Дисбаланс классов

Дисбаланс классов (англ. class imbalance) — это непропорциональное соотношение классов в данных, когда наблюдений каких-то классов значительно больше остальных.

Мажорный класс — класс, представленный значительно бóльшим числом наблюдений по сравнению с остальными.

Минорный класс — класс или один из классов, представленный значительно меньшим числом наблюдений в сравнении с мажорным.

Модели, обученные на несбалансированных данных, склонны гораздо лучше предсказывать мажорный класс и игнорировать остальные.

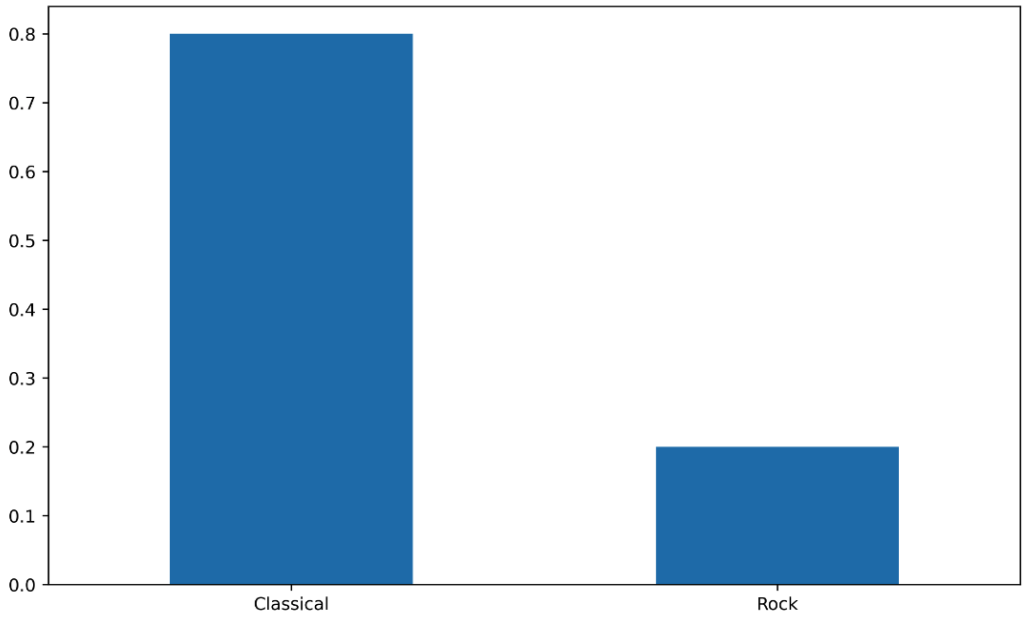
Вес класса — параметр, который определяет важность класса для модели.

Взвешивание класса — процесс настройки весов разных классов в зависимости от их доли в данных.

Порог классификации — вероятность, достаточная для отнесения объектов к старшему классу.

Обнаружение дисбаланса классов

Дисбаланс нужно выявлять во время исследовательского анализа данных.



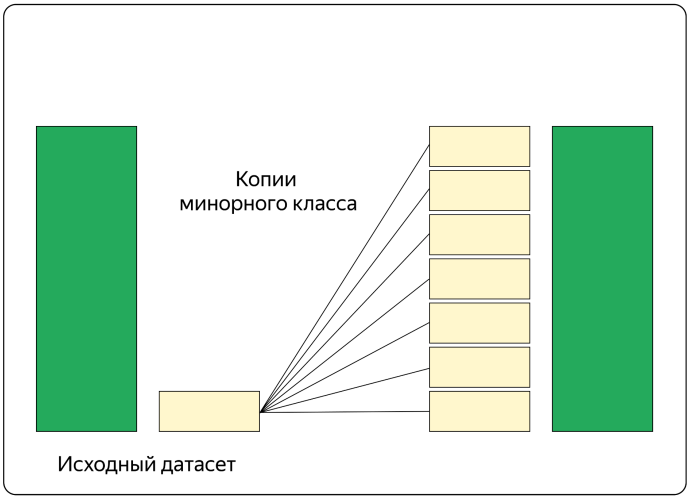
Чтобы посмотреть, как дисбаланс классов влияет на работу модели, можно сравнить её с дамми-моделью (DummyClassifier), которая всегда присваивает наблюдениям мажорный класс.

Способы борьбы с дисбалансом классов

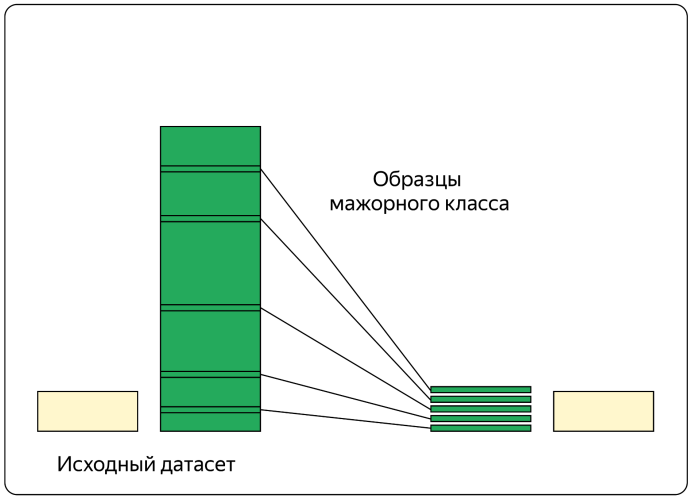
Oversampling, upsampling, оверсэмплинг — это искусственное увеличение количества объектов минорного класса в датасете до количества объектов мажорного класса.

Undersampling, downsampling, андерсэмплинг — это искусственное уменьшение количества объектов мажорного класса в датасете до количества объектов минорного класса.

Oversampling



Undersampling



- **Оверсэмплинг**

```
from imblearn.over_sampling import RandomOverSampler

# экземпляр сэмплера
sampler = RandomOverSampler(random_state=42)

# сэмплируем данные
X_resample, y_resample = sampler.fit_resample(X, y)
```

Дисбаланс классов

- **SMOTE** (англ. Synthetic Minority Oversampling Technique — «Техника синтетического оверсэмплинга минорного класса»)

```
from imblearn.over_sampling import SMOTE

# экземпляр сэмплера, можно выбрать количество ближайших соседей
sampler = SMOTE(random_state=42, k_neighbors=5)

# сэмплируем данные, в X должны быть только количественные признаки
X_train_resample, y_train_resample = sampler.fit_resample(X_train, y_train)
```

- **ADASYN** (англ. Adaptive Synthetic — «Адаптивные синтетические наблюдения»)

```
from imblearn.over_sampling import ADASYN

# экземпляр сэмплера
sampler = ADASYN(random_state=RANDOM_STATE)

# сэмплируем данные
X_resample, y_resample = sampler.fit_resample(X, y)
```

- **Undersampling**

```
from imblearn.over_sampling import RandomUnderSampler

# экземпляр сэмплера
sampler = RandomUnderSampler(random_state=42)

# сэмплируем данные
X_resample, y_resample = sampler.fit_resample(X, y)
```

- **SMOTETomek**

```
from imblearn.over_sampling import SMOTETomek

# экземпляр сэмплера
sampler = SMOTETomek(random_state=42)

# сэмплируем данные
X_resample, y_resample = sampler.fit_resample(X, y)
```

Кросс-валидация

Валидация — это проверка качества работы моделей и их настройка.

Валидационная выборка — набор объектов, на котором проверяют качество решения модели после обучения и настраивают её.

Кросс-валидация — это проверка качества модели на всех доступных данных, кроме отложенной тестовой выборки.

- KFold формирует блоки без учёта стратификации по целевому признаку:

```
from sklearn.model_selection import KFold
import numpy as np

# создание данных
X = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

kfold = KFold(
    n_splits=5, # задаёт количество блоков, на которые будут разделены данные
    shuffle=True, # определяет, перемешивать ли блоки перед разбиением данных
    random_state=42 # фиксирование случайности
)

for train_split, test_split in kfold.split(X):
    print(f"Тренировочные: {X[train_split]}, Валидационные: {X[test_split]}")
```

Дисбаланс классов


```
Тренировочные: [0 2 3 4 5 6 7 9], Валидационные: [1 8]
Тренировочные: [1 2 3 4 6 7 8 9], Валидационные: [0 5]
Тренировочные: [0 1 3 4 5 6 8 9], Валидационные: [2 7]
Тренировочные: [0 1 2 3 5 6 7 8], Валидационные: [4 9]
Тренировочные: [0 1 2 4 5 7 8 9], Валидационные: [3 6]
```

- StratifiedKFold формирует блоки с учётом стратификации по целевому признаку:

```python
from sklearn.model_selection import StratifiedKFold
import numpy as np

# загрузка данных
X = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

s_kfold = StratifiedKFold(
    n_splits=5, # задаёт количество блоков, на которые будут разделены данные
    shuffle=True, # определяет, перемешивать ли блоки перед разбиением данных
    random_state=42 # фиксирование случайности
)

# передаём y в s_kfold.split для стратификации
for train_split, test_split in s_kfold.split(X, y):
    print(f"Тренировочные: {X[train_split]}, Валидационные: {X[test_split]}")
```

- cross_val_score() — функция для кросс-валидации:

```
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier()

cross_val_score(
    model, # модель для кросс-валидации
    X_train, # входные признаки
    y_train, # целевой признак
    cv=5, # число блоков для кросс-валидации, по умолчанию их пять
    scoring='accuracy' # метрика на валидационном блоке данных
)
```

Создание своей метрики

Базовых метрик может не хватить для всех рабочих ситуаций, тогда придётся создавать собственные пользовательские метрики.

Ниже написан код для пользовательской метрики, которая будет рассчитывать F1-меру с учётом весов каждого класса:

```
# задаём веса: вес класса 0 равен 1, класса 1 – 10
class_weights = {0: 1, 1: 10}

# создаём функцию для оценки качества модели
def custom_metric(y_true, y_pred, class_weights):
    # вычисляем F1-score с учётом весов классов
    metric = f1_score(y_true, y_pred, sample_weight=[class_weights[y] for y in y
_true])
    return metric

# создаём пользовательскую метрику
scorer = make_scorer(
    custom_metric,
    greater_is_better=True, # чем модель качественнее, тем выше метрика
    class_weights=class_weights # прочие настройки
)
```

Дисбаланс классов

Ниже написан код для пользовательской метрики, которая будет рассчитывать FPR:

```
from sklearn.metrics import make_scorer, confusion_matrix

# создаём функцию для подсчёта метрики FPR
def custom_metric(y_true, y_pred):
    conf_matrix = confusion_matrix(y_true, y_pred)
    return conf_matrix[0][1] / (conf_matrix[0][1] + conf_matrix[0][0])

# создаём пользовательскую метрику
scorer = make_scorer(
    custom_metric,
    greater_is_better=False,  # чем модель качественнее, тем ниже метрика
)
```