

## Ансамблевое обучение

Ансамблевое обучение – это раздел машинного обучения, в котором для решения различных задач используется ни одна модель, а ансамбль моделей. Ансамблевое обучение делится на 3 основных вида:

- **Stacking.** Строится  $n$  разных моделей, результаты которых являются мета-факторами для мета-модели. То есть мета-модель – это модель, которая обучается на результатах базовых моделей.
- **Bagging** (Случайный лес). Строится  $n$  независимых друг от друга моделей, затем, в случае решения задачи регрессии, их результаты усредняются, в случае классификации – берется наиболее встречаемый результат моделей.
- **Boosting** (XGBoost, CatBoost, Adaboost, GBDT). Строится последовательно  $n$  моделей, которые зависимы друг от друга.

Рассмотрим подробнее.

### Stacking

Обучение происходит на предсказаниях разных моделей, т. е. каждая последующая модель учитывает предыдущие предсказания как новые признаки.

Этапы стекинга моделей:

- Обучаем несколько разных моделей
- Производим предсказания для каждой модели (получаем мета-факторы)
- Эти предсказания используем как признаки в дополнение к изначальной выборке
- Получившийся датасет используем как новую обучающую выборку для мета-модели

### Bagging

Существует такое понятие, как «мудрость масс». Коллективное мнение широкой группы случайно собранных людей, возможно более точное чем мнение отдельно взятого человека, даже если этот человек является экспертом в данном вопросе.

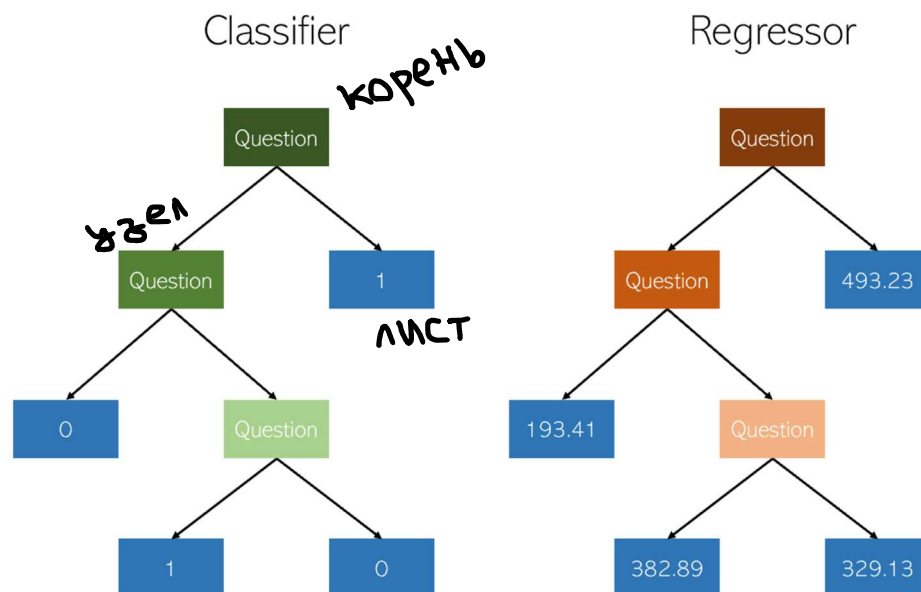
Стандартным примером бэггинга является случайный лес ( $n$  деревьев решений). Но мы не ограничены только этим алгоритмом. Можно построить  $n$  моделей линейной регрессии и найти среднее значение их результатов, которое будет результатом ансамблевого обучения. Можно построить  $n$  логистических регрессий и в качестве результата ансамблевого обучения взять моду выводов моделей.

Рассмотрим случайный лес. Случайный лес состоит из нескольких деревьев решений, что позволяет получить более точные и стабильные прогнозы. Рассмотрим, что представляет собой дерево решений.

Дерево решений – это средство поддержки принятия решений. Метод представления решающих правил в определенной иерархии

Дерево решений к данным задает вопросы. И в зависимости от результата, делит данные на две группы. Деление на группы продолжается до некоторой точки останова. Дерево состоит из корневой вершины (root), далее дерево содержит узлы, вершины, которые содержат «дочерние узлы», если их нет, то такие вершины называются листьями.

Дерево решений может решать задачу регрессии и классификации. В вершинах дерева лежат: условие (вопрос к данным), значение ошибки, количество объектов, которые удовлетворяют условию (вопросу), предсказанное значение. В случае регрессии – предсказанное значение – это среднее значение элементов в вершине. В случае классификации – мода значений в вершине. В листьях нет вопросов, там только объекты, которые удовлетворяют некоторому условию и значение ошибки.



Качество вершины можно оценить с помощью нескольких метрик, которые минимизируются при построении дерева решений. Для задачи регрессии – это средний квадрат ошибок (MSE), для задачи классификации – это индекс Джини или энтропия. Цель алгоритма – добиться максимальной «чистоты» вершин для задачи классификации или наименьшей ошибки MSE для задачи регрессии.

У деревьев решений есть три основных критерия останова:

- max depth (максимальная глубина дерева, чтобы дерево сильно не ветвилось);
- min samples leaf (минимальное количество примеров в листе);
- max leaf nodes (максимальное количество листьев).

Они необходимы для того, чтобы дерево не подгонялось под исходные данные (модель не переобучалась).

Для случайного леса деревьям решений на вход подаются не все данные (которых может быть очень много, и дерево будет считаться неприлично долго), а подвыборки из данных. Результаты работы всех деревьев усредняются или берется их мода и подается на выход случайного леса.

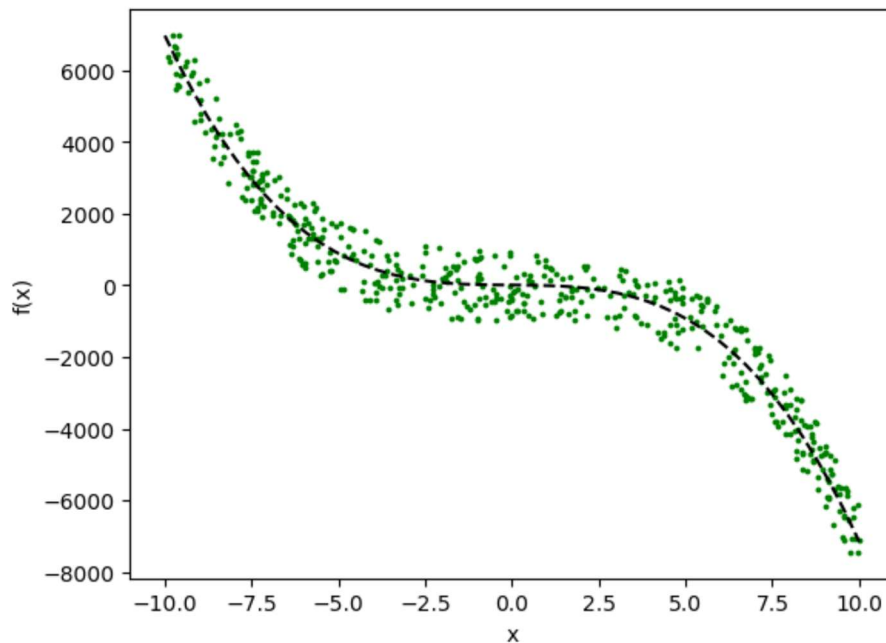
Рассмотрим небольшой пример того, как случайный лес уменьшает дисперсию деревьев решений:

```
1 import numpy as np
2 import pandas as pd
3 import warnings
4 import matplotlib.pyplot as plt
5 from sklearn.model_selection import train_test_split
6 from sklearn import tree
7 from sklearn import ensemble
8
9 warnings.filterwarnings('ignore')
```

```
1 # исходная функция
2 def f(x):
3     return 6 - 6*x - x**2 - 7*x**3
```

```
1 #добавим немного шума к исходной прямой
2 #создадим 10 подвыборок
3 x_datasets = []
4 y_datasets = []
5
6 for i in range(10):
7     xx = np.random.uniform(-10,10,50) #равномерно распределенные значение на интервале от -10 до 10
8     x_datasets.append(xx)
9     y_datasets.append([f(i) for i in xx] + np.random.uniform(-1000,1000,50)) #добавление шума к каждому y(xi)
```

```
1 # необходимо найти зависимость зеленых точек
2 x = np.linspace(-10,10,50)
3 y = f(x)
4 plt.xlabel('x')
5 plt.ylabel('f(x)')
6 for i in range(10):
7     plt.scatter(x_datasets[i],y_datasets[i], c = 'green', s = 3)
8 plt.plot(x,y, '--', color = 'black')
9 plt.show()
```



```

1 #обучим на каждой подвыборке дерево решений
2 models = []
3 for i in range(10):
4     model_tree = tree.DecisionTreeRegressor(max_depth=8, random_state=1)
5     model_tree.fit(x_datasets[i].reshape(-1,1), y_datasets[i])
6     models.append(model_tree)
7

```

```

1 #получили 10 моделей деревьев решений, которые обучены на подвыборках
2 models

```

```

[DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1),
 DecisionTreeRegressor(max_depth=8, random_state=1)]

```

```

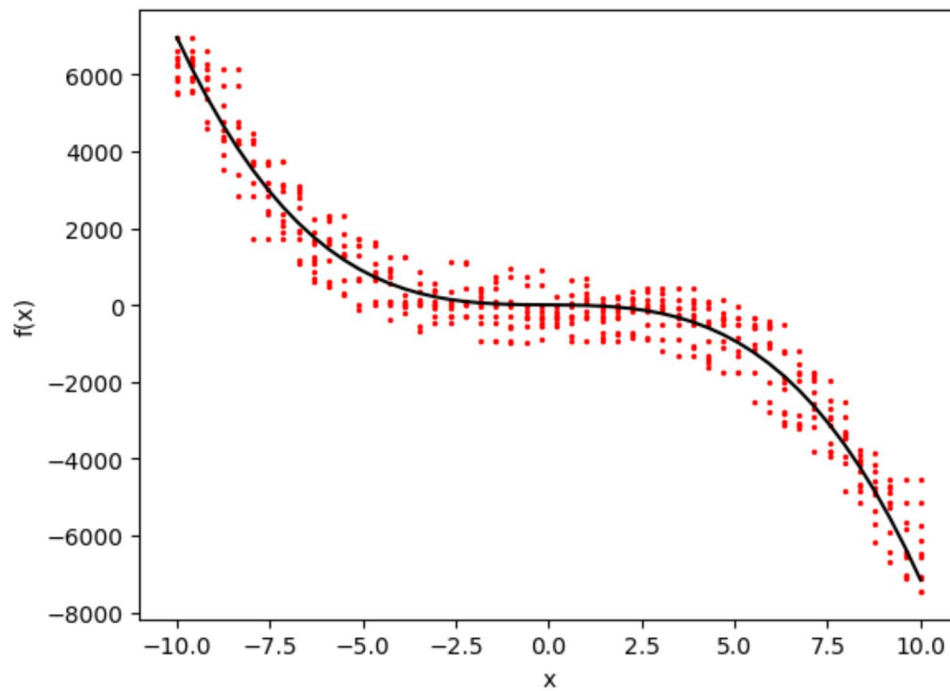
1 #находим прогноз каждого дерева
2 y_pred = []
3 for i in range(len(models)):
4     y_pred.append(models[i].predict(x.reshape(-1,1)))

```

```

1 # красные точки -- это то, что прогнозируют деревья
2 plt.xlabel('x')
3 plt.ylabel('f(x)')
4 for i in range(10):
5     plt.scatter(x,y_pred[i], c = 'red', s = 2)
6 plt.plot(x,y, color = 'black')
7 plt.show()

```



```

1  #берем среднее значение результатов моделей для каждого  $y(x_i)$ 
2  mean_pred = np.array(y_pred).mean(axis=0)

```

```

1  mean_pred

```

```

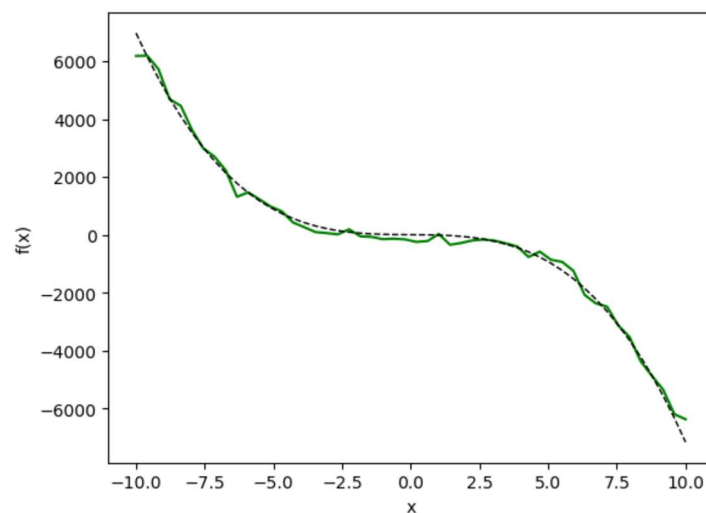
]: array([ 6175.92085668,  6184.41256171,  5719.60685949,  4679.04855232,
  4450.99150764,  3614.42680953,  2986.88776533,  2696.32023528,
  2226.64444955,  1314.03511536,  1465.9725869 ,  1226.61445165,
   979.05434422,   811.77715993,   427.30062761,   264.29365983,
    95.32569513,    60.78415485,    18.0479749 ,   191.12855377,
   -49.65398333,   -70.46330225,  -148.62817552,  -132.20917619,
  -157.15858763,  -242.68266975,  -212.73839569,   29.08819716,
  -342.59674376,  -279.25928841,  -192.54919775,  -160.97844053,
  -202.71852851,  -297.08463799,  -403.11559139,  -762.85029966,
  -579.06885583,  -859.50572712,  -933.79352298, -1241.91520627,
 -2072.27377017, -2365.00508869, -2478.17418384, -3127.6333605 ,
 -3521.62506794, -4364.2424685 , -4867.51788711, -5341.71548212,
 -6186.07748867, -6365.74503603])

```

```

1  #усредненный результат всех моделей, то есть результат случайного леса
2  plt.xlabel('x')
3  plt.ylabel('f(x)')
4  plt.plot(x, mean_pred, c = 'green', zorder = 2)
5  plt.plot(x, y, '--', color = 'black', lw = 1)
6  plt.show()

```

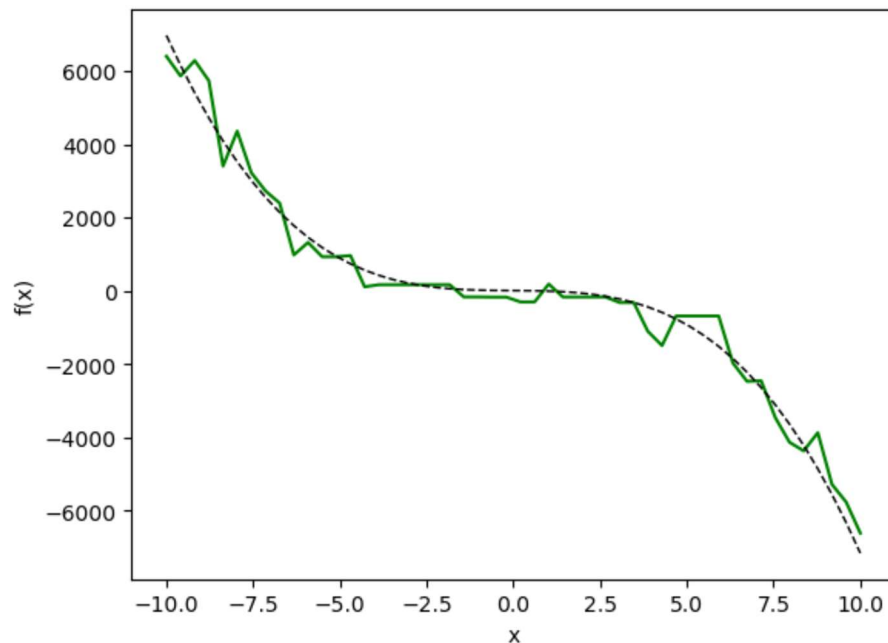


А теперь обучим одно дерево решений на всем наборе данных.

```
1 #обучим одно дерево на всех данных
2 model_tree = tree.DecisionTreeRegressor(max_depth=8, random_state=1)
3 one_model = model_tree.fit(np.array(x_datasets).reshape(-1,1), np.array(y_datasets).reshape(-1,1))
```

```
1 one_pred = one_model.predict(x.reshape(-1,1))
```

```
1 plt.xlabel('x')
2 plt.ylabel('f(x)')
3 plt.scatter(x,one_pred, c = 'green', s = 16, zorder = 2)
4 plt.plot(x,y,'--',color = 'black',lw = 1.5)
5 plt.show()
```



Коэффициент детерминации для случайного леса:

```
1 print('R2 для случайного леса', r2_score(mean_pred, f(x)))
```

R2 для случайного леса 0.9930031433318117

Коэффициент детерминации для одного дерева решений:

```
1 print('R2 для одного дерева решений', r2_score(one_pred, f(x)))
```

R2 для одного дерева решений 0.9760275915485984

Видим, что случайный лес дает более точные результаты.

## Boosting

Бустинг, использующий деревья решений в качестве базовых алгоритмов, называется градиентным бустингом над решающими деревьями, Gradient Boosting on Decision Trees, GBDT. Он отлично работает на выборках с неоднородными данными. Например, описание пользователя через его возраст, пол, среднее число поисковых запросов в день, число заказов такси и так далее. Такой бустинг способен эффективно находить нелинейные

зависимости в данных различной природы. Этим свойством обладают все алгоритмы, использующие деревья решений, однако именно GBDT обычно выигрывает в подавляющем большинстве задач.

Можно использовать и другие алгоритмы, например, линейные модели в качестве базовых, эта возможность реализована в XGBoost – первая успешная разработка, но работает достаточно долго, но можно переложить модель на GPU, т.е. ускорить процесс обучения.

Существуют еще алгоритмы градиентного бустинга, например CatBoost – разработка от Яндекс. CatBoost очень хорошо работает с данными, где есть категориальные значения.

Основная суть работы градиентного бустинга:

Для начала у нас есть первое дерево, которое выдает нам некоторый прогноз  $\hat{y}_1(x)$ . Считается разница между реальным значением и результатом модели, то есть ошибка:

$$r_1 = t - \hat{y}_1(x).$$

Мы хотим скорректировать результат первой модели  $\hat{y}_1(x)$  с помощью следующего дерева  $\hat{y}_2(x)$  так, чтобы оно идеально предсказывало  $r_1$ . Тогда ансамбль моделей  $a$  выглядит следующим образом:

$$a_2 = \hat{y}_1(x) + \hat{y}_2(x) = \hat{y}_1(x) + r_1 = \hat{y}_1(x) + t - \hat{y}_1(x) = t.$$

То есть второе дерево будет предсказывать ошибку предыдущего дерева. Тогда

$$r_2 = t - \hat{y}_1(x) - \hat{y}_2(x)$$

$$\begin{aligned} a_3 &= \hat{y}_1(x) + \hat{y}_2(x) + \hat{y}_3(x) = \hat{y}_1(x) + \hat{y}_2(x) + r_2 \\ &= \hat{y}_1(x) + \hat{y}_2(x) + t - \hat{y}_1(x) - \hat{y}_2(x) = t \end{aligned}$$

и т.д.

Бустинг называется градиентным, так как для каждого объекта в наборе данных каждое следующее дерево обучается предсказывать антиградиент функции стоимости по предсказанию предыдущей модели.

Рассмотрим пример работы баггинга и бустинга для задачи классификации на примере данных лесного покрова. Стоит задача прогнозирования типа лесного покрова по картографическим переменным (без данных дистанционного зондирования). Набор данных взят из библиотеки sklearn.

```
!pip3 install catboost
```



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from sklearn.metrics import f1_score
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.datasets import fetch_covtype
from sklearn.ensemble import RandomForestClassifier
import catboost as cb
```

```
data = fetch_covtype()
```

```
predicts = data.data
target = data.target
```

```
print(predicts)
```

```
[[2.596e+03 5.100e+01 3.000e+00 ... 0.000e+00 0.000e+00 0.000e+00]
 [2.590e+03 5.600e+01 2.000e+00 ... 0.000e+00 0.000e+00 0.000e+00]
 [2.804e+03 1.390e+02 9.000e+00 ... 0.000e+00 0.000e+00 0.000e+00]
 ...
 [2.386e+03 1.590e+02 1.700e+01 ... 0.000e+00 0.000e+00 0.000e+00]
 [2.384e+03 1.700e+02 1.500e+01 ... 0.000e+00 0.000e+00 0.000e+00]
 [2.383e+03 1.650e+02 1.300e+01 ... 0.000e+00 0.000e+00 0.000e+00]]
```

```
print(target)
```

```
[5 5 2 ... 3 3 3]
```

```
print(predicts.shape)
```

```
(581012, 54)
```

Разделим исходный датасет на обучающую и тестовую выборки с помощью `train_test_split()`.

```
A_train, A_test, y_train, y_test = train_test_split(predicts, target, train_size = 0.8)
```

```
print(A_train.shape)
print(A_test.shape)
```

```
(464809, 54)
```

```
(116203, 54)
```

Обучим случайный лес. По умолчанию у случайного леса не установлены критерии останова. Если их не установить, то случайный лес переобучится, то есть просто подстроится под исходные данные, а на других данных будет давать точность прогноза ниже.



```
random_forest = RandomForestClassifier(max_depth=15,min_samples_split=10).fit(A_train, y_train)
```

```
y_preds_d = random_forest.predict(A_train)
print('F1 мера для тренировочных данных',f1_score(y_preds_d,y_train,average='macro'))
```

F1 мера для тренировочных данных 0.7733409137057139

```
y_pred = random_forest.predict(A_test)
```

```
print('F1 мера для тестовых данных',f1_score(y_pred,y_test,average = "macro"))
```

F1 мера для тестовых данных 0.7429112778012742

Проблема заключается в том, что эти параметры нужно каким-то образом выбрать. Можно провести перебор параметров и для каждой их комбинации обучить модели, затем выбрать лучшую. Это можно сделать с помощью GridSearchCV.

```
random_forest = RandomForestClassifier()

params_grid = {
    "max_depth": [12, 18],
    "min_samples_leaf": [3, 10],
    "min_samples_split": [6, 12], #минимум примеров с вершине, при котором можно продолжить деление
}
```

```
grid_search_random_forest = GridSearchCV(estimator=random_forest,
                                          param_grid=params_grid,
                                          scoring="f1_macro",
                                          cv = 4)
```

```
grid_search_random_forest.fit(A_train, y_train)
```

```
GridSearchCV(cv=4, estimator=RandomForestClassifier(n_jobs=-1),
             param_grid={'max_depth': [12, 18], 'min_samples_leaf': [3, 10],
                          'min_samples_split': [6, 12]},
             scoring='f1_macro')
```

```
best_model = grid_search_random_forest.best_estimator_
```

```
y_preds_d = best_model.predict(A_train)
print('F1 мера для тренировочных данных',f1_score(y_preds_d,y_train,average='macro'))
```

F1 мера для тренировочных данных 0.8332805258046768

```
y_pred = best_model.predict(A_test)
```

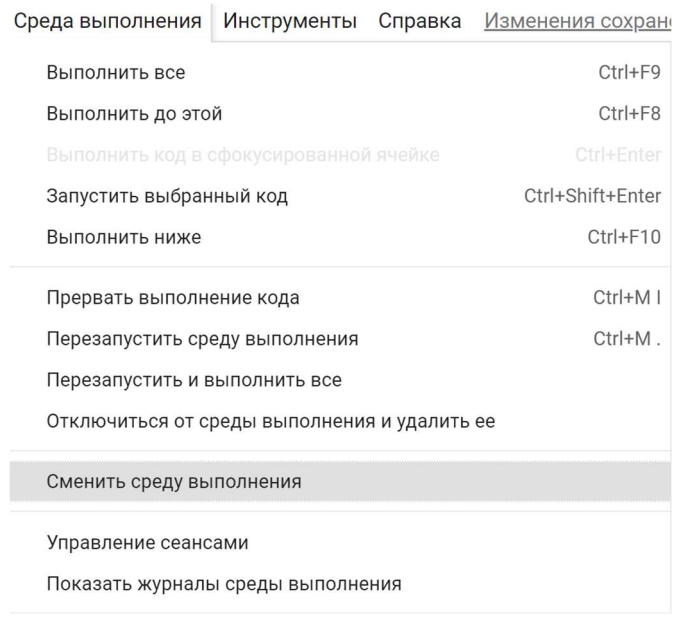
```
print('F1 мера для тестовых данных',f1_score(y_pred,y_test,average = "macro"))
```

F1 мера для тестовых данных 0.793147709970959

Результат стал немного лучше.


Перейдем к бустингу.

Необходимо перевести данные на GPU. Это в разы ускорит процесс обучения модели. Необходимо перевести среду Colab на GPU.



## Настройки блокнота

### Аппаратный ускоритель

GPU   
None  
GPU  
TPU

активно использовать Colab, ускоритель (GPU), если он вам не нужен. [Подробнее...](#)

☐ Исключить выходные данные кодовой ячейки при сохранении блокнота

Отмена Сохранить

Если этого не сделать, то процесс обучения модели на выбранных данных займет около 18 минут.

```
model_catboost_clf = cb.CatBoostClassifier(iterations=3000,  
                                           task_type="GPU",  
                                           devices='0')  
model_catboost_clf.fit(A_train, y_train)
```

Learning rate set to 0.060254

0:	learn: 1.7788133	total: 561ms	remaining: 18m 41s
1:	learn: 1.6529444	total: 1s	remaining: 16m 43s
2:	learn: 1.5507283	total: 1.42s	remaining: 15m 48s
3:	learn: 1.4660548	total: 1.85s	remaining: 15m 22s
4:	learn: 1.3942799	total: 2.27s	remaining: 15m 6s

Время обучения на графическом процессоре полторы минуты.

0:	learn: 1.7492704	total: 22.6ms	remaining: 1m 30s
1:	learn: 1.6076562	total: 43.2ms	remaining: 1m 26s
2:	learn: 1.4959345	total: 64.4ms	remaining: 1m 25s
3:	learn: 1.4061106	total: 85.9ms	remaining: 1m 25s
4:	learn: 1.3307801	total: 107ms	remaining: 1m 25s
5:	learn: 1.2673781	total: 128ms	remaining: 1m 25s

```
y_preds_t = model_catboost_clf.predict(A_train,task_type="CPU")  
print('F1 мера для тренировочных данных',f1_score(y_preds_t,y_train,average='macro'))
```

F1 мера для тренировочных данных 0.921544189245645

```
y_preds = model_catboost_clf.predict(A_test,task_type="CPU")  
print('F1 мера для тестовых данных',f1_score(y_preds,y_test,average='macro'))
```

F1 мера для тестовых данных 0.9202616383980983

Для реализации практической работы необходимо использовать ColabNotebook.

### **Практическое задание**

- 1) Найти данные для задачи классификации или для задачи регрессии (данные не должны повторяться в группе).
- 2) Реализовать баггинг.
- 3) Реализовать бустинг на тех же данных, что использовались для баггинга.
- 4) Сравнить результаты работы алгоритмов (время работы и качество моделей). Сделать выводы.
- 5) Оформить отчет о проделанной работе.