Mudcard

- **I find it tricky to determine the appropriate number of bins when plotting histograms.**
  - it is tricky. Try a couple of options and see what works best. This is as much of art as science.
- **when to split continuous variables and how to determine cut off points.**
  - why would you want to split continuous variables?
- **When should we convert a continuous variable to a categorical one? How are the cutoffs between the categories decided (e.g. <= 50k and > 50k in the gross income example)?**
  - honestly, I don't know how that cutoff was determined.
  - maybe 50k was the top 10% earner cutoff in 1990?
- **What packages would you recommend for visualizing 3D plots? Or is it not advised to plot data at that dimensionality?**
  - I fiercely HATE 3D plots so I dont have recommendations :)
  - they are clanky and difficult to make sense of at a glance
  - this is just my subjective opinion though, you won't lose points if you create 3D plots in a problem set of report
- **If you have one categorical and one continuous variable, there are multiple suggestions to use: "category-specific histograms, box plot, violin plot", so which among these should be chosen?**
  - your personal choice
  - there is no right or wrong choice here, go with the one you like the most
- **During the exploratory data analysis process, is it best to begin with a scatter matrix and then conduct other visualizations (histogram, boxplot, violin plot, etc.)? Or should one begin with the "simpler" plots first?**
  - the scatter matrix is a good start to make sense of the numerical features but you need other plots for categorical and ordinal features
- **for two categorical axises, isn't stack bar plot the same as the violin plot?**
  - nope, the violin plot shows the distrubtion of continuous features
  - categorical features have no distributions
- **I'm unclear about the difference between violin plots and bar plots.**
  - bar plots show summary stats of the continuous feature like the mean, +/- 1 standard deviation, 1 percentlie and 99 percentile
  - on the other hand, the violin plot shows the distribution of the continuous feature
- **I am not sure why we are setting bins= the squared root of n.**
  - that's just a rule of thumb
  - for distributions that are close to gaussian, bins = sqrt(n) gives a good histogram
- **muddiest part of the lecture is interpreting the scatter matrix that holds 36 different visualizations. Does only compare the 2 axis per visualization or does it**

show a relationship among all visualizations?
  - it contains one scatter plot for each possible feature pair
- **How to figure out those code in a quick way?**
  - I don't know but if you figure out, let me know :)
  - I don't think there is a quick way here, you need to put in the work and the hours
- **I didn't understand what a logspace normal distribution meant**
  - print out that part of the code and read the manual of np.logspace
- **is it necessary to visualize every column/feature before conducting an analysis? What if there are many columns?**
  - it is highly recommended
  - if there are many columns, automate the plotting process
  - loop through each column to prepare plots
- **How did we use plt.semilogx() today? What is it helpful for in the context of the exercise that we have seen in class?**
- **I was a bit confused on the log scale, how will we truly know when it is appropriate in practice?**
- **How do you know when to use transformations like a log scale on your data?**
  - semilogx is helpful if a quantity varies over several orders of magnitudes like the capital gain which can be anything from a few dollars to 100k
- **How do you convert a column from 'object' to a specific datatype?**
  - check on stackoverflow or in the pandas manual
- **If I want to look at two categorical variables (similar to the stacked barplot data with 'race' and 'gross-income') but I have rows where I know the race but the gross income is unknown, should the data be normalized to exclude rows with the missing data or should the bar graph have a separate stack for unknown income?**
  - I'd add a separate unknown income stack

Split iid data

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

- apply basic split to iid datasets
- apply k-fold split to iid datasets
- apply stratified splits to imbalanced data

The supervised ML pipeline

The goal: Use the training data (X and y) to develop a model which can accurately predict the target variable (y_new') for previously unseen data (X_new).

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 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)

- 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)

Why do we split the data?

- we want to find the best hyper-parameters of our ML algorithms
  - fit models to training data
  - evaluate each model on validation set
  - we find hyper-parameter values that optimize the validation score
- we want to know how the model will perform on previously unseen data
  - apply our final model on the test set

We need to split the data into three parts!

Recap from the second lecture

- **the learner's input**

  - Domain set $\mathcal{X}$ - a set of objects we wish to label.
  - Label set $\mathcal{Y}$ - a set of possible labels.
  - Training data $S = ((x_1, y_1), \ldots, (x_m, y_m))$ - a finite sequence of pairs from $\mathcal{X}$, $\mathcal{Y}$. This is what the learner has access to.
    - $X = (x_1, \ldots, x_m)$ is the feature matrix which is usually a 2D matrix, and $Y = (y_1, \ldots, y_m)$ is the target variable which is a vector.
- let's denote the probability distribution over $\mathcal{X}$ by D.

- let's assume there is some correct labeling function $f : \mathcal{X} \rightarrow \mathcal{Y}$.

- **a training example is then generated by sampling x_i from D, and the label y_i is generated using f.**

I.I.D. assumption

- **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 to reflect more accurately D and f
- 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)

Split iid data

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

- **apply basic split to iid datasets**
- apply k-fold split to iid datasets
- apply stratified splits to imbalanced data

Splitting strategies for iid data: basic approach

- 60% train, 20% validation, 20% test for small datasets
- 98% train, 1% validation, 1% test for large datasets
    - if you have 1 million points, you still have 10000 points in validation and test which is plenty to assess model performance

Let's work with the adult data!

In [1]:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

df = pd.read_csv('data/adult_test.csv')

# let's separate the feature matrix X, and target variable y
y = df['gross-income'] # remember, we want to predict who earns more than 50k c
X = df.loc[:, df.columns != 'gross-income'] # all other columns are features
print(y)
print(X.head())
```

```
0        <=50K.
1        <=50K.
2         >50K.
3         >50K.
4        <=50K.
          ...
16276    <=50K.
16277    <=50K.
16278    <=50K.
16279    <=50K.
16280     >50K.
Name: gross-income, Length: 16281, dtype: object
   age   workclass  fnlwgt      education  education-num       marital-status
\
0   25     Private  226802           11th              7        Never-married
1   38     Private   89814        HS-grad              9   Married-civ-spouse
2   28   Local-gov  336951     Assoc-acdm             12   Married-civ-spouse
3   44     Private  160323   Some-college             10   Married-civ-spouse
4   18           ?  103497   Some-college             10        Never-married

            occupation relationship    race      sex  capital-gain  \
0   Machine-op-inspct    Own-child   Black     Male             0
1     Farming-fishing      Husband   White     Male             0
2     Protective-serv      Husband   White     Male             0
3   Machine-op-inspct      Husband   Black     Male          7688
4                   ?    Own-child   White   Female             0

   capital-loss  hours-per-week  native-country
0             0              40   United-States
1             0              50   United-States
2             0              40   United-States
3             0              40   United-States
4             0              30   United-States
```

In [2]:

```python
help(train_test_split)
```

```
Help on function train_test_split in module sklearn.model_selection._split:

train_test_split(*arrays, test_size=None, train_size=None, random_state=None,
shuffle=True, stratify=None)
    Split arrays or matrices into random train and test subsets.

    Quick utility that wraps input validation and
    ``next(ShuffleSplit().split(X, y))`` and application to input data
    into a single call for splitting (and optionally subsampling) data in a
    oneliner.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    ----------
    *arrays : sequence of indexables with same length / shape[0]
        Allowed inputs are lists, numpy arrays, scipy-sparse
        matrices or pandas dataframes.

    test_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the proportion
        of the dataset to include in the test split. If int, represents the
        absolute number of test samples. If None, the value is set to the
        complement of the train size. If ``train_size`` is also None, it will
        be set to 0.25.

    train_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the
        proportion of the dataset to include in the train split. If
        int, represents the absolute number of train samples. If None,
        the value is automatically set to the complement of the test size.

    random_state : int, RandomState instance or None, default=None
        Controls the shuffling applied to the data before applying the split.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    shuffle : bool, default=True
        Whether or not to shuffle the data before splitting. If shuffle=False
        then stratify must be None.

    stratify : array-like, default=None
        If not None, data is split in a stratified fashion, using this as
        the class labels.
        Read more in the :ref:`User Guide <stratification>`.

    Returns
    -------
    splitting : list, length=2 * len(arrays)
        List containing train-test split of inputs.

        .. versionadded:: 0.16
            If the input is sparse, the output will be a
            ``scipy.sparse.csr_matrix``. Else, output type is the same as the
            input type.

    Examples
```

```
--------
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]

>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]

>>> train_test_split(y, shuffle=False)
[[0, 1, 2], [3, 4]]
```

In [3]:

```
random_state = 42

# first split to separate out the training set
X_train, X_other, y_train, y_other = train_test_split(X,y,train_size = 0.6,ranc
print('training set:',X_train.shape, y_train.shape) # 60% of points are in trai
print(X_other.shape, y_other.shape) # 40% of points are in other

# second split to separate out the validation and test sets
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_size = 0.
print('validation set:',X_val.shape, y_val.shape) # 20% of points are in valida
print('test set:',X_test.shape, y_test.shape) # 20% of points are in test

print(X_train.head())
```

```
training set: (9768, 14) (9768,)
(6513, 14) (6513,)
validation set: (3256, 14) (3256,)
test set: (3257, 14) (3257,)
       age         workclass  fnlwgt     education  education-num  \
4050    22           Private  335950       HS-grad              9
11446   29           Private   78261       HS-grad              9
12427   74  Self-emp-not-inc  160009    Assoc-acdm             12
5702    39      Self-emp-inc   31709  Some-college             10
13058   50           Private  144084       HS-grad              9

            marital-status        occupation    relationship    race      sex  \
4050         Never-married     Other-service   Not-in-family   Black     Male
11446            Separated   Protective-serv   Not-in-family   White     Male
12427   Married-civ-spouse   Exec-managerial         Husband   White     Male
5702    Married-civ-spouse      Adm-clerical            Wife   White   Female
13058            Separated             Sales       Unmarried   White   Female

       capital-gain  capital-loss  hours-per-week  native-country
4050              0             0              70   United-States
11446             0             0              55   United-States
12427             0             0              30   United-States
5702              0             0              20   United-States
13058             0             0              55   United-States
```

Randomness due to splitting

- the model performance, validation and test scores will change depending on which points are in train, val, test
  - inherent randomness or uncertainty of the ML pipeline
- change the random state a couple of times and repeat the whole ML pipeline to assess how much the random splitting affects your test score
  - you would expect a similar uncertainty when the model is deployed

Quiz 1

What's the second train_test_split line if you want to end up with 60-20-20 in train-val-test? Print out the sizes of X_train, X_val, X_test to verify!

In [4]:
```
X_other, X_test, y_other, y_test = train_test_split(X,y,train_size = 0.8,randon
# add your line below and chose the correct solution from canvas
```

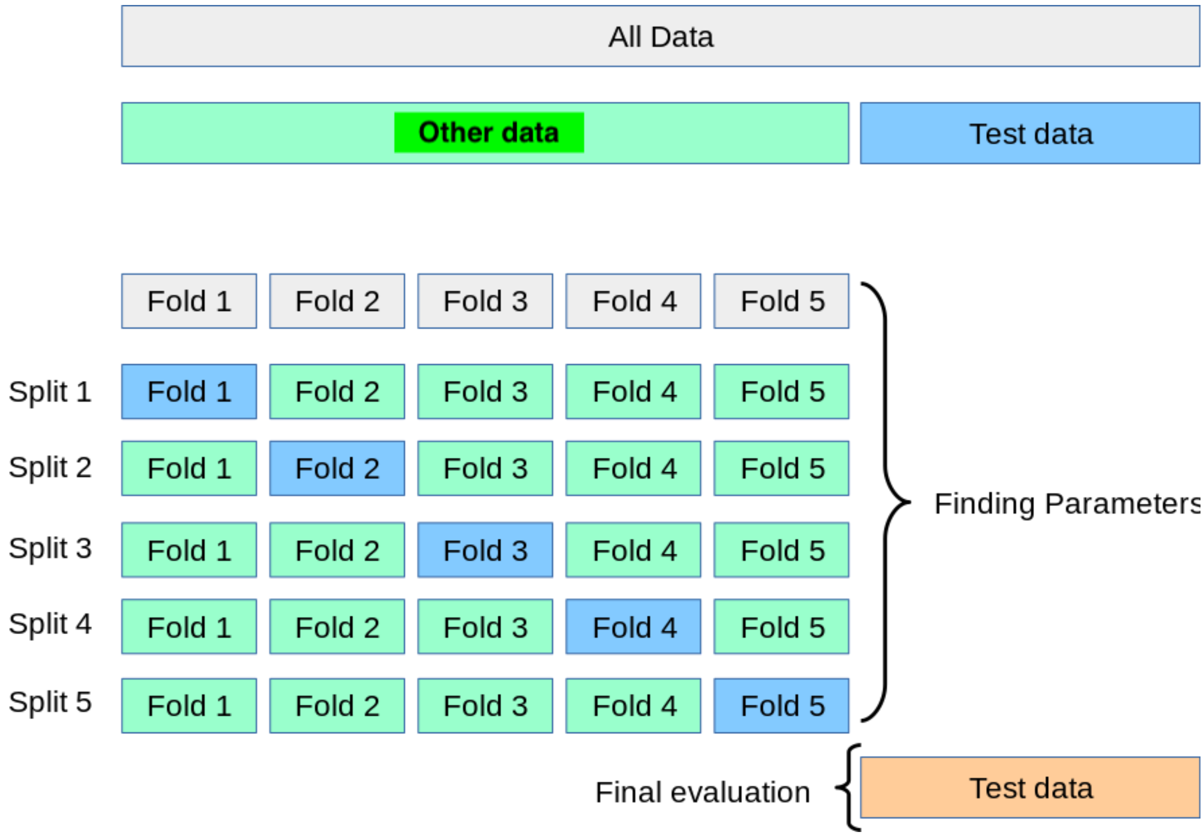
Split iid data

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

- apply basic split to iid datasets
- **apply k-fold split to iid datasets**

- apply stratified splits to imbalanced data

Other splitting strategy for iid data: k-fold splitting



In [5]:
```
from sklearn.model_selection import KFold
help(KFold)
```

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

class KFold(_BaseKFold)
 |  KFold(n_splits=5, *, shuffle=False, random_state=None)
 |
 |  K-Folds cross-validator
 |
 |  Provides train/test indices to split data in train/test sets. Split
 |  dataset into k consecutive folds (without shuffling by default).
 |
 |  Each fold is then used once as a validation while the k - 1 remaining
 |  folds form the training set.
 |
 |  Read more in the :ref:`User Guide <k_fold>`.
 |
 |  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 data before splitting into batches.
 |      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. Otherwise, this
 |      parameter has no effect.
 |      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 KFold
 |  >>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
 |  >>> y = np.array([1, 2, 3, 4])
 |  >>> kf = KFold(n_splits=2)
 |  >>> kf.get_n_splits(X)
 |  2
 |  >>> print(kf)
 |  KFold(n_splits=2, random_state=None, shuffle=False)
 |  >>> for train_index, test_index in kf.split(X):
 |  ...     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 3] TEST: [0 1]
 |  TRAIN: [0 1] TEST: [2 3]
 |
 |  Notes
 |  -----
 |  The first ``n_samples % n_splits`` folds have size
 |  ``n_samples // n_splits + 1``, other folds have size
 |  ``n_samples // n_splits``, where ``n_samples`` is the number of samples.
```

```
 |
 |  Randomized CV splitters may return different results for each call of
 |  split. You can make the results identical by setting `random_state`
 |  to an integer.
 |
 |  See Also
 |  --------
 |  StratifiedKFold : Takes group information into account to avoid building
 |      folds with imbalanced class distributions (for binary or multiclass
 |      classification tasks).
 |
 |  GroupKFold : K-fold iterator variant with non-overlapping groups.
 |
 |  RepeatedKFold : Repeats K-Fold n times.
 |
 |  Method resolution order:
 |      KFold
 |      _BaseKFold
 |      BaseCrossValidator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_splits=5, *, shuffle=False, random_state=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  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-validator.
 |
 |  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,), default=None
 |          Group labels for the samples used while splitting the dataset into
 |          train/test set.
 |
 |      Yields
 |      ------
 |      train : ndarray
 |          The training set indices for that split.
 |
 |      test : ndarray
 |          The testing set indices for that split.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from BaseCrossValidator:
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from BaseCrossValidator:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

In [6]:

```python
random_state = 42

# first split to separate out the test set
X_other, X_test, y_other, y_test = train_test_split(X,y,test_size = 0.2,random_
print(X_other.shape,y_other.shape)
print('test set:',X_test.shape,y_test.shape)

# do KFold split on other
kf = KFold(n_splits=5,shuffle=True,random_state=random_state)
for train_index, val_index in kf.split(X_other,y_other):
    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('   training set:',X_train.shape, y_train.shape)
    print('   validation set:',X_val.shape, y_val.shape)
    # the validation set contains different points in each iteration
    print(X_val[['age','workclass','education']].head())
```

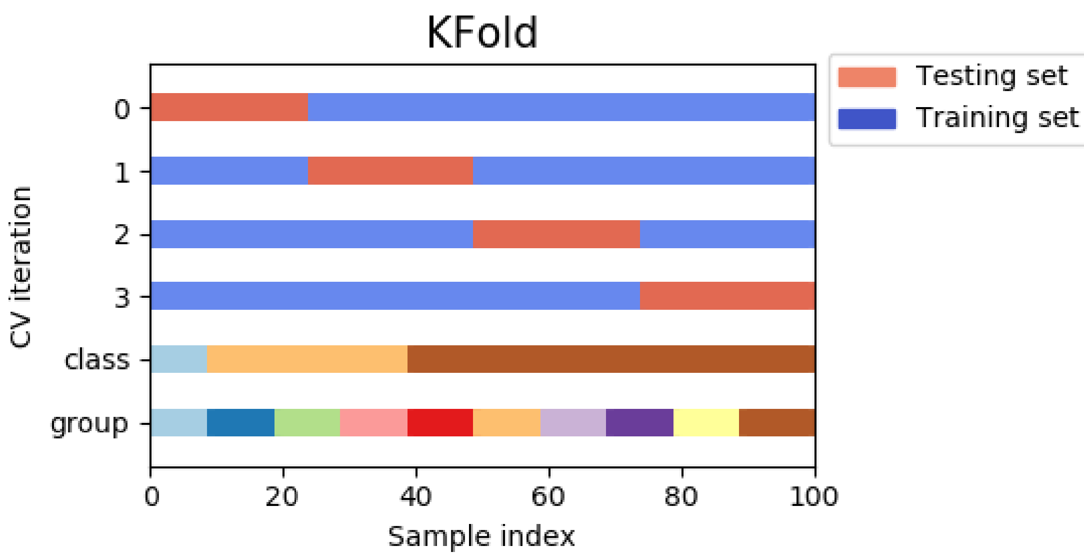
```
(13024, 14) (13024,)
test set: (3257, 14) (3257,)
   training set: (10419, 14) (10419,)
   validation set: (2605, 14) (2605,)
       age          workclass      education
9850    59            Private   Some-college
103     58   Self-emp-not-inc            9th
1383    45            Private        HS-grad
11034   49   Self-emp-not-inc      Bachelors
14876   59   Self-emp-not-inc      Bachelors
   training set: (10419, 14) (10419,)
   validation set: (2605, 14) (2605,)
       age     workclass      education
13384   60   Federal-gov      Bachelors
8471    20       Private        HS-grad
13406   21             ?   Some-college
13394   35       Private        HS-grad
15123   38       Private   Some-college
   training set: (10419, 14) (10419,)
   validation set: (2605, 14) (2605,)
       age     workclass      education
647     60             ?      Bachelors
9314    26       Private   Some-college
14499   52       Private        HS-grad
7332    53   Federal-gov     Assoc-acdm
12523   21       Private           10th
   training set: (10419, 14) (10419,)
   validation set: (2605, 14) (2605,)
       age workclass      education
5294    53   Private        HS-grad
3481    41   Private        HS-grad
7671    49   Private   Some-college
11055   39   Private      Bachelors
12751   18         ?           12th
   training set: (10420, 14) (10420,)
   validation set: (2604, 14) (2604,)
       age      workclass     education
4265    23              ?          10th
5290    23        Private       HS-grad
1157    56   Self-emp-inc   Prof-school
12344   18        Private          11th
13683   55        Private       HS-grad
```

How many splits should I create?

- tough question, 3-5 is most common
- if you do n splits, n models will be trained, so the larger the n, the most computationally intensive it will be to train the models
- KFold is usually better suited to small datasets
- KFold is good to estimate uncertainty due to random splitting of train and val, but it is not perfect
    - the test set remains the same

Why shuffling iid data is important?

- by default, data is not shuffled by Kfold which can introduce errors!



Quiz 2

Given the labels below, what are the balances of each class?

y = [0,0,0,2,2,0,0,2,0,1]

Split iid data

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

- apply basic split to iid datasets
- apply k-fold split to iid datasets
- **apply stratified splits to imbalanced data**

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

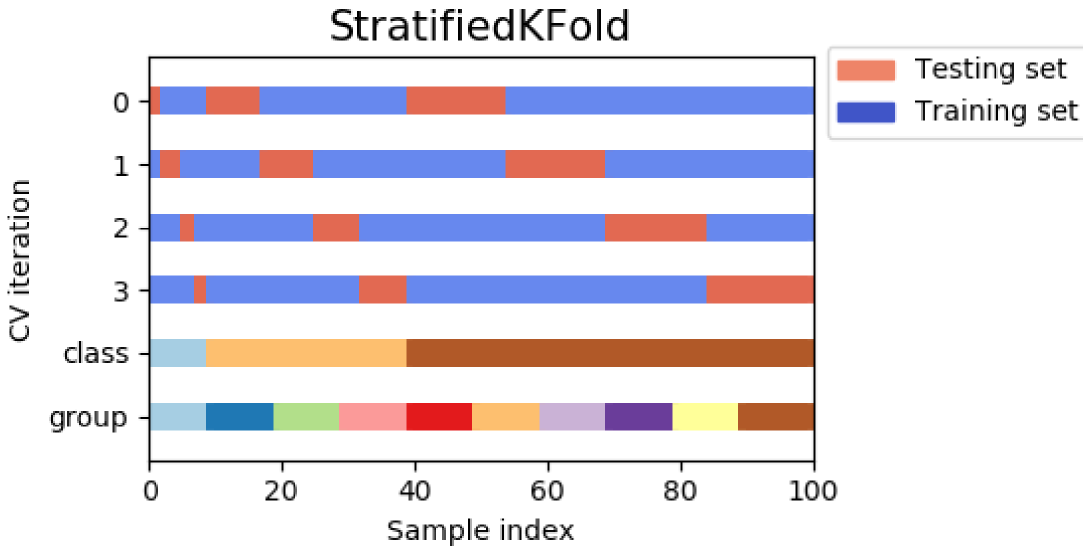
In [7]:
```
df = pd.read_csv('data/imbalanced_data.csv')

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

print(y.value_counts())
```

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

In [8]:
```
# 4 and 10

random_state = 10

X_train, X_other, y_train, y_other = train_test_split(X,y,train_size = 0.6,ranc
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_size = 0.

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,stra
X_val, X_test, y_val, y_test = train_test_split(X_other,y_other,train_size = 0.
print('**balance with stratification:**')
# very little variation (in the 4th decimal point only) which is important if 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([593,   7]))
(array([0, 1]), array([197,   3]))
(array([0]), array([200]))
**balance with stratification:**
(array([0, 1]), array([594,   6]))
(array([0, 1]), array([198,   2]))
(array([0, 1]), array([198,   2]))
```

Stratified folds



In [9]:
```
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-Folds 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>`.
 |
 |  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 batches.
 |      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 train_index, test_index in skf.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: [1 3] TEST: [0 2]
 |  TRAIN: [0 2] TEST: [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 one
 |      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
 |        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 and
 |            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 of
 |      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-validator.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from BaseCrossValidator:
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from BaseCrossValidator:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

In [10]:

```
# 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,random_
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 [11]:

```
# 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,stratif
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 [ ]: