Mud card

- **Can you talk about the variance/bias relationship one more time?**
  - overfitting / high variance model / low bias model
    - the model is too complex
    - it performs very well on the training data (aka overfits the training data)
    - but it performs poorly on new data points
  - underfitting / low variance model / high bias model
    - the model is too simple
    - it performs poorly on the training data (aka underfits)
    - and it also performs poorly on new data points

- **Can we go over how to properly split your data again?**
  - today and tomorrow :)

Cross Validation with iid data

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

- apply simple CV and k-fold CV to datasets
- use GridSearchCV with pipelines
- apply stratified splits to imbalanced data

The goals of cross validation

- we want to find the best hyper-parameters of our ML algorithms
  - fit model to training data (`.fit(X_train,y_train)`)
  - evaluate model on CV set (`.predict(X_CV,y_CV)`)
  - we find hyper-parameter values that optimize the CV score
- we want to know how the model will perform on previously unseen data
  - apply our final model on the test set (`.predict(X_test,y_test)`)

We need to split the data into three parts!

How should we split the data into train/CV/test?

- data is **Independent and Identically Distributed** (iid)
  - all samples stem from the same generative process and that the generative process is assumed to have no memory of past generated samples
  - identify cats and dogs on images
  - predict the house price
  - predict if someone's salary is above or below 50k
- examples of not iid data (more on this next time):
  - data generated by time-dependent processes
  - data has group structure (samples collected from e.g., different subjects, experiments, measurement devices)

CV steps of iid data to avoid mistakes

- shuffle and split the data
- preprocess (fit_transform train, transform the rest)
- decide on the evaluation metric
- decide ML algo, which hyper-parameters you tune, and what values you want to try
- loop over all combinations and save train and CV scores
- find best model based on optimal CV score
- report test score using the best model
- repeat a couple of times with different random states to estimate uncertainty

Cross Validation with iid data

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

- **apply simple CV and k-fold CV to datasets**
- use GridSearchCV and pipelines
- apply stratified splits to imbalanced data

Splitting strategies for iid data: basic approach

- the basic aproach:
  - 60% train, 20% CV, 20% test
  - the ratios can vary somewhat but the training set should contain most of your points
  - if you redo the split with a different random state, the results will change
    - repeat the split a couple of times to measure model uncertainty due to splitting

Let's put everything together!

In [1]:
```
import numpy as np
np.random.seed(10)
def true_fun(X):
    return np.cos(1.5 * np.pi * X)

n_samples = 100

X = np.random.rand(n_samples)
y = true_fun(X) + np.random.randn(n_samples) * 0.1
```

In [2]:
```
def ML_pipeline_basic(X,y,random_state):
    # split the data
    X_other, X_test, y_other, y_test = train_test_split(X, y, test_size=
0.2, random_state = random_state)
    X_train, X_CV, y_train, y_CV = train_test_split(X_other, y_other, te
st_size=0.25, random_state = random_state)
    # simple preprocessing
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_CV = scaler.transform(X_CV)
    X_test = scaler.transform(X_test)
    # tune ridge hyper-parameter, alpha
    alpha = np.logspace(-3,4,num=8)
    train_score = []
    CV_score = []
    regs = []
    for a in alpha:
        reg = Ridge(alpha = a)
        reg.fit(X_train,y_train)
        train_score.append(mean_squared_error(y_train,reg.predict(X_trai
n)))
        CV_score.append(mean_squared_error(y_CV,reg.predict(X_CV)))
        regs.append(reg)
    # find the best alpha
    best_alpha = alpha[np.argmin(CV_score)]
    # grab the best model
    reg = regs[np.argmin(CV_score)]
    # calculate holdout score
    test_score = mean_squared_error(y_test,reg.predict(X_test))
    return best_alpha,np.min(CV_score),test_score
```

In [3]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import matplotlib

CV_scores = []
test_scores = []
for i in range(10):
    best_alpha, CV_score, test_score = ML_pipeline_basic(X[:, np.newaxis
],y, i*42 )
    CV_scores.append(CV_score)
    test_scores.append(test_score)

print('CV MSE:',np.around(np.mean(CV_scores),2),'+/-',np.around(np.std(C
V_scores),2))
print('test MSE:',np.around(np.mean(test_scores),2),'+/-',np.around(np.s
td(test_scores),2))
```

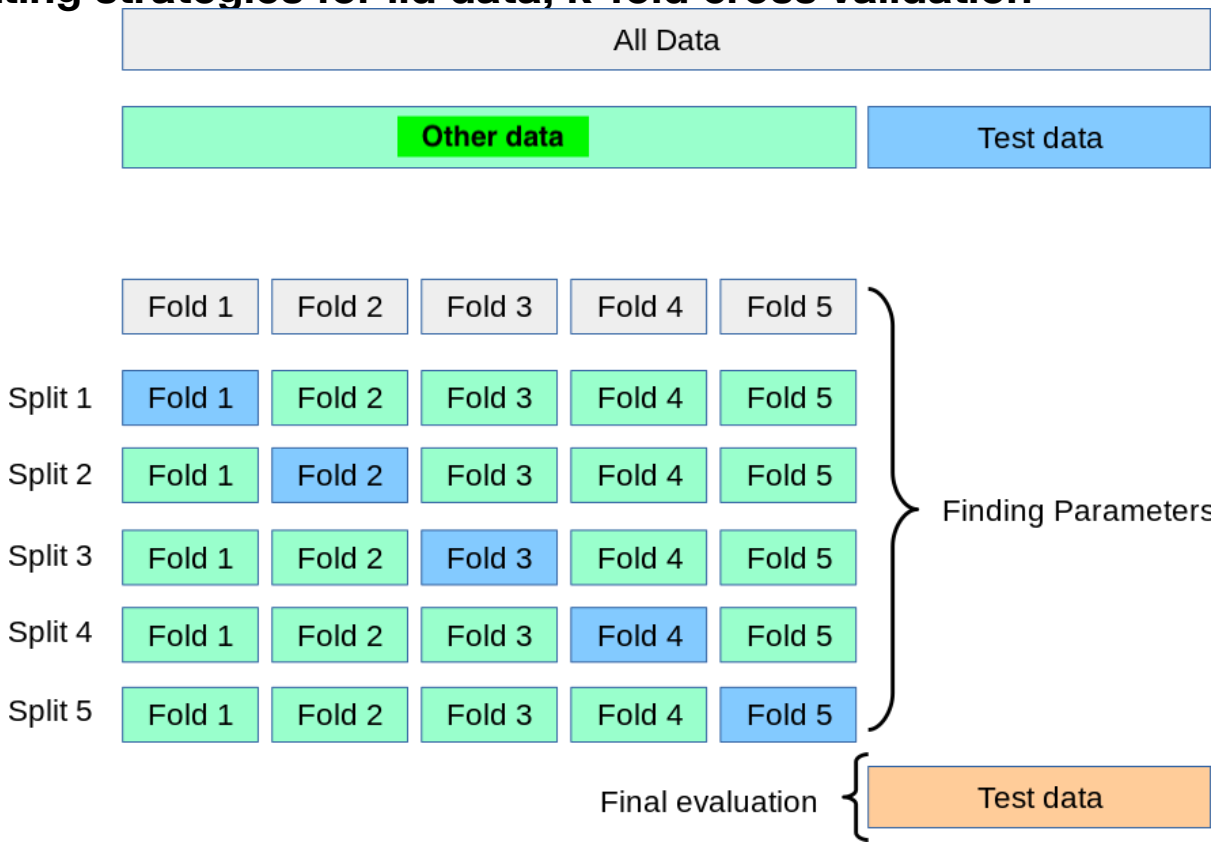
```
CV MSE: 0.18 +/- 0.04
test MSE: 0.21 +/- 0.05
```

Exercise 1

Add a couple of lines of code to `ML_pipeline_basic` to plot the train and the test scores as a function of alpha. Add x and y labels and also a legend.

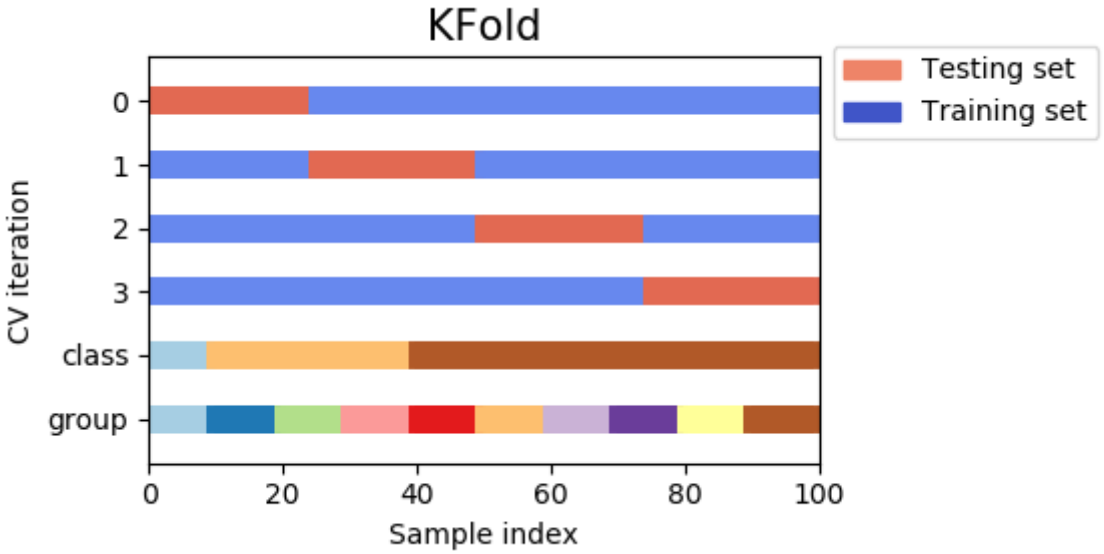
In [4]:
```

```

Splitting strategies for iid data, k-fold cross validation



Why shuffling iid data is important?

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



In [5]:
```
def ML_pipeline_kfold(X,y,random_state,n_folds):
    # split the data
    X_other, X_test, y_other, y_test = train_test_split(X, y, test_size=
0.2, random_state = random_state)
    CV_scores = []
    test_scores = []
    # k folds - each fold will give us a CV and a test score
    kf = KFold(n_splits=n_folds,shuffle=True,random_state=random_state)
    for train_index, CV_index in kf.split(X_other,y_other):
        X_train, X_CV = X_other[train_index], X_other[CV_index]
        y_train, y_CV = y_other[train_index], y_other[CV_index]
        # simple preprocessing
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_c = scaler.transform(X_CV)
        X_t = scaler.transform(X_test)
        # tune ridge hyper-parameter, alpha
        alpha = np.logspace(-5,2,num=8)
        train_score = []
        CV_score = []
        regs = []
        for a in alpha:
            reg = Ridge(alpha = a)
            reg.fit(X_train,y_train)
            train_score.append(mean_squared_error(y_train,reg.predict(X_
train)))
            CV_score.append(mean_squared_error(y_CV,reg.predict(X_c)))
            regs.append(reg)
        # find the best alpha in this fold
        best_alpha = alpha[np.argmin(CV_score)]
        # grab the best model
        reg = regs[np.argmin(CV_score)]
        CV_scores.append(np.min(CV_score))
        # calculate test score using thee best model
        test_scores.append(mean_squared_error(y_test,reg.predict(X_t)))
    return CV_scores,test_scores
```

In [6]:
```
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
import matplotlib

CV_scores, test_scores = ML_pipeline_kfold(X[:,np.newaxis],y,42,5)

print('CV MSE:',np.around(np.mean(CV_scores),2),'+/-',np.around(np.std(C
V_scores),2))
print('test MSE:',np.around(np.mean(test_scores),3),'+/-',np.around(np.s
td(test_scores),3))
```

```
CV MSE: 0.18 +/- 0.04
test MSE: 0.163 +/- 0.003
```

Some considerations

- 1) lots of lines of code were written, mistakes can be easily made!
- 2) kfold CV uses the same test set, so we do not estimate the uncertainty from random test sets
  - test score uncertainty is lower than in the basic approach
- 3) both approaches (basic and kfold) can fail if the data is imbalanced
  - if one class is infrequent, it can happen that one set or one fold contains 0 points from the rare class
  - sklearn will raise an error in that case
- 4) neither of these approaches work, if data is not iid!

Cross Validation with iid data

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

- apply simple CV and k-fold CV to datasets
- **use GridSearchCV and pipelines**
- apply stratified splits to imbalanced data

1) Let's shorten our code: GridSearchCV and pipeline in k-fold CV

In [7]:
```
def ML_pipeline_kfold(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, random_state = random_state)
    # splitter for _other
    kf = KFold(n_splits=n_folds,shuffle=True,random_state=random_state)
    # create the pipeline: preprocessor + supervised ML method
    scaler = StandardScaler()
    pipe = make_pipeline(scaler,Ridge())
    # the parameter(s) we want to tune
    param_grid = {'ridge__alpha': np.logspace(-3,4,num=8)}
    # prepare gridsearch
    grid = GridSearchCV(pipe, param_grid=param_grid,scoring = make_score
r(mean_squared_error,greater_is_better=False),
                        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 [8]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.metrics import make_scorer
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib

grid, test_score = ML_pipeline_kfold(X[:,np.newaxis],y,42,5)
results = pd.DataFrame(grid.cv_results_)
print('CV MSE:',-np.around(results[results['rank_test_score'] == 1]['mean_test_score'].values[0],2),\
      '+/-',np.around(results[results['rank_test_score'] == 1]['std_test_score'].values[0],2))
print('test MSE:',-np.around(test_score,2))
results
```

```
CV MSE: 0.19 +/- 0.03
test MSE: 0.16
```

Out[8]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_ridge__alpha	parar
0	0.000682	0.000119	0.000241	0.000034	0.001	{'ridge__alph 0.0(
1	0.000678	0.000067	0.000238	0.000024	0.01	{'ridge__alph 0.(
2	0.000609	0.000006	0.000227	0.000002	0.1	{'ridge__alph 0
3	0.000623	0.000019	0.000229	0.000002	1	{'ridge__alph 1
4	0.000611	0.000012	0.000229	0.000007	10	{'ridge__alph 10
5	0.000630	0.000054	0.000229	0.000014	100	{'ridge__alph 100
6	0.000718	0.000172	0.000245	0.000050	1000	{'ridge__alph 1000
7	0.000620	0.000024	0.000223	0.000002	10000	{'ridge__alph 10000

8 rows × 21 columns

Some notable differences

- if multiple parameters give an equally good CV score, GridSearchCV returns the largest
  - my function returns the smallest
  - it's unclear which one is better
- GridSearchCV calculates only one test score
  - my function returns n_folds scores
  - the new approach refits the best model to _other and that model is used to calculate the test score
  - it's unclear which one is better
    - my approach allows to calculate some uncertainty due to splitting (not on test)
    - the GridSearchCV approach returns one test score but it is based on more data (likely more accurate)
- 7 lines of code in GridSearchCV
  - 28 lines of code in my function

2) Estimate the uncertainty from random test sets

Exercise 2

Calculate the test score for 10 different random splits. What's the mean and std test score?

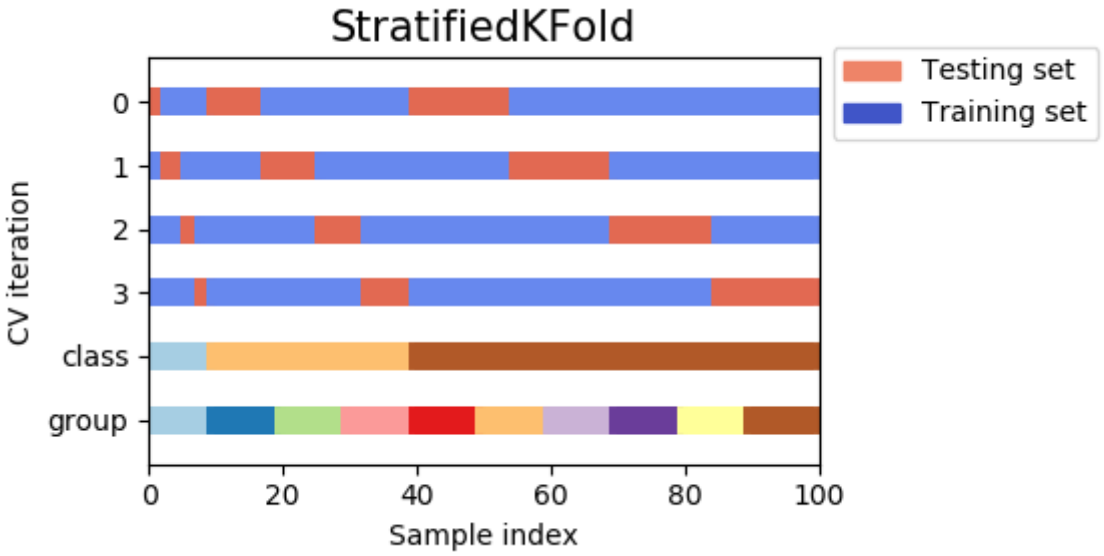
In [ ]:

Cross Validation with iid data

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

- apply simple CV and k-fold CV to datasets
- use GridSearchCV and pipelines
- **apply stratified splits to imbalanced data**

3) Imbalanced data: use stratified folds



In [10]:
```
from sklearn.model_selection import StratifiedKFold
help(StratifiedKFold)
```

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

class StratifiedKFold(_BaseKFold)
 |  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 o
f
 |  samples for each class.
 |
 |  Read more in the :ref:`User Guide <cross_validation>`.
 |
 |  Parameters
 |  ----------
 |  n_splits : int, default=3
 |      Number of folds. Must be at least 2.
 |
 |      .. versionchanged:: 0.20
 |          ``n_splits`` default value will change from 3 to 5 in v0.2
2.
 |
 |  shuffle : boolean, optional
 |      Whether to shuffle each class's samples before splitting into b
atches.
 |
 |  random_state : int, RandomState instance or None, optional, default
=None
 |      If int, random_state is the seed used by the random number gene
rator;
 |      If RandomState instance, random_state is the random number gene
rator;
 |      If None, the random number generator is the RandomState instanc
e used
 |      by `np.random`. Used when ``shuffle`` == True.
 |
 |  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)  # doctest: +NORMALIZE_WHITESPACE
 |  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
 |  -----
```

```
 |  Train and test sizes may be different in each fold, with a differen
ce of at
 |  most ``n_classes``.
 |
 |  See also
 |  --------
 |  RepeatedStratifiedKFold: Repeats Stratified K-Fold n times.
 |
 |  Method resolution order:
 |      StratifiedKFold
 |      _BaseKFold
 |      BaseCrossValidator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_splits='warn', 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, 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 spl
its and
 |          hence ``np.zeros(n_samples)`` may be used as a placeholder
for
 |          ``X`` instead of actual training data.
 |
 |      y : array-like, 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 c
all of
 |      split. You can make the results identical by setting ``random_s
tate``
 |      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-validat
or
 |
 |      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-val
idator.
 |
 |  ----------------------------------------------------------------------
---
 |  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)
```

Now you can

- apply simple CV and k-fold CV to datasets
- use GridSearchCV with pipelines
- apply stratified splits to imbalanced data

In [ ]: