Boosting

Всем ку)Сегодня мы поговорим о методе, который называется *Ансамблем алгоритмов*. Этот метод использует одновременно несколько обучающих алгоритмов для получения лучшего результата по сравнению с результатами каждого по отдельности. Состоит этот ансамбль из так называемых **базовых алгоритмов** (base learners), и есть еще алгоритм, который комбинирует полученные ответы, - **мета-алгоритм** (meta-estimator).

При использовании такого комбинирования у нас:

- повышается качество базовых алгоритмов
- повышается разнообразие за счет:
- - варьирования признаков, моделей, выборок, и самого варьирования
  Одним из популярных методов ансамбля является *Boosting*

Boosting (Бустинг)

Главная идея бустинга - последовательное построение базовых алгоритмов, каждый из которых обучается, используя ошибки предыдущего. То есть, каждый алгоритм должен избавляться от ошибок предыдущего.

Первый успешный вариант такого бустинга - AdaBoost, или же Adaptive Boosting. Основная идея же здесь - каждый следующий алгоритм строится на изменении весов объектов. Больший вес будет иметь тот объект, который классифицировался неверно.

Подробно о нем мы говорить не станем, потому что придумали **Градиентный бустинг**, который полностью вытеснил другие методы.

Градиентный бустинг

Допустим, что у нас ест выборка, состоящая из признаков x и переменных y (целевых). Идея такаого бустинга заключается в том, чтобы уменьшить ошибку предсказания классификатора, т.е. уменьшить разность $(y - \hat{y})$, чтобы она стремилась к нулю, используя нахождение методом градиентного спуска. Там еще используется функция потерь $L_{y,\hat{y}}$

Функции Потерь

Давайте теперь разберемся и выпишем, какие же функции потерь стоит использовать в задаче классификации (мы пока еще о ней говорим). Наиболее известными являются:

- Logistic loss или логистическая функция потерь: $L(y, \hat{y}) = log(1 + exp(-y\hat{y}))$ - самая используемая функция потерь в бинарной классификации;
- AdaBoost loss: $L(y, \hat{y}) = exp(-y\hat{y})$. Здесь нужно вспомнить про алгоритм AdaBoost: так получилось, что он эквивалентен методу градиентного бустинга с этой функцией потерь. Эта функция имеет более жесткий штраф по отношению к ошибкам классификации и используется реже.

В качестве же базовых алгоритмов используются решающие деревья, причем пишут крч, что лучше использовать неглубокие деревья: от 2 до 7 глубины.

XGBoost

Еще одна эффективная вариация градиентного бустинга, основанного на деревьях решений, это XGBoost (eXtreme Gradient Boosting). Здесь различия в том, что есть больше возможностей для регуляризации базовых решающих деревьев и задачи бустинга в целом. Однако это отдельная библиотека, найти и скачать ее в инете, прописав потом одну строчку в терминале очень просто, поэтому писать про это долго не будут. Параметров крч ооочень много, они разделяются на:

- общие параметры, отвечающие за выбор базового алгоритма и распараллеливание;
- параметры базового алгоритма решающих деревьев: скорость обучения (чем ниже, тем больше итераций нужно для обучения модели с хорошим качеством), максимальная глубина дерева, минимальное необходимое число объектов в каждом листе, доля выборки для обучения каждого дерева, доля признаков для каждого дерева, коэффициенты перед L1- и L2-регуляризаторами функции потерь;
- параметры задачи обучения: используемая при обучении функция потерь, метрика качества на валидационной выборке, параметр воспроизводимости.

Практика

Поговорили много, увидели мало, так что давайте посмотрим на всё своими глазами. Библиотека sklearn предоставляет нам готовую реализацию алгоритмов AdaBoost и GradientBoosting, а вот XGBoost придется импортировать из другой библиотеки.

```
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import cross_val_score
import pandas as pd
import numpy as np
```

Давайте дальше посмотрим, как эти алгоритмы будут меняться в зависимости от количества деревьев, на которых он будет построен.

Для демонстрации используем данные о выживших людях с Титаника

```
titanic = pd.read_csv('9.7_titanic.csv')
titanic.head(10)
```

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	1	22.0	1	0	7.2500	1
1	1	1	0	38.0	1	0	71.2833	2
2	1	3	0	26.0	0	0	7.9250	1
3	1	1	0	35.0	1	0	53.1000	1
4	0	3	1	35.0	0	0	8.0500	1
5	0	3	1	29.0	0	0	8.4583	3
6	0	1	1	54.0	0	0	51.8625	1
7	0	3	1	2.0	3	1	21.0750	1
8	1	3	0	27.0	0	2	11.1333	1
9	1	2	0	14.0	1	0	30.0708	2

- Survived - выжил или нет (1 или 0)
- Pclass - соц. экономический статус, где 1=высокий, 3=низкий, 2=средний
- Sex - занимался ли человек... шучу, просто пол, 1=мужик
- SibSP = количество родственников 2го порядка (муж, жена, братья, сестры)
- Parch = кол-во родственников на борту 1го порядка (мать, отец, дети)
- Fare = цена билета
- Embarked - Порт посадки. Ненужная информация на самом деле, потому что и так и так они уже на титанике, неважно где вы сели на него.

Давайте отделим отдельный столбец Survived. И не будем делить данные на тренинг тестинг группы.

```
targets = titanic.Survived # запись в переменную отдельного столбца
```

```
data=titanic.drop(columns="Survived") # запишем в другую переменную оставшиеся столбцы данных
print(data)
```

```
     Pclass  Sex   Age  SibSp  Parch     Fare  Embarked
0         3    1  22.0      1      0   7.2500         1
1         1    0  38.0      1      0  71.2833         2
2         3    0  26.0      0      0   7.9250         1
3         1    0  35.0      1      0  53.1000         1
4         3    1  35.0      0      0   8.0500         1
..      ...  ...   ...    ...    ...      ...       ...
886       2    1  27.0      0      0  13.0000         1
887       1    0  19.0      0      0  30.0000         1
888       3    0  27.0      1      2  23.4500         1
889       1    1  26.0      0      0  30.0000         2
890       3    1  32.0      0      0   7.7500         3

[891 rows x 7 columns]
```

Зададим количество деревьев в алгоритме: 1, а далее от 10 до 100 с шагом 10. В алгоритме за этот гиперпараметр отвечает n_estimtors.

Теперь последовательно обучим алгоритмы AdaBoost, GradientBoosting и XGBoost и проверим на трехкратной кросс-валидации, как изменяется точность классификаторов в зависимости от количества базовых алгоритмов. Заодно посмотрим на время обучения каждого из алгоритмов. (Для глупеньких - cross_val_score дает на выходе значения от 0 до 1, где 1 - точность описания модели хороша, 0 - плоха)

```
trees = [1] + list(range(10, 100, 10))
```

```
%%time
# для замера времени

ada_scoring = []
for tree in trees: # в цикле будем проходиться по количеству дерева
    ada = AdaBoostClassifier(n_estimators=tree)
    score = cross_val_score(ada, data, targets, scoring='roc_auc', cv=3)
    ada_scoring.append(score)
ada_scoring = np.asmatrix(ada_scoring)
```

```
CPU times: total: 2.38 s
Wall time: 2.45 s
```

```
ada_scoring
```

```
matrix([[0.77164222, 0.78587863, 0.7430975 ],
        [0.79997124, 0.85557473, 0.86624005],
        [0.80560349, 0.85092513, 0.88447896],
        [0.79925223, 0.84766561, 0.88562937],
        [0.80236794, 0.84502924, 0.87553926],
        [0.80059438, 0.8420094 , 0.87553926],
        [0.80040265, 0.84112262, 0.88050043],
        [0.80136133, 0.84210526, 0.88203432],
        [0.80299108, 0.84385486, 0.87340619],
        [0.8038539 , 0.84450197, 0.87824753]])
```

```
%%time

gbc_scoring = []
for tree in trees:
```

```
    gbc = GradientBoostingClassifier(n_estimators=tree)
    score = cross_val_score(gbc, data, targets, scoring='roc_auc', cv=3)
    gbc_scoring.append(score)
gbc_scoring = np.asmatrix(gbc_scoring)
```

```
CPU times: total: 1.34 s
Wall time: 1.35 s
```

```
gbc_scoring
```

```
matrix([[0.81430352, 0.84991851, 0.87491612],
        [0.81763493, 0.87117726, 0.88677979],
        [0.83158374, 0.87364586, 0.88179465],
        [0.81708369, 0.8781996 , 0.8825616 ],
        [0.81797047, 0.87374173, 0.88186655],
        [0.81456716, 0.87117726, 0.88730707],
        [0.81375228, 0.86949957, 0.88349631],
        [0.80979772, 0.86997891, 0.88126738],
        [0.80982169, 0.87146486, 0.87635414],
        [0.81380021, 0.87182437, 0.87501198]])
```

```
%%time

xgb_scoring = []
for tree in trees:
    xgb = XGBClassifier(n_estimators=tree)
    score = cross_val_score(xgb, data, targets, scoring='roc_auc', cv=3)
    xgb_scoring.append(score)
xgb_scoring = np.asmatrix(xgb_scoring)
```

```
CPU times: total: 2.27 s
Wall time: 1.33 s
```

```
xgb_scoring
```

```
matrix([[0.8108283 , 0.86564088, 0.87714505],
        [0.80967788, 0.89497651, 0.86614419],
        [0.81658039, 0.88625252, 0.86156648],
        [0.8127936 , 0.88191449, 0.8582111 ],
        [0.81324897, 0.87800786, 0.85694085],
        [0.81125971, 0.87865497, 0.85698878],
        [0.81056466, 0.87760042, 0.85646151],
        [0.81032499, 0.87386157, 0.86183012],
        [0.80816796, 0.8758748 , 0.85881028],
        [0.80792829, 0.87395743, 0.85665325]])
```

На первый взгляд алгоритмы работают практически идентично. Теперь посмотрим на графике, так ли это. Обратите внимание на то, что алгоритм XGBoost работает в 2 раза быстрее, чем GradienBoosting, и почти в 4 раза быстрее, чем AdaBoost.

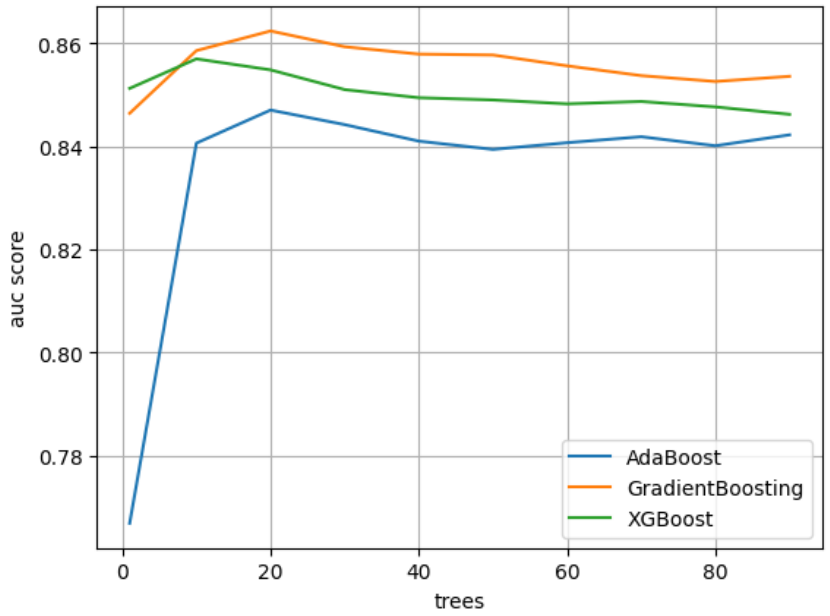
```
import matplotlib.pyplot as plt

%matplotlib inline

plt.plot(trees, ada_scoring.mean(axis=1), label='AdaBoost')
plt.plot(trees, gbc_scoring.mean(axis=1), label='GradientBoosting')
plt.plot(trees, xgb_scoring.mean(axis=1), label='XGBoost')
plt.grid(True) #сетка
```

```
plt.xlabel('trees')
plt.ylabel('auc score')
plt.legend(loc='lower right') # расположение легенды
```

```
<matplotlib.legend.Legend at 0x1addf63f700>
```



Итог

Видим ожидаемую картину: при одном дереве качество работы алгоритма было ниже, чем с увеличением количества деревьев, однако все модели показывают наиболее высокое качество примерно у отметки в 20 базовых деревьев. Вообще, в бустинге увеличение числа деревьев не всегда приводит к улучшению качества решения на тестовых данных. Число деревьев, при котором качество алгоритма максимально, зависит от темпа обучения: чем меньше темп, тем больше деревьев обычно нужно (отметим, что зависимость нелинейная).

Также, как видно по графику, алгоритм XGBoost работает не только быстрее, но и на том же уровне со своими конкурентами. Также, он может быть намного точнее своих конкурентов (здесь не фортануло гыгыгы). Это достигается за счет того, что у него больше преднастроенных гиперпараметров и он лучше оптимизирован. Если вы хотите почитать еще побольше про основы анализа, то идите по этой Ссылке, там можно почитать интересную статью на хабре про все методы что я проходил за последние две недели