Конспект по теме "Несбалансированная классификация"

Взвешивание классов

Алгоритмы машинного обучения считают все объекты обучающей выборки равнозначными по умолчанию. Если важно указать, что какие-то объекты важнее, их классу присваивается **вес**. В алгоритмах логистической регрессии, решающего дерева и случайного леса в библиотеке *sklearn* есть аргумент *class_weight*. По умолчанию он равен `None`, т. е. классы равнозначны:

```
вес класса «0» = 1.0
вес класса «1» = 1.0
```

Если указать `class_weight='balanced'`, алгоритм посчитает, во сколько раз класс «0» встречается чаще класса «1». Обозначим это число *N* (неизвестное количество раз). Новые веса классов выглядят так:

```
вес класса «0» = 1.0
вес класса «1» = N
```

Бóльший вес будет у редкого класса.

Увеличение выборки

Когда обучают модели, чтобы сбалансировать классы путём увеличения числа, такая техника называется **upsampling**.

Преобразование проходит в несколько этапов:

- Разделить обучающую выборку на объекты по классам;
- Определить тот класс, который содержит меньше объектов. Назовём его меньшим классом;
- Скопировать несколько раз объекты меньшего класса;
- С учётом полученных данных создать новую обучающую выборку;
- Перемешать данные.

Скопировать объекты несколько раз поможет синтаксис умножения списков в Python. Чтобы повторить элементы списка, он умножается на число (нужное количество раз):

```
answers = [0, 1, 0]
print(answers)
answers_x3 = answers * 3
print(answers_x3)
```

```
[0, 1, 0]
[0, 1, 0, 0, 1, 0, 0, 1, 0]
```

Чтобы соединить таблицы, используется функция **pd.concat()**. Она на вход получает список таблиц, которые нужно соединить. Исходные данные можно получить так:

```
pd.concat([table1, table2])
```

Чтобы перемешать объекты случайным образом, в библиотеке *sklearn.utils* есть метод *shuffle*:

```
features, target = shuffle(features, target, random_state=12345)
```

Функция *upsample()*:

```
def upsample(features, target, repeat):
    features_zeros = features[target == 0]
    features_ones = features[target == 1]
    target_zeros = target[target == 0]
    target_ones = target[target == 1]

    features_upsampled = pd.concat([features_zeros] + [features_ones] * repeat)
    target_upsampled = pd.concat([target_zeros] + [target_ones] * repeat)

    features_upsampled, target_upsampled = shuffle(
        features_upsampled, target_upsampled, random_state=12345)

    return features_upsampled, target_upsampled
```

Уменьшение выборки

Вместо повторения объектов меньшего класса, уберём часть обектов большего класса. Это можно сделать техникой **downsampling**.

Преобразование проходит в несколько этапов:

- Разделить обучающую выборку на объекты по классам;
- Определить тот класс, который содержит больше объектов. Назовём его большим классом;
- Случайным образом отбросить часть из объектов большего класса;
- С учётом полученных данных создать новую обучающую выборку;
- Перемешать данные.

Чтобы выбросить из таблицы случайные элементы, примените функцию *sample()*. На вход она принимает аргумент *frac*. Возвращает случайные элементы в таком количестве, чтобы их доля от исходной таблицы была равна *frac*.

```
features_sample = features_train.sample(frac=0.1, random_state=12345)
```

Функция *downsample()*:

```
def downsample(features, target, fraction):
    features_zeros = features[target == 0]
    features_ones = features[target == 1]
    target_zeros = target[target == 0]
    target_ones = target[target == 1]

    features_downsampled = pd.concat(
        [features_zeros.sample(frac=fraction, random_state=12345)] + [features_ones])
    target_downsampled = pd.concat(
        [target_zeros.sample(frac=fraction, random_state=12345)] + [target_ones])

    features_downsampled, target_downsampled = shuffle(
        features_downsampled, target_downsampled, random_state=12345)

    return features_downsampled, target_downsampled
```

Порог классификации

Чтобы определить ответ, логистическая регрессия вычисляет, к какому классу близок объект, затем сравнивает результат с нулём. Для удобства

близость к классам переведём в **вероятность классов**: модель пытается оценить, насколько вероятен тот или иной класс. У нас всего два класса (ноль и единица). Нам достаточно вероятности класса «1». Число будет от нуля до единицы: если больше 0.5 — объект положительный, меньше — отрицательный.

Граница, где заканчивается отрицательный класс и начинается положительный, называется **порогом** (*threshold*). По умолчанию он равен 0.5, но его можно поменять.

Изменение порога

В библиотеке *sklearn* вероятность классов вычисляет функция **predict_proba()**. На вход она получает признаки объектов, а возвращает вероятности:

```
probabilities = model.predict_proba(features)
```

Строки соответствуют объектам. В первом столбце указана вероятность отрицательного класса, а во втором — положительного (сумма вероятностей равна единице).

Чтобы создать цикл с нужным диапазоном, мы применили функцию *arange()* библиотеки *numpy*. Как и *range()*, функция перебирает указанные элементы диапазона, но работает не только с целыми, но и дробными числами:

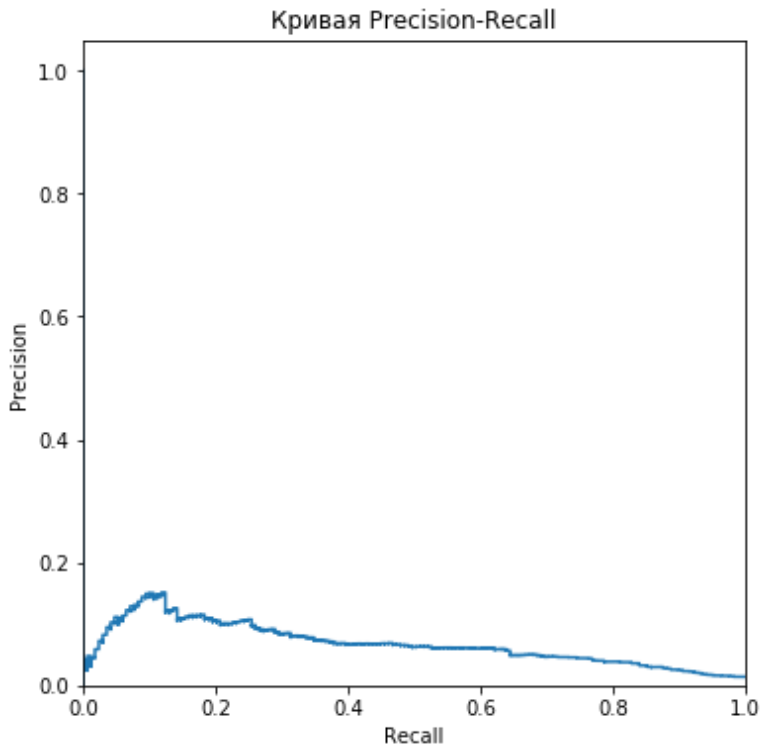
```
for value in np.arange(first, last, step):
```

PR-кривая

Изобразим на графике, как выглядят значения метрик при изменении порога.

На графике по вертикали наносится значение точности, по горизонтали — полноты. Кривая, показывающая их значения, называется **PR-кривой**

(***P**recision* и ***R**ecall*). Чем выше кривая, тем лучше модель.



TPR и FPR

Когда положительных объектов нет, точность не вычислить. Выберем другие характеристики, в которых нет деления на ноль.

Прежде чем перейти к новой кривой, дадим несколько важных определений.

Чем измеряется отношение *TP*-ответов ко всем положительным ответам? Долей истинно положительных ответов (*True Positive Rate, TPR)*. Формула выглядит так (*P* — все положительные ответы):

$$TPR = \frac{TP}{P}$$

Доля ложноположительных ответов (*False Positive Rate*, *FPR*) вычисляется аналогично. Это отношение *FP*-ответов ко всем отрицательным ответам *(N)*:

$$FPR = \frac{FP}{N}$$

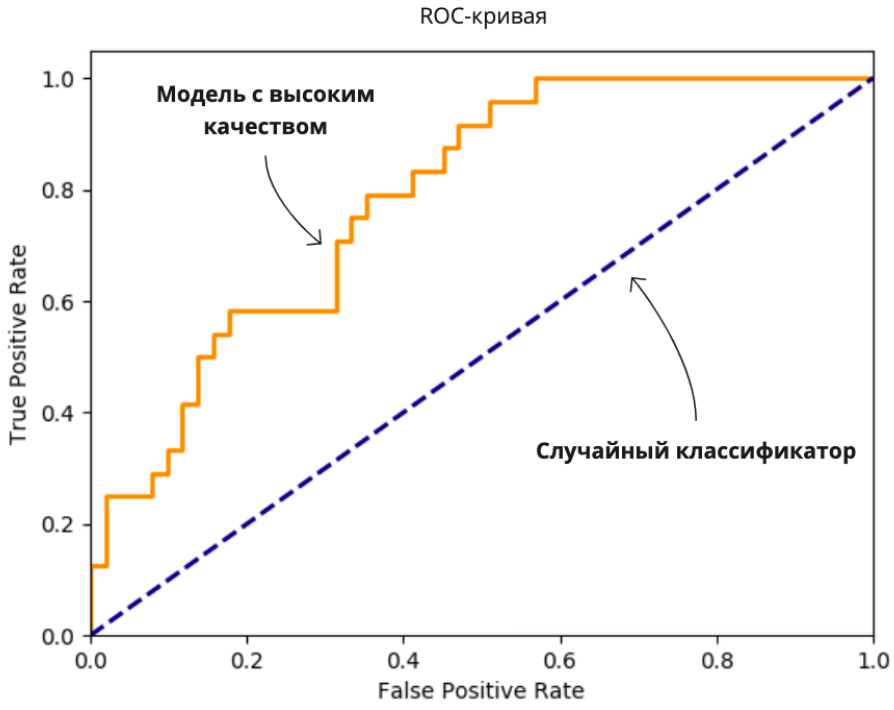
Деления на ноль не будет: в знаменателях значения, которые постоянны и не зависят от изменения модели.

ROC-кривая

Мы стали свидетелями нового противостояния — *TPR* против *FPR*. Изобразим его на графике.

По горизонтали нанесём долю ложноположительных ответов (*FPR)*, а по вертикали — долю истинно положительных ответов (*TPR*). Переберём значения порога логистической регрессии и проведём кривую. Она называется **ROC-кривая**, или **кривая ошибок**.

Для модели, которая всегда отвечает случайно, *ROC*-кривая выглядит как прямая, идущая из левого нижнего угла в верхний правый. Чем график выше, тем больше значение *TPR* и лучше качество модели.



Чтобы выявить, как сильно наша модель отличается от случайной, посчитаем площадь под *ROC*-кривой — **AUC-ROC** (*Area Under Curve ROC*). Это новая метрика качества, которая изменяется от 0 до 1. *AUC-ROC* случайной модели равна 0.5.

Построить *ROC*-кривую поможет функция *roc_curve()* из модуля *sklearn.metrics:*

```
from sklearn.metrics import roc_curve
```

На вход она принимает значения целевого признака и вероятности положительного класса. Перебирает разные пороги и возвращает три списка: значения *FPR*, значения *TPR* и рассмотренные пороги.

```
 fpr, tpr, thresholds = roc_curve(target, probabilities)
```

Для подсчёта метрики AUC-ROC, в библиотеке *sklearn* есть функция *roc_auc_score()*:

```
from sklearn.metrics import roc_auc_score
```

В отличие от других метрик, на вход она принимает не предсказания, а вероятности класса «1»:

```
auc_roc = roc_auc_score(target_valid, probabilities_one_valid)
```