ЭТАПЫ АНАЛИТИКИ ДАННЫХ

1. Введение

Как правило, в процесс аналитики данных вовлечены специалисты из различных предметных областей, которых условно можно разделить на две группы:

- Специалисты, не имеющие прямого отношения к IT и программированию (маркетологи, менеджеры по управлению финансами, руководители различного уровня, экономические аналитики). Эти специалисты хорошо разбираются в специфике предметной области (например, в продажах, организации и ведении бизнеса, управлении финансами и т.п.). Однако они не в состоянии самостоятельно использовать аналитический IT-инструментарий.
- Технические специалисты (программисты, математики, инженеры в области Data Science). Эта группа специалистов, напротив, может иметь самое отдаленное представление о бизнес-процессах предметной области. Однако, их профессиональные навыки позволяют легко справляться с настройкой и использованием технических средств анализа данных без привязки к какой-либо отдельной тематике.

Очевидно, что эти группы специалистов должны тесно взаимодействовать друг с другом, чтобы решать весь цикл аналитических задач – начиная с их постановки и заканчивая получением и интерпретацией результатов.

Для лучшего понимания всех этапов аналитического процесса, рассмотрим его на конкретном практическом примере. Пусть имеется некая торговая сеть, занимающаяся реализацией продуктов от различных брендов. В маркетинговый отдел этой сети обращаются представители крупного бренда, собирающиеся запускать программу лояльности. Цель этой программы – выявить наиболее лояльных покупателей бренда и выдать им специальные клубные карты с солидными скидками на продукцию бренда. Размеры скидок на столько большие, что карты планируется выдать только самым преданным приверженцам. Торговая сеть ведет учет всех совершенных покупок по всем имеющимся брендам и имеет соответствующий массив больших данных.

Таким образом, суть просьбы представителей бренда к аналитикам торговой сети – проанализировать имеющиеся данные по всем покупкам и найти наиболее лояльных и преданных покупателей.

2. Этапы аналитической деятельности

Рассмотрим основные этапы решения данной задачи:

1. Постановка задачи на языке бизнес-процессов. Первая стадия переговоров между представителями бренда и маркетологами торговой сети ведется на языке бизнеса. Формулировка задачи также имеет явно

выраженный маркетинговый вид: ***нахождение некоторого числа лояльных покупателей***. При этом, заранее неизвестно никаких технических деталей: сколько должно быть таких покупателей (и есть ли они вообще)? что считать лояльностью и по каким критериям ее оценивать? каких покупателей считать более лояльными, а каких менее? и т.п.

2. Переформулирование задачи бизнеса на язык измеряемых метрик. Следующий этап – передача задачи от маркетологов к аналитикам и техническим специалистам Data Science. Здесь необходимо придумать один или несколько критериев, которые можно численно рассчитать, исходя из имеющихся в распоряжении сети массивов больших данных. Например, лояльность можно оценить через количество купленных товаров одного бренда. Или – через отношение купленных товаров заданного бренда ко всему количеству покупок товаров этой группы. ***Метрикой*** *будет называть численный показатель, позволяющий явно оценить интересующие нас особенности бизнес-процесса*. Обычно, этот этап выполняется совместно маркетологами и техническими специалистами в области аналитики данных.

3. Предобработка данных. Начиная с этого этапа, работу выполняют уже только технические специалисты. Как правило, имеющиеся массивы данных гораздо больше и разнообразнее той небольшой выборки, которая потребуется нам для расчета метрик. Поэтому необходимо произвести предварительную фильтрацию данных, их адаптацию под решаемую задачу (переименование колонок, очистка данных от мусора, преобразование данных в другие форматы представления, добавление новых колонок и т.п.).

4. Разведывательный анализ. Дальнейшая поэтапная редукция данных (как с точки зрения упрощения структуры, так и с точки зрения уменьшения их объема). Необходимо оставить только те данные, которые нужны для вычисления метрических величин и сопутствующих им в аналитике показателей. Кроме того, необходимо выстроить общую статистическую картину с точки зрения базовых продуктовых метрик (сколько всего покупателей, сколько покупок, каким образом покупки распределяются по покупателям, как соотносятся покупки с брендами и др.). Например, аналитики могут сразу исключить из рассмотрения покупателей, имеющих единичное количество покупок, поскольку невозможно определить их лояльность к какому-либо бренду по такой незначительной статистике.

5. Создание и расчет метрик. Когда все необходимые данные подготовлены, остается лишь правильно использовать группировку, агрегацию и вычислительные функции Python и Pandas. Полученные результаты лучше всего сопроводить графическим представлением в виде гистограмм, графиков, распределений. Во-первых, так информация легче воспринимается. А во-вторых, для убедительности проделанной работы - бизнес-специалисты заказчика возможно захотят убедиться в адекватности и достоверности выбранных метрик.

Рассмотрим всю последовательность действий на примере конкретных данных.

3. Предварительная подготовка данных

Предположим, что автоматическая система сбора данных торговой сети произвела для нас выгрузку данных по покупателям и покупкам в виде файла ecomm.csv. Естественно, что эта система не адаптирована под данную конкретную задачу. Ее цель – просто фиксировать все покупки в некотором обобщенном виде. Поэтому нужно быть готовым к тому, что такие «сырые» данные, сразу после чтения, окажутся непригодными для работы и потребуются дополнительные действия для их предобработки. Произведем подключение библиотеки Pandas, считывание данных из файла в датафрейм и оценку объема и структуры данных (рис. 1, 2). Из рис. 1 можно оценить объем данных – около 50000 информационных строк о покупках и 21 колонка с показателями по каждой покупке.

```
In [1]: import pandas as pd

In [2]: df = pd.read_csv('ecomm.csv', encoding = 'Windows-1251')

In [6]: df.shape
Out[6]: (48129, 21)
```

Рисунок 1 – Чтение данных и оценка объема данных

```
In [1]: import pandas as pd

In [2]: df = pd.read_csv('ecomm.csv', encoding = 'Windows-1251')

In [7]: df.head()
Out[7]:
```

	Unnamed: 0	tk	pl	dia	hs	tc	cta	id_art	id_subsubfam	id_subfam	...	id
0	1242	120071109002055793	1	2007/11/09	0.505729	110000761	11000076	21895	101070640100	1010706401	...	101
1	1243	120070601004053633	1	2007/06/01	0.588519	110000761	11000076	21816	101070640100	1010706401	...	101
2	1244	120070928004076410	1	2007/09/28	0.785498	28491841	2849184	562840	101070640100	1010706401	...	101
3	1245	120070616003074261	1	2007/06/16	0.550961	95931501	9593150	28914	101070640100	1010706401	...	101

Рисунок 2 – Оценка структуры данных

На рис. 2 видна структура полученного датафрейма. И сразу же бросается в глаза первый недостаток полученных данных - информативность колонок оставляет желать лучшего. При дальнейшей работе с датафреймом, пользоваться такими малоинформативными названиями будет очень неудобно.

Поскольку на данном этапе нас интересуют только покупатели (идентификаторы покупателей хранятся в колонке `tc`) и наименования их

покупок (колонка art_sp), то мы можем сразу перенести колонки с этими данными в отдельный датафрейм и переименовать их (избавившись тем самым от всего остального массива ненужных данных) – рис. 3.

```
In [6]: user_data = df[['tc', 'art_sp']].rename(columns={'tc':'user_id', 'art_sp':'brand_info'})

In [7]: user_data

Out[7]:
```

	user_id	brand_info
0	110000761	MARAVILLA 500 G Store_Brand
1	110000761	FIDEO CABELLIN 500 G Store_Brand
2	28491841	SPAGUETTI Nє 5 500 G Brand_1
3	95931501	FIDEO FIDEUБ 500 Brand_7
4	93265591	MACARRONES GRATINAR 5 Brand_2
...	...	...
48124	45518841	FIDEOS 0 500 G Brand_4
48125	110824211	PLUMAS 3 500 G Brand_4
48126	1408670389	MACARRONES 500 G Store_Brand
48127	1408670389	SPAGHETTI 500 G Store_Brand

Рисунок 3 – Выборка отдельных колонок исходного датафрейма и их переименование

Полученный датафрейм уже более наглядный, в нем гораздо меньше лишнего для решаемой задачи. Колонки имеют информативные названия (`user_id` – идентификатор пользователя, `brand_info` – информация о купленном товаре).

Разбиение строк `split()`

Однако, некоторая избыточность данных все-таки сохраняется. Речь идет о колонке `brand_info`, где содержится строка с названием продукта, его массой и наименованием бренда. Для дальнейшей работы нам нужно только название бренда и необходимо каким-то образом выделить его из каждого значения `brand_info`. Например, в строке «**MARAVILLA 500 G Store_Brand**» нам необходима только последняя ее часть – «**Store_Brand**».

На помощь здесь приходит богатый функциональный арсенал языка Python, который очень удобен для обработки текстовых данных. В частности, нам пригодится метод `.split()`, разбивающий строку на фрагменты и помещающий их в список. По умолчанию, критерием деления являются все пустые символы (пробелы, табуляции ‘\t’, переносы строки ‘\n’). Впрочем, разделительный символ можно задать и явно, указав его в качестве параметра. Пример работы метода `split()` показан на рис. 4.

```
brand_info = 'MARAVILLA 500 G Store_Brand'
brand_info.split()
```

```
['MARAVILLA', '500', 'G', 'Store_Brand']
```

Рисунок 4 - Пример работы метода `split()`

Следовательно, если мы применим метод `split()` для строки с некоторым наименованием покупки и вытащим из результата (списка) последний элемент, то получим только лишь название бренда (рис. 5).

```
In [8]: brand_info = "MARAVILLA 500 G Store_Brand"
        brand_info.split(' ')[-1]
Out[8]: 'Store_Brand'
```

Рисунок 5 – Применение метода `split()` для выделения названия бренда

Потоковая обработка значений датафрейма методом `apply()`

Теперь нужно применить эту операцию для всех значений колонки `brand_info`, для того, чтобы вытащить название бренда каждой из покупок. Для этих целей удобно использовать функцию библиотеки Pandas `apply()`, которая применяет переданную в него функцию ко всем колонкам вызванного датафрейма. Чтобы применить функцию только к одной колонке датафрейма, можно указать её имя перед применением `apply`.

Остается лишь оформить применение функции `split()` с рис. 5 в виде отдельной функции и передать ее аргументом в функцию `apply` для колонки `brand_info` – рис. 6.

```
In [9]: def split_brand_info (full_brand_info):
            return full_brand_info.split(' ')[-1]
```

```
In [11]: user_data.brand_info.apply(split_brand_info)
Out[11]: 0        Store_Brand
         1        Store_Brand
         2            Brand_1
         3            Brand_7
         4            Brand_2
                      ...
         48124        Brand_4
         48125        Brand_4
         48126    Store_Brand
         48127    Store_Brand
         48128    Store_Brand
         Name: brand_info, Length: 48129, dtype: object
```

Рисунок 6 – Создание функции выделения названия бренда и применение ее ко всем значениям колонки `brand_info`

Чтобы не терять связи с идентификаторами покупателей, создадим в датафрейме новый столбец с полученным названием бренда (рис. 7).

```
In [9]: def split_brand_info (full_brand_info):
            return full_brand_info.split(' ')[-1]

In [12]: user_data['brand_name'] = user_data.brand_info.apply(split_brand_info)

In [13]: user_data

Out[13]:
```

	user_id	brand_info	brand_name
0	110000761	MARAVILLA 500 G Store_Brand	Store_Brand
1	110000761	FIDEO CABELLIN 500 G Store_Brand	Store_Brand
2	28491841	SPAGUETTI № 5 500 G Brand_1	Brand_1
3	95931501	FIDEO FIDEUБ 500 Brand_7	Brand_7
4	93265591	MACARRONES GRATINAR 5 Brand_2	Brand_2
...	...	...	...
48124	45518841	FIDEOS 0 500 G Brand_4	Brand_4
48125	110824211	PLUMAS 3 500 G Brand_4	Brand_4

Рисунок 7 – Создание новой колонки `brand_name` и заполнение ее данными о брендах из колонки `brand_info`.

Лямбда-функции (анонимные функции)

На практике, не всегда имеет смысл создавать отдельные функции для одноразовых действий, которые в дальнейшем нам нигде и никогда не потребуются. Это приводит к избыточному коду и захламлению программы лишними именами и функциями.

Язык Python позволяет создавать безымянные (анонимные функции), которые можно целиком передавать в качестве аргументов другим функциям или использовать в других одноразовых ситуациях.

Функция `split_brand_info`, созданная на рис. 6, является одноразовой, поскольку больше никаких подобных действий нам не потребуется. Поэтому имеет смысл заменить ее анонимной функцией и, тем самым, оптимизировать код нашей программы.

Синтаксис создания анонимных функций выглядит следующим образом:

```
lambda <аргументы>: <действия>
```

Другое название таких функций – лямбда-функции. По сути, они несут в себе только описание действия, которое нужно совершить с поступившими аргументами.

Функция `split_brand_info` в виде лямбда функции будет выглядеть так:

```
lambda x: x.split(' ')[-1]
```

В данном случае, `x` является аргументом лямбда-функции. А действие – разбиение строки на фрагменты и выделение последнего из них. Применим лямбда-функцию для достижения результата, аналогичного рис. 7, но без созданиях лишних функций вида `split_brand_info`.

```
In [14]: user_data['brand_name'] = user_data.brand_info.apply(lambda x: x.split(' ')[-1])

In [15]: user_data
```

Out[15]:

	user_id	brand_info	brand_name
0	110000761	MARAVILLA 500 G Store_Brand	Store_Brand
1	110000761	FIDEO CABELLIN 500 G Store_Brand	Store_Brand
2	28491841	SPAGUETTI № 5 500 G Brand_1	Brand_1
3	95931501	FIDEO FIDEUБ 500 Brand_7	Brand_7
4	93265591	MACARRONES GRATINAR 5 Brand_2	Brand_2
...	...	...	...
48124	45518841	FIDEOS 0 500 G Brand_4	Brand_4
48125	110824211	PLUMAS 3 500 G Brand_4	Brand_4
48126	1408670389	MACARRONES 500 G Store_Brand	Store_Brand
48127	1408670389	SPAGHETTI 500 G Store_Brand	Store_Brand

Рисунок 8 – Использование лямбда-функции

Сочетание методов `apply` и `lambda` является весьма распространенной практикой при потоковой (векторной) обработке датафреймов. По сути, этот способ позволил нам вместить в одну строку кода три важных действия:

- описать действие по разбиению строки и выделению имени бренда
- применить это действие ко всем строкам колонки `brand_info`
- поместить полученный результат в новую колонку, предварительно создав ее.

4. Разведывательный анализ

Произведя предварительную выборку и очистку данных, настало время сделать первые шаги в направлении, собственно анализа. В рассуждениях вводной части мы выдвинули вполне очевидный тезис, что искать лояльных клиентов нужно среди покупателей с большим числом покупок. Потому что, малое число покупок не позволит нам оценить статистическую картину предпочтений покупателя.

Поэтому первым аналитическим шагом будет выявление того, сколько покупок сделал каждый покупатель. Это сразу позволит нам отделить одноразовых (или не частых) покупателей от постоянных клиентов с большим числом покупок – рис. 9.

```
In [23]: users_purchase = (user_data.groupby('user_id', as_index = False)
                                    .agg({'brand_name':'count'})
                                    .rename(columns={'brand_name':'purchases'}))

In [24]: users_purchase.shape

Out[24]: (11764, 2)

In [25]: users_purchase.head()
```

Out[25]:

	user_id	purchases
0	-1236394515	1
1	1031	6
2	4241	5
3	17311	2
4	17312	2

Рисунок 9 – Сколько покупок сделал каждый покупатель

Обратите внимание на последний метод цепочки – rename. Он использован для того, чтобы получить информативное название новой колонки с количеством покупок. Попутно также узнаем, что в нашем датафрейме содержится информация о 11764 покупателях.

Следующим шагом попытаемся определить, какое количество покупок считать большим и достаточным, а какое малым (и, соответственно, недостаточным) для определения лояльности. Используем для этого нахождение медианы в нашем распределении числа покупок (рис. 10)

```
In [46]: users_purchases.purchases.median()

Out[46]: 2.0
```

Рисунок 10 – Медианное значение в распределении числа покупок

Медианное значение (50%-процентиль) показывает границу, разделяющую покупателей на две группы. Половина покупателей (50%) произвело менее 2-х покупок, половина (50%) – более 2-х. На данном этапе, мы могли бы смело отбросить первую половину пользователей и работать только со второй. Однако, 2 покупки – величина также очень небольшая для того, чтобы делать надежные статистические выводы. Достаточно большое количество покупателей, совершивших две покупки одного бренда, могли сделать это совершенно случайно, подчиняясь каким-либо сиюминутным потребностям. Другой недостаток полученного медианного разбиения в том,

что во вторую половину попадает почти 6000 покупателей, что является слишком большой цифрой для группы лояльности.

Попробуем использовать неравномерное разбиение процентилем другой величины. Чтобы подобрать нужный процентиль, воспользуемся известным нам методом `.describe()`, описанным в работе №1, который выдает полную статистическую выкладку по числовым колонкам – рис. 11.

```
In [48]: users_purchases.describe()
```

Out[48]:

	user_id	purchases
count	1.176400e+04	11764.000000
mean	7.690517e+07	4.091210
std	1.622210e+08	4.573143
min	-1.236395e+09	1.000000
25%	1.503761e+07	1.000000
50%	4.682179e+07	2.000000
75%	9.311601e+07	5.000000
max	1.408849e+09	60.000000

Рисунок 11 – Оценка распределений числа покупок с использованием метода `.describe()`

Весьма перспективным выглядит 75%-процентиль со значением 5. То есть, только 25% покупателей совершили 5 и более покупок. Будет вполне резонным отобрать именно эту малую группу покупателей для дальнейшей работы. Для этого несколько изменим цепочку методов с рис. 9, добавив последним элементом фильтр покупателей по числу покупок (5 и более) – рис. 12.

Попутно, можно отметить, что круг потенциально интересных для нашей задачи покупателей сузился до 3383 человек. Оценим, как распределяется число покупок в этой узкой группе, повторно воспользовавшись методом `.describe()` – рис. 13.

```
In [26]: users_purchase = (user_data.groupby('user_id', as_index = False)
                                    .agg({'brand_name':'count'})
                                    .rename(columns={'brand_name':'purchases'})
                                    .query('purchases >= 5') )

In [27]: users_purchase.shape
Out[27]: (3383, 2)

In [28]: users_purchase.head()
Out[28]:
```

	user_id	purchases
1	1031	6
2	4241	5
11	25971	7
14	40911	27
16	45181	5

Рисунок 12 - Сколько покупок сделал каждый покупатель (учитываются только покупатели, совершившие 5 и более покупок)

```
In [51]: users_purchases.describe()
Out[51]:
```

	user_id	purchases
count	3.383000e+03	3383.000000
mean	6.421500e+07	9.320130
std	1.504830e+08	5.623993
min	1.031000e+03	5.000000
25%	8.871271e+06	6.000000
50%	2.842547e+07	7.000000
75%	8.542964e+07	11.000000
max	1.408810e+09	60.000000

Рисунок 13 – Оценка распределений числа покупок 5 и более

Таким образом, разведывательный анализ данных показывает нам примерную общую картину имеющихся данных и во многом является весьма умозрительным. Почему для фильтрации покупателей был выбран 75-процентиль? Точного ответа на этот вопрос нет. Так мы сделали первую прикидочную выборку и можем попытаться поискать лояльных покупателей в ней. Если полученный результат нас не устроит, то можно сместить границу фильтрации и попробовать поискать еще раз.

5. Создание и расчет метрик

Существенным недостатком полученного датафрейма `user_purchases` является то, что мы не видим, как распределяются различные бренды в общем числе покупок каждого покупателя. Например, на рис. 12 видно, что покупатель с номером 1031 совершил 6 покупок. Но что это за покупки? Это покупки товаров одного бренда или разных?

Имея на руках информацию о покупателях в разрезе отдельных брендов, мы могли бы создать простейшую численную метрику лояльности – ***какая доля от всех покупок покупателя приходится на товары любимого бренда*** (причем любимых брендов может быть и несколько).

Для того, чтобы произвести это важное уточнение, вернемся к нашим «сырым» данным user_data и выполним новую группировку (groupby), но уже по двум колонкам – по номеру пользователя и по названию бренда, далее подсчитаем количество позиций каждого бренда (агрегация с использованием count) и отсортируем по убыванию как самих покупателей, так и количество позиций каждого бренда для каждого из них – рис. 14

```
In [32]: (user_data.groupby(['user_id','brand_name'], as_index=False)
                   .agg({'brand_info':'count'})
                   .sort_values(['user_id', 'brand_info'], ascending=[False, False]))
```

Out[32]:

	user_id	brand_name	brand_info
18187	1408849249	Store_Brand	1
18186	1408840919	Store_Brand	1
18185	1408832719	Brand_4	3
18184	1408825059	Brand_1	1
18183	1408817589	Store_Brand	2
...	...	...	...
3	4241	Brand_4	3
4	4241	Store_Brand	2
2	1031	Store_Brand	5
1	1031	Brand_3	1
0	-1236394515	Brand_4	1

Рисунок 14 – Распределение числа покупок в разрезе брендов для каждого покупателя

Полученный код можно еще немного нарастить, выбрав для каждого покупателя только первую строчку с максимальным числом брендов – рис. 15.

```
In [33]: (user_data.groupby(['user_id','brand_name'], as_index=False)
                   .agg({'brand_info':'count'})
                   .sort_values(['user_id', 'brand_info'], ascending=[False, False])
                   .groupby('user_id')
                   .head(1))
```

Out[33]:

	user_id	brand_name	brand_info
18187	1408849249	Store_Brand	1
18186	1408840919	Store_Brand	1
18185	1408832719	Brand_4	3
18184	1408825059	Brand_1	1
18183	1408817589	Store_Brand	2
...	...	...	...
6	17312	Brand_1	1
5	17311	Brand_4	2
3	4241	Brand_4	3
2	1031	Store_Brand	5

Рисунок 15 - Наибольшее число покупок одного бренда для каждого покупателя

Теперь осталось сохранить полученные результаты в отдельном датафрейме и переименовать колонки в соответствие с заложенным в них смыслом – `brand_name` заменим на `lovely_brand_name` (“любимый бренд”), а `brand_info` – на `lovely_brand_purchases` (“число покупок любимого бренда”) – рис. 16.

```
In [35]: lovely_brand_purchases_df =(user_data.groupby(['user_id','brand_name'], as_index=False)
                   .agg({'brand_info':'count'})
                   .sort_values(['user_id', 'brand_info'], ascending=[False, False])
                   .groupby('user_id')
                   .head(1)
                   .rename (columns={'brand_name':'lovely_brand', 'brand_info':'lovely_brand_purchases'}))
```

```
In [36]: lovely_brand_purchases_df
```

Out[36]:

	user_id	lovely_brand	lovely_brand_purchases
18187	1408849249	Store_Brand	1
18186	1408840919	Store_Brand	1
18185	1408832719	Brand_4	3
18184	1408825059	Brand_1	1
18183	1408817589	Store_Brand	2

Рисунок 16 - Наибольшее число покупок одного бренда для каждого покупателя с информативными названиями колонок

Теперь мы имеем два разных датафрейма, которые акцентируются на разных аспектах проблемы:

- **датафрейм `user_purchases`** – содержит информацию об общем числе покупок каждого покупателя (причем там содержатся данные о покупателях с 5 и более покупками);
- **датафрейм `lovely_brand_purchases_df`** – содержит информацию о любимых брендах для каждого покупателя, независимо от числа покупок.

Было бы важным иметь также информацию, сколько брендов приходится на одного покупателя. Это позволит вплотную подойти к нашей конечной цели. В самом деле, если покупатель, например, имеет 6-7 покупок и мы знаем, что на его покупки приходится только один бренд, то такого покупателя можно считать лояльным.

Создадим третий датафрейм `users_unique_brand` в котором будет храниться число уникальных брендов для каждого покупателя – рис. 17.

```
In [39]: users_unique_brands = (user_data.groupby('user_id', as_index = False)
                                         .agg ({'brand_name': pd.Series.nunique})
                                         .rename(columns={'brand_name':'unique_brands'}))

In [40]: users_unique_brands
```

Out[40]:

	user_id	unique_brands
0	-1236394515	1
1	1031	2
2	4241	2
3	17311	1
4	17312	2
...	...	...
11759	1408817589	2
11760	1408825059	1
11761	1408832719	1

Рисунок 17 – Создание датафрейма `users_unique_brand` с числом уникальных брендов для каждого покупателя

Обратите внимание на метод, используемый при агрегации: `pd.Series.nunique()`. Это метод, который считает число уникальных значений в колонке и применяться он может только серии (Series) Pandas.

Серия Pandas это специальный тип данных внешне очень похожий на список Python. Однако, основным отличительным достоинством серии является наличие большого количества готовых методов для статистических расчетов над данными, хранящимися в этой серии. Список Python не имеет таких встроенных методов.

Преобразовать список в серию Pandas очень просто – рис. 18.

```
In [84]: x = pd.Series([1, 2, 3])

In [85]: x

Out[85]: 0    1
         1    2
         2    3
         dtype: int64
```

Рисунок 18 – Преобразование списка в серию Pandas

Серии и датафреймы (небольшое теоретическое отступление)

Важно понимать, что ***каждая колонка датафрейма представляет из себя серию***. Поэтому к ней можно применять все методы серий. Например, другим полезным методом является `unique()`, возвращающий уникальные значения в колонке в виде списочной структуры. Или уже знакомый нам метод `median()`, возвращающий медианное значение для данных серии, а также многие другие – `mean()`, `max()`, `min()` и др.

Если говорить еще более общим образом, то ***датафрейм является специальным двухмерным словарем***, похожим на обычный словарь вида {‘x’:[1, 2, 3], ‘y’:[‘a’, ‘b’, ‘c’]} – за тем исключением, что ключами являются имена колонок, а значениями – содержимое колонок. Благодаря этому, можно легко преобразовать обычный словарь в датафрейм – рис. 19.

```
In [100]: d = pd.DataFrame({'x': [1, 2, 3], 'y': ['a', 'b', 'c']})

In [102]: d

Out[102]:
```

	x	y
0	1	a
1	2	b
2	3	c

Рисунок 19 – Преобразование словаря в датафрейм

Объединение датафреймов методом **`.merge()`**

Настало время объединить все три имеющихся у нас датафрейма в единый датафрейм для возможности совместного анализа всех имеющихся данных. Обратим еще раз внимание на содержимое построенных датафреймов:

- **датафрейм `user_purchases`** – общее число покупок каждого покупателя (только для покупателей с 5 и более покупками);
- **датафрейм `lovely_brand_purchases_df`** – любимый бренд для каждого покупателя, независимо от числа покупок;
- **датафрейм `user_unique_brand`** – число брендов среди покупок каждого покупателя.

У всех трех датафреймов имеется одна общая колонка – идентификатор покупателя (`user_id`). Это позволяет объединить все три датафрейма в одну единую структуру для возможности совместного анализа добытых данных.

Для объединения (джойна) двух датафреймов используется метод `merge()`.

Подсказка: чтобы быстро просмотреть описание какой-либо функции в Jupyter Notebook можно использовать такую запись (рис. 20):

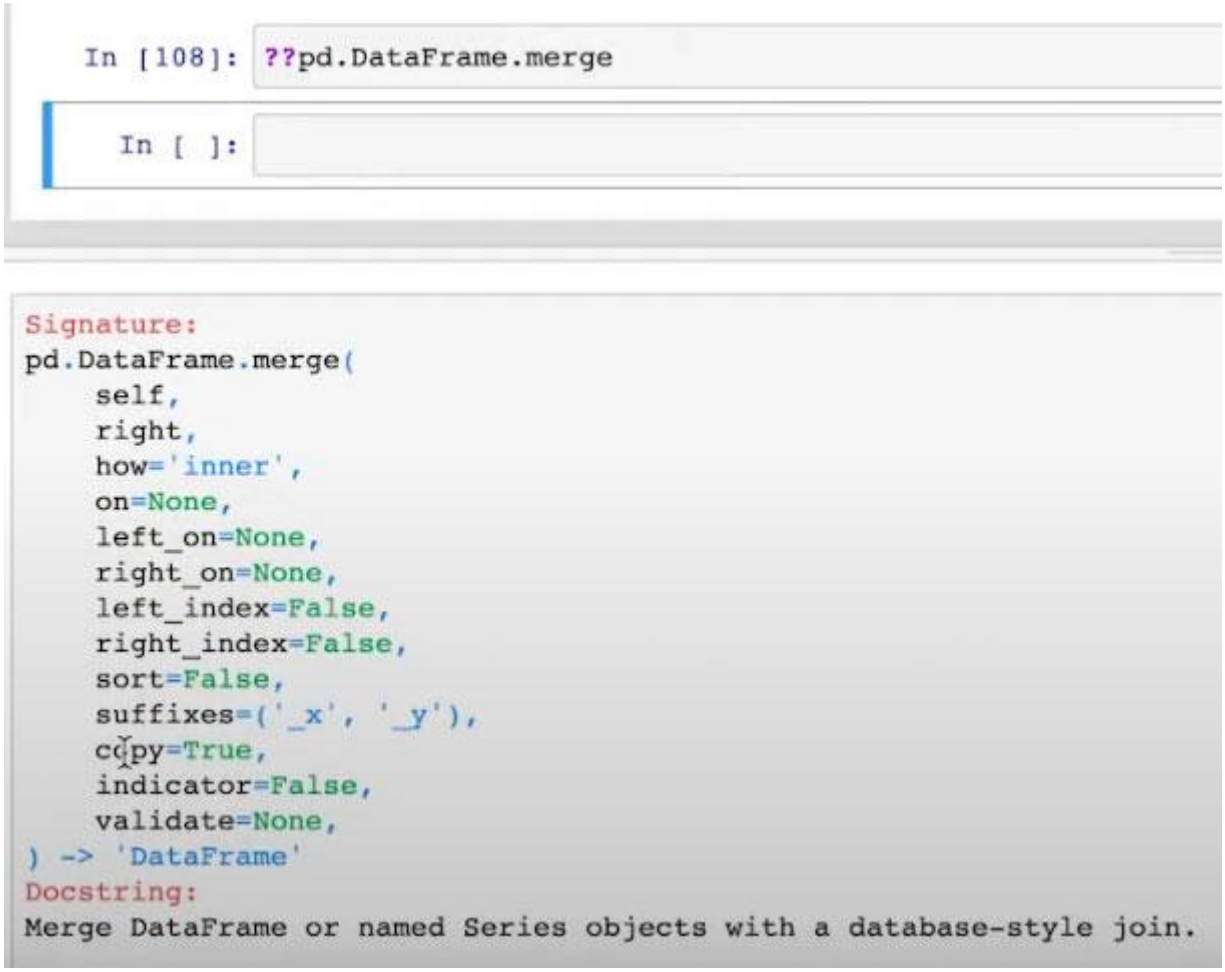


Рисунок 20 – Вызов справочного описания функции

Обязательным аргументом `merge()` является другой датафрейм, с которым планируется объединение. Объединение идёт по общей колонке (аргумент `on=`), у которой имеется одинаковый смысл и общие значения в обоих датафреймах. Если вставить несколько вызовов `merge()` в цепочку,

то так можно объединить сразу несколько датафреймов (что нам и требуется сделать) – рис. 21.

```
In [20]: loyalty_data = (users_purchase.merge(users_unique_brands, on='user_id')
                                       .merge(lovely_brand_purchases_df, on='user_id'))

In [21]: loyalty_data
```

Out[21]:

	user_id	purchases	unique_brands	lovely_brand	lovely_brand_purchases
0	1031	6	2	Store_Brand	5
1	4241	5	2	Brand_4	3
2	25971	7	2	Store_Brand	5
3	40911	27	5	Brand_4	19
4	45181	5	4	Store_Brand	2
...	...	...	...	...	...
3378	1408767189	5	1	Brand_4	5
3379	1408783189	10	2	Store_Brand	8
3380	1408783379	6	2	Brand_1	4

Рисунок 21 – Объединение трех датафреймов по колонке `user_id`

Существуют различные способы объединения таблиц A и B (аргумент `how=`): `inner`, `outer`, `left`, `right` (по умолчанию `inner`). В примере с покупателями (объединение по колонке `user_id`) это будет выглядеть следующим образом (рис. 22):

- `inner` – объединяются только те покупатели, которые есть и в A и в B (аналог – пересечение множеств);
- `left` – объединяются все покупатели таблицы A и совпадающие с ними покупатели из таблицы B;
- `right` - объединяются все покупатели таблицы B и совпадающие с ними покупатели из таблицы A;
- `outer` – объединяются покупатели, которые есть и в A и в B.

Более подробные и точные сведения о схемах объединения данных можно почитать в статьях:

https://habr.com/ru/post/448072/
https://habr.com/ru/post/450528/

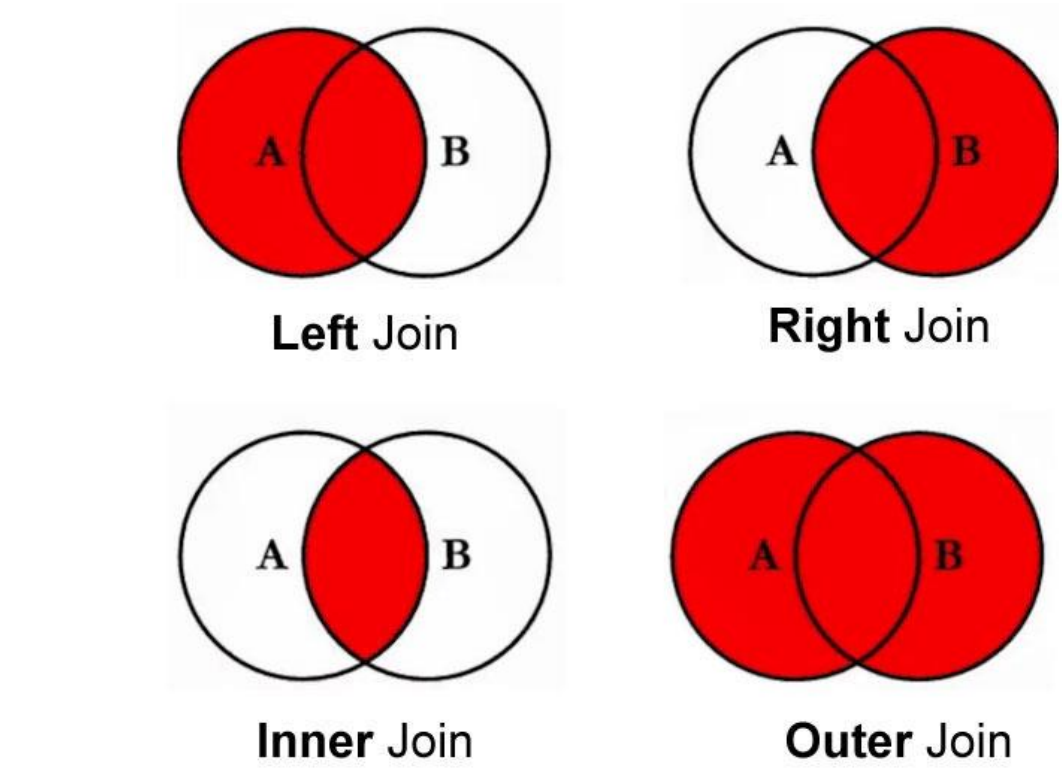


Рисунок 22 – Упрощенные схемы объединения датафреймов

В нашем случае, количество покупателей в датафреймах разное, поскольку в **`user_purchases`** мы отфильтровали данные только для покупателей с 5 и более покупками, а в двух других датафреймах содержатся все покупатели. Поэтому здесь важно не ошибиться со схемой объединения. Наилучшим образом подходит схема inner, для того, чтобы сохранить в объединенном датафрейме только покупателей с 5 и более покупками. На рис. 21 мы не указываем явно параметр «`how=inner`», потому что он принимается таковым по умолчанию.

В итоге – получили единый датафрейм **`loyalty_data`**, в котором собрана вся информация, необходимая для расчета нужной нам метрики: общее количество покупок каждого покупателя, сколько покупок приходится на любимый бренд, сколько брендов приходится на все покупки.

Теперь мы можем применить простейший фильтр – выбрать только тех покупателей, на покупки которых приходится только один бренд – рис. 23 (обратите внимание, на рисунке показан альтернативный способ фильтрации без использования метода query).

```
In [23]: loyal_users = loyalty_data [loyalty_data.unique_brands == 1]
```

```
In [24]: loyal_users
```

Out[24]:

	user_id	purchases	unique_brands	lovely_brand	lovely_brand_purchases
13	86281	14	1	Brand_4	14
18	94961	6	1	Brand_4	6
29	132061	9	1	Brand_4	9
30	134281	6	1	Brand_4	6
35	157311	12	1	Brand_4	12
...	...	...	...	...	...
3372	1010244089	9	1	Store_Brand	9
3374	1010247239	5	1	Brand_4	5
3376	1010274559	5	1	Brand_4	5

Рисунок 23 - Отбор покупателей, на покупки которых приходится только один бренд

По сути, теперь в датафрейме **`loyal_users`** содержатся покупатели с 5 и более покупками, на покупки которых приходится только один бренд. Это и есть лояльные пользователи. Можно сказать, что в первом приближении поставленная перед нами задачи решена.

Однако, более продвинутый результат решения должен иметь некоторую численную метрику, с помощью которой мы можем сравнивать лояльность покупателей между собой, ранжировать покупателей по уровню лояльности и т.п.

Создадим в нашем обобщенном датафрейме **`loyalty_data`** новую колонку `loyalty_score` , в которой для каждого покупателя рассчитаем долю покупок, приходящихся на любимый бренд: разделим значения колонки lovely_brand_purchases (число покупок любимого бренда) на значения purchases (общее число покупок) – рис. 24.

```
In [25]: loyalty_data ['loyalty_score'] = loyalty_data.lovely_brand_purchases / loyalty_data.purchases

In [26]: loyalty_data
```

Out[26]:

	user_id	purchases	unique_brands	lovely_brand	lovely_brand_purchases	loyalty_score
0	1031	6	2	Store_Brand	5	0.833333
1	4241	5	2	Brand_4	3	0.600000
2	25971	7	2	Store_Brand	5	0.714286
3	40911	27	5	Brand_4	19	0.703704
4	45181	5	4	Store_Brand	2	0.400000
...	...	...	...	...	...	...
3378	1408767189	5	1	Brand_4	5	1.000000
3379	1408783189	10	2	Store_Brand	8	0.800000
3380	1408783379	6	2	Brand_1	4	0.666667
3381	1408798879	8	3	Store_Brand	4	0.500000

Рисунок 24 – Создание метрики `loyalty_score` для каждого покупателя

Задача выполнена. Данная метрика позволяет ранжировать покупателей по уровню лояльности – рис. 25.

```
In [30]: loyalty_data.sort_values('loyalty_score', ascending=False)
```

Out[30]:

	user_id	purchases	unique_brands	lovely_brand	lovely_brand_purchases	loyalty_score
2143	54915411	10	1	Brand_4	10	1.000000
1662	27647721	6	1	Brand_4	6	1.000000
1655	27415431	8	1	Brand_4	8	1.000000
2880	94697021	9	1	Brand_4	9	1.000000
1657	27472311	7	1	Brand_4	7	1.000000
...	...	...	...	...	...	...
453	2406921	7	5	Brand_2	2	0.285714
577	3616201	7	4	Brand_1	2	0.285714
676	5046282	7	4	Brand_1	2	0.285714
2959	100947471	7	4	Brand_2	2	0.285714
985	12083322	5	5	Brand_1	1	0.200000

3383 rows × 6 columns

Рисунок 25 – Ранжирование покупателей по лояльности

Впрочем, сразу видно, что эта метрика также далека от совершенства, поскольку одинаково оценивает лояльность покупателя, совершившего 10 из 10 покупок любимого бренда, и к примеру, 7 из 7. В обоих случаях, `loyalty_score` будет равен 1, но лояльность первого покупателя явно выше. Можно было бы продолжать дальнейшее усовершенствование этой

метрики, но мы на этом остановимся, поскольку общая поставленная задача выполнена, а базовые идеи и последовательность аналитических этапов рассмотрена.

В заключении осталось рассмотреть вопрос визуализации полученных результатов. Графическое представление данных, в большинстве случаев, нагляднее и понятнее, чем табличное.

6. Простейшие методы визуализации данных

Для визуализации данных будем использовать две дополнительные библиотеки: **Seaborn** и **Matplotlib**. Добавим в первой ячейки импорта следующие команды – рис. 26:

```
In [31]: import pandas as pd

         import seaborn as sns
         import matplotlib.pyplot as plt

         %matplotlib inline
```

Рисунок 26 – Импорт библиотек seaborn и matplotlib

Строка `%matplotlib inline` – активирует интерактивную поддержку **Matplotlib** для блокнота Jupyter. То есть графики будут появляться прямо в out-ячейках Jupyter Notebook.

Seaborn – высокоуровневая библиотека для визуализации данных и выделения их статистических особенностей. Seaborn написана поверх библиотеки **Matplotlib**, но предлагает интерфейс более высокого уровня, а это значит, что вам не придется тратить время на детальное и долгое программирование внешнего вида графиков. Чтобы создать график пригодный для общего понимания данных, потребуется всего одна-две строки кода, а если требуется подготовить график для презентации или публикации, то Seaborn также предоставляет доступ к низкоуровневым настройкам.

Еще одна важная особенность библиотеки Seaborn - это заложенный в нее механизм предобработки данных, возможный благодаря тесной интеграции с библиотекой Pandas. Данные можно передавать прямо в объектах *DataFrame*, а Seaborn сама выполнит всю необходимую работу: агрегацию (выделение подмножеств), статистическую обработку (например вычисление доверительных интервалов) и визуальное выделение полученных результатов.

Точно так же как Python и Pandas позволяют сосредоточиться на задаче, а не коде с помощью которого она решается, Seaborn позволяет сконцентрироваться на важных нюансах визуального представления ваших

данных, а не коде с помощью которого эти нюансы нужно рисовать и выделять.

Пожалуй единственный минус Seaborn заключается в том, что получаемые графики не являются интерактивными - нельзя навести указатель мыши на какую-нибудь точку и увидеть соответствующее ей значение, нельзя выделить какую-то область графика и увеличить ее.

Подробное описание всех возможностей Seaborn с примерами применения можно найти на сайте: https://seaborn.pydata.org/

Для того, чтобы построить простейший график распределения полученной метрики `loyalty_score`, достаточно вызвать метод `distplot()`, указав в качестве аргумента колонку, значения которой будут откладываться по оси OX. (рис. 27). Можно также отключить математическую аппроксимацию (параметр `kde=False`) и получить в чистом виде гистограмму с абсолютным значением количества покупателей по оси OY – рис. 28.

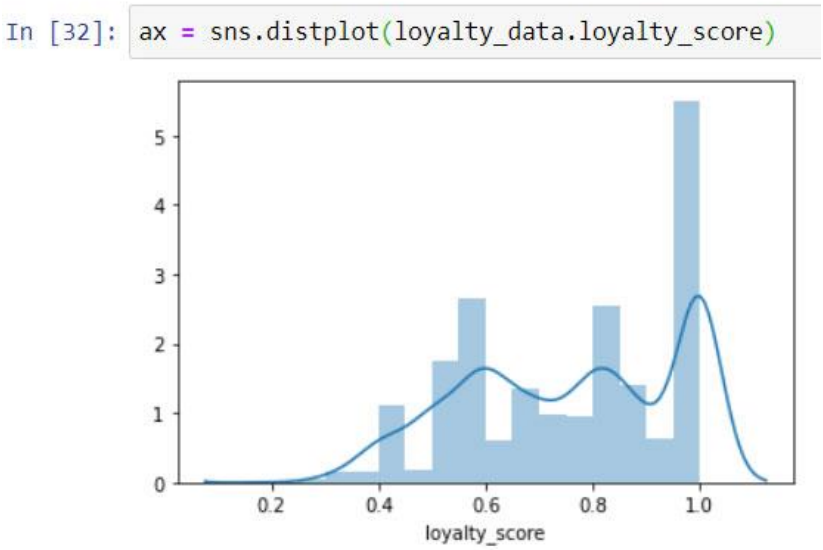
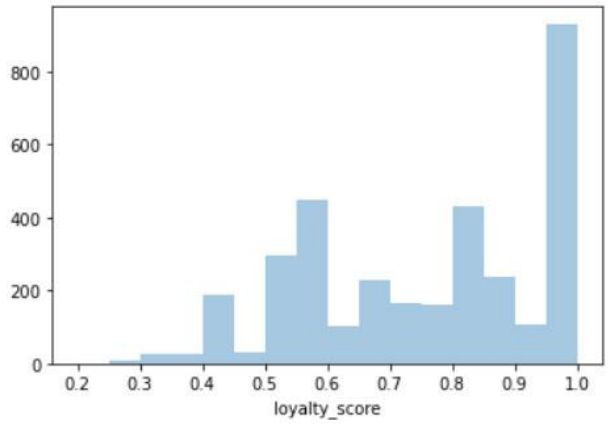


Рисунок 27 - Простейший график распределения полученной метрики `loyalty_score`

```
In [33]: ax = sns.distplot(loyalty_data.loyalty_score, kde=False)
```




```
In [34]: loyalty_data.loyalty_score.median()
Out[34]: 0.8
```

Рисунок 28 - Простейший график распределения полученной метрики `loyalty_score` без математической аппроксимации

Этот график сразу позволяет сделать предварительный вывод о большом количестве лояльных покупателей – медиана располагается в районе значения 0.8. То есть более 50% покупателей с покупками 5 и более обладают высочайшим показателем лояльности от 0.8 до 1.0

Для примера, можно сделать столь же быстрый анализ по брендам – определить, какое медианное значение `loyalty_score` у каждого бренда и у скольких покупателей этот бренд является любимым – рис. 29. После этого можно сразу построить графики другого типа, в котором явно задаются колонки, соответствующие осям (первые два аргумента) и датафрейм, из которого берутся данные для построения (третий аргумент) (рис. 30, 31).

```
In [35]: brand_loyalty = (loyalty_data.groupby('lovely_brand', as_index=False)
                                      .agg({'loyalty_score':'median', 'user_id':'count'}))
```

```
In [36]: brand_loyalty
```

Out[36]:

	lovely_brand	loyalty_score	user_id
0	Brand_1	0.679487	410
1	Brand_2	0.600000	88
2	Brand_3	0.500000	115
3	Brand_4	0.818182	2041
4	Brand_5	0.600000	5
5	Brand_7	0.444444	9
6	Store_Brand	0.750000	715

Рисунок 29 – Анализ по брендам

```
In [37]: ax = sns.barplot(x='lovely_brand', y='loyalty_score', data=brand_loyalty)
```

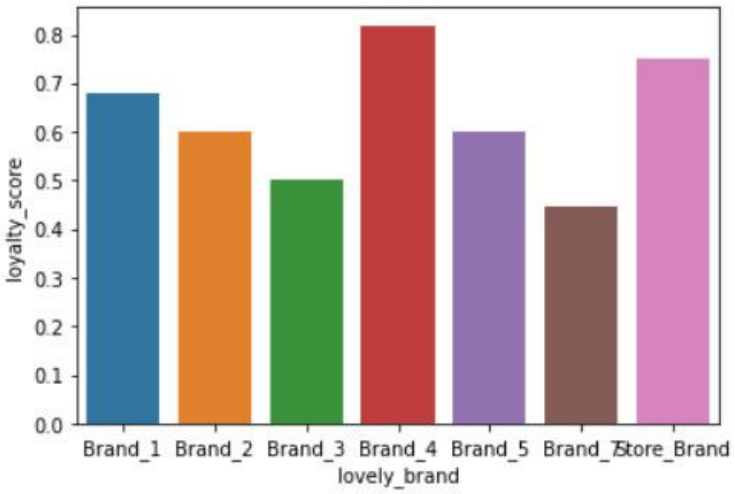


Рисунок 30 – График распределения показателя лояльности по брендам

```
In [38]: ax = sns.barplot(x='lovely_brand', y='user_id', data=brand_loyalty)
```

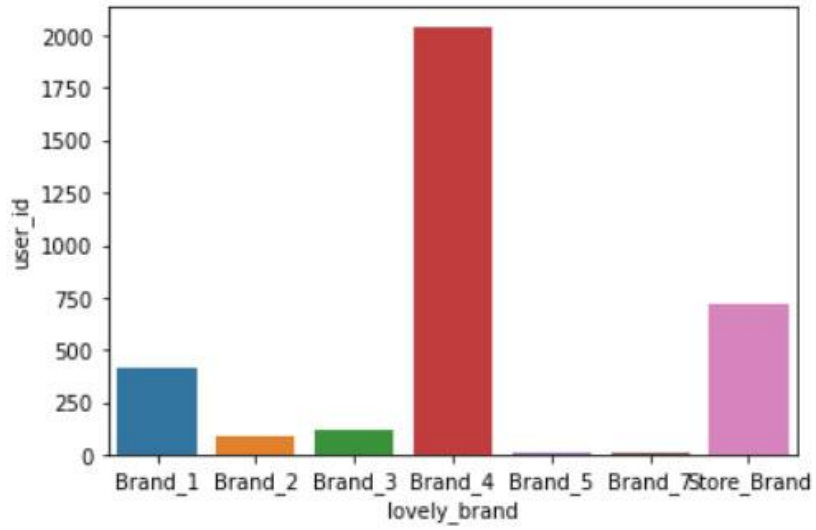


Рисунок 31 - График распределения лояльных покупателей по брендам

Выводы

Подобного рода графики служат хорошим иллюстративным материалом, для быстрого определения ***критериев принятия решений***. В данном случае, критерий принятия решения – это граничное значение показателя лояльности, выше которого мы будем считать покупателей лояльными. Принятие решение это то, ради чего был проведен весь вышеописанный анализ. В ходе принятия решения происходит возврат к бизнес-процессам, от которых мы ушли, начиная с этапа предобработки. В

нашем примере, принятие решения – это определение количества лояльных покупателей, которым бренд выдаст скидочные карты. Оно может быть разным в зависимости от величины скидки или других условий программы лояльности. Быстрый и обоснованный выбор границы лояльности – это то решение, которое может принять руководство бренда, имея на руках наглядные графики анализа данных.