In [1]:

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import LeavePOut
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import KFold
from sklearn.model_selection import TimeSeriesSplit

from sklearn.model_selection import GroupKFold
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.model_selection import LeavePGroupsOut
from sklearn.model_selection import GroupShuffleSplit

from sklearn import datasets
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression

import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.collections import LineCollection

sns.set('notebook', style='whitegrid', font_scale=1.3)
```

Валидация

В простом случае предполагается, что у нас есть данные для обучения (train set) и данные для тестирования (test set). Ответы для тестовых данных считаются неизвестными, а потому их нельзя использовать для обучения модели.

Главная цель валидации — оценить какое качество модель способна показать на тестовых данных. Отсюда вытекает и главное правило валидации:

> Валидационные данные должны быть как можно сильнее похожи на тестовые.

В этом ноутбуке мы обсудим различные стратегии валидации.

Валидация на отложенной выборке (holdout validation)

При оценке качества модели нельзя использовать данные, которые использовались для ее обучения, т.к. при таком подходе мы не сможем оценить адекватность модели на новых данных и контролировать переобучение. Для решения данной проблемы существуют подходы, использующие понятие **отложенной выборки**: X_{val}, Y_{val}. Отложенной выборкой называют размеченные данные, которые мы не используем при обучении модели.

В рассматриваемом нами случае ответы на тестовых данных неизвестны, а потому отложенную выборку мы можем сформировать лишь из обучающих данных. Такую выборку обычно называют валидационным множеством (validation set или development set).

В scikit-learn разбиение на обучающую и тестовую выборки можно легко получить с помощью функции `train_test_split` .

Рассмотрим на примере задачи классификации ирисов.

In [2]:

```
# загружаем датасет
data_full = datasets.load_iris()
print("Shape: {}".format(data_full.data.shape))
```

```
Shape: (150, 4)
```

In [3]:

```
X_train, X_test, y_train, y_test = train_test_split(
    # *arrays: принимает индексируемые объекты с совпадающей shape[0].
    # Например: list, np.array, pd.DataFrame.
    data_full.data, data_full.target,
    test_size=0.4,  # доля данных, которые берем в тестовую выборку
    random_state=0,  # фиксируем случайность
    shuffle=True,  # перемешивает данные в случайном порядке
    stratify=None  # если не None, то сохраняет доли классов при разбиении
)
```

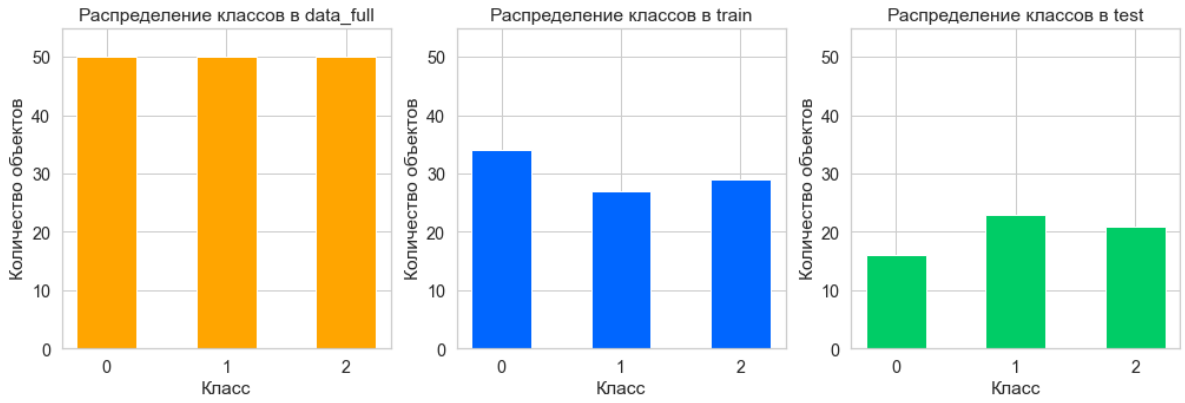
In [4]:

```
print("Shape of train data: {} {}".format(X_train.shape, y_train.shape))
print("Shape of test data: {} {}".format(X_test.shape, y_test.shape))
```

```
Shape of train data: (90, 4) (90,)
Shape of test data: (60, 4) (60,)
```

In [5]:

```
plt.figure(figsize=(17,5))
split_cases = [data_full.target, y_train, y_test]
colors = ['orange', '#0066FF', '#00CC66']
labels = ['Распределение классов в data_full',
          'Распределение классов в train',
          'Распределение классов в test']
for i in range(3):
    plt.subplot(1, 3, i + 1)
    values, counts = np.unique(split_cases[i], return_counts=True)
    plt.bar(values, counts, width=0.5, color=colors[i])
    plt.ylim(0, 55)
    plt.xticks([0, 1, 2])
    plt.xlabel('Класс')
    plt.ylabel('Количество объектов')
    plt.title(labels[i])
plt.show()
```



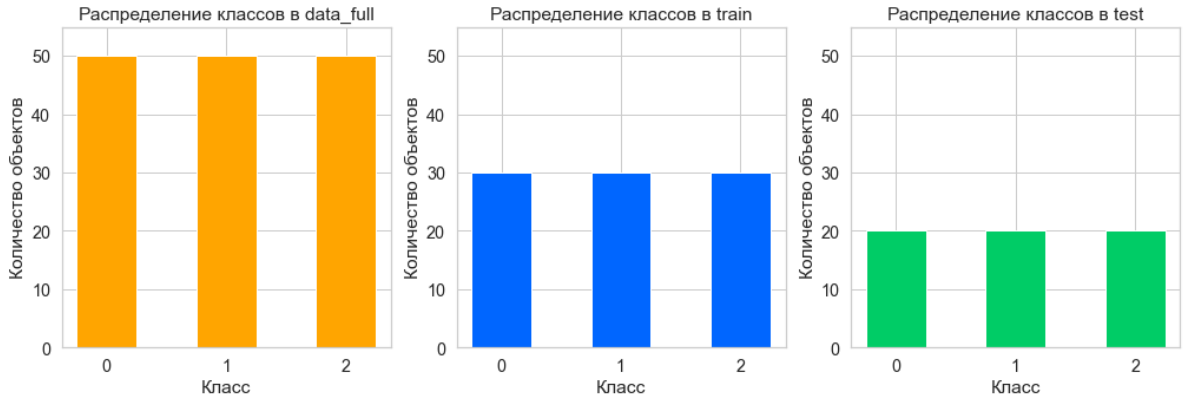
Видим, что распределение классов в обучающей и тестовой выборках отличаются. Теперь попробуем сделать разбиение с `stratify = data_full.target` . Стратификация — стратегия кросс-валидации, при которой в обучающей и тестовой выборке сохраняется одинаковое распределение целевой переменной, такое же, как во всем датасете. Зачем нужна стратификация узнаем чуть позже.

In [6]:

```
X_train, X_test, y_train, y_test = train_test_split(
    # *arrays: принимает индексируемые объекты с совпадающей shape[0].
    # Например: list, np.array, pd.DataFrame.
    data_full.data, data_full.target,
    test_size=0.4, # доля данных, которые берем в тестовую выборку
    random_state=0, # фиксируем случайность
    shuffle=True, # перемешивает данные в случайном порядке
    # сохраняем доли классов при разбиении как в таргете
    stratify=data_full.target
)
```

In [7]:

```
plt.figure(figsize = (17,5))
split_cases = [data_full.target, y_train, y_test]
colors = ['orange', '#0066FF', '#00CC66']
labels = ['Распределение классов в data_full',
          'Распределение классов в train',
          'Распределение классов в test']
for i in range(3):
    plt.subplot(1, 3, i + 1)
    values, counts = np.unique(split_cases[i], return_counts=True)
    plt.bar(values, counts, width=0.5, color=colors[i])
    plt.ylim(0, 55)
    plt.xticks([0, 1, 2])
    plt.xlabel('Класс')
    plt.ylabel('Количество объектов')
    plt.title(labels[i])
plt.show()
```



Видим, что после разбиения с `stratify = data_full.target` распределения классов в train и test не отличаются.

Подведем итог для метода отложенной выборки.

Достоинства:

- Быстрый для оценки качества модели. При использовании данной техники разбиения данных для оценки качества модели происходит одна процедура обучения на обучающей выборке, после чего качество модели оценивается на тестовых данных.

Недостатки:

- Результат сильно зависит от способа разбиения. Объекты в train и test могут получиться из разных распределений, если `stratify = False` .

- При обучении модели на обучающей выборке валидационная выборка не используется, то есть мы не задействуем все доступные данные для обучения.
- При оптимизации значения метрики на валидационном множестве модель немного переобучается под него. Таким образом, значение метрики качества на новых данных не будет соответствовать значению метрики на тестовом множестве.

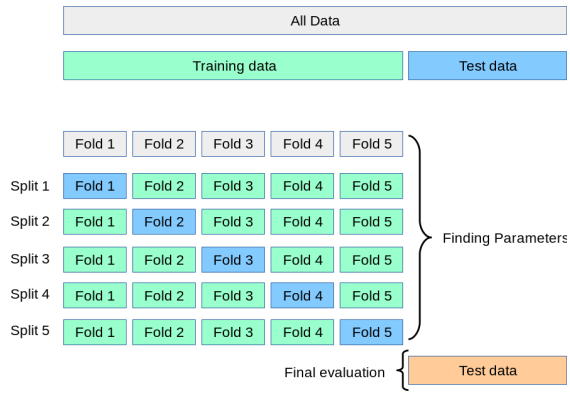
Кросс-валидация (cross-validation)

k-Fold Cross Validation

Описанные выше недостатки оказались критичны, поэтому для борьбы с ними придумали кросс-валидацию. **Кросс-валидация** — это метод оценки качества модели, при котором обучающая выборка делится на k частей, или **фолдов**. После чего для каждого из k фолдов проделывается следующая процедура:

- модель обучается на остальных $k - 1$ фолдах, которые вместе формируют обучающую выборку для данной итерации
- обученная модель оценивается на оставшемся k-ом фолде

Таким образом мы получаем k оценок качества. Итоговая метрика считается как среднее полученных оценок. Ниже представлена визуализация рассматриваемой стратегии кросс-валидации для пяти фолдов.



Рассмотрим пример. Будем использовать данные о ценах квартир в Бостоне.

In [8]:

```
boston = datasets.load_boston() # данные о ценах квартир в Бостоне
X = pd.DataFrame(data=boston['data'], columns= boston['feature_names'])
y = boston['target']
```

```
/home/mrgeekman/Programs/miniconda/envs/mipt-stats/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function load
_boston is deprecated; `load_boston` is deprecated in 1.0 and will be
removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can
refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use
of this
    dataset unless the purpose of the code is to study and educate abo
ut
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np


        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=
None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :
2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames h
ousing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.

  warnings.warn(msg, category=FutureWarning)
```

Описание датасета.

In [9]:

```
print(boston['DESCR'])
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median V
alue (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over
25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds r
iver; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1
940
        - DIS      weighted distances to five Boston employment centre
s
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of bla
ck people by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/ (ht
tps://archive.ics.uci.edu/ml/machine-learning-databases/housing/)


This dataset was taken from the StatLib library which is maintained at
Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedon
ic
prices and the demand for clean air', J. Environ. Economics & Manageme
nt,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diag
nostics
...', Wiley, 1980.   N.B. Various transformations are used in the tabl
e on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning pap
```

```
ers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influ
ential Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learn
ing. In Proceedings on the Tenth International Conference of Machine L
earning, 236-243, University of Massachusetts, Amherst. Morgan Kaufman
n.
```

In [10]:

```
X.head()
```

Out[10]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LS
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	

In [11]:

```
X.shape
```

Out[11]:

```
(506, 13)
```

При помощи функции `cross_val_score` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html) можем получить значение выбранной метрики на всех фолдах.

В качестве примера применим ее к линейной модели.

In [12]:

```
model = LinearRegression()
scores = cross_val_score(
    estimator=model, # модель, качество которой хотим оценить
    X=X, # данные для обучения (не содержат целевую переменную)
    y=y, # значения целевой переменной
    cv=5, # количество фолдов
    scoring='neg_mean_squared_error', # метрика качества
    n_jobs=-1 # количество ядер для вычислений, -1 - использование всех ядер
)
scores
```

Out[12]:

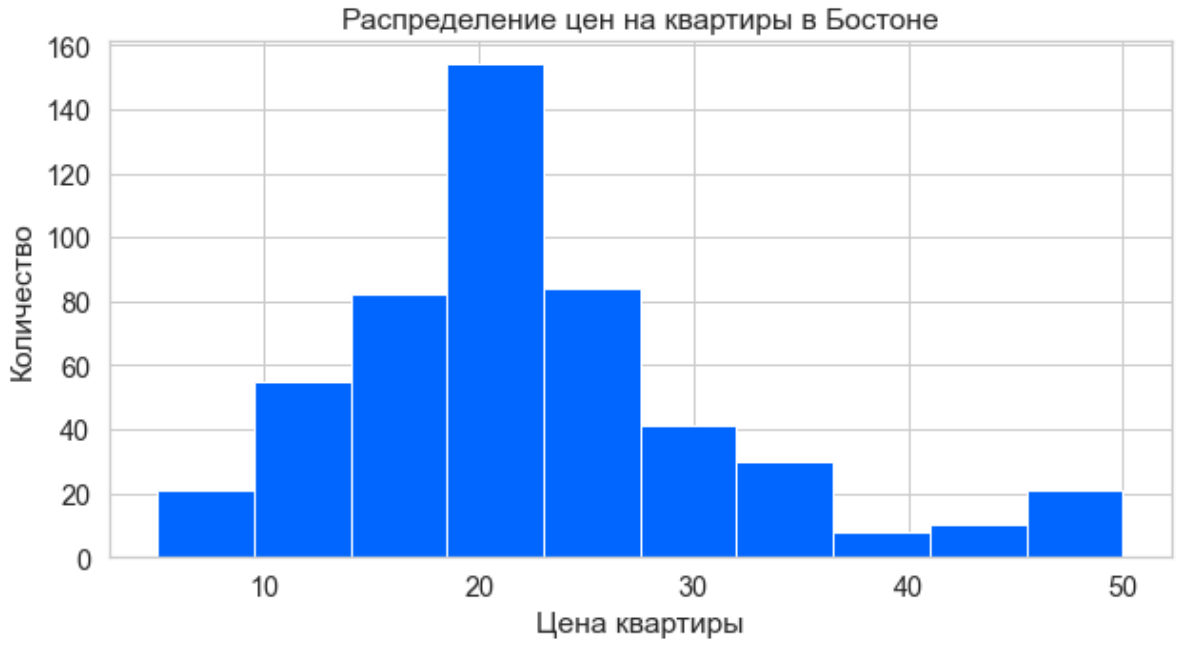
```
array([-12.46030057, -26.04862111, -33.07413798, -80.76237112,
       -33.31360656])
```

Стоит отметить, что в качестве scoring мы используем `neg_mean_squared_error` . Префикс neg показывает, что мы оптимизируем $(-1) \cdot \mathrm{MSE}$. Дело в том, что оптимизации в sklearn подразумевают **максимизацию** метрики качества.

Посмотрим на распределение цен, чтобы понимать в каком масштабе находятся значения MSE.

In [13]:

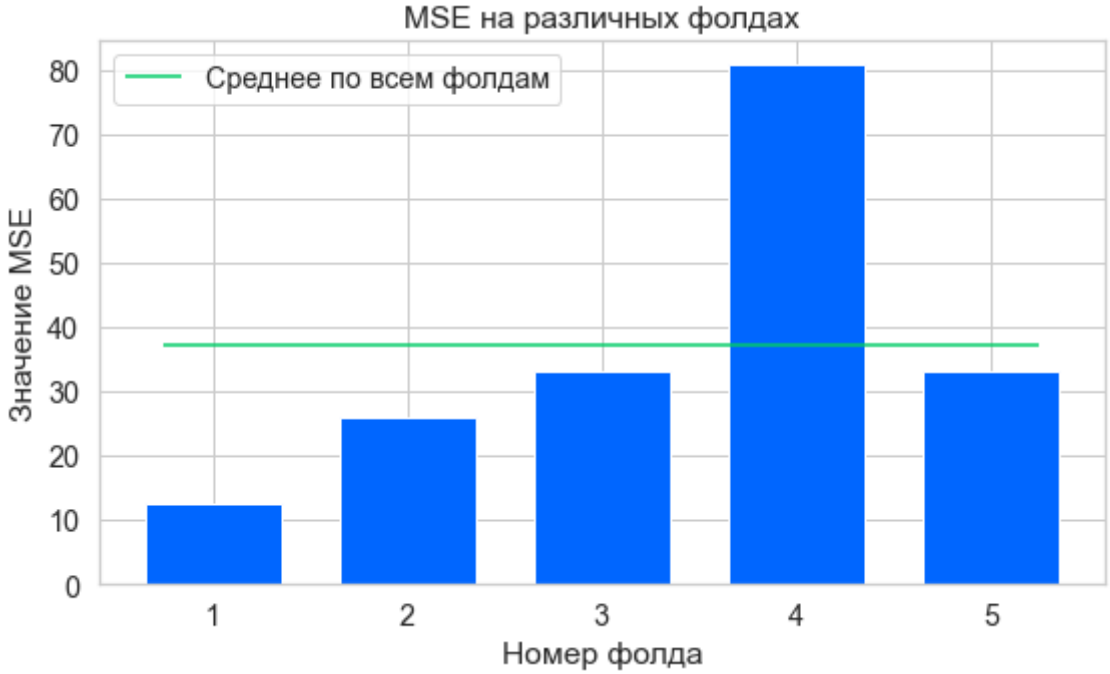
```
plt.figure(figsize=(10,5))
plt.hist(y, color='#0066FF')
plt.xlabel('Цена квартиры')
plt.ylabel('Количество')
plt.title('Распределение цен на квартиры в Бостоне')
plt.show()
```



Визуализируем MSE на всех фолдах.

In [14]:

```
plt.figure(figsize=(9,5))
plt.bar(range(1, 6), (-1)*scores, width=0.7, color='#0066FF')
plt.hlines(np.mean((-1)*scores), 0.75, 5.25, color='#00CC66',
           label='Среднее по всем фолдам')
plt.xlabel('Номер фолда')
plt.ylabel('Значение MSE')
plt.title('MSE на различных фолдах')
plt.legend()
plt.show()
```



Видим, что значения достаточно сильно различаются.

Полезно знать:

- `cross_validate` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html) — позволяет задать сразу несколько метрик для подсчета качества модели. Возвращаем значения данных метрик для каждой итерации кросс-валидации в виде словаря.
- `cross_val_predict` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_predict.html#sklearn.model_selectior — возвращает предсказания, полученные для каждого объекта выборки при кросс-валидации.

Выше мы рассмотрели функцию `cross_val_score`, которая имеет аргумент `cv`. По умолчанию данный аргумент использует стратегию кросс-валидации KFold, но ему можно передавать и другие стратегии кросс-валидации. Рассмотрим аналогичный способ использования KFold кросс-валидации, который на практике является более гибким.

Задаем стратегию кросс-валидации KFold.

In [15]:

```
kf = KFold(
    n_splits=2,  # количество фолдов
    shuffle=False  # перемешиваем ли данные перед разбиением
)
kf
```

Out[15]:

```
KFold(n_splits=2, random_state=None, shuffle=False)
```

В sklearn объекты классов, которые соответствуют стратегиям кросс-валидации, обычно имеют два метода:

- `get_n_splits` — возвращает количество итераций, которое необходимо для заданной стратегии кросс-валидации;
- `split` — возвращает генератор индексов для разбиения данных на train и test.

Замечание.

У этого класса нет никакого механизма стратификации.

In [16]:

```
kf.get_n_splits()
```

Out[16]:

```
2
```

In [17]:

```
kf.split(
    X=X  # данные для разбиения
)
```

Out[17]:

```
<generator object _BaseKFold.split at 0x7fc9e9542c80>
```

Приведем пример, демонстрирующий работу метода `split` .

In [18]:

```
data = np.array([[81, 27], [26, 45], [83, 64], [25, 98]])
data
```

Out[18]:

```
array([[81, 27],
       [26, 45],
       [83, 64],
       [25, 98]])
```

In [19]:

```
for train_index, test_index in kf.split(data):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1] TEST: [2 3]
```

Случай, когда аргумент `cv` функции `cross_val_score` принимает на вход стратегию кросс-валидации. На выходе функции получаем значения метрик для каждой итерации кросс-валидации.

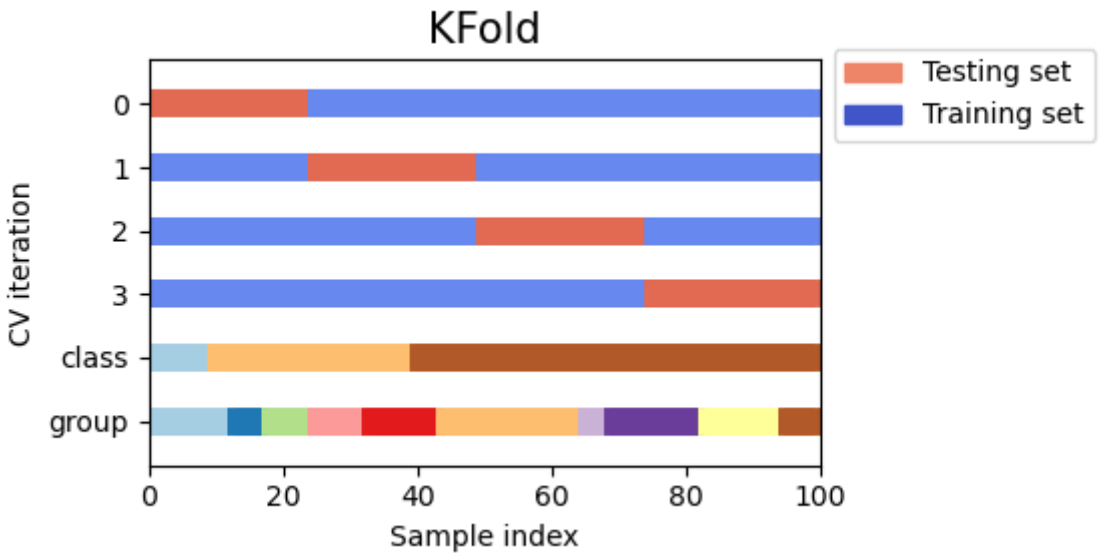
In [20]:

```
scores = cross_val_score(estimator=model, X=X, y=y, cv=kf,
                         scoring='neg_mean_squared_error')
scores
```

Out[20]:

```
array([ -27.21869349, -303.43686293])
```

Визуальная интерпретация KFold кросс-валидации:



На данном графике представлена 4-Fold CV (сокращение от Cross-Validation). Каждую горизонтальную полосу стоит понимать как одну и ту же выборку из 100 элементов. По горизонтальной оси показан номер элемента выборки. По вертикальной оси сверху отложены номера фолдов. Снизу отложены разбиения выборки по классу (целевая переменная) и какому-то категориальному признаку (группа). О группах будет сказано чуть позже.

Достоинства:

- Оценивается качество модели, полученное при обучении на всех данных.
- При подборе гиперпараметров можем контролировать переобучение, т.к. выбирается модель, показавшая лучшее качество на отложенных (тестовых) фолдах. Переобучение — это ситуация, когда модель показывает хорошее качество на обучающей выборке, но плохое качество на отложенной выборке.

Недостатки:

- Значительная вычислительная сложность. Вместо одной процедуры обучения приходится обучать модель k раз.
- Никак не учитывает распределение значений целевой переменной.
- Не учитывает разбиение объектов на группы (что это такое разберемся чуть ниже).

Leave One Out (LOO)

Данная стратегия кросс-валидации по сути является NFold CV, где N — количество элементов в обучающей выборке. На каждой итерации мы обучаем модель на $N - 1$ элементах и оцениваем качество на оставшемся элементе.

In [21]:

```
X = [1, 2, 3, 4]
loo = LeaveOneOut()

# итерируемся по разбиениям множества индексов
for train, test in loo.split(X):
    print("%s %s" % (train, test))
```

```
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

Достоинства:

- На каждой итерации при обучении модели используются все данные, за исключением одного элемента.
- Исследование отдельных объектов. Если на каком-то объекте допускается большая ошибка, может это выброс.
- В некоторых случаях выведены теоретические формулы результата LOO.

Недостатки:

- Огромная вычислительная сложность, не рекомендуется использовать на больших данных.
- Модель, полученная на конкретной итерации, не сильно отличается от моделей, которые получены на других итерациях. Таким образом ошибка сильно зависит от отложенного элемента, вследствие чего среди ошибок на отложенных элементах можно наблюдать высокий разброс.

LeavePOut

Данная стратегия кросс-валидации заключается в следующем. Пусть n — размер выборки. При $p = 1$ данная стратегия эквивалентна LOO. Для оценки модели будет обучено C_n^p моделей, где на каждой итерации для обучающей выборки будет взято $n - p$ элементов, а для тестовой p элементов. Пример использования:

In [22]:

```
X = [0.76, 0.43, 0.47, 0.82, 0.22]  # какая-то выборка размера 5
lpo = LeavePOut(p=2)  # p - количество элементов в отложенном фолде
for train, test in lpo.split(X):
     print("%s %s" % (train, test))
```

```
[2 3 4] [0 1]
[1 3 4] [0 2]
[1 2 4] [0 3]
[1 2 3] [0 4]
[0 3 4] [1 2]
[0 2 4] [1 3]
[0 2 3] [1 4]
[0 1 4] [2 3]
[0 1 3] [2 4]
[0 1 2] [3 4]
```

Достоинства:

- Является *исчерпывающей* стратегией кросс-валидации для заданного размера тестовой выборки, т.е. проверяет все возможные способы разделения исходной выборки на обучающее и тестовое множества заданного размера.

Недостатки:

- Огромная вычислительная сложность, которая быстро растет с увеличением параметра p. Не рекомендуется использовать на больших данных. Например, при $n = 100$ и $p = 30$ необходимо обучить примерно $3 \cdot 10^{25}$ моделей.
- На некоторых итерациях распределение целевой переменной в обучающей и тестовой выборке может быть слишком разным (нет стратификации).

Замечание.

Важно понимать, что `LeavePOut(p)` не является `KFold(n_splits=n_samples // p)` , т.к. KFold создает непересекающиеся тестовые множества.

ShuffleSplit

Данная стратегия состоит в следующем:

- фиксируем количество итераций `n_splits` , т.е количество разбиений, которое мы хотим получить.
- фиксируем размер тестовой выборки `test_size` , который будет одинаковым на каждой итерации.
- перемешиваем выборку и делим ее на две части: train и test. Проделываем это `n_splits` раз.

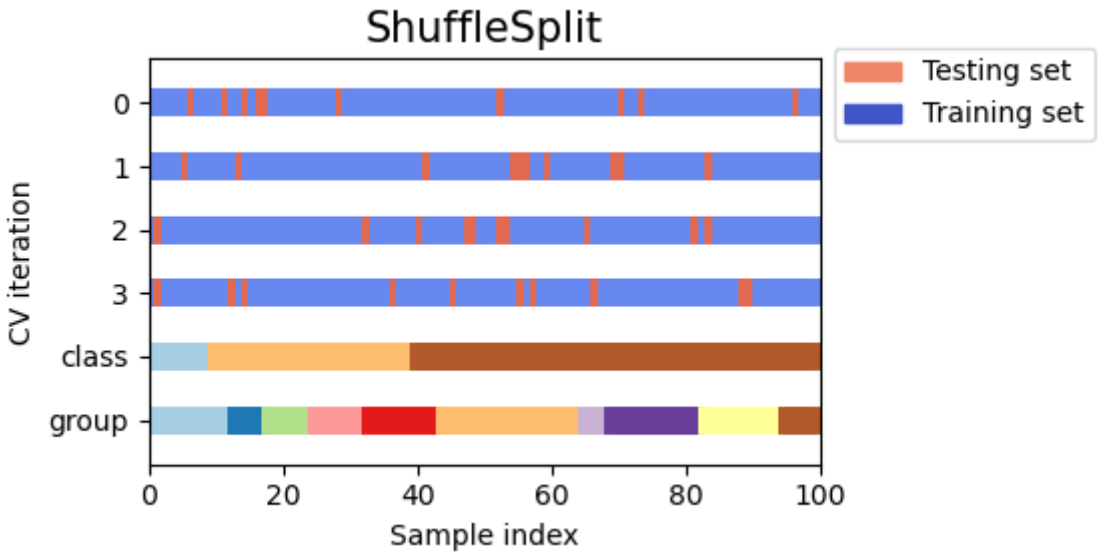
Пример работы:

In [23]:

```
X = np.arange(10)  # какая-то выборка размера 10
ss = ShuffleSplit(
    # количество итераций перемешивания с разбиением на train и test
    n_splits=5,
    # доля объектов, которые хотим класть в test на каждой итерации
    test_size=0.25,
    random_state=0
)

for train_index, test_index in ss.split(X):
    print("%s %s" % (train_index, test_index))
```

```
[9 1 6 7 3 0 5] [2 8 4]
[2 9 8 0 6 7 4] [3 5 1]
[4 5 1 0 6 9 7] [2 3 8]
[2 7 5 8 0 3 4] [6 1 9]
[4 1 0 6 8 9 3] [5 2 7]
```

Визуальная интерпретация:



Достоинства:

- Является хорошей альтернативой KFold, т.к. дает более четкий контроль над количеством итераций и разбиением на train и test.
- Результат разбиения случаен, поэтому не зависит от порядка объектов в данных.

Недостатки:

- На некоторых итерациях распределение целевой переменной в обучающей и тестовой выборке может быть слишком разным, так как разбиение случайно.
- Не учитывает разбиение объектов на группы (что это такое разберемся чуть ниже).

Замечание.

Ключевым отличием KFold от ShuffleSplit является тот факт, что в KFold каждый объект выборки в одной из итераций попадает в тестовый фолд, а в остальных итерациях используется для обучения. В ShuffleSplit разбиение каждой итерации не зависит от предыдущих итераций, объект выборки может как попасть в тестовый фолд, так и не попасть.

Stratified KFold

При рассмотрении различных стратегий кросс-валидации выше мы неоднократно отмечали, что многие стратегии не учитывают распределение целевой переменной. В некоторых задачах это может быть критично. Например, если вам нужно будет предсказать вероятность заболевания у пациента, то в выборке наверняка будет сильный дисбаланс классов. Таким образом при кросс-валидации объекты выборки из положительного класса могут просто не попасть в фолды для обучения. Для решения данной проблемы используется Stratified KFold. При таком подходе каждый фолд имеет примерно такое же распределение целевого класса, как и во всем датасете. Это нужно, т.к. мы хотим получить значение метрики, которая отражает *реальное* качество модели, а значение метрики сильно зависит от баланса классов. Например, модель, предсказывающая для всего класс 0, будет иметь хорошее качество на множестве $0, 0, 1, 0$, но плохое на $1, 1, 1, 0$. Сохраняя баланс классов, нам удастся получить значение метрики, которое более приближено к реальному качеству модели.

Замечание.

Стратификация работает только для классификации.

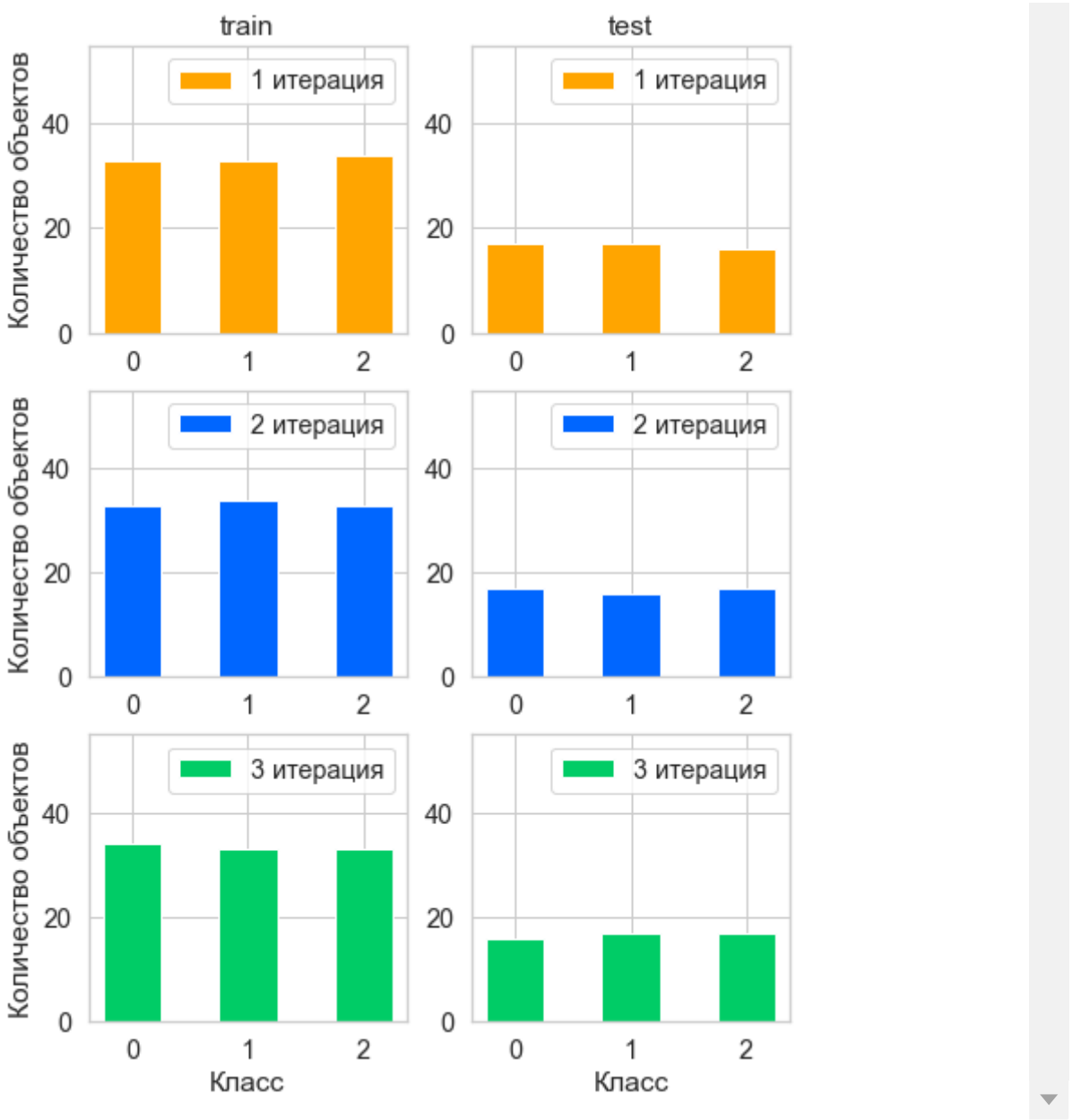
Рассмотрим применение Stratified KFold к задаче классификации ирисов. Построим графики распределения классов на каждой итерации.

In [24]:

```
iris = datasets.load_iris()
X = pd.DataFrame(data=iris['data'], columns=iris['feature_names'])
y = iris['target']
```

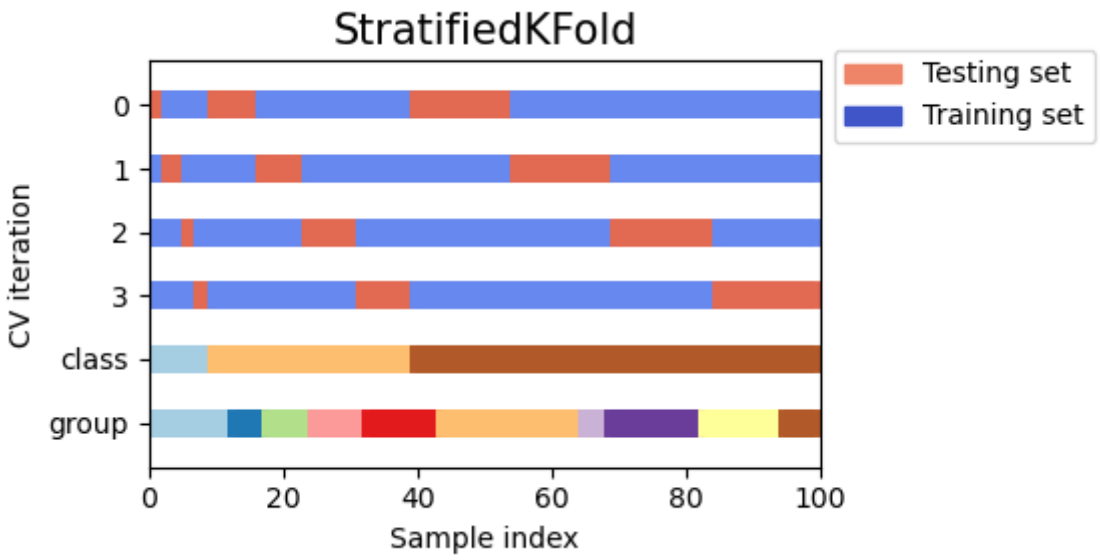
In [25]:

```
skf = StratifiedKFold(n_splits=3, shuffle=True)

colors = ['orange', '#0066FF', '#00CC66']

plt.figure(figsize=(7, 10))
for i, (train, test) in enumerate(skf.split(X, y)):
    plt.subplot(3,2,2*i + 1)
    if i == 0:
        plt.title('train')
    values, counts = np.unique(y[train], return_counts=True)
    plt.bar(values, counts, width=0.5, color=colors[i],
            label='{} итерация'.format(i + 1))
    plt.legend()
    plt.ylim(0, 55)
    plt.xticks([0, 1, 2])
    if 2*i + 1 == 5:
        plt.xlabel('Класс')
    plt.ylabel('Количество объектов')
    plt.subplot(3,2, 2*i + 2)
    if i == 0:
        plt.title('test')
    values, counts = np.unique(y[test], return_counts=True)
    plt.bar(values, counts, width=0.5, color=colors[i],
            label='{} итерация'.format(i + 1))
    plt.legend()
    plt.ylim(0, 55)
    plt.xticks([0, 1, 2])
    if 2*i + 2 == 6:
        plt.xlabel('Класс')

plt.show()
```



Ниже представлена визуальная интерпретация Stratified KFold. Стоит обратить внимание, что на каждой итерации доли каждого класса в train и test такие же, как в полном датасете.

Визуальная интерпретация:



Из построенного графика и картинки видим, что при разбиении выборки на train и test на каждой итерации кросс-валидации распределение целевого класса остается примерно одинаковым.

Достоинства:

- Учитывает распределение целевого класса при разбиении на обучающую и тестовую выборку.
- Оценивается качество модели, полученное при обучении на всех данных.
- При подборе гиперпараметров можем контролировать переобучение, т.к. выбирается модель, показавшая лучшее качество на отложенных (тестовых) фолдах.

Недостатки:

- Если не использовать `shuffle` , то результат сильно зависит от порядка объектов в данных.
- Не учитывает разбиение объектов на группы (что это такое разберемся чуть ниже).

Stratified Shuffle Split

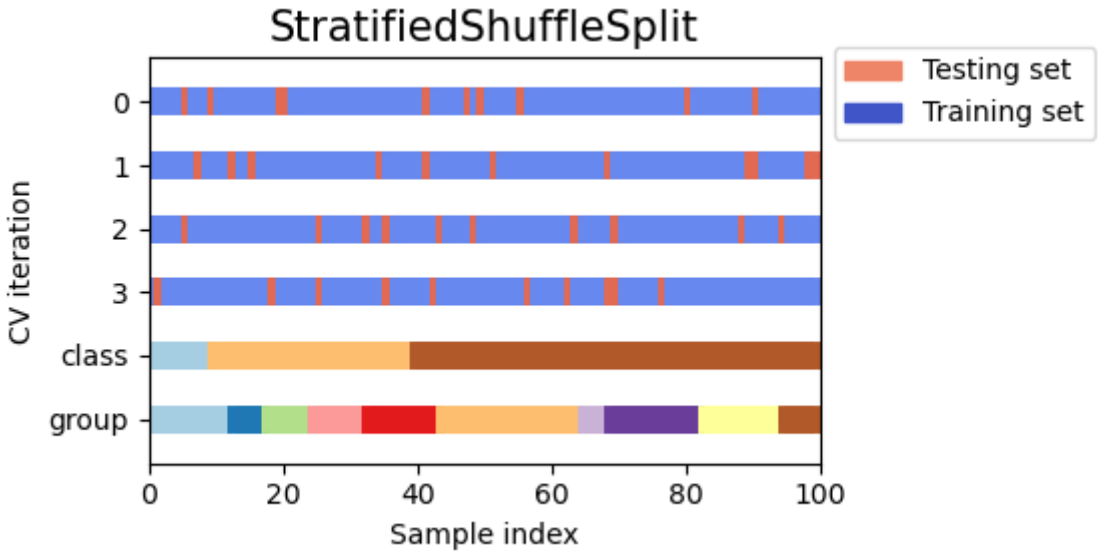
Данная стратегия кросс-валидации делает то же самое, что и Shuffle Split, только учитывает распределение целевой переменной. Несмотря на случайность разбиения, в каждой итерации распределение на train и test такое же, как и во всем датасете.

In [26]:

```
sss = StratifiedShuffleSplit(n_splits=5, test_size=0.2,
                             random_state=0)
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
y = np.array([0, 0, 0, 1, 1, 1])

for train_index, test_index in sss.split(X, y):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
TRAIN: [2 5 1 3] TEST: [0 4]
TRAIN: [0 4 3 1] TEST: [2 5]
TRAIN: [0 4 3 1] TEST: [2 5]
TRAIN: [5 4 1 2] TEST: [0 3]
TRAIN: [1 5 2 4] TEST: [0 3]
```

Визуальная интерпретация:



Достоинства:

- Учитывает распределение целевого класса.
- Результат разбиения случаен, поэтому не зависит от порядка объектов в данных.

Недостатки:

- Не учитывает разбиение объектов на группы (что это такое разберемся чуть ниже).

Групповая кросс-валидация

На практике могут возникнуть ситуации, когда в таблице появляется понятие **группы**. Это категориальная переменная, которая обладает свойством целостности. Что это означает? Понятнее всего будет показать на примере. Представьте, что у вас есть данные с записями показателя уровня сахара в крови, причем на одного человека приходится по несколько записей. Вы хотите получить модель, которая сможет предсказывать уровень сахара на новых людях. Таким образом, при обучении модели нужно избегать ситуации, когда и в train, и в test попадают записи, соответствующие одному и тому же человеку. В данном примере **группой** записей является человек (например, его id).

Данные стратегии кросс-валидации работают аналогично уже рассмотренным выше методам, поэтому рассмотрим их вкратце.

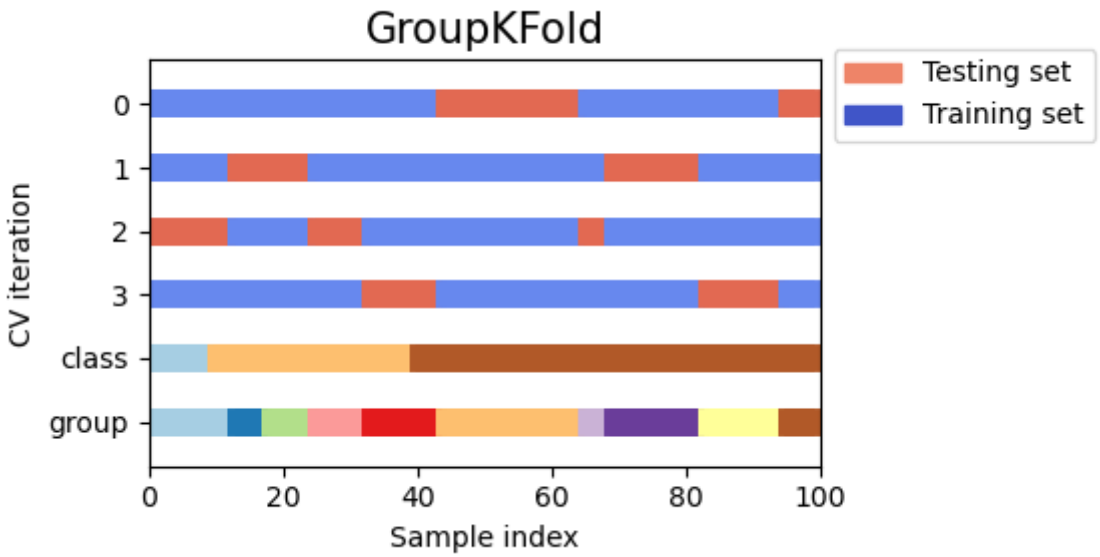
Group KFold

Разбиение на фолды происходит так, что на каждой итерации в train или test нет представителей одной и той же группы. Другими словами, представители одной группы попадают либо train фолды, либо в test фолд.

In [27]:

```
X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 8.8, 9, 10]
y = ["a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]
groups = [1, 1, 1, 2, 2, 2, 3, 3, 3, 3]

gkf = GroupKFold(n_splits=3)

for train, test in gkf.split(X, y, groups=groups):
    print("%s %s" % (train, test))
```

```
[0 1 2 3 4 5] [6 7 8 9]
[0 1 2 6 7 8 9] [3 4 5]
[3 4 5 6 7 8 9] [0 1 2]
```



Leave One Group Out

В данной стратегии на каждой итерации в качестве тестовой выборки используются только **все** представители какой-то одной группы. Таким образом, количество итераций равно количеству групп в данных.

In [28]:

```
X = [1, 5, 10, 50, 60, 70, 80]
y = [0, 1, 1, 2, 2, 2, 2]
groups = [1, 1, 2, 2, 3, 3, 3]
logo = LeaveOneGroupOut()
for train, test in logo.split(X, y, groups=groups):
    print("%s %s" % (train, test))
```

```
[2 3 4 5 6] [0 1]
[0 1 4 5 6] [2 3]
[0 1 2 3] [4 5 6]
```

Leave P Groups Out

В данной стратегии рассматриваются все возможные разбиения на обучающее и тестовое множества, где тестовое множество состоит из P полных групп. Таким образом, это аналог стратегии LeavePOut, оперирующий целыми группами.

In [29]:

```
X = np.arange(6)
y = [1, 1, 1, 2, 2, 2]
groups = [1, 1, 2, 2, 3, 3]
lpgo = LeavePGroupsOut(n_groups=2)
for train, test in lpgo.split(X, y, groups=groups):
    print("%s %s" % (train, test))
```

```
[4 5] [0 1 2 3]
[2 3] [0 1 4 5]
[0 1] [2 3 4 5]
```

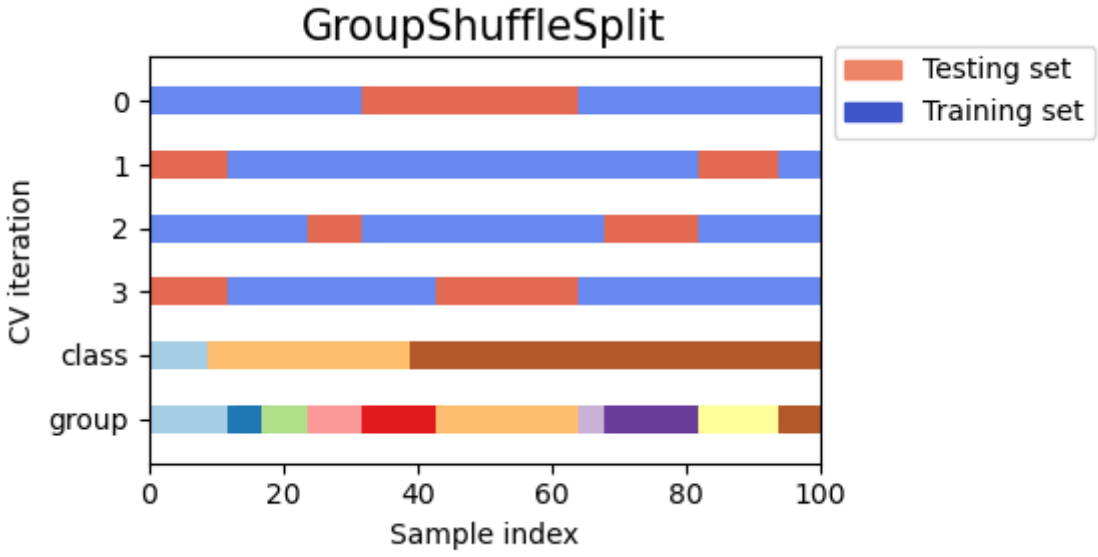
Group Shuffle Split

В данной стратегии на каждой итерации данные разбиваются на train и test случайным образом, но так, чтобы представители одной группы не попадали одновременно и в train, и в test.

In [30]:

```
X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 0.001]
y = ["a", "b", "b", "b", "c", "c", "c", "a"]
groups = [1, 1, 2, 2, 3, 3, 4, 4]
gss = GroupShuffleSplit(n_splits=4, test_size=0.5, random_state=0)
for train, test in gss.split(X, y, groups=groups):
    print("%s %s" % (train, test))
```

```
[0 1 2 3] [4 5 6 7]
[2 3 6 7] [0 1 4 5]
[2 3 4 5] [0 1 6 7]
[4 5 6 7] [0 1 2 3]
```



Итоги

Итак, выше мы рассмотрели множество стратегий кросс-валидации и узнали, в каких случаях какой из них отдавать предпочтение. О том, как использовать кросс-валидацию при настройке гиперпараметров модели, вы узнаете в ноутбуке "Поиск гиперпараметров". В качестве итога можно сказать, что кросс-валидация — это отличный метод для оценки качества вашей модели, потому что при обучении вы можете задействовать все имеющиеся данные. К сожалению, его немаловажным минусом является количество процедур обучения, так как часто требуется большое количество вычислительных мощностей, особенно если вы работаете с большими датасетами.