Lecture 11: Group-based and stratified splitting

By the end of this lecture, you will be able to

- describe the motivation and importance of group-based splitting
- apply various group-based splitting strategies
- apply stratified splitting in a classification problem

The supervised ML pipeline

0. Data collection/manipulation: you might have multiple data sources and/or you might have more data than you need

- you need to be able to read in datasets from various sources (like csv, excel, SQL, parquet, etc)
- you need to be able to filter the columns/rows you need for your ML model
- you need to be able to combine the datasets into one dataframe

1. Exploratory Data Analysis (EDA): you need to understand your data and verify that it doesn't contain errors

- do as much EDA as you can!

2. Split the data into different sets: most often the sets are train, validation, and test (or holdout)

- practitioners often make errors in this step!
- you can split the data randomly, based on groups, based on time, or any other non-standard way if necessary to answer your ML question

3. Preprocess the data: ML models only work if X and Y are numbers! Some ML models additionally require each feature to have 0 mean and 1 standard deviation (standardized features)

- often the original features you get contain strings (for example a gender feature would contain 'male', 'female', 'non-binary', 'unknown') which needs to be transformed into numbers
- often the features are not standardized (e.g., age is between 0 and 100) but it needs to be standardized

4. Choose an evaluation metric: depends on the priorities of the stakeholders

- often requires quite a bit of thinking and ethical considerations

5. Choose one or more ML techniques: it is highly recommended that you try multiple models

- start with simple models like linear or logistic regression
- try also more complex models like nearest neighbors, support vector machines, random forest, etc.

6. Tune the hyperparameters of your ML models (aka cross-validation or hyperparameter tuning)

- ML techniques have hyperparameters that you need to optimize to achieve best performance
- for each ML model, decide which parameters to tune and what values to try
- loop through each parameter combination
  - train one model for each parameter combination
  - evaluate how well the model performs on the validation set
- take the parameter combo that gives the best validation score
- evaluate that model on the test set to report how well the model is expected to perform on previously unseen data

7. Interpret your model: black boxes are often not useful

- check if your model uses features that make sense (excellent tool for debugging)
- often model predictions are not enough, you need to be able to explain how the model arrived to a particular prediction (e.g., in health care)

Recall from Lecture 05

- **the i.i.d. assumption**: the examples in the training set are independently and identically distributed according to D
  - every x_i is freshly sampled from D and then labelled by f
  - that is, x_i and y_i are picked independently of the other instances
  - S is a window through which the learner gets partial info about D and the labeling function f
  - the larger the sample gets, the more likely it is that D and f are accurately reflected
- examples of not iid data:
  - data generated by time-dependent processes
  - data has group structure (samples collected from e.g., different subjects, experiments, measurement devices)
- we will get back to this later in the term
- if there is any sort of time or group structure in your data, it is likely non-iid

    - **time series data**
        - **values are not independent**
        - stocks price
        - covid19 cases
        - weather data
    - **group structure:**
        - **samples are not identically distributed, D might be different for each group**
        - a person appears multiple times in the dataset (e.g., hospital/doctor visits)
        - data is collected on multiple instruments (e.g., equipment failure prediction)
        - geographical data (e.g., data collected about various cities, counties, states, countries)

By the end of this lecture, you will be able to

- **describe the motivation and importance of group-based splitting**
- apply various group-based splitting strategies
- apply stratified splitting in a classification problem

Importance of group based splitting

- iid is often assumed
    - most auto-ML tools assume iid because it makes the problem easy to solve
- one of the most common mistakes data science practitioners make is to assume iid when it is not correct
- consequences:
    - information leakage
    - the model performs very well on the test set (low generalization error)
    - when it is deployed, it performs poorly on new groups
- datasets with group structure have some sort of group ID
    - this can be customer ID, patient ID, instrument ID, sensor ID, etc.
    - the group ID should NOT be used as a feature
    - a unique identifier (often just a random sequence of characters) is not something an ML model can use
    - it should be separated out and used as a group ID in group-based splitting methods (more on this later)
- a categorical/ordinal feature is not usually a group ID!
    - a group ID is usually a column that does not contain info the ML model can use
    - the model can learn from categories

An example: seizure project

- you can read the publication here
- classification problem:
  - epileptic seizures vs. non-epileptic psychogenic seizures
- data from empatica wrist sensor
  - heart rate, skin temperature, EDA, blood volume pressure, acceleration
- data collection:
  - patients come to the hospital for a few days
  - eeg and video recording to determine seizure type
  - wrist sensor data is collected
- question:
  - Can we use the wrist sensor data to differentiate the two seizure types on new patients?

In [1]:

```
import pandas as pd
import numpy as np

df = pd.read_csv('../data/seizure_data.csv')
print(df[df['patient ID'] == 32])
```

```
    patient ID          seizure_ID  ACC_mean  BVP_mean  EDA_mean    HR_mea
n  \
5           32  ID32__day3_arm_1_sz1  1.028539 -0.092102  0.112795  64.74816
7
6           32  ID32__day3_arm_1_sz1  1.027986  0.745437  0.130486  63.71566
7
7           32  ID32__day2_arm_1_sz0  1.002146  0.150810  0.189272  61.83850
0
8           32  ID32__day2_arm_1_sz0  1.005410  0.482859  1.226038  66.24083
3
9           32  ID32__day1_arm_1_sz0  0.997017 -0.925122  0.200990  56.10366
7
10          32  ID32__day1_arm_1_sz0  1.009207  1.618456  1.679754  64.66816
7
27          32  ID32__day1_arm_1_sz0  1.000290  0.046690  0.123165  54.28950
0
28          32  ID32__day1_arm_1_sz0  1.010351  0.125039  0.471180  65.06066
7
29          32  ID32__day2_arm_1_sz0  1.018163  0.254302  0.206010  61.87583
3
30          32  ID32__day2_arm_1_sz0  1.016785  1.242893  0.954649  66.21616
7
34          32  ID32__day3_arm_1_sz1  1.008867  0.070180  0.195966  65.99566
7
35          32  ID32__day3_arm_1_sz1  1.009554  0.222872  0.229909  63.87100
0
58          32  ID32__day3_arm_1_sz0  1.008873 -0.550857  0.177822  67.75083
3
79          32  ID32__day3_arm_1_sz0  1.026840  0.355953  0.205273  69.12466
7

    TEMP_mean  ACC_stdev   BVP_stdev  EDA_stdev  ...  BVP_50th  EDA_50th  \
5   36.944833   0.007469   36.486091   0.003905  ...     1.815  0.112710
6   36.676333   0.028190   84.964155   0.018598  ...     2.210  0.131921
7   38.600333   0.003747   64.194294   0.024278  ...     6.985  0.186026
8   39.296083   0.035257  165.665784   0.891139  ...     1.140  1.062333
9   34.656667   0.022648   77.013336   0.132008  ...     3.800  0.142159
10  34.678000   0.046047  146.515297   0.438236  ...     5.585  1.690537
27  38.467417   0.019826   51.176639   0.014530  ...     7.765  0.124259
28  38.448000   0.077142   61.205657   0.156170  ...     3.290  0.510114
29  37.681583   0.006805   40.982246   0.017099  ...     1.455  0.202632
30  37.979500   0.032493  219.277839   0.612229  ...    -5.785  1.028171
34  40.659458   0.021812   49.981175   0.013259  ...     3.480  0.198570
35  40.481333   0.048531   37.409681   0.031963  ...     0.695  0.228677
58  39.906667   0.021431   27.472002   0.003085  ...     1.955  0.178073
79  34.490167   0.008165   40.742936   0.003550  ...     3.090  0.206207

    HR_50th  TEMP_50th  ACC_75th  BVP_75th  EDA_75th  HR_75th  TEMP_75th  \
5    65.060      36.95  1.029947   16.3725  0.115591  65.8175     36.990
6    62.175      36.81  1.029947   21.1625  0.147611  66.2100     36.840
7    61.840      38.61  1.006085   43.8850  0.209086  61.9000     38.790
8    62.325      39.37  1.008872   49.4325  2.313129  71.0625     39.390
9    56.110      34.66  0.996821   35.2700  0.176739  56.6050     34.660
10   65.790      34.66  1.021497   70.4800  1.998868  67.7725     34.735
27   53.960      38.49  1.002073   39.8525  0.133226  54.7425     38.500
28   65.285      38.45  1.014302   25.4625  0.577047  69.4975     38.530
```

```
29   61.910      37.68  1.022811   29.2125  0.219282  61.9300     37.750
30   64.700      38.00  1.022811   65.5000  1.503002  69.5725     38.030
34   66.145      40.68  1.013700   13.1300  0.199852  67.0425     40.710
35   64.395      40.49  1.016106   12.9650  0.260383  65.9625     40.530
58   68.170      39.93  1.015264   17.8625  0.179354  68.5725     40.030
79   69.810      34.37  1.033260   13.4550  0.207488  70.0000     34.680

    label
5     0.0
6     0.0
7     0.0
8     0.0
9     0.0
10    0.0
27    0.0
28    0.0
29    0.0
30    0.0
34    0.0
35    0.0
58    0.0
79    0.0

[14 rows x 48 columns]
```

In [2]:
```python
y = df['label']
patient_ID = df['patient ID']
seizure_ID = df['seizure_ID']
X = df.drop(columns=['patient ID','seizure_ID','label'])
classes, counts = np.unique(y,return_counts=True)
print(classes, counts)
print('balance:',np.max(counts/len(y)))
```

```
[0. 1.] [ 86 190]
balance: 0.6884057971014492
```

In [3]:
```python
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

def ML_pipeline_kfold_GridSearchCV(X,y,random_state,n_folds):
    # create a test set
    X_other, X_test, y_other, y_test = train_test_split(X, y, test_size=0.2,
    # splitter for _other
    kf = StratifiedKFold(n_splits=n_folds,shuffle=True,random_state=random_s
    # create the pipeline: preprocessor + supervised ML method
    scaler = StandardScaler()
    pipe = make_pipeline(scaler,SVC())
    # the parameter(s) we want to tune
    param_grid = {'svc__C': np.logspace(-3,4,num=8),'svc__gamma': np.logspac
```

```
    # prepare gridsearch
    grid = GridSearchCV(pipe, param_grid=param_grid,scoring = make_scorer(ac
                        cv=kf, return_train_score = True)
    # do kfold CV on _other
    grid.fit(X_other, y_other)
    return grid, grid.score(X_test, y_test)
```

In [4]:
```
test_scores = []
for i in range(5):
    grid, test_score = ML_pipeline_kfold_GridSearchCV(X,y,i*42,5)
    print(grid.best_params_)
    print('best CV score:',grid.best_score_)
    print('test score:',test_score)
    test_scores.append(test_score)
print('test accuracy:',np.around(np.mean(test_scores),2),'+/-',np.around(np.
```

```
{'svc__C': np.float64(100.0), 'svc__gamma': np.float64(0.001)}
best CV score: 0.9363636363636363
test score: 0.8214285714285714
{'svc__C': np.float64(10.0), 'svc__gamma': np.float64(0.01)}
best CV score: 0.9136363636363635
test score: 0.9285714285714286
{'svc__C': np.float64(10.0), 'svc__gamma': np.float64(0.01)}
best CV score: 0.9227272727272726
test score: 0.9464285714285714
{'svc__C': np.float64(10.0), 'svc__gamma': np.float64(0.01)}
best CV score: 0.9318181818181819
test score: 0.8928571428571429
{'svc__C': np.float64(10.0), 'svc__gamma': np.float64(0.001)}
best CV score: 0.9272727272727274
test score: 0.875
test accuracy: 0.89 +/- 0.04
```

This is wrong! A very bad case of data leakage!

- the textbook case of data/information leakage!
- if we just do KFold CV blindly, the points from the same patient end up in different sets
  - when you deploy the model and apply it to data from new patients, that patient's data will be seen for the first time
- the ML pipeline needs to mimic the intended use of the model!
  - we want to split the points based on the patient ID!
  - we want all points from the same patient to be in either train/CV/test

Group-based split: GroupKFold

No description has been provided for this image

In [5]:
```
from sklearn.model_selection import GroupKFold
from sklearn.model_selection import GroupShuffleSplit
```

```
def ML_pipeline_groups_GridSearchCV(X,y,groups,random_state,n_folds):
    # create a test set based on groups
    splitter = GroupShuffleSplit(n_splits=1,test_size=0.2,random_state=rando
    for i_other,i_test in splitter.split(X, y, groups):
        X_other, y_other, groups_other = X.iloc[i_other], y.iloc[i_other], g
        X_test, y_test, groups_test = X.iloc[i_test], y.iloc[i_test], groups
    # check the split
#     print(pd.unique(groups))
#     print(pd.unique(groups_other))
#     print(pd.unique(groups_test))
    # splitter for _other
    kf = GroupKFold(n_splits=n_folds)
    # create the pipeline: preprocessor + supervised ML method
    scaler = StandardScaler()
    pipe = make_pipeline(scaler,SVC())
    # the parameter(s) we want to tune
    param_grid = {'svc__C': np.logspace(-3,4,num=8),'svc__gamma': np.logspac
    # prepare gridsearch
    grid = GridSearchCV(pipe, param_grid=param_grid,scoring = make_scorer(ac
                        cv=kf, return_train_score = True)
    # do kfold CV on _other
    grid.fit(X_other, y_other, groups=groups_other)
    return grid, grid.score(X_test, y_test)
```

In [6]:

```
test_scores = []
for i in range(5):
    grid, test_score = ML_pipeline_groups_GridSearchCV(X,y,patient_ID,i*42,5
    print(grid.best_params_)
    print('best CV score:',grid.best_score_)
    print('test score:',test_score)
    test_scores.append(test_score)
print('test accuracy:',np.around(np.mean(test_scores),2),'+/-',np.around(np.
```

```
{'svc__C': np.float64(10.0), 'svc__gamma': np.float64(0.001)}
best CV score: 0.7609139784946237
test score: 0.6410256410256411
{'svc__C': np.float64(0.1), 'svc__gamma': np.float64(0.01)}
best CV score: 0.6522727272727272
test score: 0.2711864406779661
{'svc__C': np.float64(10.0), 'svc__gamma': np.float64(0.001)}
best CV score: 0.5720073891625616
test score: 0.9390243902439024
{'svc__C': np.float64(10.0), 'svc__gamma': np.float64(0.001)}
best CV score: 0.7061742424242425
test score: 0.43243243243243246
{'svc__C': np.float64(10000.0), 'svc__gamma': np.float64(0.001)}
best CV score: 0.6082407407407406
test score: 0.8901098901098901
test accuracy: 0.63 +/- 0.26
```

The takeaway

- an incorrect cross validation pipeline gives misleading results
    - usually the model appears to be pretty accurate

    - but the performance is poor when the model is deployed
- this can be avoided by a careful cross validation pipeline
    - think about how your model will be used
    - mimic that future use in CV

By the end of this lecture, you will be able to

- describe the motivation and importance of group-based splitting
- **apply various group-based splitting strategies**
- apply stratified splitting in a classification problem

Let's take a look at group splitters using toy datasets

Group-based split: GroupKFold

No description has been provided for this image

In [7]:
```
from sklearn.model_selection import GroupKFold
import numpy as np

X = np.ones(shape=(8, 2))
y = np.ones(shape=(8, 1))
groups = np.array([1, 1, 2, 2, 2, 3, 3, 3])

group_kfold = GroupKFold(n_splits=3)

for train_index, test_index in group_kfold.split(X, y, groups):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
TRAIN: [0 1 2 3 4] TEST: [5 6 7]
TRAIN: [0 1 5 6 7] TEST: [2 3 4]
TRAIN: [2 3 4 5 6 7] TEST: [0 1]
```

In [8]:
```
help(GroupKFold)
```

```
Help on class GroupKFold in module sklearn.model_selection._split:

class GroupKFold(GroupsConsumerMixin, _BaseKFold)
 |  GroupKFold(n_splits=5, *, shuffle=False, random_state=None)
 |
 |  K-fold iterator variant with non-overlapping groups.
 |
 |  Each group will appear exactly once in the test set across all folds (th
e
 |  number of distinct groups has to be at least equal to the number of fold
s).
 |
 |  The folds are approximately balanced in the sense that the number of
 |  samples is approximately the same in each test fold when `shuffle` is Tr
ue.
 |
 |  Read more in the :ref:`User Guide <group_k_fold>`.
 |
 |  For visualisation of cross-validation behaviour and
 |  comparison between common scikit-learn split methods
 |  refer to :ref:`sphx_glr_auto_examples_model_selection_plot_cv_indices.py
`
 |
 |  Parameters
 |  ----------
 |  n_splits : int, default=5
 |      Number of folds. Must be at least 2.
 |
 |      .. versionchanged:: 0.22
 |          ``n_splits`` default value changed from 3 to 5.
 |
 |  shuffle : bool, default=False
 |      Whether to shuffle the groups before splitting into batches.
 |      Note that the samples within each split will not be shuffled.
 |
 |      .. versionadded:: 1.6
 |
 |  random_state : int, RandomState instance or None, default=None
 |      When `shuffle` is True, `random_state` affects the ordering of the
 |      indices, which controls the randomness of each fold. Otherwise, this
 |      parameter has no effect.
 |      Pass an int for reproducible output across multiple function calls.
 |      See :term:`Glossary <random_state>`.
 |
 |      .. versionadded:: 1.6
 |
 |  Notes
 |  -----
 |  Groups appear in an arbitrary order throughout the folds.
 |
 |  Examples
 |  --------
 |  >>> import numpy as np
 |  >>> from sklearn.model_selection import GroupKFold
 |  >>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
 |  >>> y = np.array([1, 2, 3, 4, 5, 6])
```

```
 |  >>> groups = np.array([0, 0, 2, 2, 3, 3])
 |  >>> group_kfold = GroupKFold(n_splits=2)
 |  >>> group_kfold.get_n_splits(X, y, groups)
 |  2
 |  >>> print(group_kfold)
 |  GroupKFold(n_splits=2, random_state=None, shuffle=False)
 |  >>> for i, (train_index, test_index) in enumerate(group_kfold.split(X,
y, groups)):
 |  ...     print(f"Fold {i}:")
 |  ...     print(f"  Train: index={train_index}, group={groups[train_inde
x]}")
 |  ...     print(f"  Test:  index={test_index}, group={groups[test_inde
x]}")
 |  Fold 0:
 |    Train: index=[2 3], group=[2 2]
 |    Test:  index=[0 1 4 5], group=[0 0 3 3]
 |  Fold 1:
 |    Train: index=[0 1 4 5], group=[0 0 3 3]
 |    Test:  index=[2 3], group=[2 2]
 |
 |  See Also
 |  --------
 |  LeaveOneGroupOut : For splitting the data according to explicit
 |      domain-specific stratification of the dataset.
 |
 |  StratifiedKFold : Takes class information into account to avoid building
 |      folds with imbalanced class proportions (for binary or multiclass
 |      classification tasks).
 |
 |  Method resolution order:
 |      GroupKFold
 |      GroupsConsumerMixin
 |      _BaseKFold
 |      BaseCrossValidator
 |      sklearn.utils._metadata_requests._MetadataRequester
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_splits=5, *, shuffle=False, random_state=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  set_split_request(self: sklearn.model_selection._split.GroupKFold, *, gr
oups: Union[bool, NoneType, str] = '$UNCHANGED$') -> sklearn.model_selectio
n._split.GroupKFold from sklearn.utils._metadata_requests.RequestMethod.__ge
t__.<locals>
 |      Request metadata passed to the ``split`` method.
 |
 |      Note that this method is only relevant if
 |      ``enable_metadata_routing=True`` (see :func:`sklearn.set_config`).
 |      Please see :ref:`User Guide <metadata_routing>` on how the routing
 |      mechanism works.
 |
 |      The options for each parameter are:
 |
 |      - ``True``: metadata is requested, and passed to ``split`` if provid
```

```
ed. The request is ignored if metadata is not provided.
 |
 |      - ``False``: metadata is not requested and the meta-estimator will n
ot pass it to ``split``.
 |
 |      - ``None``: metadata is not requested, and the meta-estimator will r
aise an error if the user provides it.
 |
 |      - ``str``: metadata should be passed to the meta-estimator with this
given alias instead of the original name.
 |
 |      The default (``sklearn.utils.metadata_routing.UNCHANGED``) retains t
he
 |      existing request. This allows you to change the request for some
 |      parameters and not others.
 |
 |      .. versionadded:: 1.3
 |
 |      .. note::
 |          This method is only relevant if this estimator is used as a
 |          sub-estimator of a meta-estimator, e.g. used inside a
 |          :class:`~sklearn.pipeline.Pipeline`. Otherwise it has no effect.
 |
 |      Parameters
 |      ----------
 |      groups : str, True, False, or None,                     default=skle
arn.utils.metadata_routing.UNCHANGED
 |          Metadata routing for ``groups`` parameter in ``split``.
 |
 |      Returns
 |      -------
 |      self : object
 |          The updated object.
 |
 |  split(self, X, y=None, groups=None)
 |      Generate indices to split data into training and test set.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Training data, where `n_samples` is the number of samples
 |          and `n_features` is the number of features.
 |
 |      y : array-like of shape (n_samples,), default=None
 |          The target variable for supervised learning problems.
 |
 |      groups : array-like of shape (n_samples,)
 |          Group labels for the samples used while splitting the dataset in
to
 |          train/test set.
 |
 |      Yields
 |      ------
 |      train : ndarray
 |          The training set indices for that split.
 |
```

```
 |      test : ndarray
 |          The testing set indices for that split.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _BaseKFold:
 |
 |  get_n_splits(self, X=None, y=None, groups=None)
 |      Returns the number of splitting iterations in the cross-validator.
 |
 |      Parameters
 |      ----------
 |      X : object
 |          Always ignored, exists for compatibility.
 |
 |      y : object
 |          Always ignored, exists for compatibility.
 |
 |      groups : object
 |          Always ignored, exists for compatibility.
 |
 |      Returns
 |      -------
 |      n_splits : int
 |          Returns the number of splitting iterations in the cross-validato
r.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from BaseCrossValidator:
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.utils._metadata_requests._MetadataRequest
er:
 |
 |  get_metadata_routing(self)
 |      Get metadata routing of this object.
 |
 |      Please check :ref:`User Guide <metadata_routing>` on how the routing
 |      mechanism works.
 |
 |      Returns
 |      -------
 |      routing : MetadataRequest
 |          A :class:`~sklearn.utils.metadata_routing.MetadataRequest` encap
sulating
 |          routing information.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from sklearn.utils._metadata_requests._MetadataR
```

```
equester:
 |
 |  __init_subclass__(**kwargs)
 |      Set the ``set_{method}_request`` methods.
 |
 |      This uses PEP-487 [1]_ to set the ``set_{method}_request`` methods.
It
 |      looks for the information available in the set default values which
are
 |      set using ``__metadata_request__*`` class attributes, or inferred
 |      from method signatures.
 |
 |      The ``__metadata_request__*`` class attributes are used when a metho
d
 |      does not explicitly accept a metadata through its arguments or if th
e
 |      developer would like to specify a request value for those metadata
 |      which are different from the default ``None``.
 |
 |      References
 |      ----------
 |      .. [1] https://www.python.org/dev/peps/pep-0487
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.utils._metadata_requests._Metada
taRequester:
 |
 |  __dict__
 |      dictionary for instance variables
 |
 |  __weakref__
 |      list of weak references to the object
```

Group-based split: GroupShuffleSplit

No description has been provided for this image

In [9]:
```python
from sklearn.model_selection import GroupShuffleSplit

gss = GroupShuffleSplit(n_splits=10, train_size=.8, random_state=0)

for train_idx, test_idx in gss.split(X, y, groups):
    print("TRAIN:", train_idx, "TEST:", test_idx)
```

```
TRAIN: [0 1 2 3 4] TEST: [5 6 7]
TRAIN: [0 1 2 3 4] TEST: [5 6 7]
TRAIN: [2 3 4 5 6 7] TEST: [0 1]
TRAIN: [0 1 2 3 4] TEST: [5 6 7]
TRAIN: [0 1 2 3 4] TEST: [5 6 7]
TRAIN: [0 1 5 6 7] TEST: [2 3 4]
TRAIN: [0 1 5 6 7] TEST: [2 3 4]
TRAIN: [2 3 4 5 6 7] TEST: [0 1]
TRAIN: [2 3 4 5 6 7] TEST: [0 1]
TRAIN: [0 1 5 6 7] TEST: [2 3 4]
```

Quiz 1

Go back to the GroupKFold example above. What happens when you change n_splits to 4? Why?

Why could we set the n_splits argument to 10 in GroupShuffleSplit? Check the manual of both methods to find the answer.

Explain your answer in a couple of sentences!

By the end of this lecture, you will be able to

- describe the motivation and importance of group-based splitting
- apply various group-based splitting strategies
- **apply stratified splitting in a classification problem**

Imbalanced data

- imbalanced data: only a small fraction of the points are in one of the classes, usually ~5% or less but there is no hard limit here
- examples:
    - people visit a bank's website. do they sign up for a new credit card?
        - most customers just browse and leave the page
        - usually 1% or less of the customers get a credit card (class 1), the rest leaves the page without signing up (class 0).
    - fraud detection
        - only a tiny fraction of credit card payments are fraudulent
    - rare disease diagnosis
- the issue with imbalanced data:
    - if you apply train_test_split or KFold, you might not have class 1 points in one of your sets by chance
    - this is what we need to fix

Solution: stratified splits

In [10]:
```
df = pd.read_csv('../data/imbalanced_data.csv')

X = df[['feature1','feature2']]
y = df['y']

print(y.value_counts())
```

```
y
0    990
1     10
Name: count, dtype: int64
```

In [11]:
```
# 4 and 10

random_state = 4

X_train, X_other, y_train, y_other = train_test_split(X,y,train_size = 0.6,r
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_size =

print('**balance without stratification:**')
# a variation on the order of 1% which would be too much for imbalanced data
print(np.unique(y_train,return_counts=True))
print(np.unique(y_val,return_counts=True))
print(np.unique(y_test,return_counts=True))

X_train, X_other, y_train, y_other = train_test_split(X,y,train_size = 0.6,s
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_size =
print('**balance with stratification:**')
# very little variation (in the 4th decimal point only) which is important i
print(np.unique(y_train,return_counts=True))
print(np.unique(y_val,return_counts=True))
print(np.unique(y_test,return_counts=True))
```

```
**balance without stratification:**
(array([0, 1]), array([591,   9]))
(array([0]), array([200]))
(array([0, 1]), array([199,   1]))
**balance with stratification:**
(array([0, 1]), array([594,   6]))
(array([0, 1]), array([198,   2]))
(array([0, 1]), array([198,   2]))
```

Stratified folds

No description has been provided for this image

In [12]:
```
from sklearn.model_selection import StratifiedKFold
help(StratifiedKFold)
```

```
Help on class StratifiedKFold in module sklearn.model_selection._split:

class StratifiedKFold(_BaseKFold)
 |  StratifiedKFold(n_splits=5, *, shuffle=False, random_state=None)
 |
 |  Stratified K-Fold cross-validator.
 |
 |  Provides train/test indices to split data in train/test sets.
 |
 |  This cross-validation object is a variation of KFold that returns
 |  stratified folds. The folds are made by preserving the percentage of
 |  samples for each class.
 |
 |  Read more in the :ref:`User Guide <stratified_k_fold>`.
 |
 |  For visualisation of cross-validation behaviour and
 |  comparison between common scikit-learn split methods
 |  refer to :ref:`sphx_glr_auto_examples_model_selection_plot_cv_indices.py
`
 |
 |  Parameters
 |  ----------
 |  n_splits : int, default=5
 |      Number of folds. Must be at least 2.
 |
 |      .. versionchanged:: 0.22
 |          ``n_splits`` default value changed from 3 to 5.
 |
 |  shuffle : bool, default=False
 |      Whether to shuffle each class's samples before splitting into batche
s.
 |      Note that the samples within each split will not be shuffled.
 |
 |  random_state : int, RandomState instance or None, default=None
 |      When `shuffle` is True, `random_state` affects the ordering of the
 |      indices, which controls the randomness of each fold for each class.
 |      Otherwise, leave `random_state` as `None`.
 |      Pass an int for reproducible output across multiple function calls.
 |      See :term:`Glossary <random_state>`.
 |
 |  Examples
 |  --------
 |  >>> import numpy as np
 |  >>> from sklearn.model_selection import StratifiedKFold
 |  >>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
 |  >>> y = np.array([0, 0, 1, 1])
 |  >>> skf = StratifiedKFold(n_splits=2)
 |  >>> skf.get_n_splits(X, y)
 |  2
 |  >>> print(skf)
 |  StratifiedKFold(n_splits=2, random_state=None, shuffle=False)
 |  >>> for i, (train_index, test_index) in enumerate(skf.split(X, y)):
 |  ...     print(f"Fold {i}:")
 |  ...     print(f"  Train: index={train_index}")
 |  ...     print(f"  Test:  index={test_index}")
 |  Fold 0:
```

```
 |      Train: index=[1 3]
 |      Test:  index=[0 2]
 |    Fold 1:
 |      Train: index=[0 2]
 |      Test:  index=[1 3]
 |
 |  Notes
 |  -----
 |  The implementation is designed to:
 |
 |  * Generate test sets such that all contain the same distribution of
 |    classes, or as close as possible.
 |  * Be invariant to class label: relabelling ``y = ["Happy", "Sad"]`` to
 |    ``y = [1, 0]`` should not change the indices generated.
 |  * Preserve order dependencies in the dataset ordering, when
 |    ``shuffle=False``: all samples from class k in some test set were
 |    contiguous in y, or separated in y by samples from classes other than
k.
 |  * Generate test sets where the smallest and largest differ by at most on
e
 |    sample.
 |
 |  .. versionchanged:: 0.22
 |      The previous implementation did not follow the last constraint.
 |
 |  See Also
 |  --------
 |  RepeatedStratifiedKFold : Repeats Stratified K-Fold n times.
 |
 |  Method resolution order:
 |      StratifiedKFold
 |      _BaseKFold
 |      BaseCrossValidator
 |      sklearn.utils._metadata_requests._MetadataRequester
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_splits=5, *, shuffle=False, random_state=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  split(self, X, y, groups=None)
 |      Generate indices to split data into training and test set.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Training data, where `n_samples` is the number of samples
 |          and `n_features` is the number of features.
 |
 |          Note that providing ``y`` is sufficient to generate the splits a
nd
 |          hence ``np.zeros(n_samples)`` may be used as a placeholder for
 |          ``X`` instead of actual training data.
 |
 |      y : array-like of shape (n_samples,)
```

```
 |          The target variable for supervised learning problems.
 |          Stratification is done based on the y labels.
 |
 |      groups : object
 |          Always ignored, exists for compatibility.
 |
 |      Yields
 |      ------
 |      train : ndarray
 |          The training set indices for that split.
 |
 |      test : ndarray
 |          The testing set indices for that split.
 |
 |      Notes
 |      -----
 |      Randomized CV splitters may return different results for each call o
f
 |      split. You can make the results identical by setting `random_state`
 |      to an integer.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _BaseKFold:
 |
 |  get_n_splits(self, X=None, y=None, groups=None)
 |      Returns the number of splitting iterations in the cross-validator.
 |
 |      Parameters
 |      ----------
 |      X : object
 |          Always ignored, exists for compatibility.
 |
 |      y : object
 |          Always ignored, exists for compatibility.
 |
 |      groups : object
 |          Always ignored, exists for compatibility.
 |
 |      Returns
 |      -------
 |      n_splits : int
 |          Returns the number of splitting iterations in the cross-validato
r.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from BaseCrossValidator:
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
```

```
 |  Methods inherited from sklearn.utils._metadata_requests._MetadataRequest
er:
 |
 |  get_metadata_routing(self)
 |      Get metadata routing of this object.
 |
 |      Please check :ref:`User Guide <metadata_routing>` on how the routing
 |      mechanism works.
 |
 |      Returns
 |      -------
 |      routing : MetadataRequest
 |          A :class:`~sklearn.utils.metadata_routing.MetadataRequest` encap
sulating
 |          routing information.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from sklearn.utils._metadata_requests._MetadataR
equester:
 |
 |  __init_subclass__(**kwargs)
 |      Set the ``set_{method}_request`` methods.
 |
 |      This uses PEP-487 [1]_ to set the ``set_{method}_request`` methods.
It
 |      looks for the information available in the set default values which
are
 |      set using ``__metadata_request__*`` class attributes, or inferred
 |      from method signatures.
 |
 |      The ``__metadata_request__*`` class attributes are used when a metho
d
 |      does not explicitly accept a metadata through its arguments or if th
e
 |      developer would like to specify a request value for those metadata
 |      which are different from the default ``None``.
 |
 |      References
 |      ----------
 |      .. [1] https://www.python.org/dev/peps/pep-0487
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.utils._metadata_requests._Metada
taRequester:
 |
 |  __dict__
 |      dictionary for instance variables
 |
 |  __weakref__
 |      list of weak references to the object
```

In [13]:
```
# what we did before: variance in balance on the order of 1%
random_state = 2

X_other, X_test, y_other, y_test = train_test_split(X,y,test_size = 0.2,ranc
```

```
print('test balance:',np.unique(y_test,return_counts=True))

# do KFold split on other
kf = KFold(n_splits=4,shuffle=True,random_state=random_state)
for train_index, val_index in kf.split(X_other,y_other):
    print('new fold')
    X_train = X_other.iloc[train_index]
    y_train = y_other.iloc[train_index]
    X_val = X_other.iloc[val_index]
    y_val = y_other.iloc[val_index]
    print(np.unique(y_train,return_counts=True))
    print(np.unique(y_val,return_counts=True))
```

```
test balance: (array([0, 1]), array([198,   2]))
new fold
(array([0, 1]), array([596,   4]))
(array([0, 1]), array([196,   4]))
new fold
(array([0, 1]), array([593,   7]))
(array([0, 1]), array([199,   1]))
new fold
(array([0, 1]), array([592,   8]))
(array([0]), array([200]))
new fold
(array([0, 1]), array([595,   5]))
(array([0, 1]), array([197,   3]))
```

In [14]:
```
# stratified K Fold: variation in balance is very small (4th decimal point)
random_state = 42

# stratified train-test split
X_other, X_test, y_other, y_test = train_test_split(X,y,test_size = 0.2,stra
print('test balance:',np.unique(y_test,return_counts=True))

# do StratifiedKFold split on other
kf = StratifiedKFold(n_splits=4,shuffle=True,random_state=random_state)
for train_index, val_index in kf.split(X_other,y_other):
    print('new fold')
    X_train = X_other.iloc[train_index]
    y_train = y_other.iloc[train_index]
    X_val = X_other.iloc[val_index]
    y_val = y_other.iloc[val_index]
    print(np.unique(y_train,return_counts=True))
    print(np.unique(y_val,return_counts=True))
```

```
test balance: (array([0, 1]), array([198,   2]))
new fold
(array([0, 1]), array([594,   6]))
(array([0, 1]), array([198,   2]))
new fold
(array([0, 1]), array([594,   6]))
(array([0, 1]), array([198,   2]))
new fold
(array([0, 1]), array([594,   6]))
(array([0, 1]), array([198,   2]))
new fold
(array([0, 1]), array([594,   6]))
(array([0, 1]), array([198,   2]))
```

Mudcard

In [ ]:

```

```