  8.776737|
|  259486|       8592|     7| 6.6117353|
|  300472|       4519|     7| 7.0859327|
+--------+-----------+------+----------+
only showing top 10 rows
```

In [22]:
```
predictions_train.show(10)
```

```
+--------+-----------+------+----------+
|user_uid|element_uid|rating|prediction|
+--------+-----------+------+----------+
|    7634|       8592|     5| 4.5959888|
|   16791|       4519|     8| 7.1391373|
|   16916|       9427|     8|  8.227643|
|   18209|       7340|     9|  8.413807|
|   18613|       1238|     8| 6.5373917|
|   19509|       4519|    10|  9.228175|
|   21307|       1238|     9|  8.248903|
|   26583|       6397|    10|  8.461562|
|   30769|       4519|     8|  8.393911|
|   34677|       4519|     9|  8.743662|
+--------+-----------+------+----------+
only showing top 10 rows
```

Уже по первым 10-ти строкам видно, что модель довольно неплохо приближает тренировочный датасет (а значение метрики rmse даёт понять, что в среднем на нём она ошибается примерно на единицу), хотя, что интересно, имеет тенденцию занижать оценки. На тестовом датасете она справляется хуже и иногда выдаёт совсем нерелевантные прогнозы (например, оценка пользователем 252099 элемента 2866), однако в среднем ошибается на две единицы. При этом тренировочный датасет содержит 130_000 строк, а тестовый --- 35_000.

Посмотрим, как увеличение датасета повлияет на качество модели:

In [23]:
```
# отбираю 260_000 строк для обучения и 70_000 строк для тестирования
# с помощью стратифицированной по элементам выборки
sss = StratifiedShuffleSplit(n_splits=1, train_size=260_000,
                             test_size=70_000, random_state=0)
for _, (train_index, test_index) in enumerate(sss.split(ratings_main, ratings_main['elemer
    ratings_train = ratings_main.iloc[train_index].copy()
    ratings_test = ratings_main.iloc[test_index].copy()
```

In [24]:
```
ratings_train = ratings_train[['user_uid', 'element_uid', 'rating']]
ratings_test = ratings_test[['user_uid', 'element_uid', 'rating']]
```

In [25]:
```
# создаю SparkSession под названием 'pandas to spark'
spark = SparkSession.builder.appName('pandas to spark').getOrCreate()
spark.sparkContext.setLogLevel("ERROR")

# Apache Arrow используется для конвертации
# Pandas DataFrame в pySpark DataFrame
spark.conf.set('spark.sql.execution.arrow.pyspark.enabled', 'true')

# создаю DataFrame
spark_train = spark.createDataFrame(ratings_train)
spark_test = spark.createDataFrame(ratings_test)
spark_train.printSchema()
```

```
root
 |-- user_uid: long (nullable = true)
 |-- element_uid: long (nullable = true)
 |-- rating: long (nullable = true)
```

In [26]:
```
%%time

als = ALS(rank=rank, regParam=regParam, maxIter=10, seed=0)
als.setUserCol('user_uid')
```

```
als.setItemCol('element_uid')
# обучение модели
model = als.fit(spark_train)
model.setColdStartStrategy('drop');
predictions_test = model.transform(spark_test) # прогнозы на тестовых данных
```

```
[Stage 5116:=================================================>      (9 + 1) / 10]
CPU times: user 90.7 ms, sys: 0 ns, total: 90.7 ms
Wall time: 1min 11s
```

In [27]:

```
predictions_train = model.transform(spark_train) # прогнозы на тренировочных данных
evaluator = RegressionEvaluator(metricName='rmse',
                                labelCol='rating',
                                predictionCol='prediction')
rmse_test = evaluator.evaluate(predictions_test)
print(f'{rank} latent factors and regularization = {regParam}: '
      f'test RMSE is {rmse_test}')
rmse_train = evaluator.evaluate(predictions_train)
print(f'{rank} latent factors and regularization = {regParam}: '
      f'train RMSE is {rmse_train}')
```

```
100 latent factors and regularization = 0.7: test RMSE is 2.036019546805109
[Stage 5279:>                                                         (0 + 8) / 8]
100 latent factors and regularization = 0.7: train RMSE is 1.3244280391467262
```

Интересно, что увеличение тренировочного и тестового датасетов в два раза при тех же параметрах не помогло добиться заметного улучшения прогностической способности модели: значение метрики на тестовом датасете уменьшилось на 0.2, а на тестовом выросло на те же 0.2. При этом на обучение модели ушло лишь на 30% времени больше.

Попробуем подобрать оптимальные параметры для этого датасета:

In [28]:

```
df1 = tune_ALS(spark_train, spark_test,
               [0.1, 0.4, 0.7, 1],
               [50, 60, 70, 80, 90, 100])
```

In [31]:

```
df1 = df1.iloc[1:]
df1
```

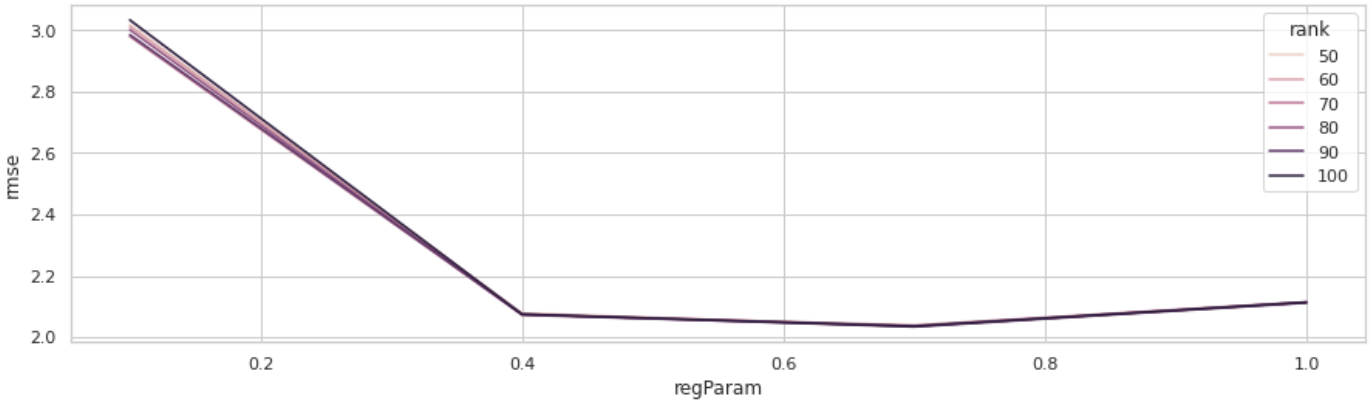
Out[31]:

	rank	regParam	rmse
1	50	0.1	3.017379
2	50	0.4	2.074984
3	50	0.7	2.037107
4	50	1.0	2.113955
5	60	0.1	3.012323
6	60	0.4	2.078101
7	60	0.7	2.035633
8	60	1.0	2.113324
9	70	0.1	2.979702
10	70	0.4	2.073439

	rank	regParam	rmse
11	70	0.7	2.038862
12	70	1.0	2.114951
13	80	0.1	3.001354
14	80	0.4	2.072774
15	80	0.7	2.035811
16	80	1.0	2.113807
17	90	0.1	2.985055
18	90	0.4	2.074635
19	90	0.7	2.034023
20	90	1.0	2.112332
21	100	0.1	3.033050
22	100	0.4	2.074893
23	100	0.7	2.036020
24	100	1.0	2.114097

In [32]:

```
sns.lineplot(x='regParam', y='rmse', data=df1,
             hue='rank')
plt.xlabel('Значение параметра регуляризации')
plt.ylabel('RMSE')
plt.savefig('ALS3.png', transparent=True)
plt.show()
```



In [33]:

```
df1[df1['rmse']==df1['rmse'].min()]
```

Out[33]:

	rank	regParam	rmse
19	90	0.7	2.034023

Итак, оптимальное значение количества латентных факторов для этого датасета отличается, но значение метрики лучше лишь на 0.002. Вероятно, причина кроется в способе генерации выборки. Она осуществлялась стратифицированно, поэтому новой информации при увеличении датасета не добавилось.

Количество итераций для всех попыток выбрано равным значению по умолчанию в библиотечной реализации ALS (10). Также было опробовано значение 5, но оно давало худшие результаты.

Пример построения рекомендаций для пользователя 231944:

In [34]:
```
user = 231944
```

In [36]:
```
user_subset = spark_train.where(spark_train.user_uid==user)
user_subset_recs = model.recommendForUserSubset(user_subset, 10)
```

In [43]:
```
recs = user_subset_recs.select('recommendations.element_uid', 'recommendations.rating').fi
recs['element_uid']
```

Out[43]:
```
[7593, 340, 6574, 6789, 6616, 10076, 5826, 1235, 4533, 8662]
```

In [44]:
```
recs['rating']
```

Out[44]:
```
[9.373523712158203,
 9.136528968811035,
 8.975046157836914,
 8.963576316833496,
 8.918926239013672,
 8.804742813110352,
 8.799222946166992,
 8.738356590270996,
 8.714048385620117,
 8.70267391204834]
```

In [ ]:
```

```