Санкт–Петербургский государственный университет

Папернюк Александр Олегович

Выпускная квалификационная работа

Оптимизация алгоритмов рекомендательной системы

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2017 «Прикладная математика, фундаментальная информатика и программирование»
Кафедра «математического моделирования энергетических систем»

Научный руководитель:
кандидат физ.-мат. наук, старший
преподаватель кафедры математической
теории игр и статистических решений
Кумачева Сурия Шакировна

Рецензент:
инженер-программист, Общество
с ограниченной ответственностью «Мэйл.Ру»
Макаров Юрий Сергеевич

Санкт-Петербург

2021 г.

Содержание

Введение 3

Постановка задачи 8

Обзор Литературы 11

Глава 1. Алгоритмы рекомендательных систем 12

1.1. Alternating Least Squares (ALS) 12

1.2. Применение функции BM25 15

1.3. Lightfm with warp loss function 16

1.4. Метод ближайших соседей 19

1.5. Применение функции TF-IDF 21

1.6. Autoencoder 22

Глава 2. Практические результаты 27

Заключение 35

Список литературы 36

Приложение 38

Введение

Задача разработки рекомендательных систем появилась относительно недавно – в век развития Интернета и информационных технологий количество доступной информации увеличилось настолько, что человек не способен проанализировать её полностью, чтобы выбрать только интересную ему.

Поэтому многие современные сервисы создают рекомендательные системы, которые на основе информации о профиле пользователя и его предыдущего поведения в системе пытаются определить, какие объекты, товары или услуги могут быть ему интересны. Объектами могут быть товары, книги, музыка, фильмы, новости и т. д.

Многие известные по всему миру сайты уже используют рекомендательные системы: Ozon, eBay, Amazon, Кинопоиск, IMDb, Pandora и др., но ни одна из систем не может гарантировать 100% точность сформированной рекомендации. Методы построения прогноза нуждаются в усовершенствовании. К примеру, со 2 октября 2006 года Netflix проводил открытое соревнование Netflix prize, в котором командам необходимо было усовершенствовать алгоритм рекомендательной системы для лучшего предсказания оценок, которые поставят пользователи фильмам. Главный приз данного соревнования составлял $1,000,000.

Как стало понятно, рекомендательные системы могут служить инструментом для увеличения продаж, продажи более разнообразных объектов, а также улучшения понимания пользовательских потребностей и желаний. Поэтому они быстро набирают популярность и начинают широко применяться в электронной коммерции, при поиске фильмов, музыки, ПО, научных статей, а также на новостных сайтах и в справочных центрах. Таким образом, задача разработки эффективных рекомендательных систем ныне является актуальной.

То есть рекомендательные системы — это комплексы алгоритмов, программ и сервисов, основная задача которых предсказать, какие объекты будут интересны пользователю и как он на них отреагирует, ориентируясь на информацию о его профиле либо иные данные. Такие данные могут быть получены как явными, так и неявными способами. К явным относят следующие методы: пользователь ставит оценку объектам из заданного диапазона или ранжирует группу объектов, выбирает наиболее привлекательный из двух или небольшой группы объектов и др. К неявным методам относятся: число просмотров объекта (например, сколько раз была прослушана аудиозапись), категории просматриваемых объектов (например, в интернет-магазинах, где есть точное разделение на категории) и др.

Различают 4 основных типа рекомендательных систем:

- Коллаборативная фильтрация (Collaborative Filtering) —

рекомендации, в которых используется информация о поведении пользователей в прошлом - например, информация о покупках или оценках, данным некоторым товарам. Такие системы основаны на схожести предпочтений пользователей. При этом могут учитываться неявные характеристики, которые сложно было бы учесть при создании профиля. Основное допущение метода состоит в следующем: те, кто одинаково оценивали какие-либо предметы в прошлом, склонны давать похожие оценки другим предметам и в будущем. Прогнозы составляются индивидуально для каждого пользователя, хотя используемая информация собрана от многих участников. Главным недостатком методов коллаборативной фильтрации является наличие проблемы «холодого» старта. Это ситуация, когда объекту еще никто не поставил никакую оценку или таких оценок слишком мало. При использовании этого подхода новые объекты не будут рекомендоваться пользователям в силу отсутствия оценок этого объекта.

- Основанные на контенте (content-based) — рекомендации, основанные на данных, собранных о каждом конкретном объекте и пользователе. Пользователю рекомендуются объекты, похожие на те, которыми он ранее интересовался, а также объекты, которые, скорее всего, будут интересны, исходя из профиля пользователя. Похожесть оценивается по содержимому объектов. Когда новый пользо-

ватель, у которого пустой профиль, начинает использовать рекомендательную систему, основанную на контенте, ему необходимо дать некоторую информацию о себе. Это можно сделать разными способами: путем задания предпочтительных значений характеристик, указанием интересных ему объектов (при сравнении с которыми можно будет выделить ключевые характеристики для этого пользователя), а также добавлением личной информации. При использовании рекомендательных систем такого типа основной проблемой, с которой сталкиваются разработчики, является проблема нахождения значений параметров-характеристик объекта, так как количество таких характеристик может быть очень большим. Преимуществом рекомендательных систем основанных на контенте является отсутствие проблемы «холодного» старта. При использовании этого метода пользователям достаточно заполнить хотя бы частично свои профили, после чего им можно рекомендовать как новые, так и давно имеющиеся в базе и уже рекомендованные другим пользователям объекты.

Модели, основанные на контенте, как правило, гораздо более точны, чем методы коллаборативной фильтрации, но сильно проигрывают им в скорости.

- Гибридные (hybrid) — рекомендации основаны на комбинировании коллаборативных и контентных подходов, что позволяет избежать большинства недостатков, свойствен-

ных каждой системе.

Постановка задачи

Пусть имеется n пользователей и m музыкальных треков. Каждый пользователь прослушивает произвольное количество музыкальных композиций. Следовательно, формируется матрица R размера $n \times m$. Элемент $r_{u,i} \in [0, +\infty)$ этой матрицы соответствует количеству прослушиваний пользователя $u \in U$ трека $i \in I$, где U - множество всех пользователей тренировочной матрицы, I - множество всех аудиозаписей тренировочной матрицы. В каждой строке матрицы R нулевые элементы характеризуют аудиозаписи, которые данный пользователь еще ни разу не слушал. Данная работа посвящена практической реализации алгоритмов коллаборативной фильтрации и их сравнению с целью выявления модели, которая рекомендует наиболее релевантные и персонализированные треки для каждого пользователя.

Для сравнения алгоритмов были подобраны 3 метрики, характеризующие необходимые свойства для рассматриваемой задачи.

- HR (Hit Ratio). Пусть переменная rec_u обозначает множество объектов, рекомендуемых моделью, для пользователя u, переменная $k = |rec_u| = const$ и $real_u$ — это множество объектов, с которыми пользователь u взаимодействует в тестовых данных. Метрика Hit Ratio вычисляется по формуле:

$$HR@k = \sum_{u=1}^{n} \frac{rec_u \cap real_u}{min(k, |real_u|)},$$

  где $HR@k \in [0, 1]$. То есть, чем больше треков находятся на пересечении множеств rec_u и $real_u$ у пользователя, тем выше значение метрики.

- NDCG (Normalized Discounted Cumulative Gain). Метрика NDCG отвечает за то, насколько релевантные треки находятся наверху списка рекомендаций. То есть, предпочтительней, чтобы пользователи слу-

шали треки, находящиеся наверху рекомендаций, нежели внизу. Обозначим за $rec_{u,i}$ трек, который рекомендуется пользователю u и находится на i-ом месте в списке рекомендаций. Величина DCG вычисляется по формуле:

$$DCG@k = \sum_{i=1,j=1}^{k} \frac{I(rec_{u,i} \in real_u)}{\log_2(j+1)},$$

где I - это индикаторная функция. Также вводится переменная $IDCG$, значение которой соответствует максимально возможному значению DCG:

$$IDCG@k = \sum_{j=1}^{min(k,real_u)} \frac{1}{\log_2(j+1)}$$

И наконец зададим метрику NDCG@k:

$$NDCG@k = \frac{DCG@k}{IDCG@k},$$

где $NDCG \in [0, 1]$.

- APR (average popularity rank). Метрика APR показывает, насколько наша модель рекомендует персонализированные (непопулярные) треки. Для данной метрики необходимо сформировать список треков, сортированный по убыванию количества прослушиваний, который назовем топ-чартом треков. Обозначим за top_i позицию i-ого рекомендованного трека в топ-чарте, тогда:

$$APR@k = 1 - \frac{k \cdot (k+1)}{2 \cdot \sum_{i=1}^{k} top_i},$$

где $APR@k \in [0, 1)$. Чем ближе значение метрики к 0, тем более популярные треки модель рекомендует, и наоборот.

В данной работе термин «метрика» используется в значении меры некоторой характеристики модели. Данная мера может иметь различные

математические определения, в зависимости от поставленной задачи.

Для каждого из расмотренных алгоритмов коллаборативной фильтрации были посчитаны метрики на тестовых данных, чтобы выявить модель, которая одновременно рекомендует наиболее подходящие и наименее популярные треки, дабы список рекомендаций был основан исключительно на предпочтениях каждого клиента.

Обзор Литературы

В течение последних нескольких десятилетий, с появлением Youtube, Amazon, Netflix и многих других подобных веб-сервисов, рекомендательные системы занимают все больше и больше места в нашей жизни. От электронной коммерции (предлагать покупателям статьи, которые могут их заинтересовать) до онлайн-рекламы (предлагать пользователям правильный контент, соответствующий их предпочтениям), рекомендательные системы сегодня неизбежны в наших ежедневных онлайн-путешествиях.

Метрика, используемая для оценки модели, такая, как NDCG подробно изучена в [16], а затем модифицирована в данной работе для случая с неявными оценками объектов. Метод оценки качества Hit Ratio и идея построения метрики APR ранее были описаны в [17].

Первый рассматриваемый алгоритм - ALS, описанный в [1] и улучшенный по времени в [2], был адаптирован для задачи рекомендации подходящих музыкальных треков. Исходя из вычисленных метрик, ALS был улучшен с добавлением функции ранжирования bm25, которая обозревается в книге [15]. Наиболее популярный алгоритм, используемый для построения рекомендаций, Lightfm с функцией потерь warp освещается в статьях [6] и [7]. Для исходной задачи была адаптирована функция tf-idf, используемая для анализа текстов и описываемая в [15]. Основная идея классического метода задачи построения рекомендаций ближайших соседей была изучена по источнику [14].

В данной работе используется модель autoencoder, у которой в идельном случае работы алгоритма необязательно должны совпадать входные и выходные векторы, в отличие от модели, описанной в [9]. Выходные векторы используемой модели должны предсказывать вектор прослушиваний конкретного пользователя, полученный спустя некоторое время.

Глава 1. Алгоритмы рекомендательных систем

1.1 Alternating Least Squares (ALS)

Пусть, как уже говорилось выше, $r_{u,i}$ — значения матрицы взаимодействий R. Введем индикаторную переменную $p_{u,i}$, которая указывает на предпочтение пользователя $u \in U$ объекта $i \in I$:

$$p_{u,i} = \begin{cases} 1 & \text{if } r_{u,i} > 0 \\ 0 & \text{if } r_{u,i} = 0 \end{cases}$$

И также введем:

$$c_{u,i} = 1 + r_{u,i}$$

С заданной индикаторной переменной $p_{u,i}$ и весовой переменной $c_{u,i}$ необходимо найти вектор $x_u \in \mathbb{R}^p$ для каждого пользователя и вектор $y_i \in \mathbb{R}^p$ для каждого объекта, который будет учитывать предпочтения пользователя. Иными словами, переменная предпочтений получается посредством векторного произведения: $p_{u,i} = x_u^T y_i$. Векторы пользователей и объектов подбираются таким образом, чтобы значение функции потерь было минимальным:

$$\min_{x_*, y_*} \sum_{u,i} c_{u,i} \left(p_{u,i} - x_u^T y_i\right)^2 + \lambda \left(\sum_u ||x_u|| + \sum_i ||y_i|| \right)$$

Член $\lambda(\sum_u ||x_u|| + \sum_i ||y_i||)$ необходим для регуляризации модели то есть, чтобы модель не переобучалась на тренировочных данных.

Сперва модель инициализирует векторы пользователей и объектов, значения которых подчиняются равномерному распределению на отрезке[0, 0,01].

Пусть все векторы объектов могут быть получены из матрицы Y с

размерностью $m \times f$. Для каждого агента определим диагональную матрицу C^u с размерностью $m \times m$, где $C^u_{ii} = c_{ui}$ и вектор $p(u) \in \mathbb{R}^m$, который содержит все предпочтения пользователя u.

Зафиксировав векторы объектов, дифференцируем функцию потерь по векторам пользователей x_u и приравниваем к нулю. Затем выразим эти векторы и получим уравнение

$$x_u = \left(Y^T C^u Y + \lambda I\right)^{-1} Y^T C^u p(u) \tag{1}$$

Для экономии времени приближенное решение уравнения (1) ищется с помощью метода сопряженных градиентов [2], так как при $\lambda > 0$ матрица $Y^T C^u Y + \lambda I$ является симметричной положительно определенной матрицей. Решение системы (1) эквивалентно нахождению минимума соответствующей квадратичной формы, который вычисляется с помощью нахождения сопряжённых направлений [2].

Аналогично векторам объектов, определяем $(n \times f)$-матрицу X, для каждого объекта задаем диагональную $(n \times n)$-матрицу C^i_{uu}, где $C^i_{uu} = c_{ui}$ и вектор $p(i) \in \mathbb{R}^n$. Зафиксировав векторы пользователей, дифференцируем функцию потерь по векторам объектов y_i и полученное выражение приравниваем к нулю. Выразив векторы объектов, получаем уравнение, аналогичное (1):

$$y_i = \left(X^T C^i X + \lambda I\right)^{-1} X^T C^i p(i) \tag{2}$$

Далее векторы объектов (2) считаются с помощью метода сопряженных градиентов [2].

Поочередно фиксируя векторы и обновляя их, добиваемся сходимости метода. Главный плюс ALS – скорость и возможность легкого распараллеливания.

После подсчёта итоговых векторов пользователей и музыкальных композиций, для любого пользователя $u \in U$ можно составить список рекомендуемых треков. Для этого вектор пользователя x_u^T векторно домножается на векторы треков y_i. Далее формируется список из k треков, у которых результаты произведения оказались наибольшими. Этот список сортируется по убыванию результатов, и в итоге получаем треки, подобранные по предпочтениям клиента u. В случае, если мы считаем метрики на тестовых данных, то в рассмотрении не учитываются треки, которые уже были прослушаны пользователем, так как наша задача – предлагать пользователям треки, которые они не слушали.

1.2 Применение функции BM25

Для каждого элемента тренировочной матрицы применяется функция ранжирования BM25, которая обычно используется для работы с текстами:

$$r_{u,i}^{new} = IDF(i)\frac{r_{u,i}(k_1+1)}{r_{u,i} + k_1\left[(1-b) + b\cdot\frac{|I_u|}{\overline{|I|}}\right]},$$

где $|I_u|$ – общее количество взаимодействий клиента с разными объектами. $\overline{|I|}$ – математическое ожидание величины $|I_u|$, k_1 и b – настраиваемые гиперпараметры, контролирующие количество прослушиваний трека пользователем и активность пользователя соответственно. Функция $IDF(i)$:

$$IDF(i) = \log_{10}\frac{|U|}{|U_i|},$$

где $|U|$– общее количество клиентов, $|U_i|$ - количество пользователей, которые взаимодействовали с объектом i. После применения функции BM25, модель будет рекомендовать менее популярные, но более персонализированные треки пользователям.

Формируется итоговая весовая матрица и подается на вход ALS-модели.

1.3 Lightfm with warp loss function

Lightfm также ищет вектор $x_u \in \mathbb{R}^p$ для каждого пользователя и вектор $y_i \in \mathbb{R}^p$ для каждого объекта, где число p заранее определяется. Если p будет недостаточно большим, то исходные векторы не будут в полной мере характеризовать пользователя/объект, если же значение будет большим, то велика вероятность переобучения алгоритма.

Сперва рассматриваемая модель задает векторы пользователей и объектов, значения которых подчиняются равномерному распределению на отрезке $[-(\frac{0.5}{p}), \frac{0.5}{p}]$.

Введем переменную $f_i(u)$, которая показывает, насколько пользователю u предпочтителен объект i:

$$f_i(u) = x_u^T \cdot y_i + b_u + b_i,$$

где b_u и b_i - смещения для клиента u и объекта i соответственно, которые также задаются равномерным распределением на отрезке $[-(\frac{0.5}{p}), \frac{0.5}{p}]$. Функция, которую алгоритм минимизирует, подбирая значения векторов клиентов, объектов и их смещений, представима в таком виде:

$$err = \sum_{u \in U} L(rank(u, i)), \tag{3}$$

где функция L равна частичной сумме убывающего числового ряда. Чаще всего задают $L(k) = \sum_{j=1}^{k} \frac{1}{j}$, однако в реальных вычислениях ради повышения скорости модели эту функцию можно аппроксимировать логарифмом.

У каждого пользователя почти все объекты расставлены по приоритетам. То есть, независимо от того, что пользователь взаимодействовал с двумя объектами, скорее всего, какой-то для него является более предпочтительным. Данный алгоритм для каждого пользователя строит некоторую иерархию объектов, основываясь на значениях в тренировочной матрице - $r_{u,i}$. Соответственно, модель может ошибочно предсказать, какие треки

предпочтительнее трека i для клиента u. Значение функции $rank(u, i)$ и обозначает количество таких объектов.

$$rank(u, i) = \sum_{j \in I : r_{u,i} > r_{u,j}} I(f_i(u) \leq f_j(u))$$

$I(x)$ здесь обозначает индикаторную функцию, которая равна единице, если неравенство верно и нулю – иначе.

Преобразовав уравнение (3) и заменив индикаторную функцию на hinge loss, функция err может быть аппроксимирована:

$$\overline{err} = \sum_{u \in U} \sum_{j \in I : r_{u,i} > r_{u,j}} L(rank^1(u, i)) \frac{\max(0, 1 - f_i(u) + f_j(u))}{rank^1(u, i)},$$

где

$$rank^1(u, i) = \sum_{j \in I : r_{u,i} > r_{u,j}} I(f_i(u) \leq f_j(u) + 1).$$

Чаще всего, чтобы сэкономить время и упростить вычисления $rank^1(u, i)$ аппроксимируют делением $|I|-1$ на N, где $|I|$ - общее количество объектов, N - количество шагов, необходимое для нахождения объекта j, удовлетворяющего неравенству $1 + f_j(u) > f_i(u)$.

Чтобы минимизировать ошибку (4), среди ненулевых значений тренировочной матрицы случайным образом выбирается элемент, соответствующий, к примеру, пользователю u и объекту i. Затем модель подбирает случайные объекты, среди треков, у которых приоритет ниже, чем у i, пока не найдется объект j, для которого выполняетя неравенство $1 + f_j(u) > f_i(u)$ и тогда делается градиентный шаг для минимизации функции:

$$L(\lfloor \frac{|I| - 1}{N} \rfloor) \cdot \max(0, 1 - f_i(u) + f_j(u))$$

В данном случае градиентный шаг - это составляющая метода градиентного спуска, которая измененяет параметры модели посредством при-

бавления к вектору исходных параметров вектор антиградиента. В свою очередь, вектор антиградиента - это вектор противоположный вектору, состоящему из производных минимизируемой функции, взятых по соответствующим параметрам. Направление вектора антиградиента указывает на минимум рассматриваемой функции в пространстве параметров, таким образом, совершая градиентные шаги, модель минимизирует исходную функцию.

В алгоритме используется adagrad [19], который является модификацией стохастического алгоритма градиентного спуска [18] с отдельной для каждого параметра скоростью обучения. Идея Adagrad [19] заключается в том, чтобы уменьшать обновления для элементов, которые и так часто обновляются. Эта стратегия увеличивает скорость сходимости по сравнению со стандартным стохастическим методом градиентного спуска в условиях, когда данные редки и соответствующие параметры более информативны.

Безусловным плюсом рассматриваемой модели является то, что она учитывает порядок приоритетов объектов, сложившийся у каждого пользователя. К примеру алгоритм может отличить трек, который клиент слушал множество раз, и трек, который был прослушан всего несколько раз.

После подсчёта векторов x_u, b_u для каждого пользователя $u \in U$ и векторов y_i, b_i для каждого объекта $i \in I$, любому клиенту $u \in U$ можно составить список рекомендуемых треков. То есть, сначала для каждого трека считается величина $f_i(u) = x_u^T \cdot y_i + b_u + b_i$, где значение u зафиксировано. Далее формируется список из k треков, у которых значения $f_i(u)$ оказались наибольшими. Этот список сортируется по убыванию результатов, и в итоге получаем треки, подобранные по предпочтениям клиента u.

1.4 Метод ближайших соседей

Метод ближайших соседей (Nearest Neighbors) [14] относится к метрическим алгоритмам классификации, основанным на оценивании сходства объектов. Активному пользователю (для которого строится рекомендация) предлагаются музыкальные треки, которые наиболее похожи на прослушанные им ранее. В данном алгоритме используется косинусная мера сходства для определения похожести двух объектов:

$$cos(i_1, i_2) = \frac{r_{i_1} \cdot r_{i_2}}{||r_{i_1}|| \cdot ||r_{i_2}||},$$

где $r_{i_1} \in \mathbb{R}^{|U|}$ и $r_{i_1} = (r_{i_1,1}, r_{i_1,2}, ...r_{i_1,|U|})$, то есть это вектор с количеством прослушиваний трека i_1 каждым пользователем. В равенстве (5) используется Эвклидова норма.

Стоит заметить, что в рассматриваемой задаче все точки, соответствующие объектам, лежат в первой четверти, так как значения в тренировочной матрице положительные. Значение косинуса уже принадлежит отрезку [0; 1], причем, если сравнить трек с самим собой, то косинус будет равен 1, что соответствует полному совпадению характеристик объектов. То есть, чем больше значение косинуса, тем более похожи рассматриваемые треки.

Для начала алгоритм строит матрицу схожести с размерностью ($|I| \times |I|$), элементы которой являются косинусным расстоянием между объектами, соответствующими номеру столбца и номеру строки определённого элемента. Чтобы рекомендовать пользователю u k аудиозаписей, формируется нулевой вектор $item\_score$, длина которого - общее количество треков в данных (индексы соответствуют аудиозаписям). Рассматривается музыкальный трек i, прослушанный пользователем u, для него в матрице схожести в строке i берутся значения косинусного расстояния и умножаются на значение $r_{u,i}$, выступающее в роли веса. Получившийся вектор прибавляется к вектору $item\_score$ и проделываются такие манипуляции для всех треков, прослушанных клиентом u. По итогу рекомендуем пользователю k

треков, имеющих наибольшие значения в векторе $item\_score$.

1.5 Применение функции TF-IDF

В общем понимании TF-IDF [15] - это статистическая мера, используемая для оценки важности слова в контексте документа. Мера TF-IDF часто используется в задачах анализа текстов и информационного поиска. Однако в данной задаче она применяется для каждого элемента тренировочной матрицы, чтобы вкусы клиентов больше характеризовали треки, которые не особо популярны и захватывают только специфическую аудиторию.

Таким образом каждый элемент в тренировочной матрице заменяется на элемент:

$$r_{u,i}^{new} = tf(u) \cdot idf(i) = \frac{|I_u|}{|I|} \cdot \log \frac{|U|}{|U_i|},$$

где $|I_u|$ - количество объектов, с которыми пользователь u взаимодействовал, $|I|$ - общее количество объектов, $|U|$– общее количество клиентов, $|U_i|$ - количество пользователей, которые взаимодействовали с объектом i.

Обновлённая тренировочная матрица подаётся методу ближайших соседей на вход.

1.6 Autoencoder

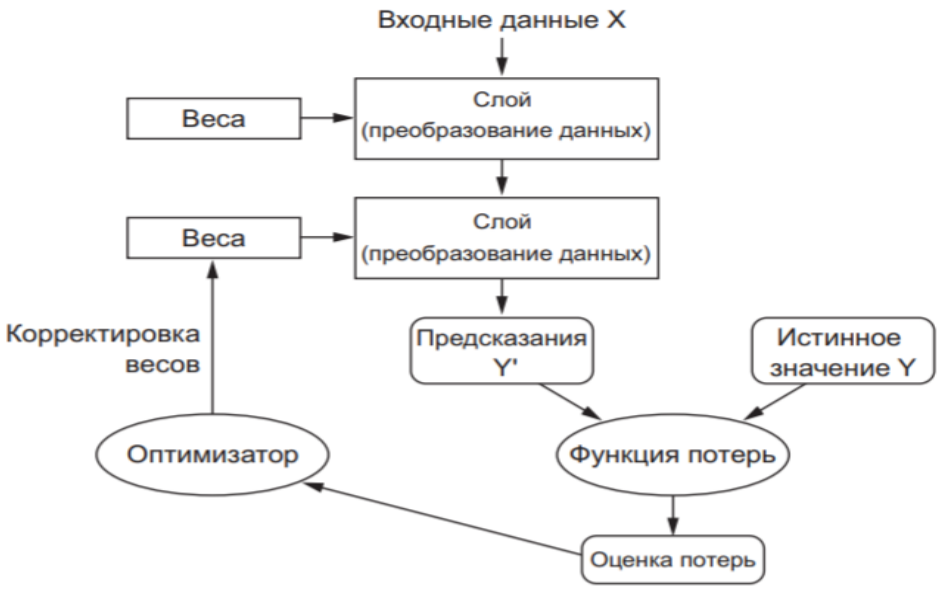
Данный метод основан на построении и применении нейронной сети [21]. Искуственные нейронные сети - это математические модели, а также их программные или аппаратные реализации, построенные по принципу организации и функционирования биологических нейронных сетей - сетей нервных клеток живого организма. Нейронная сеть представляет собой сеть искуственных нейронов, связанных между собой синаптическими соединениями. Сеть обрабатывает входную информацию и в процессе измененияcвоего состояния во времени формирует совокупность выходных сигналов. Все модели нейронных сетей требуют обучения или расчета весов связей. В общем случае обучение - такой выбор параметров сети, при котором сеть лучше всего справляется с поставленной проблемой [21].

Обучение нейронных сетей [20] сосредоточено на следующих объектах:

- слоях, которые объединяются в сеть (или модель);
- исходных данных и соответствующих им целях;
- функции потерь, которая определяет сигнал обратной связи, используемый для обучения;
- оптимизаторе, определяющем, как происходит обучение.

Их связь можно представить, как показано на рис. 1: сеть состоит из слоев, которые объединяются в цепочку и отображают исходные данные в предсказания. Затем функция потерь сравнивает эти предсказания с целями и возвращает значение потери: меру соответствия предсказания, произведенного сетью, ожидаемому результату. Оптимизатор использует это значение потери для изменения весов сети.

Рис. 1: Связь между сетью, слоями, функцией потерь и оптимизатором



Значение выхода у каждого нейрона представимо в виде формулы:

$$y = \phi(\sum_{i=1}^{n} \omega_i x_i + b),$$

где b - смещение данного нейрона, ϕ-функция активации, ω_i - веса нейрона и x_i - входные значения нейрона.

Нейроны располагаются слоями, при этом каждый нейрон из следующего слоя связан со всеми нейронами предыдущего. Основными видами слоев в нейронной сети являются входные слои, скрытые слои и выходные слои.

В исходном алгоритме используются две функции активации, преобразующие выходные сигналы нейронов: сигмоида и ReLU. Первая представима в виде $\sigma(x) = \frac{1}{1+e^{-x}}$ и чаще всего используется для решения задачи классификации.

Вторая представима в виде $f(x) = \max(0, x)$ и, в отличие от сигмоиды, исправляет проблему исчезающего градиента, при котором обучаемые веса почти не обновляются из-за маленького градиента.

Рис. 2: Сигмоида

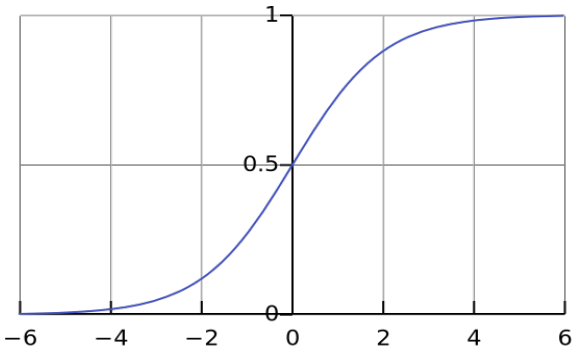
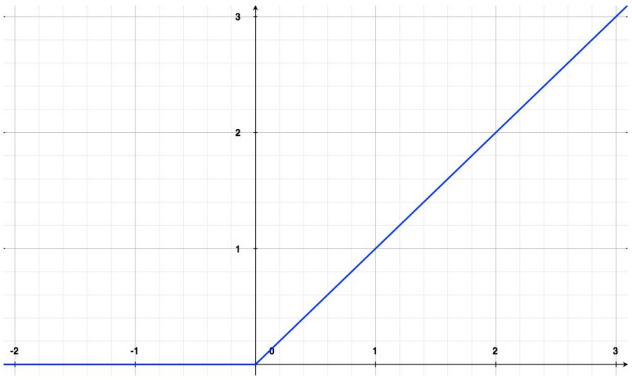


Рис. 3: ReLU



Данный алгоритм обучается с помощью метода обратного распостранения ошибки (Backpropagation). При обучении нейронной сети методом градиентного спуска вычисляется функция потерь, которая показывает, насколько далеко предсказания сети от истинных значений. Обратное распространение позволяет вычислить градиент функции потерь по отношению к каждому из весов сети. Благодаря чему, каждый вес обновляется индивидуально, чтобы постепенно уменьшить функцию потерь на протяжении многих итераций обучения. Обратное распространение включает в себя вычисление градиента, проходящего в обратном направлении через прямую сеть от последнего слоя сети до первого.

Каждому пользователю на момент времени t соответствует вектор прослушиваний $p_u^{(t)} = (p_{u,1}^{(t)}, p_{u,2}^{(t)}, ..., p_{u,m}^{(t)})$, где $p_{u,i}^{(t)} = 1$, если $r_{u,i}^{(t)} > 0$, то есть, если пользователь u хотя бы раз прослушал трек i до момента времени t,

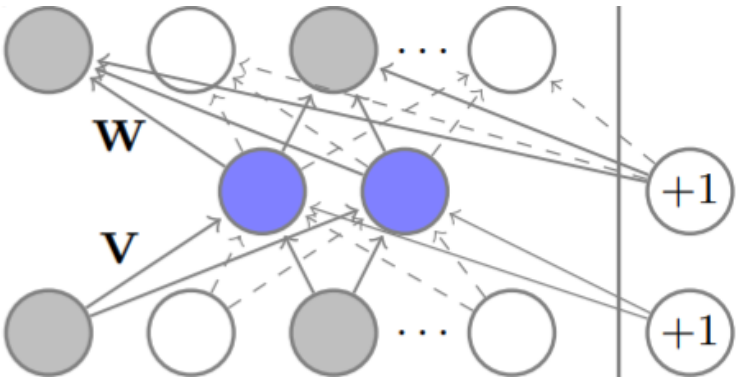
где $t \in [0, +\inf]$, а если наоборот - $r_{u,i}^{(t)} = 0$, то $p_{u,i}^{(t)} = 0$. Вектор $p_u^{(t)}$ подаётся на вход нейронной сети, которая на выходе пытается предсказать, каков будет вектор прослушиваний у клиента через некоторое время $p_u^{(t+1)}$. Автоэнкодер в данном случае состоит из трёх слоёв: входного, скрытого и выходного. Для каждого нейрона в скрытом слое используется функция активации relu. Затем для каждого выходного нейрона используется сигмоида, благодаря которой на выходе получаем значения, лежащие в промежутке (0,1).

Таким образом, для пользователя $u \in U$ на выходе получем вектор:

$$h(p_u^{(t)}, \theta) = f(W \cdot g(V p_u^{(t)} + \mu) + b).$$

$f(.)$ и $g(.)$ - функции активации сигмоида и relu соответственно. Здесь $\theta = \{W, V, \mu, b\}$, обучаемые веса $W \in \mathbb{R}^{m \times k}$, $V \in \mathbb{R}^{k \times m}$, $\mu \in \mathbb{R}^k$ и $b \in \mathbb{R}^m$. То есть, количество нейронов в скрытом слое k, данное число настраиваемое и подбирается, исходя из результатов работы алгоритма на валидационных данных. Параметры θ обучаются, используя метод обратного распостранения ошибки, вычисляющий градиенты, которые используются при обновлении весов нейронной сети.

Рис. 4: Архитектура autoencoder



Исходная модель минимизирует логистическую функцию с положи-

тельным весом:

$$err = \sum_{u \in U} \frac{1}{|I|} \sum_{i \in I} w \cdot p_{u,i}^{(t+1)} \cdot \log h(p_u^{(t)}, \theta) + (1 - p_{u,i}^{(t+1)}) \cdot \log(1 - h(p_u^{(t)}, \theta)),$$

где w - вес для положительных значений $p_{u,i}^{(t+1)}$. В исходной задаче необходимо, чтобы w был больше единицы, так как выходной вектор модели состоит в основном из нулей и нам важно, чтобы модель преимущественно обучалась на положительных примерах, то есть, училась предсказывать треки для клиентов, которые он, скорее всего, прослушает.

После обучения модели для того, чтобы рекомендовать список из k треков пользователю $u \in U$, на вход сети подается вектор прослушиваний пользователя u. Затем на выходе треки, которые клиент ещё не слушал, сортируются по соответствующим результатам, выданным моделью, и k треков с наибольшими значениями на выходе рекомендуются пользователю.

Глава 2. Практические результаты

Для анализа результатов на языке Python в открытом приложении Jupyter Notebook была реализована программа. В ней использовались такие библиотеки, как scipy, implicit и другие.

В основу исследования легло задание, полученное при прохождении практики в отделе музыки компании ООО «Мейл.Ру». Цель исследования – реализация модели коллаборативной фильтрации, которая для каждого пользователя рекомендует наиболее подходящие треки таким образом, чтобы они были персонализированными, т. е. подобранными исключительно по предпочтениям клиента. Необходимым условием создания продукта было, чтобы алгоритм обучался не более двух часов, таким образом нейронные сети с большим количеством слоёв и сети с более сложной структурой не были рассмотрены в данном исследовании.

В качестве данных были предоставлены две разреженные матрицы, каждая размерности 6631477 $\times$ 119990, где первое значение представляет количество пользователей, а второе – количество музыкальных треков. Первая матрица сформирована на конец сентября 2020 года, и в ней содержится 370 357 786 ненулевых значений, а вторая – на конец октября 2020 года, и в ней содержится 376 479 076 ненулевых значений. Посредством обнуления во второй матрице элементов, которые уже содержатся в первой, была сформирована третья матрица. Обозначим первую матрицу за тренировочные данные, а третью– за валидационные данные.

Итого в валидационной матрице осталось 948588 пользователей с хотя бы одним прослушанным треком, 69484 клиентов с двадцатью и более прослушанными треками и 15656600 ненулевых значений. А в тренировочных данных пользователей, прослушавших 20 или более треков – 4904258 и в среднем каждый клиент прослушал около 55 различных треков.

Сравнение моделей производится с использованием шести метрик: HR@10, $NDCG$@10, APR@10, HR@100, $NDCG$@100, APR@100. То есть, оценива-

ются списки из 10 и 100 аудиозаписей, предложенных моделью для каждого пользователя, так как музыкальная платформа в социальной сети «Одноклассники» может рекомендовать для клиентов только списки из 10 и 100 треков.

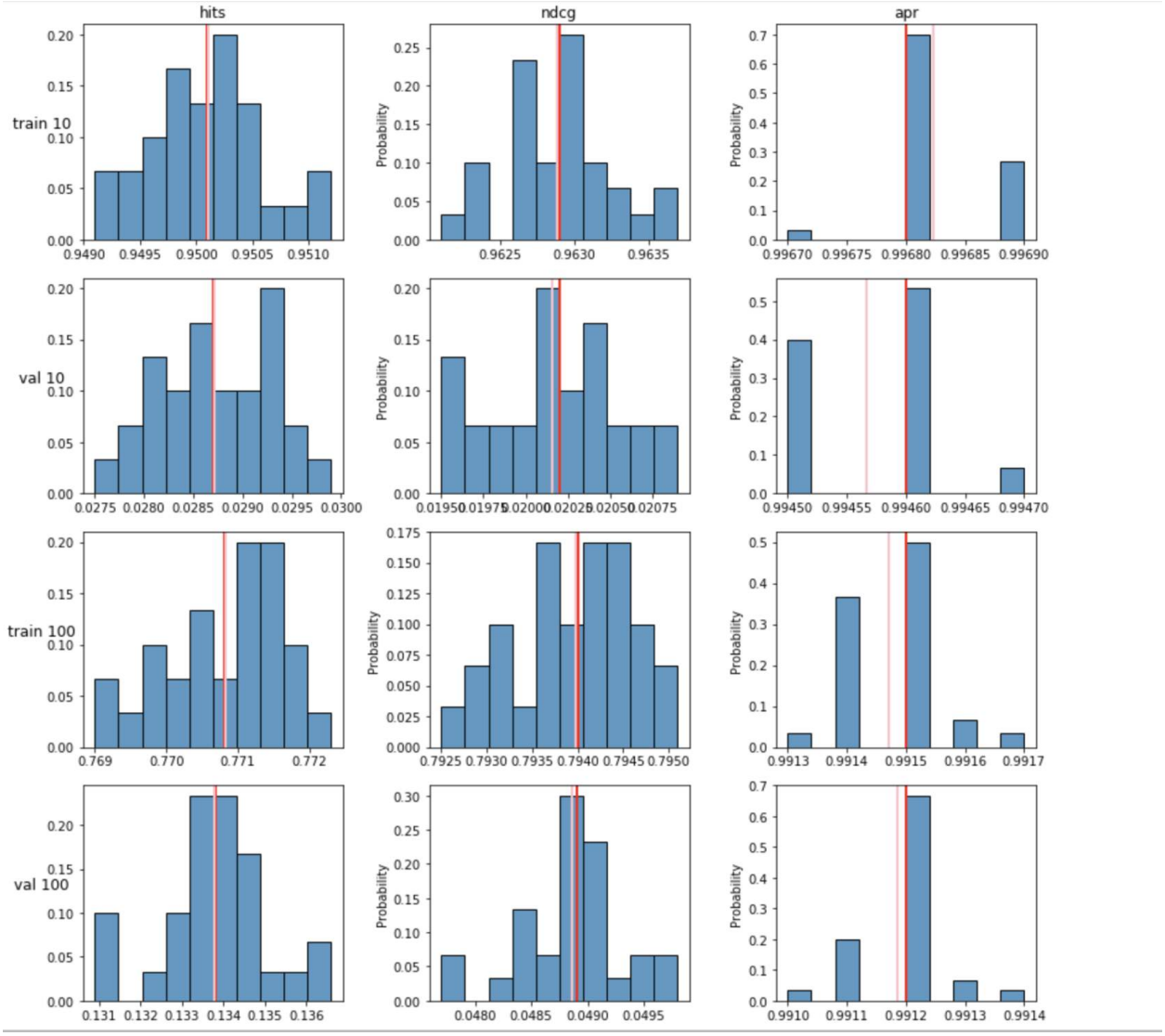
Подсчет метрик на валидационных данных, после обучения модели на тренировочных, осуществлялся на 10% случайно выбранных пользователях, которые прослушали хотя бы один трек. Но даже для оценки 10% случайно выбранных клиентов уходит в среднем 15 минут.

Дабы убедиться, что оценки метрик для 10% клиентов являются несмещенными, то есть, чтобы для каждой метрики была справедлива формула:

$$E(\hat{\theta}) = \theta, \tag{4}$$

где $\hat{\theta}$ - это значение метрики, посчитанное для 10% пользователей из валидационных данных, θ - значение метрики, посчитанное для всех пользователей из валидационных данных. Была построена таблица из гистограмм, где по столбцам обозначены метрики, а по строкам - на какой матрице из двух считались метрики и какова длина списка рекомендованных треков. Все значения показателей качества были посчитаны для алгоритма ближайших соседей с функцей tf-idf, обученной на тренировочной матрице, то есть, модель обучалась, затем 30 раз рандомным образом выбирались 10% клиентов и уже для них считались метрики качества.

Рис. 5: Гистограммы со значениями разных метрик, посчитанных на 30-ти разных тестовых выборках.



В каждом графике розовая линия - это математическое ожидание значений конкретной метрики, посчитанной на десяти процентах от общего количества, а красной обозначается значение показателя при подсчете на всех пользователях, прослушавших хотя бы одну аудиозапись. Из гистограмм можно сделать два вывода: во-первых, уравнение (4) справедливо, а во-вторых, разброс значений во всех графиках невелик (не доходит даже до $\pm 0,01$).

Следовательно, чтобы сэкономить временя, было принято решение считать метрики по десяти процентам клиентов из валидационной матри-

цы. Ниже представлены таблицы со значениями метрик разных алгоритмов, обученных и тестируемых на тренировочных и валидационных данных соответственно.

В первом случае после обучения алгоритмов, модели тестировались на тренировочных данных, затем на валидационных.

Таблица 1: Работы разных алгоритмов на тренировочных данных

модели	HR@10	NDCG@10	APR@10
Als	0.5305	0.5863	0.9625
Als с bm25	0.4097	0.4266	0.9807
Lightfm с warp	0.3318	0.3487	0.9867
Nearest Neighbors с TF-IDF	0.9505	0.9632	0.9968
Nearest Neighbors	0.9298	0.9448	0.9986
модели	HR@100	NDCG@100	APR@100
Als	0.3641	0.405	0.9298
Als с bm25	0.3998	0.3811	0.9603
Lightfm с warp	0.3563	0.3216	0.9729
Nearest Neighbors с TF-IDF	0.7697	0.7933	0.9915
Nearest Neighbors	0.8352	0.8429	0.9977

Из таблицы 1 можно сделать вывод, что алгоритмы Nearest Neighbors с TF-IDF и Nearest Neighbors слишком хорошо запоминают обучающие взаимодействия клиентов и аудиозаписей, в связи с этим обобщающая способность этих алгоритмов невелика.

Таблица 2: Работы разных алгоритмов на валидационных данных

модели	HR@10	NDCG@10	APR@10
Als	0,036	0,0257	0,972
Als с bm25	**0,0382**	**0,0273**	**0,9854**
Lightfm с warp	0,0334	0,0239	0,9876
Nearest Neighbors с TF-IDF	0,0286	0,0202	0,9945
Nearest Neighbors	0,011	0,0076	0,9997
модели	HR@100	NDCG@100	APR@100
Als	0,1545	0,0575	0,9413
Als с bm25	**0,1723**	**0,0634**	**0,9677**
Lightfm с warp	0,1647	0,0594	0,9743
Nearest Neighbors с TF-IDF	0,1305	0,0481	0,9912
Nearest Neighbors	0,0557	0,02	0,9985

Исходя из таблицы 2, алгоритм als с bm25 рекомендует наиболее подходящие и уникальные треки, выбор которых на предпочтениях пользователей.

Стоит упомянуть, что для каждого алгоритма подбирались гиперпараметры по значениям метрик, посчитанных на валидационной матрице. В таблице 1 и 2 представлены посчитанные метрики для моделей, с наиболее подходящими гиперпараметрами.

Для алгоритма ALS оптимальная размерность векторов пользователей и треков равна 120, коэффициент регуляризации - 0,01 и количество итераций - 10, то есть, это количество циклов, в которых сначала вычисляются векторы пользователей, затем векторы треков. Для lightfm наиболее подходящая размерность вектора клиентов - 150, количество эпох - 10, иными словами, алгоритм 10 раз обучался на тренировочной матрице, постепенно изменяя параметры обучения модели. В функции bm25 параметры, контролирующие количество прослушиваний трека пользователем и активность клиента, равны 2,25 и 0,8 соответственно. То есть для каждой модели подбирались такие гиперпараметры, с которыми алгоритм показывал наибольшие значения метрик на валидационных данных.

Далее был проведён эксперимент, с целью убедиться, что с увеличением тренировочных данных построенная модель будет рекомендовать более подходящие треки клиентам. Среди всех аудиозаписей рандомным образом было выбрано 15% треков от общего количества, столбцы которых в валидационных данных были добавлены в новую тестовую матрицу, которая изначально была нулевой. Столбцы остальных 85% треков были добавлены к соответствующим столбцам в тренировочной матрице. Модель Nearest Neighbors с tf-idf несколько раз была обучена на тренировочных матрицах, к которым были добавлены 70, 75, 80, и 85 процентов столбцов валидационной матрицы, и все эти разы алгоритм оценивался на тестовой матрице.

Таблица 3: Результаты работы алгоритма ближайших соседей с функцией tf-idf, обученного на увеличенной тренировочной выборке.

Процент добавочных столбцов	HR@10	NDCG@10	APR@10
70	0,0485	0,0296	0,9945
75	0,0509	0,0317	0,9945
80	0,0521	0,0327	0,9945
85	0,0536	0,0341	0,9945
Процент добавочных столбцов	HR@100	NDCG@100	APR@100
70	0,1929	0,063	0,9914
75	0,1972	0,0655	0,9914
80	0,1995	0,0668	0,9914
85	0,2027	0,0686	0,9914

Из приведённой таблицы 3 можно сделать вывод, что с увеличением информации о прослушанных клиентом треков, увеличивается и предсказательная способность модели. Следовательно, если бы построенные модели были встроены в музыкальную платформу, необходимо было бы периодически заново обучать модели с уже новыми накопившимися данными.

Для обучения алгоритма autoencoder была сформирована промежуточная дополнительная матрица, в которой были все взаимодействия изначальной тренировочной матрицы и были добавлены столбцы из валидационной таблицы, соответствующие 50% объектов, выбранных рандомным

образом. Столбцы случайно выбранных объектов также обнуляются в валидационной матрице. Это делается для того, чтобы смоделировать ситуацию, где тренировочная матрица соответствует матрице прослушиваний в момент времени t, промежуточная матрица соответствует моменту времени $(t+1)$, валидационная - $(t+2)$. После этого подаются на вход векторы каждого клиента из тренировой таблицы, и на выходе модель должна предсказывать вектор прослушиваний каждого клиента, соответствующий моменту времени $(t+1)$. А метрики уже считаются для случая, когда на вход подаётся промежуточная матрица и на выходе ожидаются векторы прослушиваний клиентов новой валидационной матрицы. Аналогично, на промежуточной матрице был обучен алгоритм als с bm25 и протестирован на валидационной матрице.

Таблица 4: Метрики, посчитанные для каждого из двух алгоритмов, обученных на промежуточной матрице.

модели	HR@10	NDCG@10	APR@10
Als с bm25	0,0493	0,0318	0,9841
Autoencoder	**0,0578**	**0,0406**	**0,9834**
модели	HR@100	NDCG@100	APR@100
Als с bm25	0,2058	0,0692	0,9665
Autoencoder	**0,2383**	**0,0871**	**0,967**

В таблице 4 приводятся результаты для autoencoder с 2000 нейронами в скрытом слое, с весом для положительных примеров, равным 2000, с коэффициентом обучения 0,00001 и с 10 эпохами обучения, т.е. вектор прослушиваний каждого клиента подавался алгоритму десять раз. Примечательно, что алгоритм показывал очень низкие значения качества на валидации для часто используемых значений коэффициентов обучения: 0,01, 0,001, 0,005.

В статье [9] также рассматривается вариант, когда алгоритм работает не с векторами прослушиваний пользователей, а с векторами треков. Размерность каждого вектора треков в нашем случае равна 6631477, и в

связи с этим, подавая вектор такой размерности на вход модели и принимая на выходе вектор аналогичной размерности, количество обучаемых весов алгоритма autoencoder становится во много раз больше, чем количество подаваемых на вход векторов (тренировочных примеров). Так как в данной задаче всего около 120 тысяч тренировочных примеров, рассматриваемая нейронная сеть, учитывающая векторы треков, не сможет достаточтно хорошо подобрать веса, чтобы показать высокое качество работы на тестовых данных.

Из таблицы 4 видно, что два алгоритма рекомендуют примерно одинаковые по популярности музыкальные композиции, однако модель autoencoder предлагает более подходящие треки для клиентов, следовательно, она лучшая среди моделей, описанных в таблице 2.

Заключение

Таким образом, в данной работе рассмотрено несколько подходов к реализации алгоритмов коллаборативной фильтрации.

Были построены алгоритмы для рекомендательных систем и подобраны для них гиперпараметры. Также были реализованы метрики, каждая из которых по-разному характеризует рекомендуемые списки аудиозаписей. Для каждой модели были посчитаны метрики. Было показано, что оценки метрик, посчитанных на 10% клиентов, являются несмещёнными для метрик, посчитанных на всех пользователях. Оказалось, что алгоритм Als рекомендует наиболее подходящие треки для пользователей, если каждый элемент тренировочной матрицы преобразовать с помощью функции ранжирования bm25. Такая модель работает даже лучше, чем Lightfm с warp loss функцией, хотя этот алгоритм достаточно популярен в таком роде задач. Но все же модель, показавшая себя лучше всех, среди рассмотренных, это autoencoder. Также было показано, что с ростом тренировочных данных, качество рекомендаций модели увеличивается на тестовых данных.

В будущем планируется реализовать алгоритмы для гибридных рекомендательных систем, то есть, помимо информации об истории прослушиваний треков каждого пользователя, модель будет использовать дополнительную информацию о музыкальных треках и пользователях.

Список литературы

[1] Y. Hu, Y. Koren, C. Volinsky, Collaborative filtering for implicit feedback datasets, Eighth IEEE International Conference on Data Mining, 2008.

[2] G. Takács, I. Pilászy, D. Tikk, Applications of the conjugate gradient method for implicit feedback collaborative filtering, RecSys '11: Proceedings of the fifth ACM Conference on Recommender Systems, 2011.

[3] «Ranking evaluation metrics for recommender systems» [В интернете]: URL: https://towardsdatascience.com/ranking-evaluation-metrics-for-recommender-systems-263d0a66ef54 [дата обращения: 10.01.2021].

[4] «Рекомендации в Okko: как заработать сотни миллионов, перемножив пару матриц» [В интернете]: URL: https://habr.com/ru/company/okko/blog/454224/ [дата обращения: 10.01.2021].

[5] «Методы оптимизации нейронных сетей» [В интернете]: URL: https://habr.com/ru/post/318970/ [дата обращения: 10.01.2021].

[6] J. Weston, S. Bengio, N. Usunier, WSABIE: Scaling Up To Large Vocabulary Image Annotation, International Joint Conferences on Artificial Intelligence, 2011.

[7] M. Kula, Metadata Embeddings for User and Item Cold-start Recommendations, ArXiv.org (Cornell University Library), 2015.

[8] N. Usunier, D. Buffoni, P. Gallinari, Ranking with ordered weighted pairwise classification, Proceedings of the 26th International Conference on Machine Learning, p. 1057–1064, Montreal, 2009.

[9] S. Sedhain†, A. Krishna Menon, S. Sanner, L. Xie, AutoRec: Autoencoders Meet Collaborative Filtering, Association for Computing Machinery (ACM), Montreal, 2016.

[10] F. Ricci, L. Rokach, B. Shapira and P. B. Kantor, Recommender Systems Handbook, LLC: Springer Science+Business Media, 2011.

[11] «Рекомендательные системы сегодня –необходимость для бизнеса» [В интернете]: URL: https://blog.heyml.com/ [дата обращения: 10.01.2021].

[12] Т. Сегаран, Программируем коллективный разум, СПб.: Символ-Плюс, 2008.

[13] Н. Арзуманян, М. Смирнов, М. Смирнова, Анализ использования различных мер сходства в коллаборативной фильтрации, Процессы управления и устойчивость том 3, p. 342 - 347, 2016.

[14] К. В. Воронцов, Лекции по метрическим алгоритмам классификации, p. 2-7, 2008.

[15] Х. Лейн, Х. Хапке, К. Ховард, Обработка естественного языка в действии, p. 132-139, 2020.

[16] K.Falk, Practical Recommender Systems, p. 227- 229, 2019.

[17] F. Kane, Building Recommender Systems with Machine Learning and AI, p. 39 - 47, 2018.

[18] P. Achlioptas, Stochastic Gradient Descent in Theory and Practice, Stanford, p. 1 - 2, 2019.

[19] J. Duchi, E. Hazan, and Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, The Journal of Machine Learning Research, 2011.

[20] Ш. Франсуа, Глубокое обучение на Python, p. 81 - 85, 2018.

[21] В.С. Ростовцев, Искусственные нейронные сети, Издательство "Лань p. 4 - 7, 2021

Приложение

Ниже представлен программный код, реализующий описанные выше алгоритмы.

Загрузка необходимых библиотек:

```
import math
from typing import List
from heapq import nlargest
from itertools import product
import time
from multiprocessing import Pool

import pandas as pd
import numpy as np
from scipy import sparse
import scipy
import pickle
from lightfm import LightFM
from implicit.als import AlternatingLeastSquares as Als
from implicit import nearest_neighbours
import multiprocessing
```

Определяем класс с методами, реализующими подсчёт метрик:

```
class Metrics:

    def __init__(self):
        pass

    def calculate_hits_k(self, real: List[int], recommendations: List[int], k: int = 10)\
        -> float:
        '''
            recommendations: list of recommended tracks
            real: list of tracks which user have listened
        '''
        try:
            return len(set(recommendations[:k]) & set(real))/ min(k, len(real))
        except ZeroDivisionError:
            print("k or length of list tracks which user have listened equals 0 (hits_k)")
            return 0

    def calculate_ndcg_k(self, real: List[int], recommendations: List[int], k: int = 10) \
        -> float:
        real_s = set(real)
        dcg = np.sum([i in real_s for i in recommendations]/ np.log2(np.arange(2, k + 2)))
        border = min(k, len(real_s))
        idcg = np.sum(np.ones(border)/ np.log2(np.arange(2, border + 2)))
        try:
            return dcg/idcg
        except ZeroDivisionError:
            print("k or length of list tracks which user have listened equals 0 (ndcg_k)")
            return 0
```

```
    def calculate_apr_k(self, recommendations: List[int], top_tracks: np.ndarray,\
        k:int = 10) -> float:
        '''
            This function calculate scaled average popularity rank. Calculating mean
            popularity rank of k the most popular tracks and then then it is divided
            by mean popularity
            of recommended tracks. Further we consider the difference to one and
            the calculated number.

            If this function return 0, our k recommendation tracks are the most popular now.
            If on the contrary our function returns 1, our k recommendation tracks are
            the most
            unpopular now.
            Args:
                recommendations: recommendation list of tracks
                top_tracks: sorted list of all tracks by the number of times they
                were played
                k: number of first recommendations which we want to use
        '''
        recommendations = recommendations[:k]
        best_position_in_top = (len(recommendations) + 1)/2
        try:
            assert len(np.nonzero(np.in1d(top_tracks, recommendations))[0]) == \
                len(recommendations)
            average_popularity_rank = sum(np.nonzero(np.in1d(top_tracks, recommendations))
                \[0] + 1)/len(recommendations)
        except AssertionError:
            print("There is one or more tracks in recommendation"+\
            "list which isn't in sorted list of all tracks")
            print(f"{len(np.nonzero(np.in1d(top_tracks, recommendations))[0])}"+\
            "!="+f"{len(recommendations)}")
            raise
        except ZeroDivisionError:
            print("Length of recommendation list is 0")
            return 0
        return 1 - best_position_in_top/ average_popularity_rank
```

Реализованы две функции, благодаря которым подсчёт метрик для каждого клиента производится параллельно.

```
# Two functions helping with parallel computing inModelScoring class
def fun(f, q_in, q_out):
    while True:
        i, x = q_in.get()
        if i is None:
            break
        q_out.put((i, f(x)))


def parmap(f, X, nprocs=multiprocessing.cpu_count()):
    q_in = multiprocessing.Queue(1)
    q_out = multiprocessing.Queue()

    proc = [multiprocessing.Process(target=fun, args=(f, q_in, q_out))
            for _ in range(nprocs)]
    for p in proc:
```

```
        p.daemon = True
        p.start()

    sent = [q_in.put((i, x)) for i, x in enumerate(X)]
    [q_in.put((None, None)) for _ in range(nprocs)]
    res = [q_out.get() for _ in range(len(sent))]

    [p.join() for p in proc]

    return [x for i, x in sorted(res)]
```

Ниже, для каждой модели реализован метод, который возвращает список рекомендованных треков для случаев тестирования модели на тренировочных и тестовых данных. В методе inference рандомным образом выбирается некоторая часть пользователей, для которых будет тестироваться алгоритм, далее метод вызывает функцию оценки соответствующей модели и по итогу возвращает посчитанные значения метрик.

```
class ModelScoring(Metrics):

    def __init__(self):
        super().__init__()

    def lightfm_scoring(self, user_K: List[int]) -> List[float]:
        """
            user_K: list containing id of user and list of k
        """
        K = user_K[1]
        user = user_K[0]
        max_K = max(K)
        train_items = set(self.train[user].nonzero()[1])
        train_real = list(train_items)
        val_real = self.validation[user].nonzero()[1]

        train_scores = self.model.predict(user, list(range(self.train.shape[1])))
        train_dict_items_score = dict(zip(range(self.train.shape[1]), train_scores))
        train_recommendations = nlargest(max_K, train_dict_items_score, key =\
        train_dict_items_score.get)

        items_without_train = np.delete(range(self.train.shape[1]), list(train_items))
        val_scores = np.delete(train_scores, list(train_items))
        val_dict_items_score = dict(zip(items_without_train, val_scores))
        val_recommendations = nlargest(max_K, val_dict_items_score, \
        key = val_dict_items_score.get)
        return self.calculate_metrics(train_real, val_real, train_recommendations,\
        val_recommendations, K)

    def als_scoring(self, user_K: List[int]) -> List[float]:
        K = user_K[1]
        user = user_K[0]
        max_K = max(K)
        train_real = self.train[user].nonzero()[1]
        val_real = self.validation[user].nonzero()[1]
```

```
    train_recommendations = np.array(self.model.recommend(user, user_items= self.train,\
    N = max_K, filter_already_liked_items= False))[:, 0]
    val_recommendations = np.array(self.model.recommend(user, user_items= self.train,\
    N = max_K, filter_already_liked_items= True))[:, 0]
    return self.calculate_metrics(train_real, val_real, train_recommendations,\
    val_recommendations, K)

def item_neighbours_scoring(self, user_K: List[int]) -> List[float]:
    K = user_K[1]
    user = user_K[0]
    max_K = max(K)
    train_real = self.train[user].nonzero()[1]
    val_real = self.validation[user].nonzero()[1]
    train_recommendations = np.array(self.model.recommend(user, user_items= self.train,\
    N = max_K, filter_already_liked_items= False))[:, 0]
    val_recommendations = np.array(self.model.recommend(user, user_items= self.train,\
    N = max_K, filter_already_liked_items= True))[:, 0]
    return self.calculate_metrics(train_real, val_real, train_recommendations,\
    val_recommendations, K)

def calculate_metrics(self, train_real: List[int], val_real: List[int],\
    train_recommendations: List[int], val_recommendations: List[int], K: int)\
        -> List[float]:
    """
        Args:
            train_real: list of tracks which user have listened in training dataset.
            val_real: list of tracks which user have listened in validation dataset.
            train_recommendations: list of tracks which our algorithm have recommended
            in training dataset.
            val_recommendations: list of tracks which our algorithm have recommended
            in validation dataset.
        return:
            result: list of all resulting metrics

    """
    result = []
    for k in K:
        result.extend([self.calculate_hits_k(train_real, train_recommendations[:k], k),\
            self.calculate_ndcg_k(train_real, train_recommendations[:k], k),\
            self.calculate_apr_k(train_recommendations[:k], self.train_top_tracks , k),\
            self.calculate_hits_k(val_real, val_recommendations[:k], k),\
            self.calculate_ndcg_k(val_real, val_recommendations[:k], k), \
            self.calculate_apr_k(val_recommendations[:k], self.val_top_tracks , k)])
    return result

def inference(self, function_for_scoring, fraction, num_threads):
    """
        Function that testing model and returning all necessary metrics
        Args:
            function_for_scoring: name of scoring function. It depends on model which
            were trained.
            fraction: part of users in validation data that will be used in inference
        return:
            list of metrics
    """
    FRACTION_OF_VALID = fraction
    users_in_valid = np.unique(self.validation.nonzero()[0])
    part_of_users_in_valid = np.random.permutation(users_in_valid)[:\
```

```
            int(len(users_in_valid)* FRACTION_OF_VALID)]
        results = []
        K_s = [10, 100]
        results.append(parmap(function_for_scoring, zip(part_of_users_in_valid,\
            np.full((len(part_of_users_in_valid), len(K_s)), K_s)), num_threads))
        results = np.array(results[0])
        metrics = []
        for i in range(2):
            mean_hits_K_train, mean_ndcg_K_train, mean_apr_K_train, mean_hits_K_val,\
                mean_ndcg_K_val, mean_apr_K_val = [int(sum(results[:, j])/\
                    len(part_of_users_in_valid)*10**4)/10**4 \
                    for j in range(i * 6 , (i+1) * 6)]
            metrics.append((mean_hits_K_train, mean_ndcg_K_train, mean_apr_K_train,
                            mean_hits_K_val, mean_ndcg_K_val, mean_apr_K_val))
        return metrics
```

Методы следующего класса обучают модели с заданными гиперпараметрами и возвращают посчитанные метрики на тренировочных и тестовых данных.

```
class Model(ModelScoring):

    def __init__(self, train, validation, fraction_validation=0.1, num_threads=50,\
    train_top_tracks, result_top_tracks):
        """
            Args:
                fraction_validation: part of users in validation data that will be used in
                inference.
                num_threads: number of threads for training.
                train_top_tracks: top listened tracks in training dataset.
                result_top_tracks: top listened tracks in training data and validation data.
        """
        super().__init__()
        self.train = train
        self.validation = validation
        self.fraction_validation = fraction_validation
        self.num_threads = num_threads
        self.train_top_tracks = train_top_tracks
        self.val_top_tracks = result_top_tracks

    def form_top_tracks(self, user_item_matrix: scipy.sparse.csr.csr_matrix) -> np.ndarray:
        '''
            return: sorted list of all tracks by the number of times they were played
        '''
        return np.argsort(user_item_matrix.sum(axis = 0).A.reshape(-1))[::-1]

    def fit_lightfm(self, no_components=150, learning_schedule='adagrad', loss='warp',\
        learning_rate=0.05, rho=0.95, epsilon=1e-06, max_sampled=10, random_state=None, \
        epochs=10):
        self.model = LightFM(no_components=no_components,learning_schedule=learning_schedule,\
            loss=loss, learning_rate=learning_rate, rho=rho, epsilon=epsilon, max_sampled=\
            max_sampled, random_state=random_state)
        self.model.fit(self.train, epochs=epochs, num_threads=self.num_threads)
        return self.inference(self.lightfm_scoring, self.fraction_validation, self.num_threads)

    def fit_als(self, bm_25=True, K1=100, B=1, factors=120, regularization=0.01, use_cg=True,\
```

```
    iterations=15, calculate_training_loss=False, random_state=None):
    """
        K1, B: for bm25 function.
    """
    self.model = Als(factors=factors,regularization=regularization, use_cg=use_cg,\
    iterations=iterations, num_threads=self.num_threads)
    if bm_25:
        self.train_transformed = nearest_neighbours.bm25_weight(self.train, K1, B)
        self.model.fit(self.train_transformed.T)
    else:
        self.model.fit(self.train.T)
    return self.inference(self.als_scoring, self.fraction_validation, self.num_threads)

def fit_als_big_train(self, bm_25=True, K1=1.25, B=0.8, factors=120, regularization=0.01,\
    use_cg=True, iterations=15, calculate_training_loss=False, random_state=None):
    self.model = Als(factors=factors,regularization=regularization,
                        use_cg=use_cg, iterations=iterations, num_threads=self.num_threads)
    base_train = self.train.copy()
    base_validation = self.validation.copy()
    nonzero_val_items = np.unique(self.validation.nonzero()[1])
    rand_val_items = np.random.permutation(nonzero_val_items)
    val_items = rand_val_items[:int(len(nonzero_val_items) * 0.5)]
    set_val_items = set(val_items)
    indices_vaildation = np.vstack(self.validation.nonzero())
    indices_nonzero_valid = [i for i in range(indices_vaildation.shape[1]) \
    if indices_vaildation[1,i] in set_val_items]
    indices_valid_model = indices_vaildation[:,indices_nonzero_valid]
    self.train[indices_valid_model[0], indices_valid_model[1]] +=\
        self.validation[indices_valid_model[0], indices_valid_model[1]]
    self.validation[indices_valid_model[0], indices_valid_model[1]] = 0
    if bm_25:
        self.train_transformed = nearest_neighbours.bm25_weight(self.train, K1, B)
        self.model.fit(self.train_transformed.T)
    else:
        self.model.fit(self.train.T)
    all_metrics = self.inference(self.als_scoring, self.fraction_validation,\
    self.num_threads)
    self.train = base_train.copy()
    self.validation = base_validation.copy()
    return all_metrics

def fit_inference_N_knn(self, K=100, n_inference=30):
    self.model = nearest_neighbours.TFIDFRecommender(K = K, num_threads = self.num_threads)
    self.train = self.train.astype("float")
    self.model.fit(self.train.T)
    all_metrics = []
    self.fraction_validation = 0.1
    for i in range(n_inference):
        all_metrics.append(self.inference(self.item_neighbours_scoring,\
        fraction_validation, self.num_threads))
    return all_metrics

def fit_big_train_knn(self, K = 100, train_percents=[0.7, 0.75, 0.8, 0.85],\
    percent_test = 0.15):
    all_metrics = []
    base_train = self.train.copy()
    base_validation = self.validation.copy()
    print(len(train_percents))
```

```
        val_items = np.unique(validation.nonzero()[1])
        rand_val_items = np.random.permutation(val_items)
        test_items = rand_val_items[:int(len(val_items) * percent_test)]
        other_items = rand_val_items[int(len(val_items) * percent_test):]
        set_test_items = set(test_items)
        for train_percent in train_percents:
            self.model = nearest_neighbours.TFIDFRecommender(K = K, num_threads =\
                self.num_threads)
            train_items = np.random.permutation(other_items)[: int(len(val_items) *\
                (train_percent))]
            set_train_items = set(train_items)
            indices_vaildation = np.vstack(self.validation.nonzero())
            addition_train = [i for i in range(indices_vaildation.shape[1])\
            if indices_vaildation[1,i] in set_train_items]
            indices_new_train = indices_vaildation[:,addition_train]
            self.train[indices_new_train[0], indices_new_train[1]] +=\
                self.validation[indices_new_train[0], indices_new_train[1]]
            test_interactions = [i for i in range(indices_vaildation.shape[1])\
                if indices_vaildation[1,i] in set_test_items]
            indices_test = indices_vaildation[:,test_interactions]
            test = sparse.csr_matrix(train.shape, dtype=train.dtype)
            test[indices_test[0], indices_test[1]] = self.validation[indices_test[0],\
                indices_test[1]]
            self.validation = test.copy()
            self.train = self.train.astype("float")
            self.model.fit(self.train.T)
            all_metrics.append(self.inference(self.item_neighbours_scoring, 1,\
                self.num_threads))
            self.train = base_train.copy()
            self.validation = base_validation.copy()
        return all_metrics

    def fit_item_neighbours(self, algorithm = 'tfidf', K=100):
        """
            algorithm: {cos, tfidf}.
        """
        if algorithm == 'cos':
            self.model = nearest_neighbours.CosineRecommender(K = K, num_threads =\
                self.num_threads)
        elif algorithm == 'tfidf':
            self.model = nearest_neighbours.TFIDFRecommender(K = K, num_threads =\
                self.num_threads)
        else:
            raise ("Specify the algorithm correctly")
        self.train = self.train.astype("float")
        self.model.fit(self.train.T)
        return self.inference(self.item_neighbours_scoring, self.fraction_validation,\
        self.num_threads)
```

Функция, формируемая список треков, отсортированных по общему количеству прослушиваний (по популярности).

```
def form_top_tracks(user_item_matrix: scipy.sparse.csr.csr_matrix) -> np.ndarray:
    return np.argsort(user_item_matrix.sum(axis = 0).A.reshape(-1))[::-1]
```

Далее, представлен программный код, формируемый тренировочные и валидационные данные.

Для начала, загружаются датасеты, сформированные на конец сентября и конец октября.

```
# have formed in September
data1 = sparse.load_npz('data_matrix_500.npz')
# have formed in October
data2 = sparse.load_npz('data_matrix_510.npz')
```

Из двух датасетов удаляются клиенты, которые до конца сентября ещё ничего не слушали.

```
# delete new users, which haven't listened any tracks in data1
new_users = np.delete(range(data1.shape[0]), np.unique(data1.nonzero()[0]))
data1 = data1[:new_users[0]]
data2 = data2[:new_users[0]]
sparse.save_npz("train.npz", data1)
sparse.save_npz("data2.npz", data2)
train = sparse.load_npz('train.npz')
data2 = sparse.load_npz('data2.npz')
```

Из двух датасетов удаляются треки, которые в конце сентября ещё никто не слушал, это делается для того, чтобы не было проблемы холодного старта.

```
# delete new tracks, which haven't been listened by users in data1
un = np.unique(train.nonzero()[1])
new_items = list(set(range(119998)) - set(un))
print(new_items)
train = train[:, :new_items[0]]
data2 = data2[:, :new_items[0]]
```

Обнуляем элементы второй таблицы, которые не равны нулю в первой. Это делается для того, чтобы обученные модели рекомендовали пользователям треки, которые ранее они не слушали. Таблицу сформированную на конец сентября называем тренировочной и вторую называем валидационной.

```
# create validation data where each cell in training data is zeroed out.
difference = data2 - train
train_indices = train.nonzero()
pairs_of_indices = [(train_indices[0][i], train_indices[1][i])\
                    for i in range(len(train_indices[0]))]
set_of_pairs_indices = set(pairs_of_indices)
validation = sparse.csr_matrix(difference, copy = True)
difference_nonzero = difference.nonzero()
```

```
for i in range(len(difference_nonzero[0])):
    if (difference_nonzero[0][i], difference_nonzero[1][i]) in set_of_pairs_indices:
        validation[difference_nonzero[0][i], difference_nonzero[1][i]] = 0
sparse.save_npz("train.npz", train)
sparse.save_npz("data2.npz", data2)
sparse.save_npz("validation_matrix.npz", validation)
```

Ниже представлен пример того, как можно посчитать метрики для заданного алгоритма.

```
train_top_tracks = form_top_tracks(train)
val_top_tracks = form_top_tracks(data2)
model = Model(train, validation, 0.1, 50, train_top_tracks, val_top_tracks)
metrics = model.fit_als()
```