Пояснительная записка

Самара 2023

СОДЕРЖАНИЕ

[1.1. Наименование системы 3](#_Toc124368114)

[1.2. Цели и метрики системы 3](#_Toc124368115)

[1.3. Описание формата входных данных 3](#_Toc124368116)

[1.3.1 Events 3](#_Toc124368117)

[1.3.2 Item\_properties 3](#_Toc124368118)

[1.3.3 Category\_tree 4](#_Toc124368119)

[1.4. Описание трансформации датасетов и построения валидации 4](#_Toc124368120)

[1.4.1 Описание трансформации исходного датасета events.csv и построения валидации 4](#_Toc124368121)

[1.4.2 Описание трансформации датасета properties.csv 6](#_Toc124368122)

[1.5. Описание модели, на которой построена рекомендательная система 6](#_Toc124368123)

[1.5.1 LightFM 6](#_Toc124368124)

[1.5.2 Метрики полученные на валидационном датасете 7](#_Toc124368125)

[1.5.3 Общая логика выдачи рекомендаций 7](#_Toc124368126)

[1.6. Поднятие docker-образа системы 8](#_Toc124368127)

[1.7. Описание API сервиса 9](#_Toc124368128)

[1.7.1 Запуск сервиса 9](#_Toc124368129)

[1.7.2 Описание методов сервиса 9](#_Toc124368130)

[1.7.3 Описание /сheck\_html\_response 10](#_Toc124368131)

[1.7.4 Описание /model\_train 10](#_Toc124368132)

[1.7.5 Описание /get\_userid\_dict 11](#_Toc124368133)

[1.7.6 Описание / recommendation 12](#_Toc124368134)

[1.7.7 Endpoint для снятия метрик 12](#_Toc124368135)

1. Общие положения
   1. Наименование системы

Рекомендательная система товаров для компании-ритейлера.

* 1. Цели и метрики системы

Целью и бизнес метрикой создания рекомендательной системы являются повышение прибыли от допродаж в интернет-магазине для компании-ритейлера на 20%.

Техническими метриками являются:

* Precision@3 - показывает, какую долю из 3 рекомендованных товаров, купил пользователь:
* Recall@3 - показывает, какая доля из того, что купил пользователь, приходится на рекомендованное:
  1. Описание формата входных данных
     1. Events

Events — датасет с событиями (файл формата .csv).

Таблица Параметры датасета Events

|  |  |  |
| --- | --- | --- |
| **Наименование параметра** | **Расшифровка** | **Тип данных** |
| timestamp | время события | int64 |
| visitorid | идентификатор пользователя | int64 |
| event | тип события | object |
| itemid | идентификатор объекта | int64 |
| transactionid | идентификатор транзакции, если она проходила | float64 |

* + 1. Item\_properties

Item\_properties — файл с свойствами товаров (файл формата .csv).

Таблица Параметры датасета Item\_properties

|  |  |  |
| --- | --- | --- |
| **Наименование параметра** | **Расшифровка** | **Тип данных** |
| timestamp | момент записи значения свойства | int64 |
| itemid | идентификатор объекта | int64 |
| property | хешированное свойство объекта | object |
| value | значение свойства | object |

* + 1. Category\_tree

Category\_tree — файл с деревом категорий (файл формата .csv).

Таблица Параметры датасета Category\_tree

|  |  |  |
| --- | --- | --- |
| **Наименование параметра** | **Расшифровка** | **Тип данных** |
| categoryid | идентификатор категорий | int64 |
| parentid | идентификатор родительской категории | float64 |

* 1. Описание трансформации датасетов и построения валидации
     1. Описание трансформации исходного датасета events.csv и построения валидации

За исходный датасет взят файл events.csv.

Трансформации проведенные с датасетом:

1. Датасет отсортирован по колонке timestamp:

events\_data = events\_data.sort\_values('timestamp')

1. Добавлен столбец с датой:

events\_data['time'] = events\_data['timestamp'].apply(lambda x: datetime.fromtimestamp(float(x / 1000)))

1. Добавлен столбец с признаком транзакции 1 – транзакция была, 0 – нет:

events\_data['transactionid\_cnt'] = events\_data['transactionid'].apply(lambda x: 0 if math.isnan(x) else 1)

1. Датасет разделен на тренировочный и тестовый сеты по времени. На валидацию выделен 1 месяц. Минимальная дата исходного датасета - 2015-05-03 07:00:04.384, максимальная дата исходного датасета - 2015-09-18 06:59:47.788. За точку для разделения исходного датасета взята 31 день назад от максимальной даты - 2015-08-18.

split\_time = events['time'].max() - pd.Timedelta("31 days") train\_events = events\_data.loc[events\_data['time'] < split\_time]  
test\_events = events\_data.loc[events\_data['time'] >= split\_time]

1. Для обучения модели из тренировочного сета выделены 20 топовых itemid по количеству транзакций:

top\_transact = train\_events['itemid'].loc[train\_events['event'] == 'transaction'].value\_counts().index[  
 :20].values.tolist()

1. Далее для построения модели были взяты только те строки, где покупка была осуществлена

train\_events\_model = train\_events.loc[(train\_events['event'] == 'transaction') &  
 (train\_events['itemid'].isin(top\_transact))]  
test\_events\_model = test\_events.loc[(test\_events['event'] == 'transaction') &  
 (test\_events['visitorid'].isin(train\_events\_model.visitorid.unique()))]

1. Удалены дубликаты по сочетанию 'visitorid', 'itemid'

train\_events\_model = train\_events\_model.drop\_duplicates(subset=['visitorid', 'itemid'])  
test\_events\_model = test\_events\_model.drop\_duplicates(subset=['visitorid', 'itemid'])

1. Созданы отдельные датасеты с нужными нам факторами - 'timestamp', 'visitorid', 'itemid', 'transactionid\_cnt'

train\_events\_lfm = train\_events\_model[['timestamp', 'visitorid',  
 'itemid', 'transactionid\_cnt']].reset\_index(drop=True)  
test\_events\_lfm = test\_events\_model[['timestamp', 'visitorid',  
 'itemid', 'transactionid\_cnt']].reset\_index(drop=True)

1. Были переименованы факторы для удобства 'visitorid': 'user\_id', 'itemid': 'item\_id', 'transactionid\_cnt': 'buy'
2. Созданы pivot\_table для train и test:

# Pivoting TRAIN  
train\_pivot = pd.pivot\_table(train\_events\_lfm,  
 index='user\_id',  
 columns='item\_id',  
 values='buy')  
  
# Pivoting TEST  
test\_pivot = pd.pivot\_table(test\_events\_lfm,  
 index='user\_id',  
 columns='item\_id',  
 values='buy')

1. Была создана сводная таблица из таблицы events, и заполнена нулями. Для тех товаров, которые пользователь купил, ячейки равны 0, для остальных — пропуску.

events\_shell = pd.concat([train\_events\_lfm, test\_events\_lfm])  
shell = pd.pivot\_table(  
 events\_shell,  
 index='user\_id',  
 columns="item\_id",  
 values="buy",  
 aggfunc=lambda x: 0  
)

1. Получены тренировочная и тестовая user-item таблицы, сложив таблицу shell с соответствующими таблицами train\_pivot и test\_pivot. 0 — если пользователь не покупал товар, 1 — если пользователь покупал товар.

train\_pivot = shell + train\_pivot  
test\_pivot = shell + test\_pivot  
train\_pivot = train\_pivot.fillna(0)  
test\_pivot = test\_pivot.fillna(0)

1. Получены разреженные матрицы, используется функция csr\_matrix() из модуля sparse библиотеки scipy:

train\_pivot\_sparse = scipy.sparse.csr\_matrix(train\_pivot.values)  
test\_pivot\_sparse = scipy.sparse.csr\_matrix(test\_pivot.values)

* + 1. Описание трансформации датасета properties.csv

Трансформации проведенные с датасетом properties:

1. Были выделены только самые распространенные proprties топ 10

top\_properties = properties.drop\_duplicates(['itemid', 'property']).groupby("property")['itemid'].count().sort\_values(ascending=False)[:10]

1. Из датасета создан отфильтрованный датасет properties по топ 10 свойствам товаров

properties\_filtered = properties[properties['property'].isin(set(top\_properties.index))]

1. Создан вспомогательный датасет сгруппированный по property с подсчетом каждого itemid в каждом property:

properties\_filtered['itemid\_cnt'] = 1  
items\_cnt = properties\_filtered[['property', 'itemid', 'itemid\_cnt']].groupby([  
 'property', 'itemid'])['itemid\_cnt'].sum().reset\_index()

1. Создадим словарь с топ 10 itemid для топ10 свойств товаров. Для групп, у которых топ 10 первых itemid похожи на остальные itemid сделан случайный выбор itemid из всего пула:

top10\_4\_top\_prop\_dict = {}  
for i in set(np.unique(items\_cnt.property)):  
 if items\_cnt.loc[items\_cnt['property'] == str(i), 'itemid\_cnt'].max() == 1:  
 top10\_4\_top\_prop\_dict[i] = np.random.choice(items\_cnt.loc[items\_cnt['property'] == str(i), 'itemid'],  
 size=10)  
 else:  
 top10\_4\_top\_prop\_dict[i] = items\_cnt.loc[items\_cnt['property'] == str(i)].sort\_values(  
 'itemid\_cnt', ascending=False)[:10]['itemid'].tolist()

* 1. Описание модели, на которой построена рекомендательная система
     1. LightFM

Для модели рекомендательной системы была выбрана матричная факторизация и использована библиотека LightFM c WARP (Weighted Approximate-Rank Pairwise) loss. LightFM – это реализация на Python ряда популярных алгоритмов рекомендаций как для неявной, так и для явной обратной связи. LightFM также позволяет включать, как элементы, так и пользовательские метаданные в традиционные алгоритмы матричной факторизации. Он представляет каждого пользователя и элемент, как сумму скрытых представлений их функций, что позволяет обобщать рекомендации для новых элементов (через свойства элементов) и для новых пользователей (через пользовательские свойства).

WARP Loss – эта функция потерь лучше других показывает себя в задачах ранжирования. Она работает с тройками (user, positive\_item, negative\_item) и имеет одну очень важную особенность – выбор негативных примеров происходит не случайно, а таким образом, чтобы выбранные негативные примеры «ломали» текущее ранжирование модели, т.е. были выше, чем позитивный пример.

Таким образом, процедура обучения модели с использованием WARP loss будет примерно следующей:

Для пары (user, positive\_item) выберем случайный негативный пример среди всех остальных айтемов (важно понимать, что так стоит делать только в том случае, когда вероятность выбрать негативный пример, который на самом деле будет позитивным, достаточно мала). Посчитаем предсказание модели на парах (user, positive\_item) и (user, negative\_item) и, если не произошло нарушения порядка (то есть, модель предсказала больший score для позитивного примера), то продолжаем сэмплировать негативные примеры до тех пор, пока нарушение не произойдет.

Если мы получили нарушение с первой попытки, то можно сделать большой градиентный шаг, так как это означает, что на данный момент модель часто ставит негативные примеры выше позитивных и надо сильно обновлять ее веса. Если же нам пришлось долго искать подходящий негативный пример, то делаем маленький шаг — модель уже достаточно хорошо обучена.

Пример кода:

model\_lfm = LightFM(no\_components=20, loss='warp', random\_state=42, learning\_rate=0.03, max\_sampled=10)  
model\_lfm.fit(train\_pivot\_sparse, epochs=10)

где:

* no\_components – размерность скрытых вложений признаков;
* learning\_rate - начальная скорость обучения для расписания обучения адаптивного градиентного спуска;
* max\_sampled - максимальное количество отрицательных образцов, используемых во время обучения с WARP loss.
  + 1. Метрики полученные на валидационном датасете

Метрики полученные на валидационном датасете модели LightFM:

* Precision@3 = 15,78 %

Из 3 рекомендуемых товаров пользователи в среднем покупают 15,78 % из них.

* Recall@3 = 6,19 %

Из 3 рекомендуемых товаров 6,19 %, что купил пользователь, приходится на рекомендованное.

* + 1. Общая логика выдачи рекомендаций

Общая логика выдачи рекомендаций:

1. Если у пользователя не было транзакции, но был просмотр или добавление в корзину в тренировочной выборке, то ему будут рекомендоваться топ-3 itemid из топ-10 товаров из топ-10 групп свойств. Относительно последнего действия пользователя в тренировочной выборке определяются свойства из топ-10, в которые мог бы входить его последний itemid. Если у пользователя последний itemid не входит ни в одно свойства из топ-10, то ему выдается рекомендация из топ-3 самых просматриваемых itemid в тренировочном датасете;
2. Для новых пользователей или пользователей, которые есть в тестовой выборке, но нет в тренировочной, выдается рекомендация из топ-3 самых просматриваемых itemid в тренировочном датасете;
3. Для пользователей, у которых была транзакция, выдается рекомендация модели, обученной топ-20 купленных товарах на LightFM user-item.
   1. Поднятие docker-образа системы

Docker – это платформа для разработки, доставки и запуска контейнерных приложений. Docker позволяет создавать контейнеры, автоматизировать их запуск и развертывание, управляет жизненным циклом. Он позволяет запускать множество контейнеров на одной хост-машине.

Контейнеры – это способ стандартизации развертки приложения и отделения его от общей инфраструктуры. Экземпляр приложения запускается в изолированной среде, не влияющей на основную операционную систему.

Контейнеры позволяют упаковать в единый образ приложение и все его зависимости: библиотеки, системные утилиты и файлы настройки. Это упрощает перенос приложения на другую инфраструктуру.

Поднятие Docker образа системы app\_diplom.tar выполняется следующим образом:

1. Скачиваем Docker образ системы **app\_diplom.tar**;
2. Открываем терминал CMD;
3. Переходим в репозиторий, где лежит Docker образ системы **app\_diplom.tar**;
4. Выполняем команду по загрузке образа системы:

* **docker load --input app\_diplom.tar**

1. Выполняем команду по запуску контейнера с системой с указанием хоста и порта для поднятия, и создания томов для хранения данных контейнера:

* **docker run -it --rm -p 0.0.0.0:8015:8015 -v logs:/usr/local/app\_ml/logs -v models:/usr/local/app\_ml/models -v dataset:/usr/local/app\_ml/dataset --name app\_diplom app\_diplom**

Просмотр лога событий, происходящих после запуска одного из методов в API системы:

1. В терминале CMD вводим следующую команду:

* **docker exec -it app\_diplom bash**

1. Переходим в папку с логами с помощью команды **cd logs**;
2. Для просмотра наименования файлов с логами вводим команду **ls**;
3. Для открытия файла с логами вводим команду (название файла указано для примера) **cat API\_log\_20230111.log**
   1. Описание API сервиса

Сервис построен на основе веб-фреймворка FastAPI (далее описание взято с официальной страницы FastAPI: <https://fastapi.tiangolo.com/ru>).

FastAPI — это современный, быстрый (высокопроизводительный) веб-фреймворк для создания API используя Python 3.6+, в основе которого лежит стандартная аннотация типов Python.

Ключевые особенности:

* **Скорость**: Очень высокая производительность, на уровне **NodeJS** и **Go** (благодаря Starlette и Pydantic). Один из самых быстрых фреймворков Python.
* **Быстрота разработки**: Увеличение скорости разработки примерно на 200–300%.
* **Меньше ошибок**: Сокращение примерно на 40% количество ошибок, вызванных человеком (разработчиком).
* **Интуитивно понятный**: Отличная поддержка редактора. Автозавершение везде. Меньше времени на отладку.
* **Лёгкость**: Разработан так, чтобы его было легко использовать и осваивать. Меньше времени на чтение документации.
* **Краткость**: Сведение к минимуму дублирование кода. Каждый объявленный параметр - определяет несколько функций. Меньше ошибок.
* **Надежность**: Получение готового к работе кода. С автоматической интерактивной документацией.
* **На основе стандартов**: Основан на открытых стандартах API и полностью совместим с ними: [OpenAPI](https://github.com/OAI/OpenAPI-Specification" \t "_blank) (ранее известном как Swagger) и [JSON Schema](https://json-schema.org/).
  + 1. Запуск сервиса

Запуск сервиса в браузере:

1. Открываем браузер;
2. Для открытия документации типа Swagger переходим по ссылке <http://localhost:8015/docs>
   * 1. Описание методов сервиса

Методы, реализованные в сервисе:

1. **/сheck html response** – проверка ответа от сервера;
2. **/model train** – обучение модели для пользователей, у которых были транзакции;
3. **/get\_userid\_dict** – получение словаря со списком всех userid пользователей в системе;
4. **/recommendation** – получение рекомендации по userid пользователя
   * 1. Описание /сheck\_html\_response

Запуск метода GET:

1. Раскрываем метод по нажатию на его поле;
2. Нажимаем кнопку **Try it out**;
3. Нажимаем кнопку **Execute**;
4. При удачном ответе получаем строку: **<b>Product recommendation system API</b>**
5. При неудачном ответе получаем ошибку 500.
   * 1. Описание /model\_train

Запуск метода POST:

1. Раскрываем метод по нажатию на его поле;
2. Нажимаем кнопку **Try it out**;
3. Для запуска метода необходимо в поле **Request body** вставить исходный датасет в формате .json. За основу был взят файл events.csv и переведен в формат .json с помощью следующего кода:

*# Импортируем библиотеку json*

***import json***

*# Создаем словарь из датасета*

**data\_for\_json\_dict = events\_data.to\_dict('records')**

*# Преобразуем словарь в нужную форму*

**data\_for\_f = {"Data": data\_for\_json\_dict}**

*# Сохраняем файл*

**with open("data\_for\_train\_lightfm.json", "w") as write\_file:**

**json.dump(data\_for\_f, write\_file)**

1. Пример тела файла .json (приведено 2 строки с данными):

{'Data': [{'timestamp': 1433221332117,

'visitorid': 257597,

'event': 'view',

'itemid': 355908,

'transactionid': nan},

{'timestamp': 1433224214164,

'visitorid': 992329,

'event': 'view',

'itemid': 248676,

'transactionid': nan},

…

]}

1. Нажимаем кнопку **Execute**;
2. При удачном ответе получаем json:

{'Training\_time': '0:01:16.100641',

'Status': 'Success',

'Message': 'Обучение модели рекомендация товаров на LightFM прошло успешно.',

'precision\_at\_3': '0.11667',

'recall\_at\_3': '0.05784'}

где:

* 'Training\_time' – время обработки данных и обучения модели;
* 'Status' – статус выполнения метода;
* 'Message' – информационное сообщение о завершении обучения модели;
* 'precision\_at\_3', 'recall\_at\_3' – метрики получившиеся на валидационном сете (метрики описаны в 2.2 Цели и метрики системы)

1. При неудачном ответе получаем:

* ошибку 500 при недоступности сервиса;
* ошибку 422 при некорректности тела исходного запроса;
* ошибки по обработке исключений в коде системы.

\*Данный метод запускать необязательно, в контейнер уже подгружены файлы модели и необходимые датасеты.

\* При вставке большого количества данных в поле **Request body**, интерфейс через браузер может зависнуть. Нужно написать запрос к сервису:

import json  
import requests  
with open("data\_for\_train\_lightfm.json", "r") as open\_file:  
 data = json.load(open\_file)  
url\_train = "http://127.0.0.1:8015/model\_train"  
headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}  
r = requests.post(url\_train, data=json.dumps(data), headers=headers)  
r.json()

* + 1. Описание /get\_userid\_dict

Запуск метода GET:

1. Раскрываем метод по нажатию на его поле;
2. Нажимаем кнопку **Try it out**;
3. Нажимаем кнопку **Execute**;
4. При удачном ответе получаем словарь со всеми уникальными userid из исходного датасета;
5. При неудачном ответе получаем:

* ошибку 500 при недоступности сервиса;
* ошибки по обработке исключений в коде системы.

\* Словарь содержит много записей, интерфейс через браузер может зависнуть. Нужно написать запрос к сервису:

import requests  
url\_getd = "http://localhost:8015/get\_userid\_dict"  
r = requests.get(url\_getd)  
r.json()

* + 1. Описание / recommendation

Запуск метода GET:

1. Раскрываем метод по нажатию на его поле;
2. Нажимаем кнопку **Try it out**;
3. В поле **user\_id** вводим необходимое нам значение в формате **int**;
4. Нажимаем кнопку **Execute**;
5. При удачном ответе получаем json:

{

"Forecast\_time": "0:00:19.755312",

"Status": "Прогноз выполнен успешно",

"Recommendations": {

"recommendation\_1": 154485,

"recommendation\_2": 449442,

"recommendation\_3": 20431

}

}

где:

* ' Forecast\_time ' – время обработки данных и выдачи рекомендации;
* 'Status' – статус выполнения метода;
* 'Recommendations' – 3 рекомендации для указанного userid.

1. При неудачном ответе получаем:

* ошибку 500 при недоступности сервиса;
* ошибки по обработке исключений в коде системы.
  + 1. Endpoint для снятия метрик

В системе реализованы два endpoint для снятия метрик precision\_at\_3 и recall\_at\_3 для метода /model\_train. Сбор метрик реализован с помощью Pushgateway, который передает метрики далее в Prometheus. Актуальные метрики передаются в Prometheus, при каждом запуске метода /model\_train. Для снятия метрик на компьютере необходимо иметь развернутый Pushgateway на хосте - localhost:9091

Реализация endpoint:

# Импортируем библиотеки

from prometheus\_client import Gauge, CollectorRegistry, push\_to\_gateway

# Задаем переменные для снятия мгновенных метрик

registry = CollectorRegistry()

pr = Gauge('LightFM\_metric\_precision\_at\_3', 'Metric precision\_at\_3 after train LightFM model', registry=registry)  
rec = Gauge('LightFM\_metric\_recall\_at\_3', 'Metric recall\_at\_3 after train LightFM model', registry=registry)

# Передадим метрики в prometheus через push\_to\_gateway  
try:  
 pr.set(map\_at3) # Set to a given value  
 push\_to\_gateway('localhost:9091', job='batchA', registry=registry)  
 logger.info('Метрика precision\_at\_k передана push\_to\_gateway.')  
except:  
 logger.info('Метрика precision\_at\_k не передана push\_to\_gateway.')  
 pass  
try:  
 rec.set(rec\_at3) # Set to a given value  
 push\_to\_gateway('localhost:9091', job='batchB', registry=registry)  
 logger.info('Метрика recall\_at\_k передана push\_to\_gateway.')  
except:  
 logger.info('Метрика recall\_at\_k не передана push\_to\_gateway.')  
 pass