Федеральное государственное бюджетное образовательное учреждение
высшего образования Финансовый университет при Правительстве Российской Федерации
Факультет Информационных технологий и анализа больших данных
Департамент анализа данных и машинного обучения

Курсовая работа
По дисциплине
Технологии разработки приложений для мобильных устройств
“ Решение задачи классификации объектов с помощью градиентного бустинга решающих деревьев”

Выполнил:
Студент 3 курса
Группы ПИ20-6
Савин Алексей
Научный руководитель:
Куликов Александр Анатольевич

г. Москва
2023

Оглавление

Введение

Решение задач классификации является важной областью машинного обучения. В этой задаче объекты необходимо отнести к заданным классам. Для достижения этой цели могут использоваться различные алгоритмы, такие как деревья решений и градиентный бустинг.

Задачи классификации являются задачами машинного обучения и требуют построения алгоритмов, которые могут классифицировать объекты на несколько классов на основе определенных признаков. Классификация является одной из основных задач обработки данных и может применяться в различных областях, таких как компьютерное зрение, биоинформатика и финансы.

Дерево решений - это графическое представление алгоритма, используемого для принятия решений на основе входных данных. Дерево решений имеет узлы, представляющие условия разделения данных на две или более групп, и листья, представляющие конечный результат решения. Построение дерева решений основано на выборе наиболее полезных признаков для разделения данных на классы.

Градиентный бустинг - это метод машинного обучения, при котором слабая модель, такая как дерево решений, обучается последовательно для повышения точности прогнозирования. Градиентный бустинг направлен на минимизацию ошибки прогнозирования путем итеративного обучения моделей на основе градиента функции потерь. Функция потерь используется для измерения разницы между фактическими и прогнозируемыми целевыми переменными.

Формула для бустинга на основе градиента выглядит следующим образом

$F_m(x) = F_{m-1}(x) + \gamma_m f_m(x))$

где $F_{m-1}(x)$ - предыдущее приближение, $f_m(x)$ - новый базовый алгоритм, а γ_m - коэффициент шага.

Градиентный бустинг используется для решения различных задач машинного обучения, таких как классификация, регрессия и ранжирование. С его

помощью можно добиться высокой точности предсказания, особенно при работе с большими объемами данных.

В качестве набора данных для курсовой работы я использовал набор данных типов лесного покрова, а основным инструментом обучения для модели классификации была библиотека python sklearn.

Датасет Forest cover types

Набор данных "Типы лесного покрова" - это набор данных, описывающий информацию о лесном покрове западной части США. Набор данных содержит информацию о географических и климатических условиях, таких как высота, уклон, экспозиция и тип почвы, а также спутниковые снимки и общие карты характеристик из географических информационных систем. Каждое наблюдение в наборе данных описывает конкретный участок леса и соответствующее покрытие. Покрытие представлено категориями, которые являются целевыми переменными данного набора данных. Всего существует семь категорий покрытия, включая сосновый лес, лиственный лес и смешанный лес.

Этот набор данных может быть использован для решения задач классификации, требующих определения класса покрытия участка леса на основе его описания. Такой анализ может быть полезен для планирования лесохозяйственной деятельности, оценки влияния изменения климата на лесные массивы и даже для понимания экосистем. Для решения задач классификации можно использовать алгоритмы машинного обучения для предсказания классов покрытия для новых участков, не описанных в наборе данных. Для этого модель должна быть обучена на существующих данных и протестирована на новых наблюдаемых данных.

В данном случае градиентный бустинг дерева решений является хорошим выбором для решения задачи классификации набора данных о типе лесного покрова, поскольку он решает сложные задачи классификации, может

работать с категориальными признаками и может быть использован для прогнозирования вероятности отнесения территории к каждому классу покрова. Формула градиентного бустинга выражается в терминах функции потерь и алгоритма оптимизации, который может быть настроен в зависимости от конкретной проблемы классификации.

Метод градиентного бустинга

Метод градиентного бустинга является одним из алгоритмов машинного обучения, реализованных в библиотеке Scikit-Learn. Метод представляет собой модификацию алгоритма градиентного бустинга, предназначенного для решения задачи классификации.

Gradient boosting - это метод, который объединяет несколько простых моделей, таких как деревья решений, для создания более сложной модели, которая может выполнить задачу более точно, чем отдельные модели. При градиентном бустинге каждое последующее созданное дерево учитывает ошибку предыдущей модели и корректирует ее, чтобы минимизировать эту ошибку. В результате, после повторения процедуры обучения на наборе данных, наша модель рассчитывает взвешенную сумму всех предсказаний деревьев.

Gradient Boosting Classifier - это реализация градиентного бустинга в библиотеке Scikit-Learn. Метод подходит для задач классификации, когда набор данных содержит числовые, категориальные или кодируемые категориальные признаки. Метод классификации gradient boosting позволяет проводить классификацию, используя различные критерии разбиения, такие как "Джини" и "энтропия", а также желаемое количество деревьев. Они также имеют возможность регулировать глубину дерева и шаги обучения, что позволяет более тонко настроить модель под поставленную задачу.

В целом, Gradient Boosting Classifier является эффективным методом для решения задач классификации в случае наличия набора данных с наличием обширного количества признаков и большого числа записей.

Функция Gradient Boosting Classifier из библиотеки sklearn имеет следующие позиционные аргументы:

- n_estimators: количество деревьев, которые будут участвовать в градиентном бустинге. По умолчанию равно 100.
- learning_rate: коэффициент, определяющий вес каждого дерева во время обучения. Маленький коэффициент даёт меньший вес каждому дереву, что увеличивает вероятность сходимости при обучении. По умолчанию равно 0.1.
- max_depth: максимальная глубина каждого дерева. Если значение None, то деревья будут каждое со своей максимальной глубиной. По умолчанию равно 3.
- criterion: функция измерения качества разбиения параметризована для определения качества деления в вершине. Поддерживается два критерия: «gini» для индекса Джини и «entropy» для прироста информации. По умолчанию используется значение «friedman_mse».

Практическая работа

Загрузка данных

Загружаю датафрейм с помощью встроенных методов sklearn.

```
1 ctypes = sklearn.datasets.fetch_covtype(as_frame = True)
```

```
1 ctypes.keys()
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'feature_names', 'DESCR'])
```

Из структуры нам интересны: датафрейм data, массив целевой переменной target и описание DESCR.

```
1 print(ctypes.get('DESCR'))
```

```
.. _covtype_dataset:

Forest covertypes
-----------------

The samples in this dataset correspond to 30×30m patches of forest in the US,
collected for the task of predicting each patch's cover type,
i.e. the dominant species of tree.
There are seven covertypes, making this a multiclass classification problem.
Each sample has 54 features, described on the
`dataset's homepage <https://archive.ics.uci.edu/ml/datasets/Covertype>`__.
Some of the features are boolean indicators,
while others are discrete or continuous measurements.

**Data Set Characteristics:**

    =================   ============
    Classes                        7
    Samples total             581012
    Dimensionality                54
    Features                     int
    =================   ============
```

Выгружаем данные в отдельные переменные для более быстрого доступа.

```
1 data = ctypes.get('data')
2 target = ctypes.get('target')
```

Исходный датафрейм содержит 54 колонки (соответственно 54 признака), которые будут влиять на определение класса объекта.

```
1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 581012 entries, 0 to 581011
Data columns (total 54 columns):
```

Массив target состоит из одной колонки – целевого класса для каждой записи.

```
1 target.info()
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 581012 entries, 0 to 581011
Series name: Cover_Type
Non-Null Count   Dtype
--------------   -----
581012 non-null  int32
```

Датафреймы изначально не имели отсутствующие значения, соответственно обработка данных не потребует замену данных средними значениями в случае пропусков.

Целевая переменная содержит значения от 1 до 7, при этом есть явный количественный перевес двух классов над другими.

```
[ ]  1 print(target.unique())
     2 print(target.value_counts().sort_index())
```

```
[5 2 1 7 3 6 4]
1    211840
2    283301
3     35754
4      2747
5      9493
6     17367
7     20510
Name: Cover_Type, dtype: int64
```

Работа с сырыми данными

Для начала я разделил сырые данные на тестовую и тренировочную выборки, чтобы посмотреть на эффективность модели при работе с датасетом без предварительной обработки.

```
1 x_train, x_test, y_train, y_test = train_test_split(data, target, test_size=0.2, random_state=True)
```

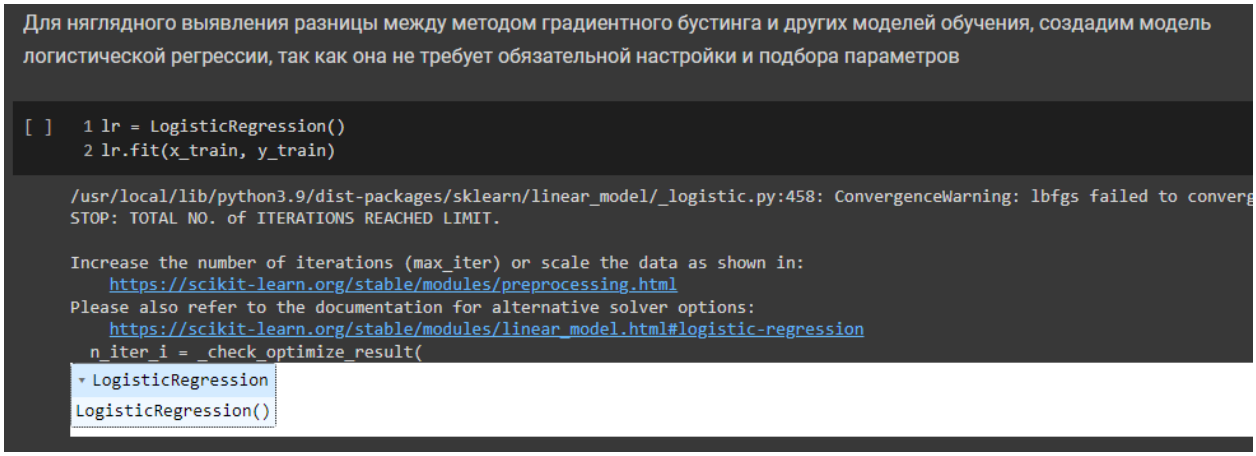
Разделим сырые данные на обучающую и тестовую выборки, чтобы посмотреть эффективность моделей обучения на них

Для того, чтобы получить наглядное сравнение метода градиентного бустинга с другими, я использовал метод логистической регрессии. Логистическая регрессия - это метод систематизации, используемый для отнесения объектов к набору дискретных классов. В отличие от линейной регрессии, которая

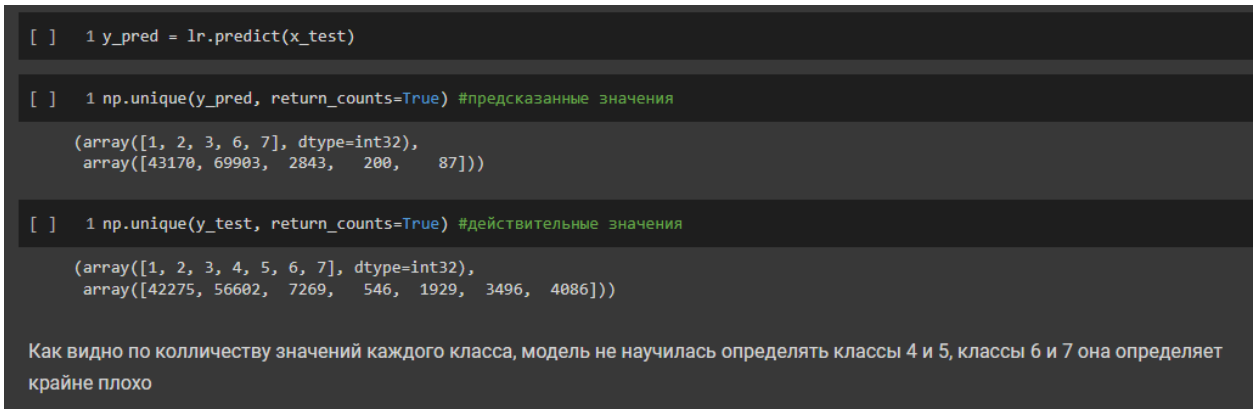
выводит непрерывные числа, LogReg преобразует отдельные результаты с сигмоидальной поддержкой для получения значений вероятности, которые затем могут быть округлены до одного из дискретных классов.

Модель логистической регрессии

Создаю модель логистической регрессии без внесения изменений в стандартные позиционные аргументы.

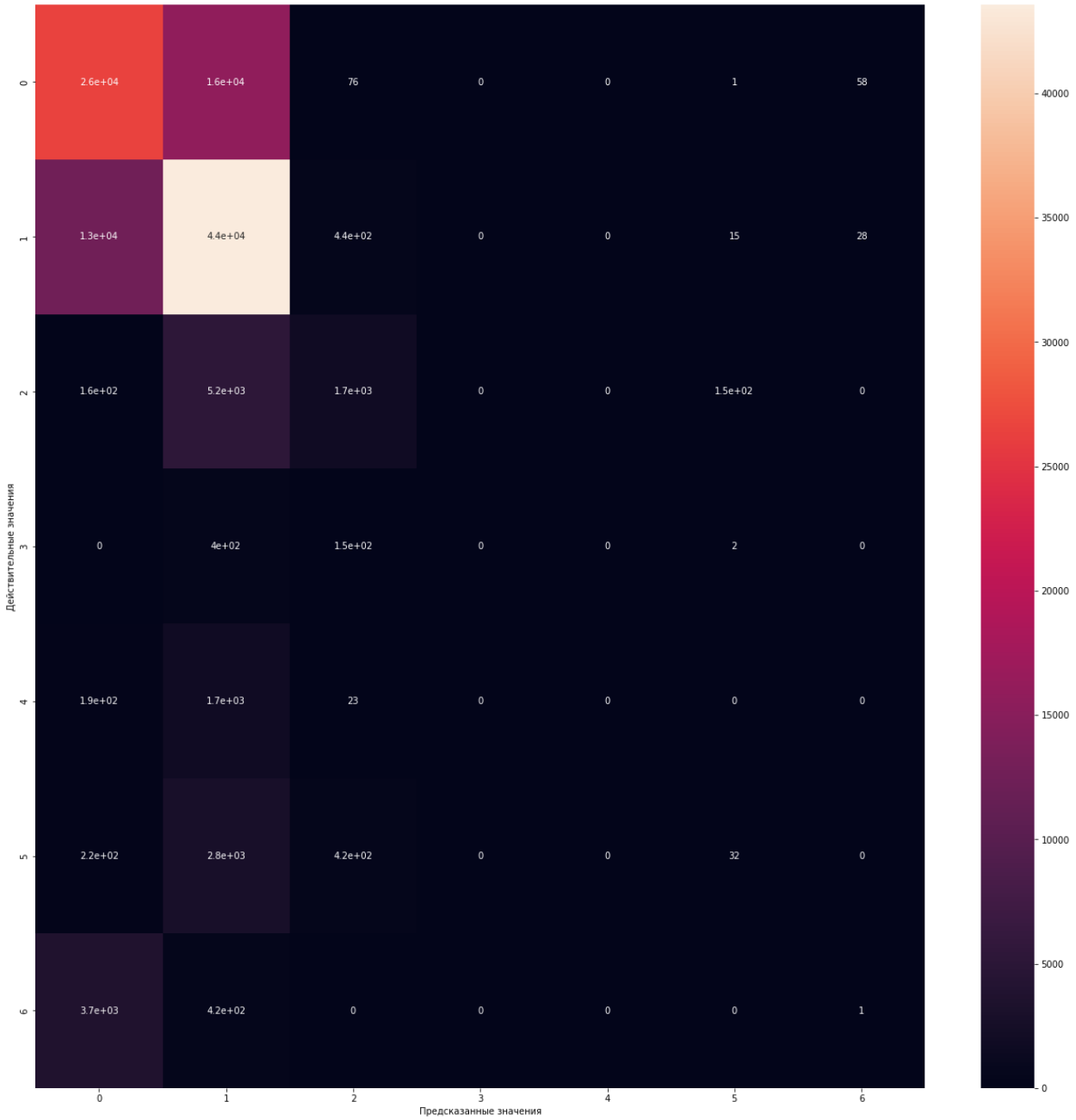


После обучения, которое заняло малое количество времени, предскажем значения на тестовой выборке.



В итоге модель логистической регрессии крайне плохо определяет классы 6 и 7, а классы 4 и 5 не предсказывает вообще.

Таблица 1 Тепловая карта предсказанных значений



Отчет по метрике аккуратности, а также подробный отчет по метрикам precision, recall, f1-score и support подтверждает то, что модель не эффективна на сырых данных.

```
[ ]  1 metrics.accuracy_score(y_test, y_pred)

0.6173764876982522

[ ]  1 print(metrics.classification_report(y_test, y_pred))

              precision    recall  f1-score   support

           1       0.61      0.62      0.62     42275
           2       0.62      0.77      0.69     56602
           3       0.61      0.24      0.34      7269
           4       0.00      0.00      0.00       546
           5       0.00      0.00      0.00      1929
           6       0.16      0.01      0.02      3496
           7       0.01      0.00      0.00      4086

    accuracy                           0.62    116203
   macro avg       0.29      0.23      0.24    116203
weighted avg       0.57      0.62      0.58    116203

/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
  _warn_prf(average, modifier, msg_start, len(result))
```

Соответственно метрика точности у неё крайне малая

Модель градиентного бустинга

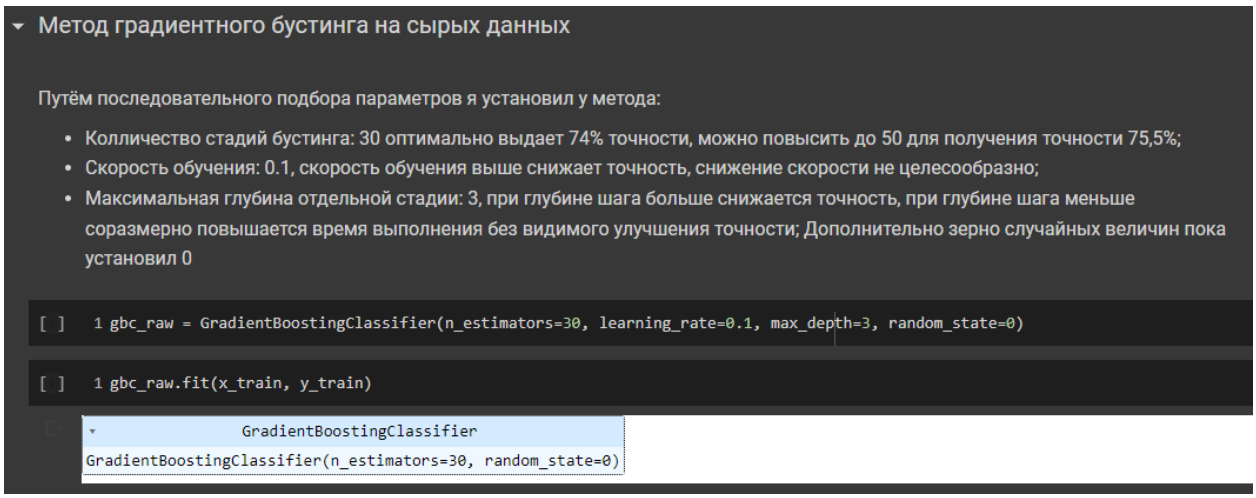
Далее я составил и обучил модель градиентного бустинга. Путем нескольких итераций обучения получил следующие значения позиционных аргументов:

Колличество стадий бустинга: 30 оптимально выдает 74% точности, можно повысить до 50 для получения точности 75,5%;

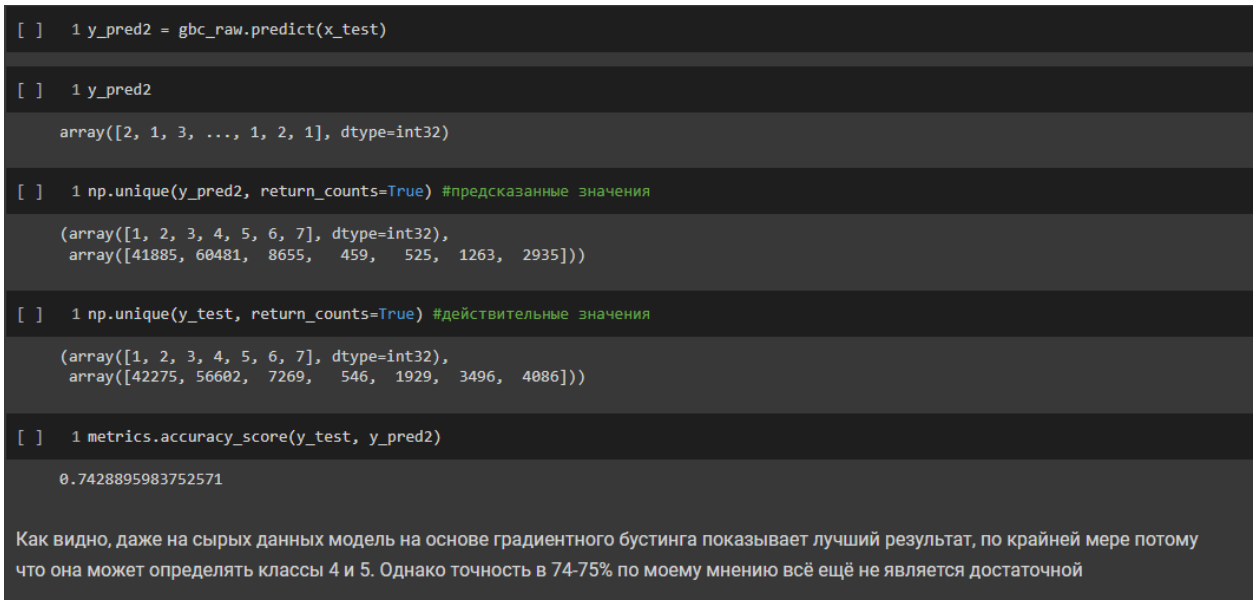
Скорость обучения: 0.1, скорость обучения выше снижает точность, снижение скорости не целесообразно;

Максимальная глубина отдельной стадии: 3, при глубине шага больше снижается точность, при глубине шага меньше соразмерно повышается время выполнения без видимого улучшения точности.

Дополнительно зерну случайных величин установил значение 0.



Обучение этой модели заняло куда больше времени, по сравнению с предыдущей. Это объясняется в том числе тем, что данный метод библиотеки не поддерживает многопоточность и в следствие каждая итерация обучения малых деревьев происходит последовательно, а не параллельно друг другу.

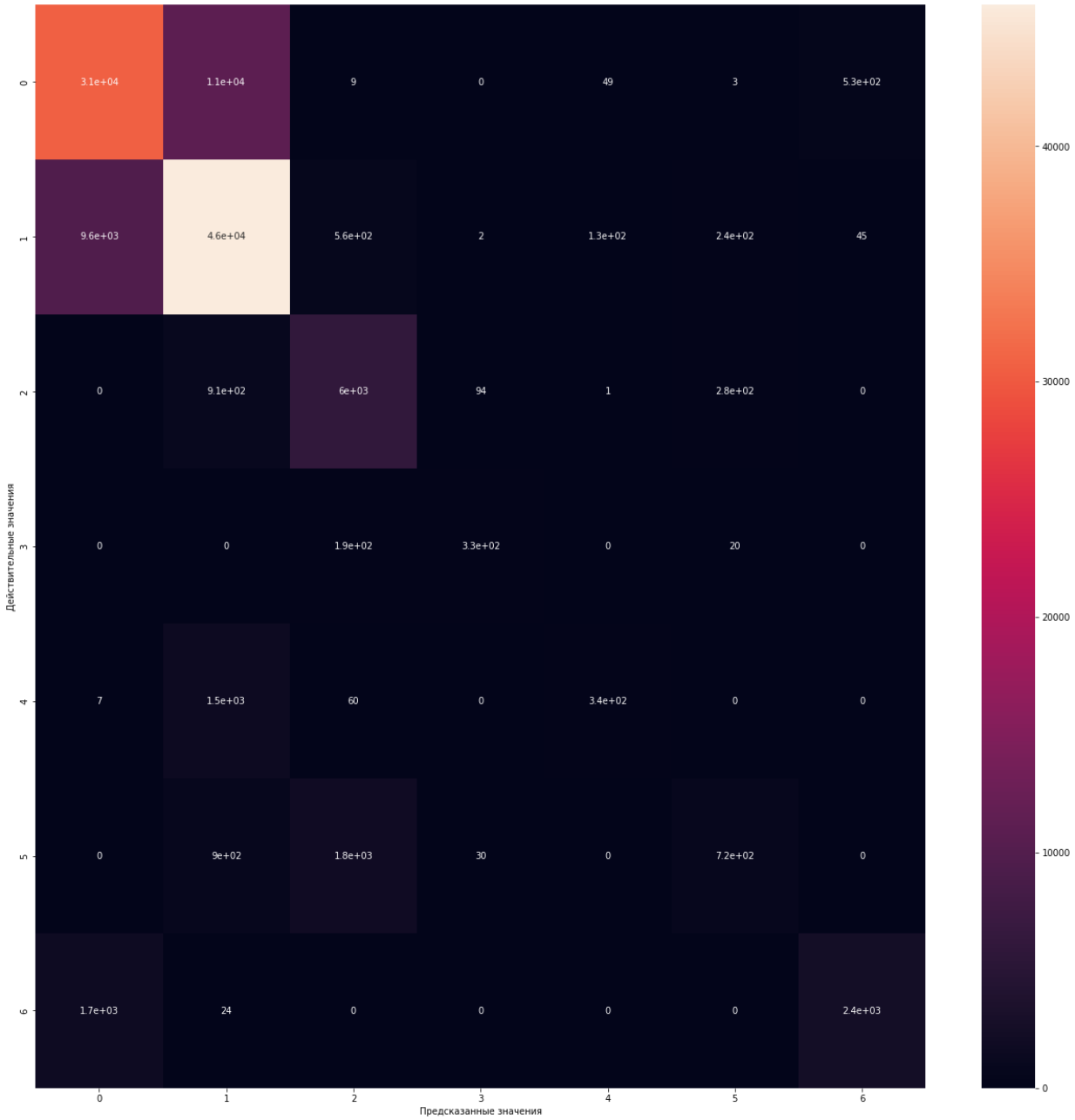


В итоге обучения модель на тестовой выборке показала результат лучше, чем логистическая регрессия. Путем калибровки аргументов метода я смог добиться стабильной точности в 74-75%, однако эта точность достигнута на сырых данных. Соответственно, я посчитал, что это значение можно улучшить путем обработки датасета.

```
[ ]   1 print(metrics.classification_report(y_test, y_pred2))
```

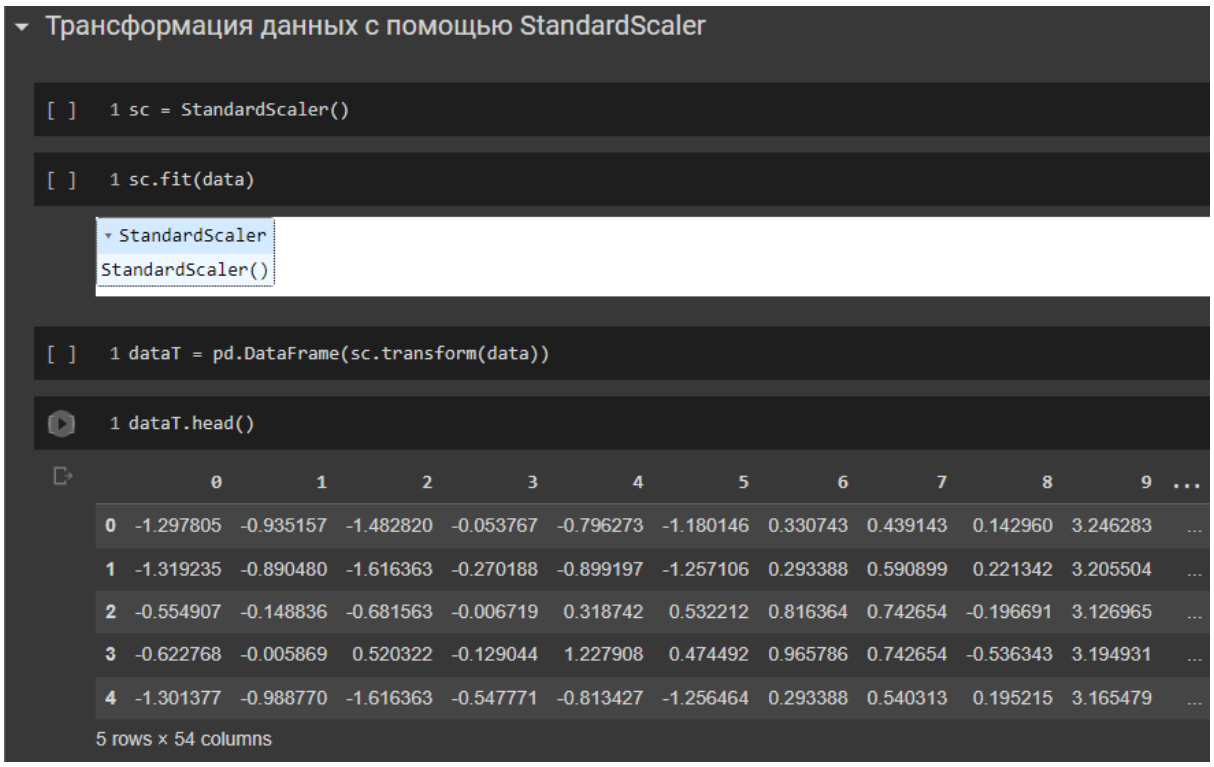
	precision	recall	f1-score	support
1	0.73	0.72	0.73	42275
2	0.76	0.81	0.79	56602
3	0.69	0.82	0.75	7269
4	0.73	0.61	0.66	546
5	0.65	0.18	0.28	1929
6	0.57	0.21	0.30	3496
7	0.80	0.58	0.67	4086
accuracy			0.74	116203
macro avg	0.70	0.56	0.60	116203
weighted avg	0.74	0.74	0.73	116203

Таблица 2 Тепловая карта предсказанных значений



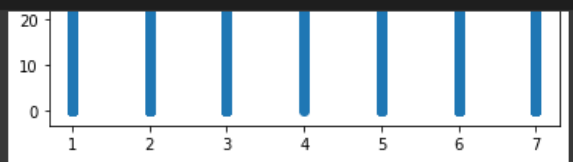
Начало обработки данных

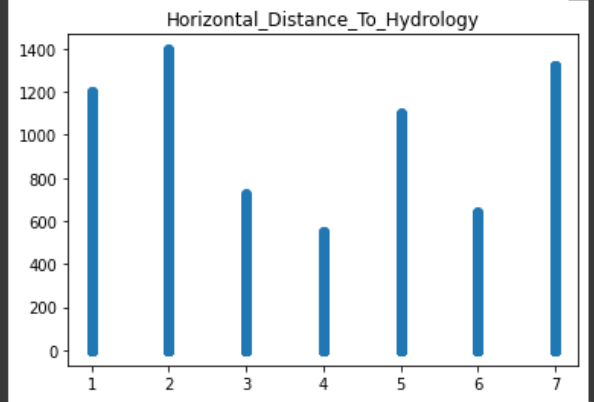
Для дальнейшей работы с датасетом я подключил метод StandartScaler, который преобразует значения столбцов и приводит их к единому виду, без изменения размера датасета.

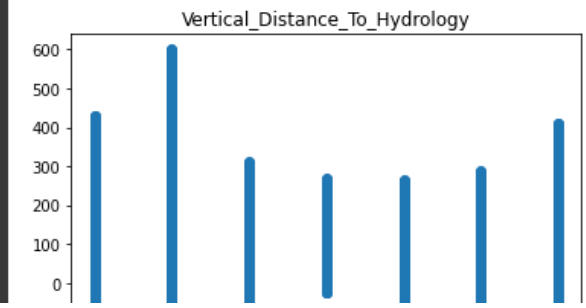


Следующим шагом был поиск зависимостей между целевым атрибутом и признаками, вычисление корреляции признаков и целевого атрибута, а также поиск столбцов с выбросами. К сожалению, максимальная корреляция между признаками составила чуть больше 32%, у большинства признаков она не превышает 15% как положительно, так и отрицательно. Следовательно, очистка датасета не обоснована, так как замена значений приведет к понижению точности модели на сырых данных.

```
[ ]  1 for column in columns:
     2     plt.scatter(target, data[column])
     3     plt.title(column)
     4     plt.show()
```

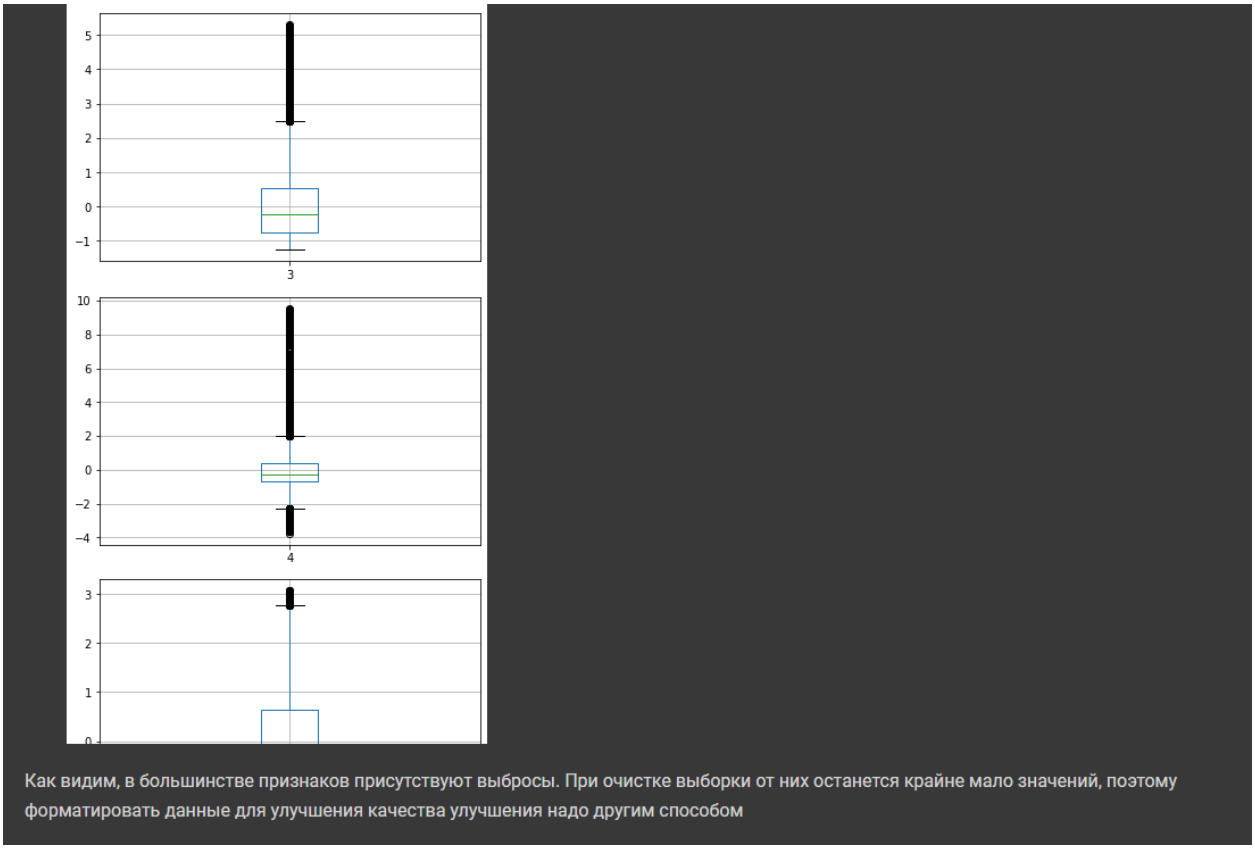






```
for column in range(54):
  print("{} - corr: {}".format(column, dataT[column].corr(dataT["target"], method='pearson')))
```

```
0 - corr: -0.26955377763050464
1 - corr: 0.017079802032774274
2 - corr: 0.14828540507947907
3 - corr: -0.02031662163693506
4 - corr: 0.08166402150981635
5 - corr: -0.15344975909042438
6 - corr: -0.035415003646699444
7 - corr: -0.09642600166233192
8 - corr: -0.04828973004683914
9 - corr: -0.10893553610322682
10 - corr: -0.20391321381925592
11 - corr: -0.048058949741315325
12 - corr: 0.06684564296338946
13 - corr: 0.32319955390915317
14 - corr: 0.09082815211746842
15 - corr: 0.11813526031668656
16 - corr: 0.06806445519051268
17 - corr: 0.09967186439491588
18 - corr: 0.07788996111809382
19 - corr: 0.11295827807212458
20 - corr: -0.0004955165229846606
21 - corr: -0.0036667935899179524
22 - corr: -0.006109533783057859
23 - corr: 0.24387630152794465
24 - corr: 0.035378737686072824
25 - corr: -0.023601133704748575
26 - corr: 0.02440365592014329
27 - corr: 0.06556207167657435
28 - corr: 0.0064248452527425545
29 - corr: 0.00984434530257407
30 - corr: 0.09058230355856872
31 - corr: 0.007390381557645007
32 - corr: -0.03645192377424674
33 - corr: -0.028664710189508686
34 - corr: -0.025400202457916656
35 - corr: -0.14174611949946497
36 - corr: -0.13505517100440448
37 - corr: -0.06874587894793496
38 - corr: -0.006449047448999298
39 - corr: -0.0003748428010107029
40 - corr: -0.014406539765595526
41 - corr: -0.001702393047555508
42 - corr: -0.12493259757283441
43 - corr: -0.010436448835294931
44 - corr: -0.06534706912172909
45 - corr: -0.0755620259959868
46 - corr: -0.06250174683524469
47 - corr: 0.0046426232473868005
48 - corr: 0.08031505015427497
49 - corr: 0.02539653702908038
50 - corr: 0.08027147359471183
51 - corr: 0.16016960859426624
```



Обработка датасета

В качестве одного из методов решения я попытался сбалансировать обучающую выборку. Для этого я выявил наименее малочисленные классы – классы 4, 5 и 6, составил новый датафрейм, который содержал 40,000 экземпляров классов 1 и 2, 30,000 экземпляров класса 7, а классы 4, 5 и 6 объединил в один.

```
[ ]   1 dataL.shape

      (169607, 55)

[ ]   1 print(dataL["target"].unique())
      2 print(dataL["target"].value_counts().sort_index())

      [1 2 3 4 7]
      1    40000
      2    40000
      3    30000
      4    29607
      7    30000
      Name: target, dtype: int64
```

В последствие датасет, содержащий истинные значения классов 4-6, будет использоваться для обучения микромодели, определяющей истинный класс для предсказанного значения 4 основной моделью, и они обе будут работать в тандеме. Фрейм DataL в свою очередь получается сбалансирован по колличеству элементов классов.

Обучение моделей на обработанных данных

После этого я составил обучающие и тестовые выборки для двух моделей, а также обучил сами модели.

```
[ ]  1 x_train_bal, x_test_bal, y_train_bal, y_test_bal = train_test_split(dataL, new_target, test_size=0.2, random_state=True)
```

```
[ ]  1 gbc = GradientBoostingClassifier(n_estimators=30, learning_rate=0.1, max_depth=100, random_state=0, verbose=3)
```

```
[ ]  1 x_train_mini, x_test_mini, y_train_mini, y_test_mini = train_test_split(merged, merged_target, test_size=0.2, random_state=True)
```

```
[ ]  1 gbc_mini = GradientBoostingClassifier(n_estimators=30, learning_rate=0.1, max_depth=10, random_state=0, verbose=3)
```

```
[ ]  1 gbc.fit(x_train_bal, y_train_bal)
```

```
      Iter       Train Loss   Remaining Time
         1           1.2210           5.72m
         2           0.9749           7.64m
         3           0.7946           8.09m
         4           0.6555           8.16m
         5           0.5449           8.05m
         6           0.4555           7.86m
         7           0.3823           7.63m
         8           0.3218           7.36m
         9           0.2715           7.04m
        10           0.2294           6.73m
        11           0.1942           6.42m
        12           0.1645           6.08m
        13           0.1395           5.76m
        14           0.1184           5.42m
        15           0.1006           5.09m
        16           0.0855           4.76m
        17           0.0727           4.41m
        18           0.0618           4.08m
        19           0.0526           3.75m
        20           0.0447           3.48m
        21           0.0381           3.16m
        22           0.0324           2.81m
        23           0.0276           2.48m
        24           0.0235           2.14m
        25           0.0200           1.82m
        26           0.0170           1.47m
        27           0.0145           1.11m
        28           0.0124          45.05s
        29           0.0105          22.44s
        30           0.0090           0.00s
```

```
[ ]   1 gbc_mini.fit(x_train_mini, y_train_mini)

      Iter       Train Loss   Remaining Time
         1           0.7194          14.12s
         2           0.6016          26.65s
         3           0.5113          22.18s
         4           0.4391          20.40s
         5           0.3798          19.81s
         6           0.3296          19.12s
         7           0.2880          18.36s
         8           0.2526          16.99s
         9           0.2221          15.77s
        10           0.1954          14.68s
        11           0.1727          13.68s
        12           0.1530          12.76s
        13           0.1357          11.90s
        14           0.1207          11.07s
        15           0.1072          10.29s
        16           0.0955           9.53s
        17           0.0852           8.79s
        18           0.0761           8.06s
        19           0.0680           7.35s
        20           0.0607           6.65s
        21           0.0541           5.96s
        22           0.0485           5.28s
        23           0.0434           4.60s
        24           0.0389           3.95s
        25           0.0350           3.33s
        26           0.0316           2.70s
        27           0.0286           2.04s
        28           0.0260           1.35s
        29           0.0236           0.67s
        30           0.0215           0.00s
```

Результат работы тандема моделей показал результат точности 98% на вспомогательной модели, определяющей классы с 4 по 6, и точность 90% на основной модели.

```
1 y_pred3 = gbc.predict(x_test_bal)
```

```
1 np.unique(y_pred3, return_counts=True) #предсказанные значения
```

```
(array([1, 2, 3, 4, 7], dtype=int32), array([7633, 8726, 5967, 5631, 5965]))
```

```
1 np.unique(y_test_bal, return_counts=True) #действительные значения
```

```
(array([1, 2, 3, 4, 7], dtype=int32), array([8033, 8008, 6028, 5953, 5900]))
```

```
1 data_subclass = x_test_bal.iloc[np.where(y_pred3 == 4)].copy()
```

```
1 y_pred3_subclass = gbc_mini.predict(x_test_mini)
```

```
1 np.unique(y_pred3_subclass, return_counts=True)
```

```
(array([4, 5, 6], dtype=int32), array([ 571, 1919, 3432]))
```

```
1 np.unique(y_test_mini, return_counts=True)
```

```
(array([4, 5, 6], dtype=int32), array([ 588, 1923, 3411]))
```

```
1 data_subclass.shape
```

```
(5631, 54)
```

```
1 metrics.accuracy_score(y_test_bal, y_pred3)
```

```
0.9082306467779022
```

```
1 metrics.accuracy_score(y_test_mini, y_pred3_subclass)
```

```
0.9836203985140155
```

Проверка качества обучения

Проверку качества обучения я провел на необработанных трансформированных данных.

```
[ ]  1 dataX = dataT.copy()
```

```
[ ]  1 dataX.head()
```

	0	1	2	3	4	5	6	7	8	9	...
0	-1.297805	-0.935157	-1.482820	-0.053767	-0.796273	-1.180146	0.330743	0.439143	0.142960	3.246283	...
1	-1.319235	-0.890480	-1.616363	-0.270188	-0.899197	-1.257106	0.293388	0.590899	0.221342	3.205504	...
2	-0.554907	-0.148836	-0.681563	-0.006719	0.318742	0.532212	0.816364	0.742654	-0.196691	3.126965	...
3	-0.622768	-0.005869	0.520322	-0.129044	1.227908	0.474492	0.965786	0.742654	-0.536343	3.194931	...
4	-1.301377	-0.988770	-1.616363	-0.547771	-0.813427	-1.256464	0.293388	0.540313	0.195215	3.165479	...

5 rows × 54 columns

```
[ ]  1 x_train, x_test, y_train, y_test = train_test_split(dataX, target, test_size=0.2, random_state=True)
```

```
[ ]  1 res = gbc.predict(x_test)
```

```
     1 np.unique(res, return_counts=True)
```

```
(array([1, 2, 3, 4, 7], dtype=int32),
 array([41344, 56005,  7542,  6775,  4537]))
```

```
[ ]  1 data_sub = x_test_bal.iloc[np.where(res == 4)].copy()
```

```
[ ]  1 data_sub.shape
```

```
(6775, 54)
```

```
[ ]  1 temp = np.where(res == 4)[0]
     2 indexes = []
     3 for element in range(len(temp)):
     4   indexes.append([element, temp[element]])
     5 indexes
```

```
[28, 686],
[29, 716],
[30, 725],
[31, 754],
[32, 783],
[33, 787],
[34, 789],
[35, 790],
```

```
[ ]  1 res_sub = gbc_mini.predict(data_sub)
```

```
[ ]  1 for i in indexes:
     2   res[i[1]] = res_sub[i[0]]
```

```
[ ]  1 np.unique(res, return_counts=True)
```

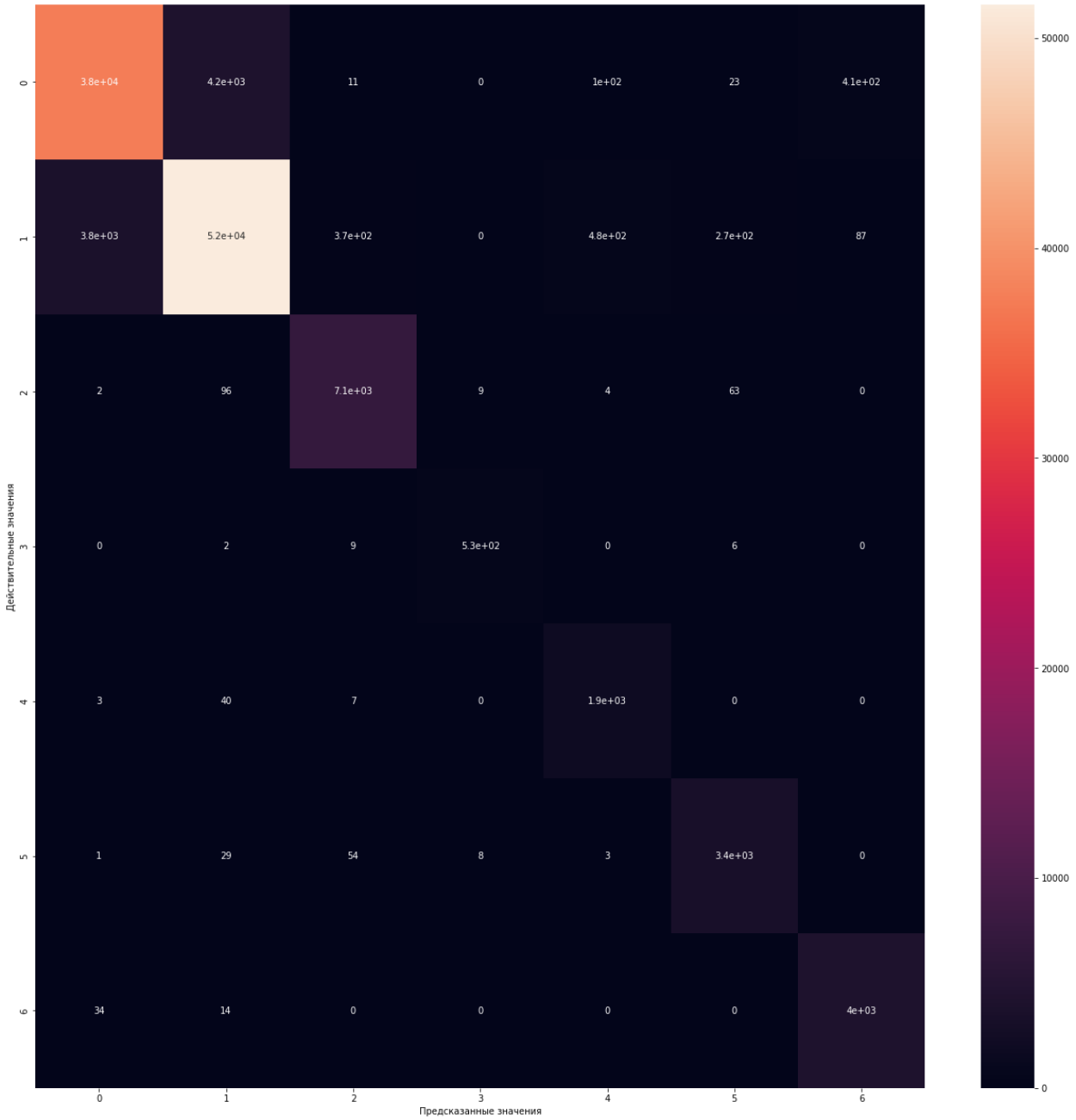
```
(array([1, 2, 3, 4, 5, 6, 7], dtype=int32),
 array([41344, 56005,  7542,   546,  2463,  3766,  4537]))
```

Проверка метрик показала, что по сравнению с моделью, обученной на сырых данных, текущая показала лучший результат, при этом не переобучившись. Значение метрики precision, однако, всё ещё западает на классе 5, составляя 0.76, но всё равно показывая прирост в 0.11 по сравнению с предыдущей моделью.

```
1 print(metrics.classification_report(y_test, res))
```

	precision	recall	f1-score	support
1	0.91	0.89	0.90	42275
2	0.92	0.91	0.92	56602
3	0.94	0.98	0.96	7269
4	0.97	0.97	0.97	546
5	0.76	0.97	0.86	1929
6	0.90	0.97	0.94	3496
7	0.89	0.99	0.94	4086
accuracy			0.91	116203
macro avg	0.90	0.95	0.92	116203
weighted avg	0.91	0.91	0.91	116203

Таблица 3 Тепловая карта предсказанных значений итоговой модели



Вывод

Несмотря на то, что обучение модели на крупном датасете заняло достаточно много времени, метод градиентного бустинга, по сравнению с другими альтернативными, является достаточно удобным инструментом машинного обучения. Так как деревья решений сами по себе стремятся к переобучению на данных, которые им даны, градиентный бустинг не позволяет малым моделям иметь доступ ко всем данным, способствуя повышению качества предсказания. Соответственно переобучение модели возможно только в

случаях неправильной обработки данных или неправильной настройки позиционных аргументов. В ходе данной работы я также получал такие результаты. Шаги к их получению можно посмотреть в неиспользованном коде в исходном файле.

Ссылки

https://colab.research.google.com/drive/15i72vH5V3HeJ-PSP4qYdpVgmnJ7fCbxG?usp=sharing – исходный файл с практической частью работы.

https://academy.yandex.ru/handbook/ml/article/gradientnyj-busting - Академия яндекса | Градиентный бустинг

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html - официальная документация Scikit Learn по методу GradientBoostingClassifier.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression - официальная документация Scikit Learn по методу LogisticRegression.

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler - официальная документация Scikit Learn по методу StandardScaler.

https://neurohive.io/ru/osnovy-data-science/gradientyj-busting/ - Градиентый бустинг — просто о сложном