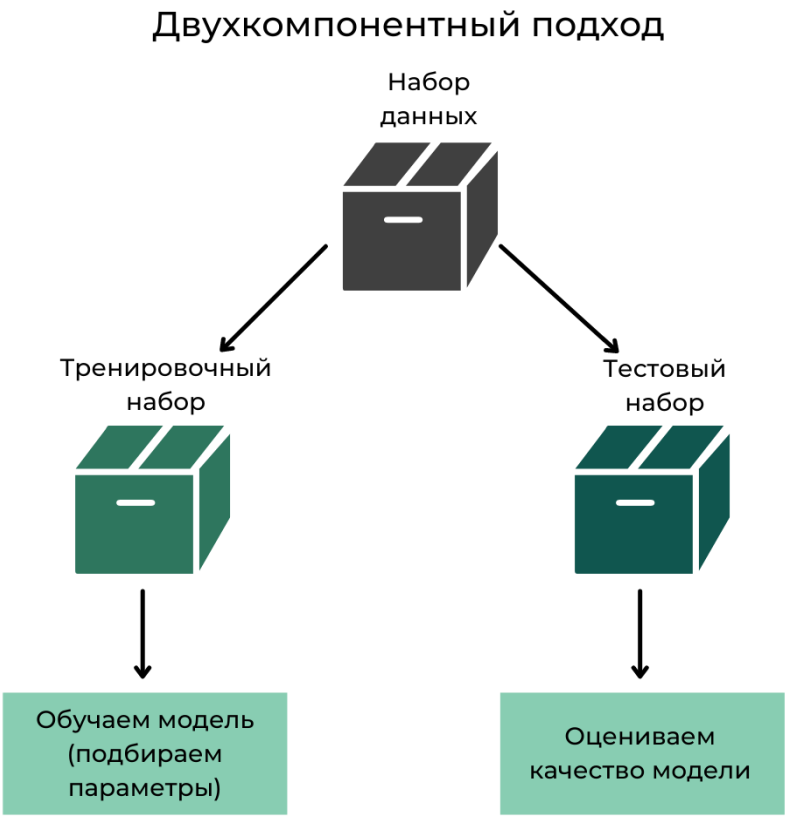
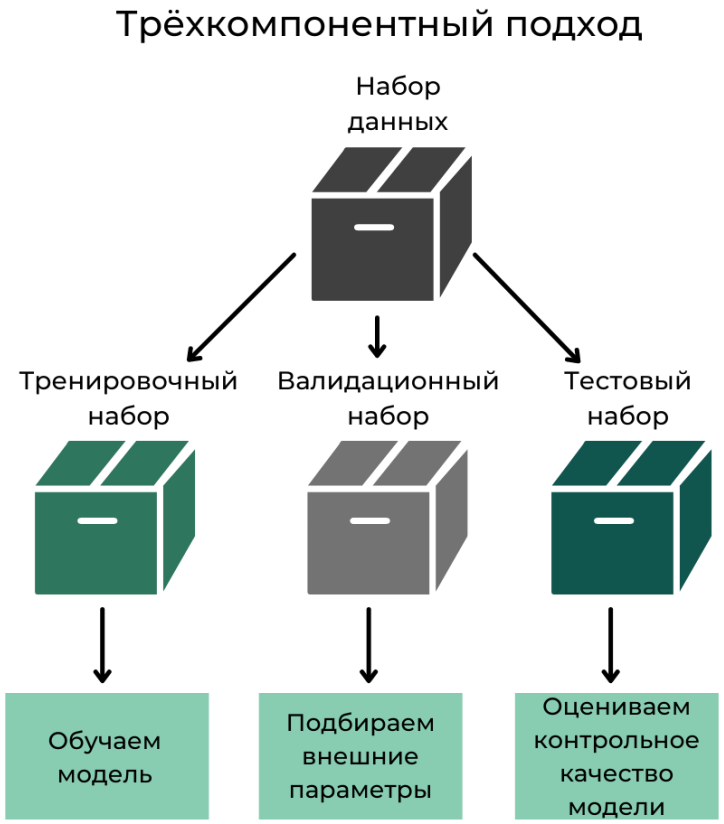
## Валидация данных

**Почему нельзя обучать модель на всех доступных данных?**
Главная цель машинного обучения — усвоить общие закономерности в данных, а не просто запомнить **обучающий (тренировочный) набор данных (training data)**.

Поэтому нам так важно иметь отложенный набор данных (с известными правильными ответами), который модель ещё не видела во время обучения. На нём мы будем оценивать качество обученной модели.

### Подходы к организации выборок:






**Основные виды выборок**, которые используются в машинном обучении:

- **Обучающая (тренировочная)** — набор данных, который используется в процессе обучения модели (подбора внутренних параметров).
- **Валидационная (проверочная)** — набор данных, на котором мы отслеживаем промежуточные результаты.
  Основная цель создания такого набора данных — отслеживание переобучения. На валидационной выборке мы производим подбор гиперпараметров — внешних параметров модели.
- **Тестовая (контрольная)** — набор данных, который имитирует работу модели в реальных условиях после подбора всех параметров. С помощью этого набора осуществляется контрольная проверка качества.

## Методы валидации

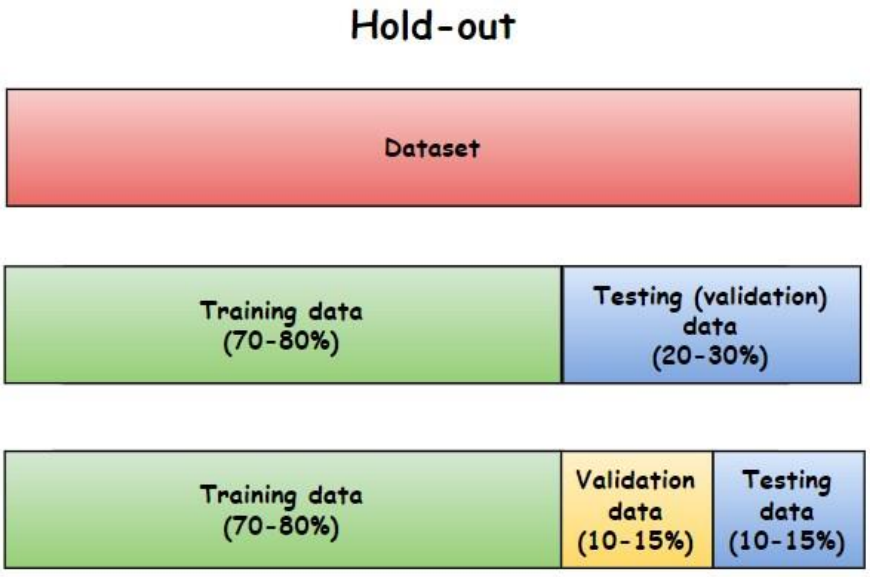
Существует множество методов (схем) проведения валидации. Наиболее распространённые из них:

- **hold-out**,
- **k-fold**,
- **leave-one-out**.

### Hold-out

Его идея в том, что для произведения проверки модели мы просто случайным образом разбиваем весь набор данных на обучающую, валидационную и тестовую выборки (последняя — по желанию).

Обычно разбиение производится в соотношении 70/30 или 80/20 при двухкомпонентном подходе, и в соотношении 70/15/15 или 80/10/10 — при трёхкомпонентном.




**Реализация hold-out в sklearn**

Для двухкомпонентного подхода:

```
X_train, X_valid, y_train, y_valid = model_selection.train_test_split(X, y, test_size=0.2, random_state=42)
```

Для трёхкомпонентного подхода:

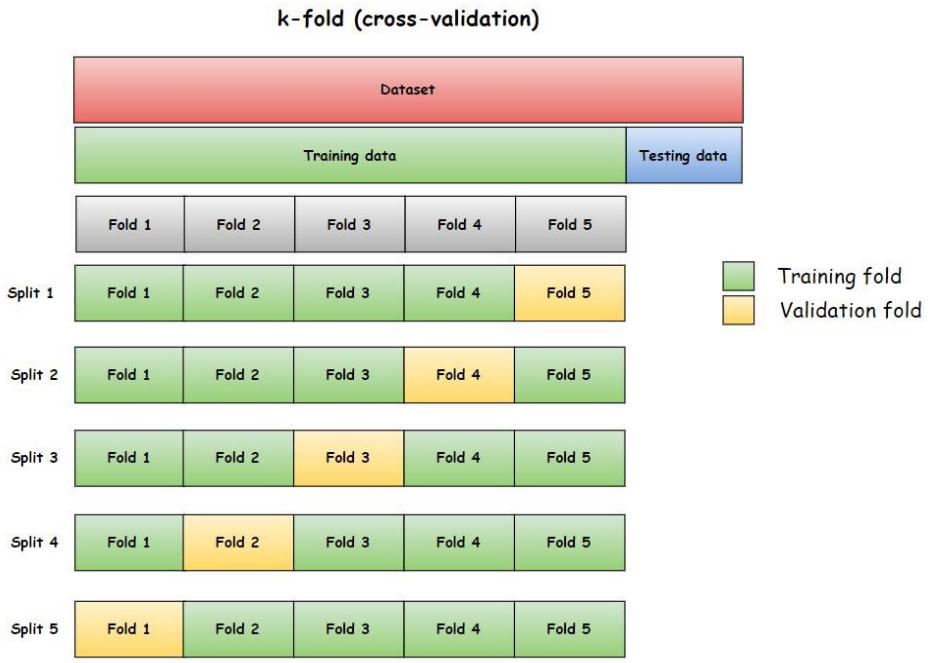
```
X_train, X_valid, y_train, y_valid = model_selection.train_test_split(X, y, test_size=0.2, random_state=42)

X_valid, X_test, y_valid, y_test = model_selection.train_test_split(X_valid, y_valid, test_size=0.5, random_state=42)
```

### K-Fold

Метод k-fold более известен как **кросс-валидация (cross validation)**, или **перекрёстный контроль**.

**Алгоритм кросс-валидации:**

1. Разбить исходную выборку на k частей — **фолдов (fold)**.
2. Повторять k раз:
    a. Обучить модель на k-1 частях — назовём их **тренировочными фолдами (training fold)**.
    b. Произвести оценку качества (вычислить метрику) на оставшейся части — назовём её **валидационным фолдом (validation fold)**.
3. Усреднить значения метрики на валидационных фолдах.




**Реализация k-fold в sklearn**

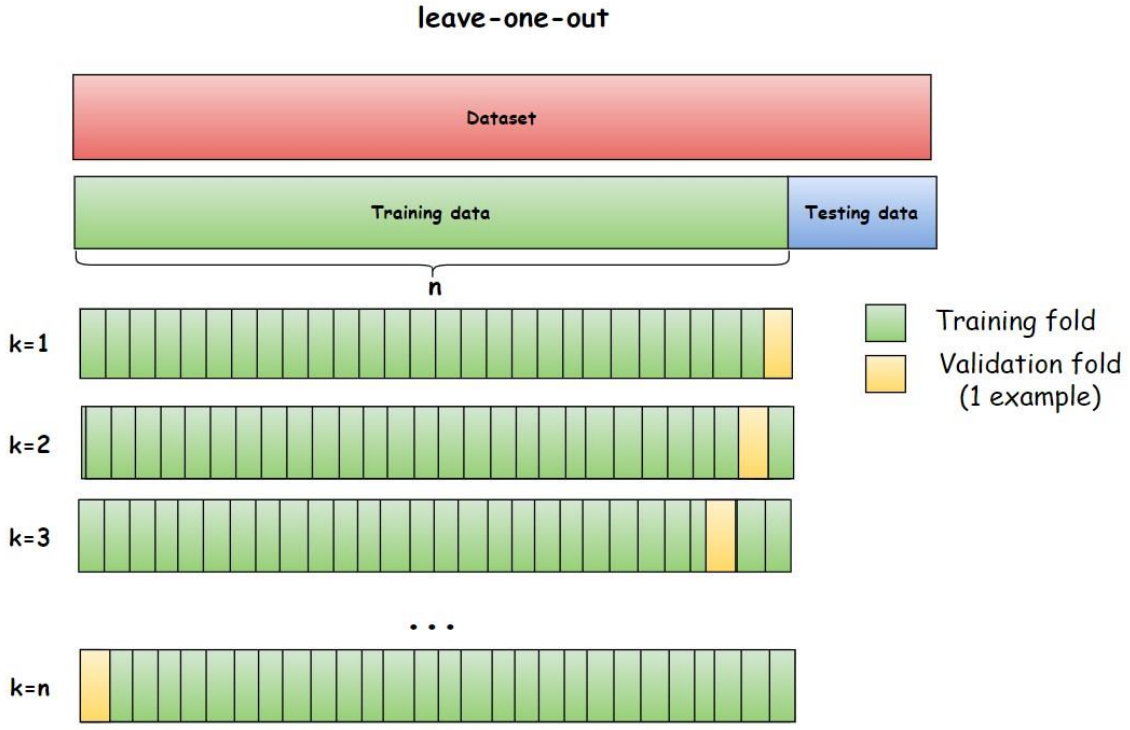
```
kf = model_selection.KFold(n_splits=5)

cv_metrics = model_selection.cross_validate( estimator=model, #модель
    X=X, #матрица наблюдений X y=y, #вектор ответов y cv=kf, #кросс-
    валидатор scoring='accuracy', #метрика return_train_score=True
    #подсчёт метрики на тренировочных фолдах
)
```

## Leave-One-Out

Метод **leave-one-out (отложенный пример)**, или **поэлементная кросс-валидация** — это частный случай кросс-валидации (k-fold), когда размер k равняется размеру всей выборки k=n, где n — количество примеров (строк в таблице).

**Алгоритм:**

1. Повторять n раз:
    a. Выбираем 1 случайный пример для валидации.
    b. Обучить модель на всех оставшихся n-1 примерах.
    c. Произвести оценку качества (вычислить метрику) на отложенном примере.
2. Усреднить значение метрик на всех примерах.




**Реализация leave-one-out в sklearn**

```
loo = model_selection.LeaveOneOut()

cv_metrics = model_selection.cross_validate(
    estimator=model, #модель
    X=X.iloc[:500], #матрица наблюдений
    X y=y.iloc[:500], #вектор ответов y
    cv=loo, #кросс-валидатор
    scoring='accuracy', #метрика
```

```
    return_train_score=True #подсчёт метрики на тренировочных фолдах
)
```

### Сравнение методов валидации

| | | | |
|---|---|---|---|
| | Очень простой и понятный.<br><br>Данный метод чаще всего применяется в случае больших датасетов в силу того, что требует значительно меньше вычислительных мощностей, чем другие методы. | Разбиение производится случайным образом, и оценка в этом методе зависит от того, какие наблюдения попали в набор для валидации. | `train_test_split()` |

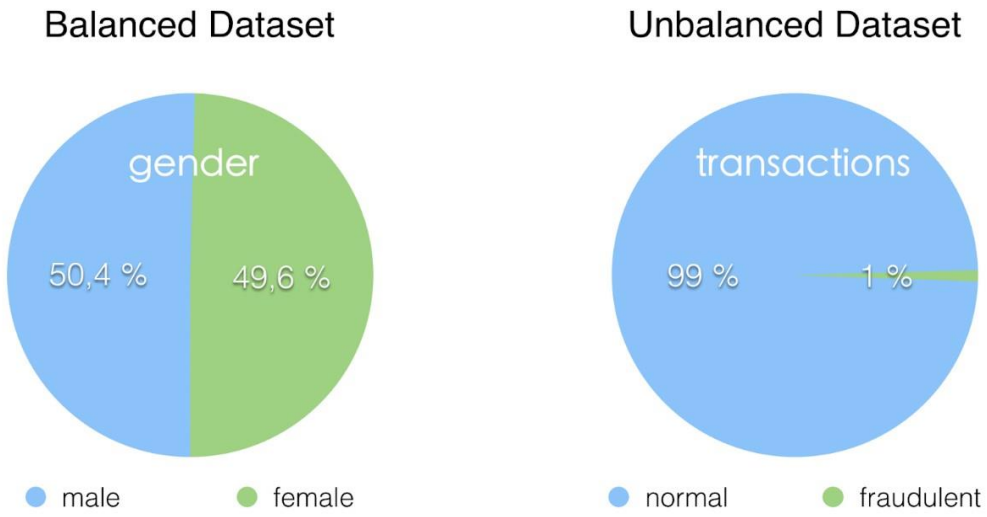
| | | | |
|---|---|---|---|
| | Более устойчивая к выбросам оценка качества модели.<br><br>Значения метрик получаются более объективными, чем в hold-out. | Обучаем одну и ту же модель k раз, что плохо сказывается на производительности. Если модель обучается медленно, то валидация может занять очень много времени. | `KFold` + `cross_validate()` |
| | Идеально подходит для небольших датасетов (менее 100 примеров).<br><br>Поскольку все доступные данные используются как для обучения, так и для валидации, значения метрик наиболее объективны и надёжны. | Обучаем одну и ту же модель n раз.<br><br>Поэтому метод не подходит для оценки качества модели на больших наборах данных, так как становится очень ресурсозатратным. | `LeaveOneOut` + `cross_validate()` |

## Дисбаланс выборки

> **Несбалансированный набор данных (unbalanced dataset)** — это выборка, в которой количества примеров каждого из классов значительно отличаются.

При этом:

➔ класс большинства называется **мажоритарным (majority) классом**;
➔ класс меньшинства называется **миноритарным (minority) классом**.




Типичные примеры задач, в которых исследователи чаще всего сталкиваются с дисбалансом выборки:

➔ обнаружение мошенничества,
➔ обнаружение оттока клиентов, ➔ распознавание лиц.

**Какие проблемы могут возникнуть из-за несбалансированной выборки?**

✖ При разбиении несбалансированной выборки на тренировочную/валидационную/тестовую увеличивается шанс попадания в одну из них объектов только одного класса, из-за чего оценка качества модели может быть необъективна.

✖ Нельзя использовать метрики, не учитывающие размеры классов, такие как accuracy.

✖ Стандартные методы ML, такие как дерево решений и логистическая регрессия, имеют тенденцию игнорировать класс меньшинства.

## Стратифицированное разбиение

> **Стратифицированное (stratified) разбиение** предполагает, что в каждый из наборов данных наблюдения, принадлежащие каждому из классов, гарантированно попадут в одинаковой пропорции.

### Реализация в sklearn (hold-out)

Для стратифицированного разбиения достаточно в функции `train_test_split()` задать параметр `stratify`, в который нужно передать столбец с метками классов, на основе которого будет производиться балансировка.

```
X_train, X_valid, y_train, y_valid= model_selection.train_test_split(X, y, stratify=y, test_size=0.2, random_state=1)
```

### Реализация в sklearn (k-fold)

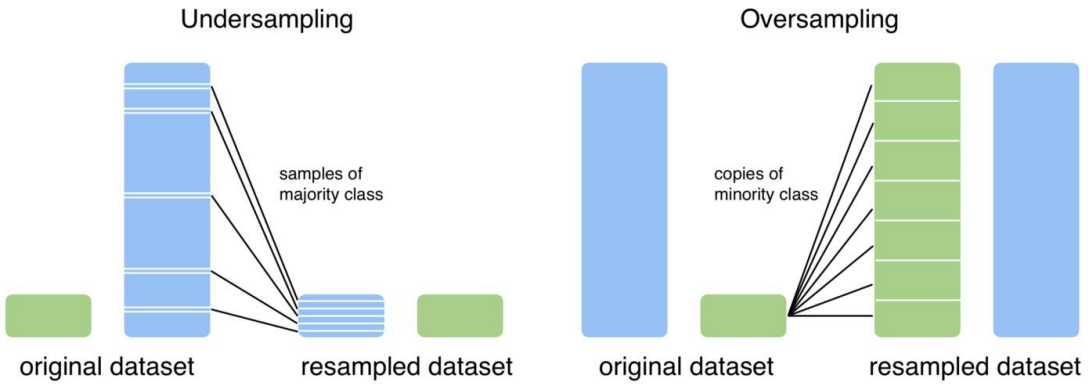
Для реализации стратификации при кросс-валидации вместо `KFold` используется кросс-валидатор `StratifiedKFold`.

```
skf = model_selection.StratifiedKFold(n_splits=5)

cv_metrics = model_selection.cross_validate( estimator=model, #модель
    X=X, #матрица наблюдений X y=y, #вектор ответов y cv=skf,
    #кросс-валидатор scoring='accuracy', #метрика
    return_train_score=True #подсчёт метрики на тренировочных
    фолдах
)
```

## Построение модели в условиях дисбаланса классов

Существует **несколько способов уменьшить влияние дисбаланса на обучение модели:**

- **Взвешивание объектов.** В функцию ошибки добавляется штраф, прямо пропорциональный количеству объектов каждого класса. Это очень похоже на регуляризацию, которую мы изучали ранее.
- **Выбор порога вероятности.** Этот подход заключается в том, что мы подбираем такой порог вероятности (по умолчанию во всех моделях он равен 0.5), при котором на валидационной выборке максимизируется целевая метрика (например, F1-score).
- **Сэмплирование (sampling)** — перебалансировка выборки искусственным путём:
  - **oversampling** — искусственное увеличение количества объектов миноритарного класса;
  - **undersampling** — сокращение количества объектов мажоритарного класса.




### Взвешивание объектов

Для того чтобы сбалансировать важность классов, обычно берут веса объектов класса-большинства (мажоритарного класса) равным $class\ weight(majority) = 1$, а веса объектов малого (миноритарного) класса

вычисляются по следующей формуле: $class$

$$weight(minority) = 1 + \frac{n_{majority}}{n_{minority}},$$

где $n_{majority}$ и $n_{minority}$ — число объектов в мажоритарном и миноритарном классах соответственно. Подобная установка весов заставляет алгоритм $n^{majority}$ $n^{minority}$ обращать большее внимание на объекты менее популярного класса.
Для того чтобы задать веса классам по приведённым выше формулам, достаточно в инициализаторе модели выставить параметр `class_weight='balanced'`.

**Пример:**

```
model = tree.DecisionTreeClassifier( criterion='entropy',
    #критерий информативности max_depth=7,
    #максимальная глубина min_samples_leaf=5,
    #минимальное число объектов в листе random_state=42,
    #генератор случайных чисел class_weight='balanced'
    #веса классов
)
```

**Выбор порога вероятности**

Любой классификатор предсказывает для объектов вероятности их принадлежности к классу 1 ($\hat{P}$) и классу 0 ($\hat{Q} = 1 - \hat{P}$). Класс объекта по умолчанию определяется по следующему правилу:

- Если вероятность $\hat{P} > 0.5$, то объект относится моделью к классу 1.

➔ Если вероятность $P \leq 0.5$, то объект относится моделью к классу 0.

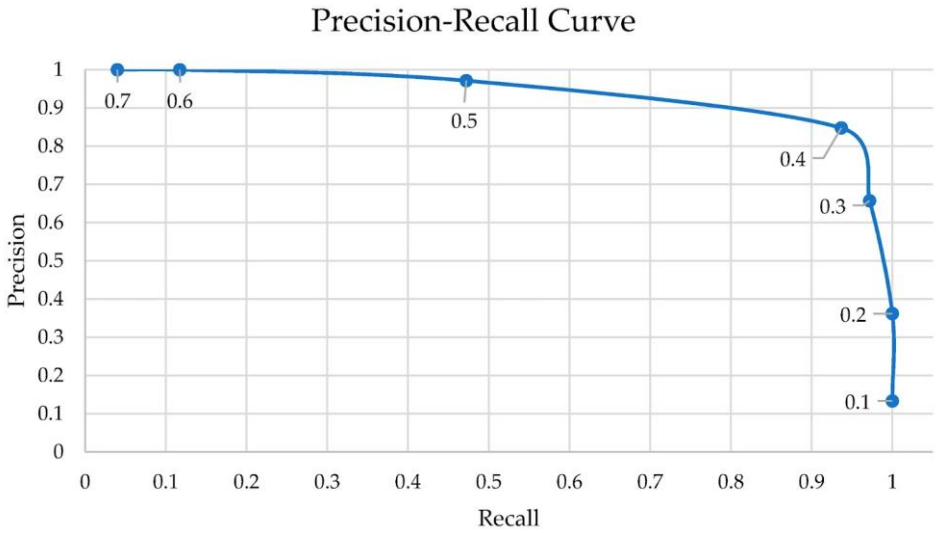
Но данный порог можно поменять и сделать его равным, например, 0.15 или 0.7.

Выбирается такой порог вероятности, который оптимален для миноритарного класса по выбранной метрике.

**PR-кривая**

**PR-кривая (precision-recall curve)** — это график зависимости precision от recall при различных значениях порога вероятности.

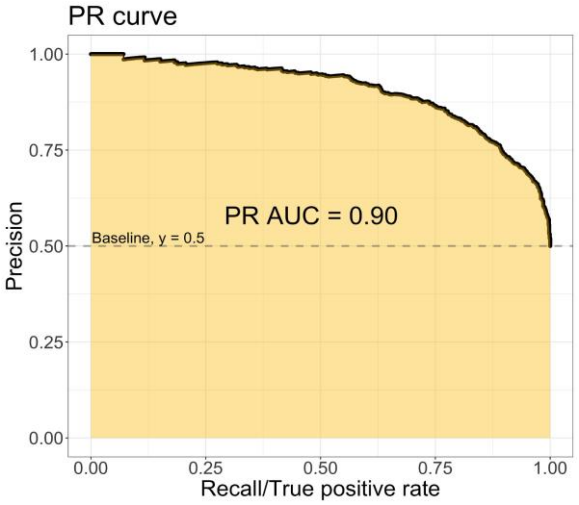
Для построения данного графика мы берём множество различных порогов вероятности (0.1, 0.15, 0.2, …1) и вычисляем метрики precision и recall при разных порогах вероятности.




**Что нам даёт такая кривая?**

➔ Во-первых, PR-кривая — это графическая метрика качества модели, она комплексно отражает и precision, и recall одновременно (как F1-мера) и особенно хороша в условиях дисбаланса классов. Качество

определяется площадью (PR AUC) под кривой: чем ближе значение площади к 1, тем лучше модель.




➔ Во-вторых, с помощью PR-кривой удобно находить оптимальный порог вероятности. Главное — определиться с критерием этой оптимальности. На кривой мы можем найти такие точки, в которых наблюдается наилучшее значение precision либо recall, либо среднее геометрическое между ними ($F_1$-score).

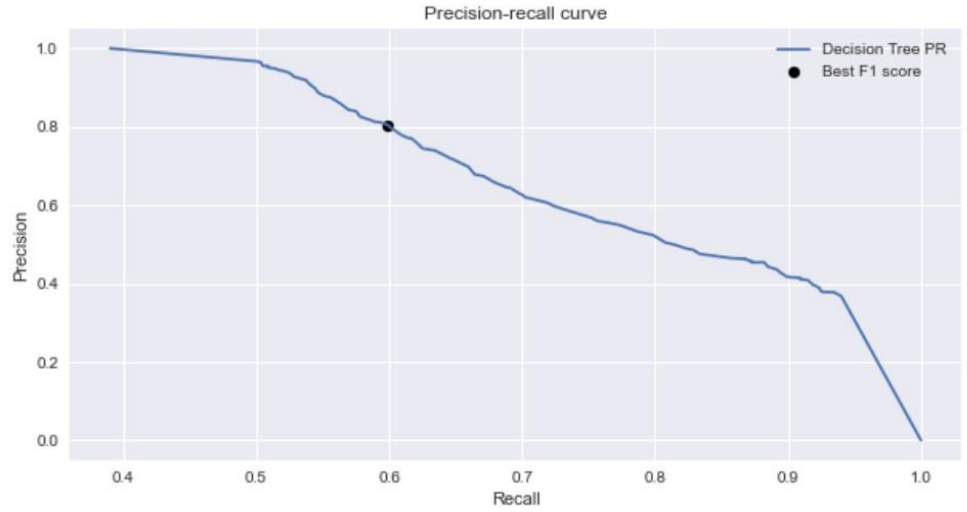
**Пример подбора порога вероятности на кросс-валидации:**

```
skf = model_selection.StratifiedKFold(n_splits=5)

y_cv_proba_pred = model_selection.cross_val_predict(model, X_train,
y_train, cv=skf, method='predict_proba') y_cv_proba_pred =
y_cv_proba_pred[:, 1] precision, recall, thresholds =
metrics.precision_recall_curve(y_train, y_cv_proba_pred)
f1_scores = (2 * precision * recall) / (precision + recall) idx = np.argmax(f1_scores)
print('Best threshold = {:.2f}, F1-Score = {:.2f}'.format(thresholds[idx],
f1_scores[idx]))
```

**Визуализация PR-кривой:**

```
fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(precision, recall, label='Decision Tree PR')
ax.scatter(precision[idx], recall[idx], marker='o', color='black', label='Best F1 score')
ax.set_title('Precision-recall curve') ax.set_xlabel('Recall') ax.set_ylabel('Precision')
ax.legend();
```

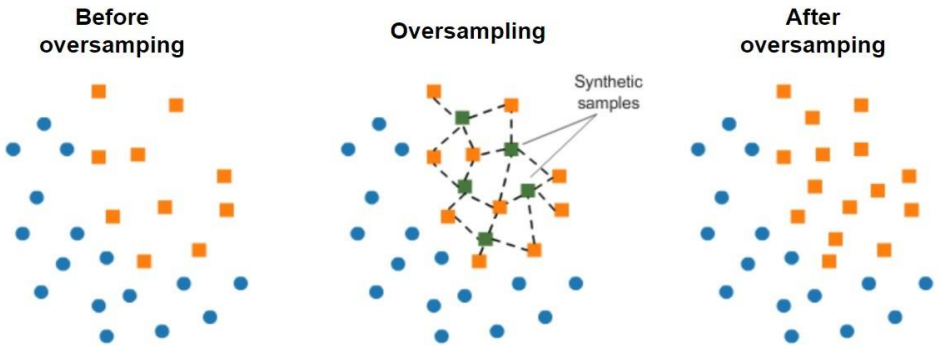
```
Best threshold = 0.33, F1-Score = 0.69
```




## Сэмплирование

Следующий подход работы в условиях дисбаланса классов, который мы рассмотрим, — сэмплирование, а точнее поговорим о **пересэмплировании (oversampling)**.

**Идея** очень проста: если у нас мало наблюдений миноритарного класса, стоит искусственно увеличить их количество.




Самый популярный алгоритм пересэмплирования — **SMOTE (Synthetic Minority Oversampling Techniques)**.

```
from imblearn.over_sampling import SMOTE
```

**Пример:**

```
sm = SMOTE(random_state=2)
X_train_s, y_train_s = sm.fit_sample(X_train, y_train)
```

## Недообучение и переобучение

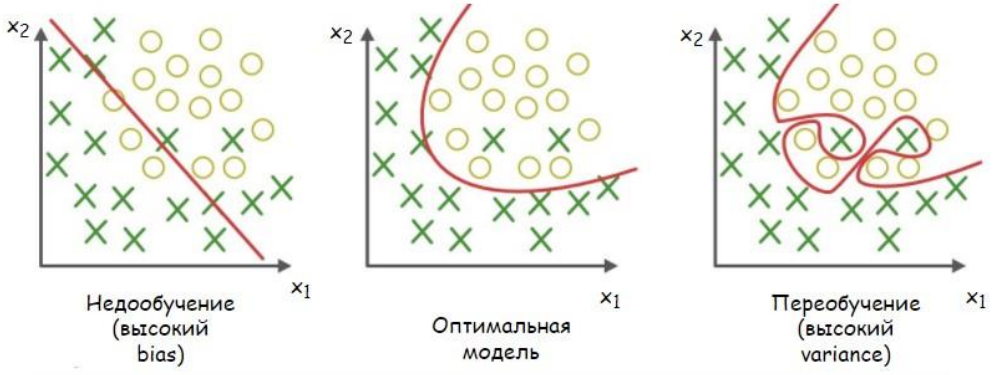
**Переобучение (overfitting)** — это проблема, при которой модель чувствительна к незначительным колебаниям в данных в процессе обучения. По сути, такая модель работает намного лучше с обучающими данными, чем с новыми. Она была чрезмерно натренирована на обнаружение уникальных характеристик обучающего набора данных, которые не являются общими закономерностями.

**Недообучение (underfitting)** — это проблема, при которой алгоритм недостаточно хорошо изучил данные и пропускает важные зависимости между признаками. В случае недообучения мы даже на обучающих данных не можем достичь приемлемых оценок для модели.

Недообучение и переобучение неразрывно связаны друг с другом: попытка бороться с одной проблемой может привести к возникновению другой, поэтому возникает **дилемма смещения-разброса (bias-variance tradeoff)**.

**Пример в случае задачи регрессии:**


**Пример в случае задачи классификации:**




**Основные способы отследить переобучение:**

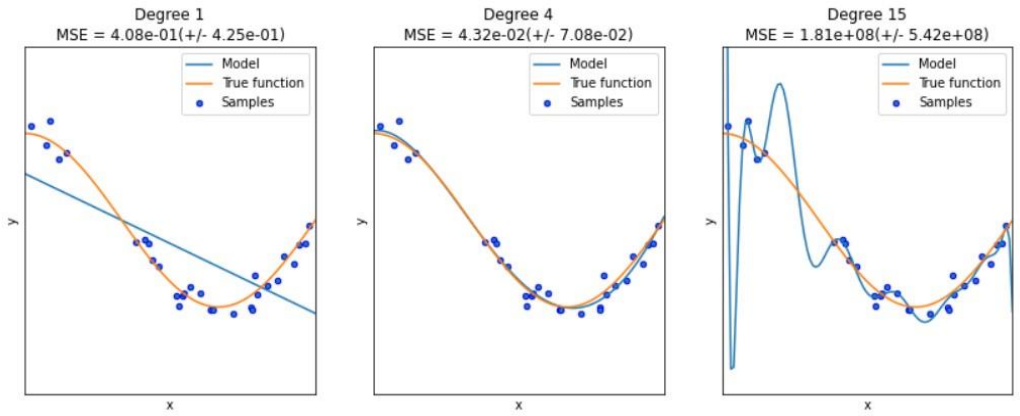
- ➔ hold-out-разбиение,
- ➔ k-fold-валидация и leave-one-out-валидация, ➔ кривые обучения (learning curves).

Если качество на валидационной выборке стабильно хуже качества на тренировочной, то это явный признак переобучения.

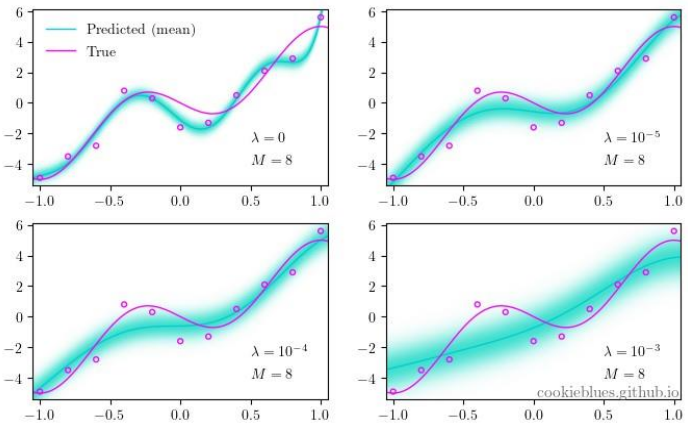
## Методы борьбы с переобучением

Ключевая идея, заложенная в каждом из методов звучит следующим образом: **снизить переобучение = уменьшить разброс (вариативность) ошибки модели**.

- **Уменьшение сложности модели.** Это основной способ борьбы с переобучением, т. к., по сути, завышенная сложность модели и является его причиной.




- **Регуляризация.** С помощью добавления штрафа в функцию потерь мы намеренно пытаемся увеличить смещение модели, чтобы уменьшить разброс.




- **Манипуляции над данными.** Ещё один верный способ побороть переобучение — это увеличить или уменьшить количество примеров, на которых обучается модель.
  - Увеличивать набор данных можно за счёт проведения новых экспериментов и сбора новой информации.

- Уменьшать набор данных можно за счёт удаления выбросов и аномалий, из-за которых отчасти и происходит переобучение модели, из обучающего набора данных.

## Утечка данных

> **Утечка данных (data leak)** — это ситуация, в которой данные, используемые для обучения модели, содержат прямую или косвенную информацию о целевой переменной.

Приведём **несколько примеров, когда может возникнуть утечка данных:**

1. **Очевидные случаи**
   - Включение целевой переменной, которую мы пытаемся предсказать, в качестве фактора.
   - Включение тестовых данных в данные по обучению модели, а затем использование этих же тестовых данных для оценки качества модели.

2. **Скрытые случаи, или giveaway-признаки**

> **Giveaway** — это признаки, которые раскрывают информацию о целевой переменной и не будут доступны после развёртывания модели в реальных условиях. Такие признаки необходимо удалять из данных перед построением модели.

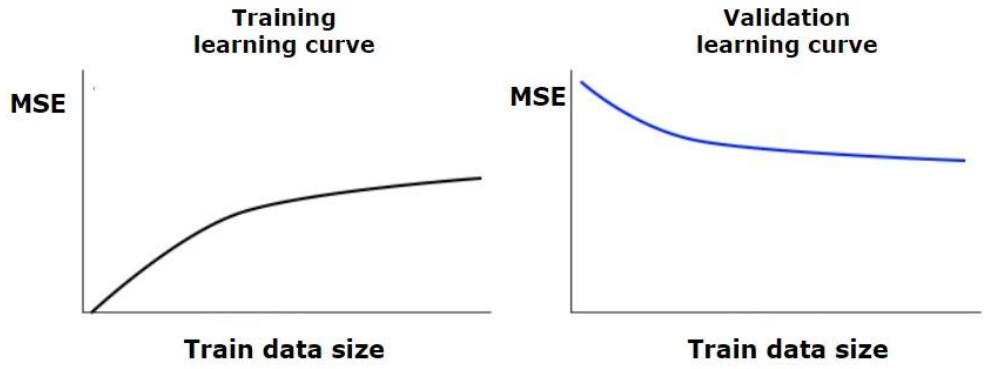
Из-за утечки данных прогноз модели становится очень оптимистичным. Вы получаете потрясающее качество во время обучения модели, радуетесь сами и радуете своего заказчика. Однако когда дело доходит до использования модели в реальных условиях, оказывается, что у вас недостаточно данных для построения прогноза.

**Как обнаружить утечку данных:**

- ➔ Читайте описание признаков.
- ➔ Проверяйте корреляции с целевым признаком.
- ➔ Относитесь скептически к подозрительно высокому качеству моделей.

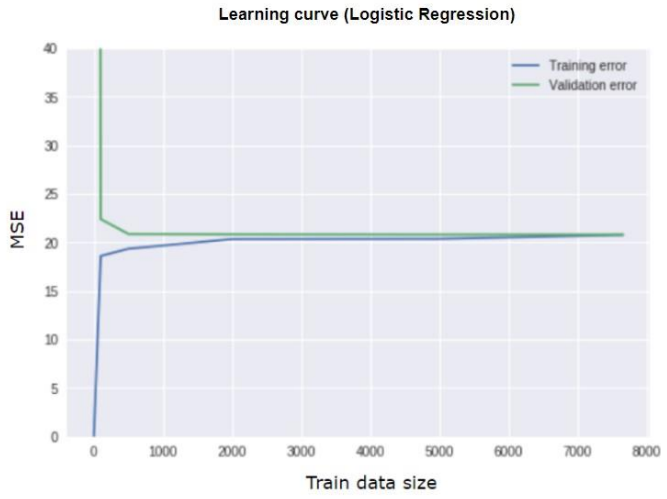
## Кривая обучения

> **Кривая обучения (learning curve)** — это график зависимости некоторой метрики на обучающем (валидационном) наборе данных от количества объектов, которые участвуют в обучении модели.

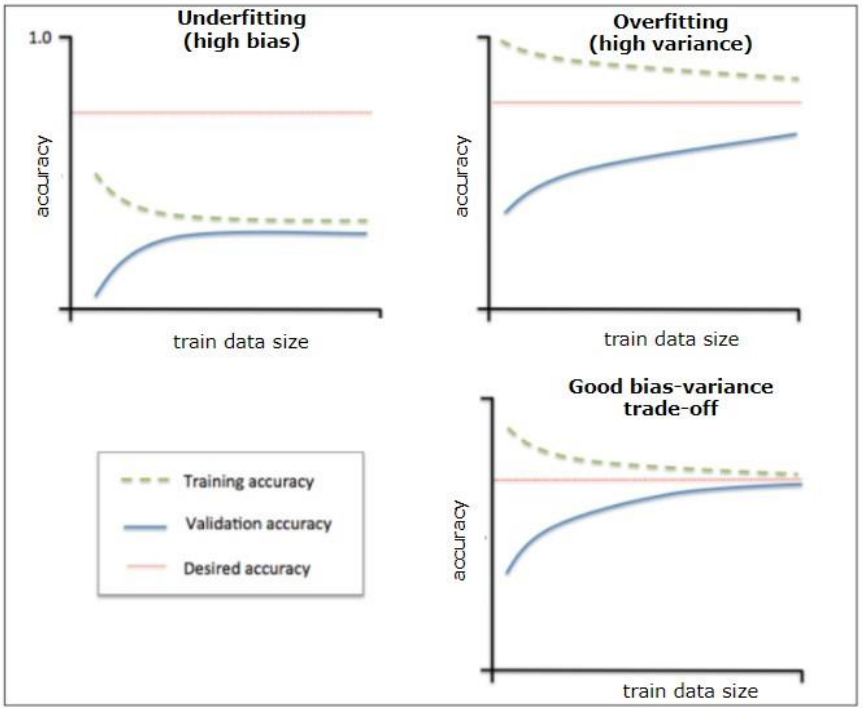



**Что нам дают такие кривые?**

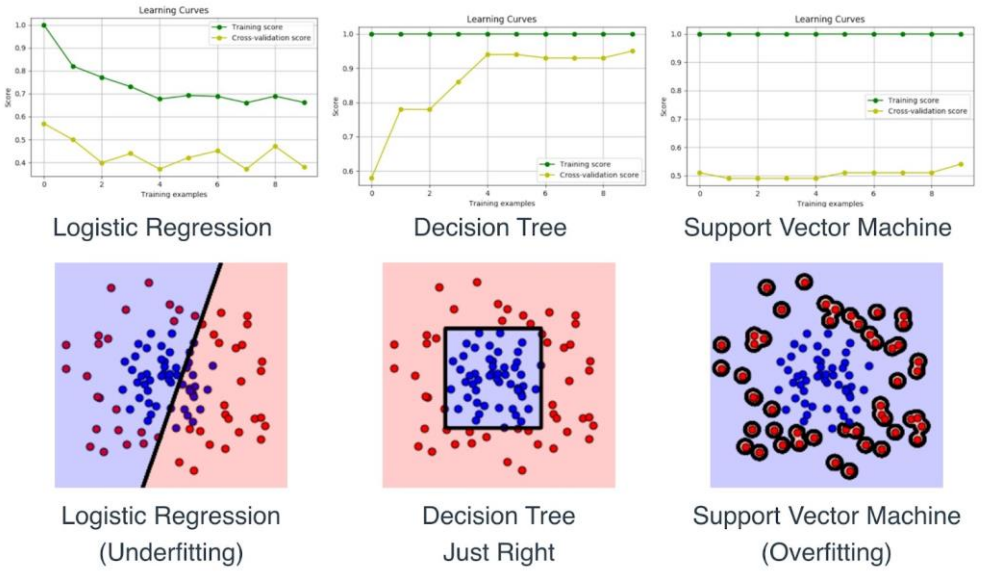
- ➔ Основное назначение кривых обучения — **мониторинг изменения метрики в процессе поступления новых данных**. Благодаря этому мы можем найти такой размер данных, начиная с которого обогащение набора данных новыми наблюдениями не приносит значительного эффекта.




➔ Благодаря кривым обучения мы можем **отслеживать недообучение и переобучение модели**.

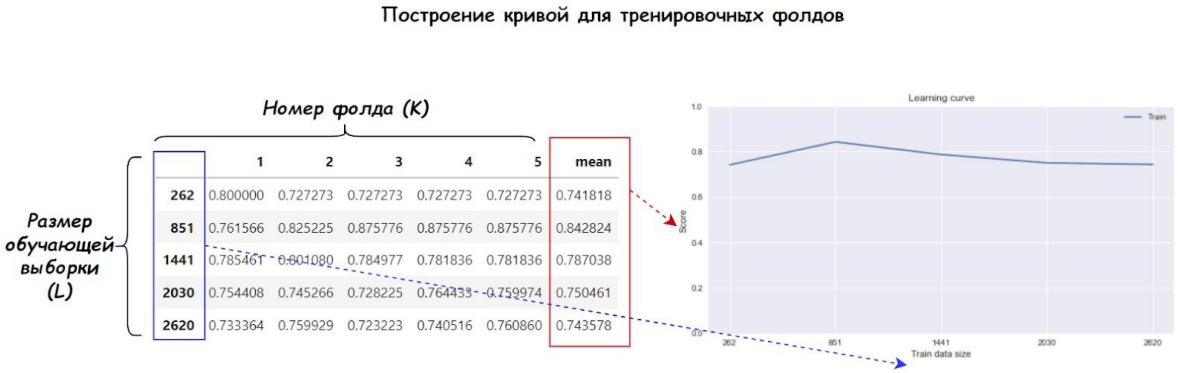



➔ Кривые обучения позволяют **визуально сравнить между собой качество различных моделей**.




На практике для наиболее эффективного построения кривых обучения используется кросс-валидация: то есть каждый набор обучающих данных, участвующих в построении кривой обучения, дополнительно проходит через k-fold-разбиение и вычисление метрик на $K$ фолдах.

В результате для тренировочных и валидационных фолдов у нас получается по таблице из метрик размером $(L, K)$, где $L$ — количество точек на оси абсцисс (x), по которым строятся кривые обучения, а $K$ — количество фолдов. Для построения кривой рассчитываются средние значения по столбцам таблицы — они и будут откладываться по оси ординат (y) при построении кривой.



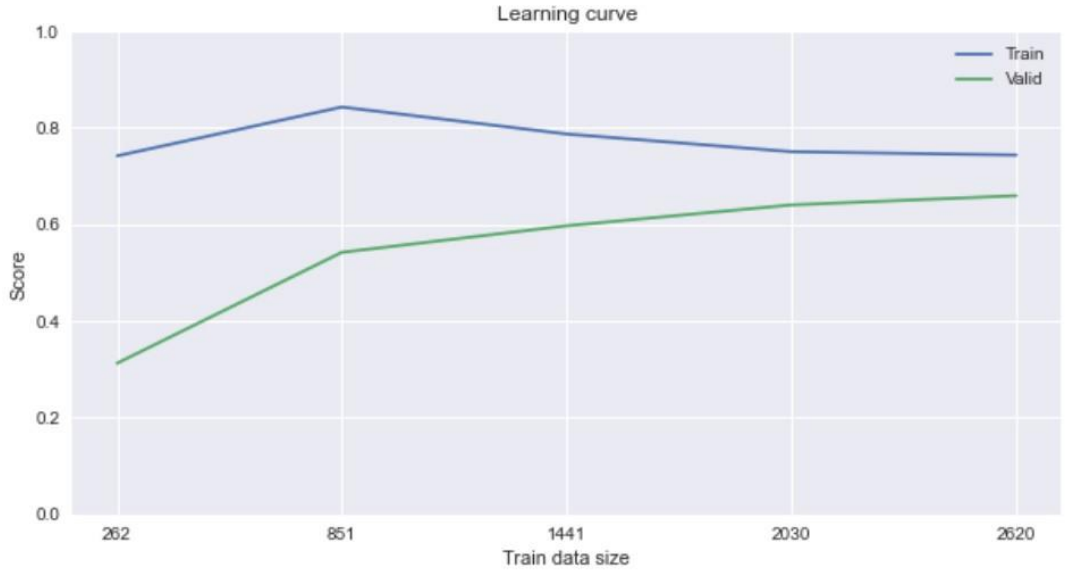

## Построение кривой обучения

Для вычисления точек для построения кривых обучения в модуле model_selection библиотеки sklearn есть функция `learning_curve()`.

**Пример:**

```
train_sizes, train_scores, valid_scores =
    model_selection.learning_curve( estimator = model, #модель X = X,
    #матрица наблюдений X y = y, #вектор ответов y cv = skf,
    #кросс-валидатор scoring = 'f1' #метрика
)
```

```
train_scores_mean = np.mean(train_scores, axis=1)
valid_scores_mean = np.mean(valid_scores, axis=1)
fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(train_sizes, train_scores_mean, label='Train')
ax.plot(train_sizes, valid_scores_mean, label='Valid')
ax.set_title('Learning curve') ax.set_xlabel('Train
data size') ax.set_ylabel('Score')
ax.xaxis.set_ticks(train_sizes) ax.set_ylim(0, 1)
ax.legend();
```


Learning curve
Train
Valid
Score
1.0
0.8
0.6
0.4
0.2
0.0
262
851
1441
2030
2620
Train data size