Supplementary Lecture: Data Splitting

이번 강의에서는 기계학습에서 주어진 데이터로부터 실험 설계를 하는 방법에 대한 내용을 소개합니다. 이 방법들을 사용하면 우리가 설계한 신경회로망이 학습에 사용되지 않은 미관측 데이터에 대한 성능을 비교적 정확히 예측할 수 있게 해 주며, 이에 따라 실제 환경에서의 성능을 예측할 수 있게 해 줍니다. 또한 이 방법들은 신경회로망 뿐만이 아니라 다른 기계학습 방법에도 별다른 수정없이 적용가능합니다.

Copyrights

> ELEC801 Pattern Recognition, Fall 2020, Gil-Jin Jang
>
> Lecture 4 Overfitting Avoidance

Customized by Gil-Jin Jang, April 14, 2021

목차

> ## Required Packages
>
> python 3.7 or higher, sys, os, numpy, sklearn,

Part 1. Holdout Validation

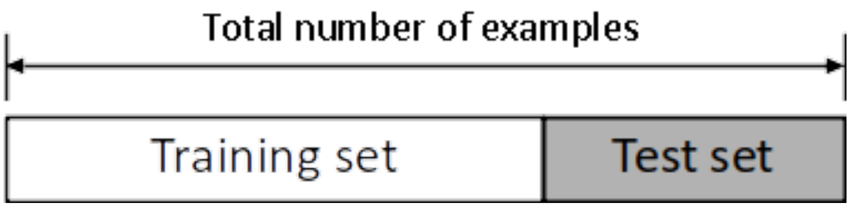
1. `Basic holdout split`
2. `Per-class holdout split`
3. `Holdout split using random sampling`
4. `Holdout split using reproducible random sampling`
5. `Holdout split using stratified random sampling`

Holdout: Overfitting

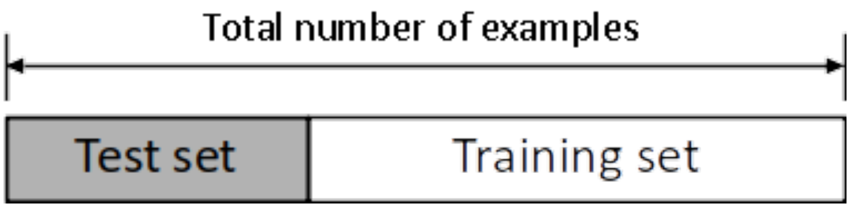
- One may be tempted to use the entire training data to select the ``optimal'' classifier, then estimate the error rate
- This naive approach has two fundamental problems
  - The final model will normally **overfit** to the training data; it often occurs 100\% correct classification on training data
  - Then the model will not be able to generalize to new data
  - The problem of overfitting is more often with models that have a large number of parameters
  - The error rate estimate is overly optimistic (lower than the true error rate)
- The techniques presented in this lecture will allow you to make the best use of your (limited) data for
  1. Training
  2. Model selection and
  3. Performance estimation

The Holdout Method

- Split dataset into two groups
  - **Training set:** used to train the classifier
  - **Test set:** used to estimate the error rate of the trained classifier



(a) Test set first

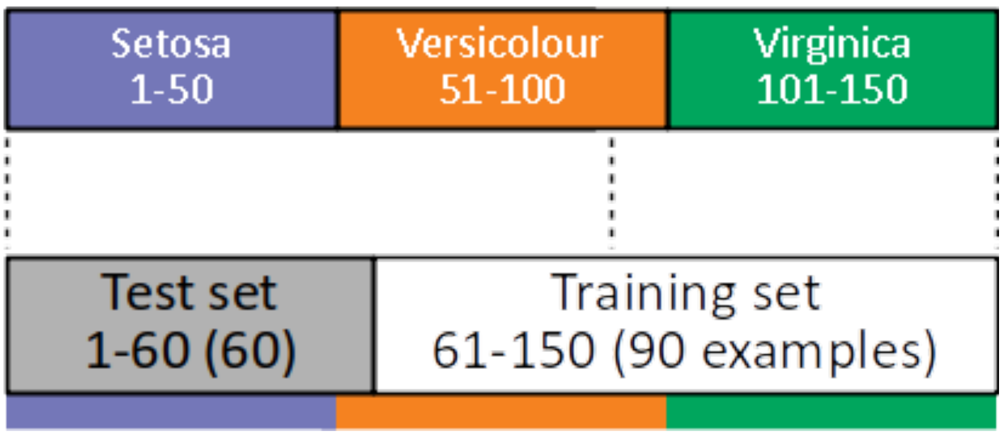


(b) Train set first

- The holdout method has two basic drawbacks

1. In problems where we have a scarce dataset we may not be able to afford to set aside a portion of the dataset for testing
2. Since it is a single train-and-test experiment, the holdout estimate of error rate will be misleading for an **unfortunate** split

HO1: Incorrect Holdout Split



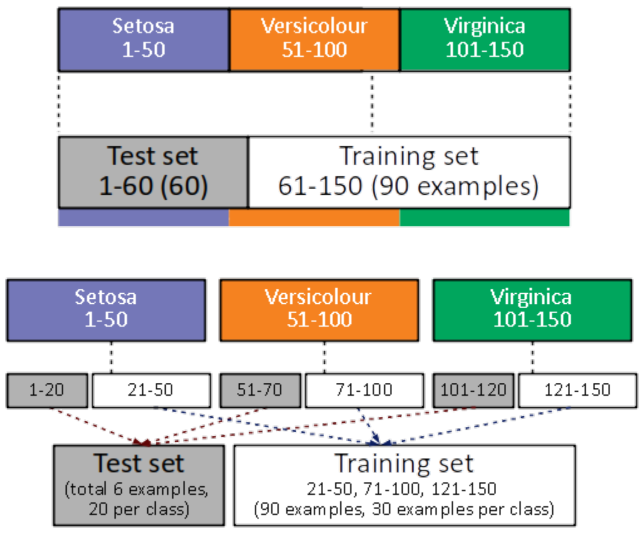
Incorrect holdout split example on Iris dataset. The data are split by test:train = 2:3. There is no Setosa examples in the training set, so the trained model cannot classify Setosa examples. On the contracy, most examples of the test set belong to Setosa class, so the test result will be biased too much.

In [ ]:
```
# two_layer_net.py
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict


class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_s
td = 0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size,
hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size,
output_size)
        self.params['b2'] = np.zeros(output_size)

        # 계층 생성
        self.layers = OrderedDict()
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1
'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2
'])

        self.lastLayer = SoftmaxWithLoss()

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    # x : 입력 데이터, t : 정답 레이블
    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x : 입력 데이터, t : 정답 레이블
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        # forward
        self.loss(x, t)

        # backward
        dout = 1
        dout = self.lastLayer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 결과 저장
        grads = {}
        grads['W1'], grads['b1'] = self.layers['Affine1'].dW, self.layers
['Affine1'].db
        grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers
['Affine2'].db

        return grads
```

In [ ]:
```
####################################
# modified from train_neuralnet.py
import sys, os
sys.path.append(os.pardir)
import numpy as np

def train_neuralnet_iris(x_train, t_train, x_test, t_test,
                         input_size=4, hidden_size=10, output_size=3,
                         iters_num = 1000, batch_size = 10, learning_rate
= 0.1,
                         verbose=True):

    network = TwoLayerNet(input_size, hidden_size, output_size)

    # Train Parameters
    train_size = x_train.shape[0]
    iter_per_epoch = max(train_size / batch_size, 1)

    train_loss_list, train_acc_list, test_acc_list = [], [], []

    for step in range(1, iters_num+1):
        # get mini-batch
        batch_mask = np.random.choice(train_size, batch_size)
        x_batch = x_train[batch_mask]
        t_batch = t_train[batch_mask]

        # 기울기 계산
        #grad = network.numerical_gradient(x_batch, t_batch) # 수치 미분 방식
        grad = network.gradient(x_batch, t_batch) # 오차역전파법 방식(압도적으로
빠르다)

        # Update
        for key in ('W1', 'b1', 'W2', 'b2'):
            network.params[key] -= learning_rate * grad[key]

        # loss
        loss = network.loss(x_batch, t_batch)
        train_loss_list.append(loss)

        if verbose and step % iter_per_epoch == 0:
            train_acc = network.accuracy(x_train, t_train)
            test_acc = network.accuracy(x_test, t_test)
            train_acc_list.append(train_acc)
            test_acc_list.append(test_acc)
            print('Step: {:4d}\tTrain acc: {:.5f}\tTest acc: {:.5f}'.forma
t(step,

train_acc,

test_acc))
    tracc, teacc = network.accuracy(x_train, t_train), network.accuracy(x_
test, t_test)
    if verbose:
        print('Optimization finished!')
        print('Training accuracy: %.2f' % tracc)
        print('Test accuracy: %.2f' % teacc)
    return tracc, teacc
```

```
In [ ]: import numpy as np
        from sklearn import datasets
        from sklearn.naive_bayes import GaussianNB

        iris = datasets.load_iris(); nsamples = iris.data.shape[0]
        ntestsamples = nsamples * 4 // 10  # `//' is integer division
        ntrainsamples = nsamples - ntestsamples   # 4:6 test:train split
        testidx = range(0,ntestsamples); trainidx = range(ntestsamples,nsamples)

        train_neuralnet_iris(iris.data[trainidx,:], iris.target[trainidx],
                             iris.data[testidx], iris.target[testidx],
                             input_size=4, hidden_size=10, output_size=3,
                             iters_num = 1000, batch_size = 10, learning_rate = 0.
        1)

        # Although only 1 Verisicolour example out of 10 is mislabeled.
        # All 50 Setosa examples are mislabeled because there is no Setosa class i
        n the model
        ####################################


        """'
        # from sklearn.naive_bayes import GaussianNB

        gnb = GaussianNB()
        gnb.fit(iris.data[trainidx,:], iris.target[trainidx])
        y_pred = gnb.predict(iris.data[testidx,:])
        nmisses = (iris.target[testidx] != y_pred).sum()
        print('Number of mislabeled out of a total %d samples : %d (%.2f%%)'
                % (len(testidx), nmisses, float(nmisses)/len(testidx)*100.0))
        # [Execution result]
        # Number of mislabeled out of a total 60 samples : 51 (85.00%)
        ''
        """
```

HO2: Per-class Holdout Split

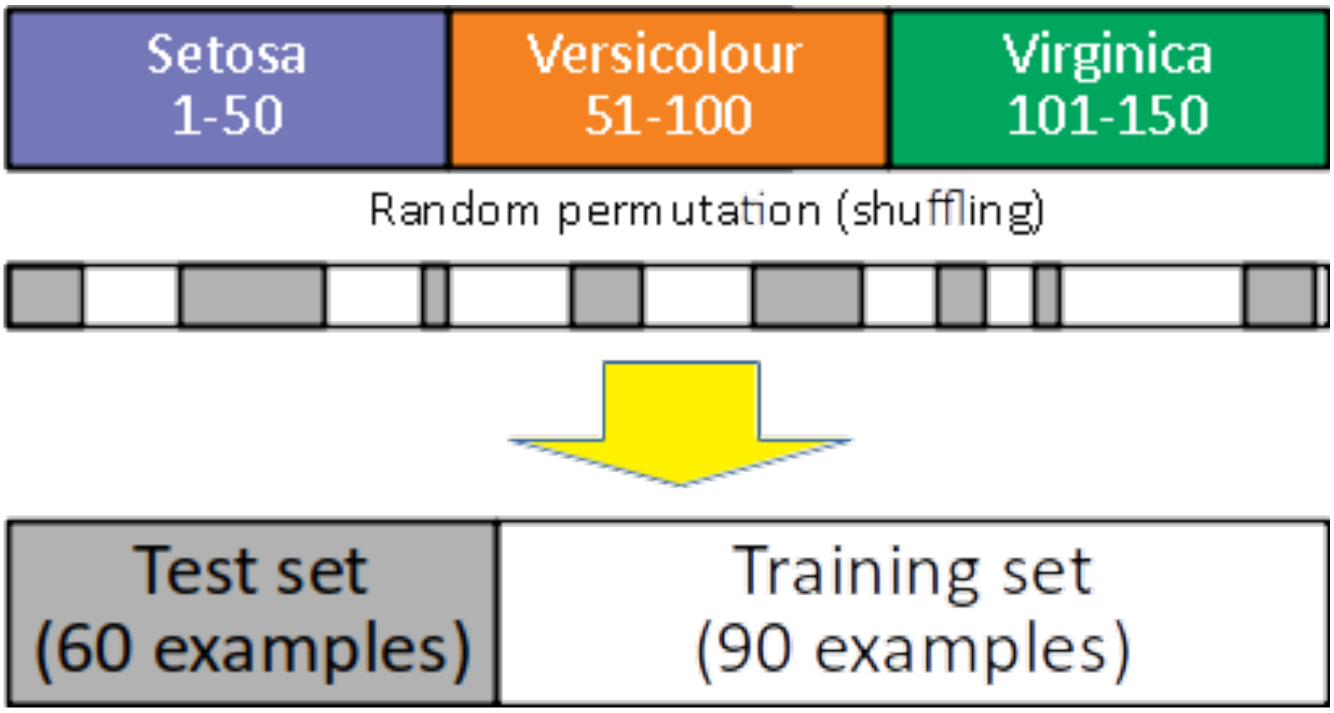


> Per-class holdout split example on Iris dataset. The data are split by test:train = 2:3, and applied to 50 samples per class. Test set is the union of the test sets from individual classes, and training set is the union of the 3 training sets. Equal number of examples are chosen for each of the classes so that the test result will not be biased.

- The splitting should be carefully designed so that at least 1 example for each of training and test set
- As an extreme case, when there are only 2 examples, it is reasonable to assign one for each of train and test sets

In [ ]:
```
# modified from: http://scikit-learn.org/stable/modules/naive_bayes.html
import numpy as np
from sklearn import datasets

iris = datasets.load_iris()
# 4:6 test:train split
ntestsamples = len(iris.target) * 4 // 10  # '//' integer division
ntestperclass = ntestsamples // 3

# allocate indices for test and training data
# Bte: boolean index for test data;  ~Bte: logical not, for training data
Bte = np.zeros(len(iris.target),dtype=bool)   # initially, False index
for c in range(0,3): Bte[range(c*50,c*50+ntestperclass)] = True

train_neuralnet_iris(iris.data[~Bte,:], iris.target[~Bte],
                     iris.data[Bte,:], iris.target[Bte],
                     input_size=4, hidden_size=10, output_size=3,
                     iters_num = 1000, batch_size = 10, learning_rate = 0.
1)


'''
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
gnb.fit(iris.data[~Bte,:], iris.target[~Bte])
y_pred = gnb.predict(iris.data[Bte,:])
nmisses = (iris.target[Bte] != y_pred).sum()
print('Number of mislabeled out of a total %d samples : %d (%.2f%%)'
        % (sum(Bte), nmisses, float(nmisses)/sum(Bte)*100.0))

#[Execution]
#Number of mislabeled out of a total 60 samples : 3 (5.00%)
'''
```

HO3: Holdout Split by Random Sampling



Holdout split using random sampling Iris dataset. The data are split by test:train = 2:3. 60 examples are selected randomly for the test set, and 90 for the training set.

- Each data split randomly selects a (fixed) number of examples **without replacement** --- *each sample has only one chance to be selected*
- Drawbacks:
    1. due to random selection, the performance may vary for different executions
    2. the number of examples per class may not be balanced

In [ ]:
```
# modified from http://scikit-learn.org/stable/modules/naive_bayes.html
import numpy as np
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB

iris = datasets.load_iris()
nsamples = iris.data.shape[0]
ntestsamples = nsamples * 4 // 10  # 4:6 test:train split
# random permutation (shuffling)
Irand = np.random.permutation(nsamples)
Ite = Irand[range(0,ntestsamples)]
Itr = Irand[range(ntestsamples,nsamples)]

train_neuralnet_iris(iris.data[Itr,:], iris.target[Itr],
                     iris.data[Ite,:], iris.target[Ite],
                     input_size=4, hidden_size=10, output_size=3,
                     iters_num = 1000, batch_size = 10, learning_rate = 0.
1)

'''
# training and testing
gnb = GaussianNB().fit(iris.data[Itr,:], iris.target[Itr])
y_pred = gnb.predict(iris.data[Ite,:])
nmisses = (iris.target[Ite] != y_pred).sum()
print('Number of mislabeled out of a total %d samples : %d (%.2f%%)'
        % (sum(Ite), nmisses, float(nmisses)/sum(Ite)*100.0))

# [Multiple Executions]
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
# Number of mislabeled out of a total 60 samples : 4 (6.67%)
# Number of mislabeled out of a total 60 samples : 0 (0.00%)
# Number of mislabeled out of a total 60 samples : 5 (8.33%)
# Number of mislabeled out of a total 60 samples : 1 (1.67%)
'''
```

sklearn.model_selection.train_test_split

`sklearn.model_selection.train_test_split(*arrays, **options)`

- Split arrays or matrices into random train and test subsets
- Parameters

*arrays	sequence of indexables with same length / shape[0]
test_size	`float` $\in [0.0, 1.0]$ or `int`, default=`None`
train_size	`float` $\in [0.0, 1.0]$ or `int`, default=`None`
shuffle	`bool`, default=`True`
random_state	`int` or `RandomState` instance, default=`None`
stratify	array-like, default=`None`

- Returns

splitting	list, length=2*len(arrays) of train and test sets

```
In [ ]: # Random sampling by sklearn.model_selection.train_test_split
        # source: https://scikit-learn.org/stable/modules/cross_validation.html
        import numpy as np
        from sklearn import datasets
        from sklearn.naive_bayes import GaussianNB
        from sklearn.model_selection import train_test_split

        # We can now quickly sample a training set while holding out
        # 40% of the data for testing (evaluating) our classifier:
        X, y = datasets.load_iris(return_X_y=True)
        Xtr,Xte,ytr,yte = train_test_split(X, y, test_size=0.4, shuffle=True)

        train_neuralnet_iris(Xtr,ytr,Xte,yte,
                             input_size=4, hidden_size=10, output_size=3,
                             iters_num = 1000, batch_size = 10, learning_rate = 0.
        1)


        '''
        # training and testing
        y_pred = GaussianNB().fit(Xtr, ytr).predict(Xte)
        nmisses = (yte != y_pred).sum()
        print('Number of mislabeled out of a total %d samples : %d (%.2f%%)'
                % (len(yte), nmisses, float(nmisses)/len(yte)*100.0))

        # [Multiple Executions]
        # Number of mislabeled out of a total 60 samples : 5 (8.33%)
        # Number of mislabeled out of a total 60 samples : 4 (6.67%)
        # Number of mislabeled out of a total 60 samples : 2 (3.33%)
        # Number of mislabeled out of a total 60 samples : 6 (10.00%)
        # Number of mislabeled out of a total 60 samples : 1 (1.67%)
        '''
```

Improving Random Sampling Holdout

- Ⓐ **due to random selection, the performance may vary for different executions**
- Ⓑ *the number of examples per class may not be balanced*

`sklearn.model_selection.train_test_split(*arrays, **options)`

- Parameters

*arrays	sequence of indexables with same length / shape[0]
test_size	`float` ∈ [0.0, 1.0] or `int`, default=`None`
train_size	`float` ∈ [0.0, 1.0] or `int`, default=`None`
shuffle	`bool`, default=`True`
random_state	**`int` or `RandomState` instance, default=`None`**
stratify	array-like, default=None

- Returns

splitting	list, length=2*len(arrays) of train and test sets

In [ ]:
```
# HO4: Reproducible Random Sampling
# Random sampling by sklearn.model_selection.train_test_split
# source: https://scikit-learn.org/stable/modules/cross_validation.html

from sklearn.model_selection import train_test_split

X, y = datasets.load_iris(return_X_y=True)
Xtr,Xte,ytr,yte = train_test_split(X, y, test_size=0.4, shuffle=True, rand
om_state=len(y))

# fix the SEED of random permutation to be the number of samples,
# to reproduce the same random sequence at every execution
np.random.seed(len(y))

train_neuralnet_iris(Xtr,ytr,Xte,yte,
                     input_size=4, hidden_size=10, output_size=3,
                     iters_num = 1000, batch_size = 10, learning_rate = 0.
1)

""" # training and testing
y_pred = GaussianNB().fit(Xtr, ytr).predict(Xte)
nmisses = (yte != y_pred).sum()
print('Number of mislabeled out of a total %d samples : %d (%.2f%%)'
        % (len(yte), nmisses, float(nmisses)/len(yte)*100.0))

# [Multiple Executions]
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
"""
```

Improving Random Sampling Holdout (2)

Ⓐ *due to random selection, the performance may vary for different executions*

Ⓑ **the number of examples per class may not be balanced**

`sklearn.model_selection.train_test_split(*arrays, **options)`

- Parameters

*arrays	sequence of indexables with same length / shape[0]
test_size	`float` $\in [0.0, 1.0]$ or `int`, default=`None`
train_size	`float` $\in [0.0, 1.0]$ or `int`, default=`None`
shuffle	`bool`, default=`True`
random_state	`int` or `RandomState` instance, default=`None`
stratify	**array-like, default=**`None`

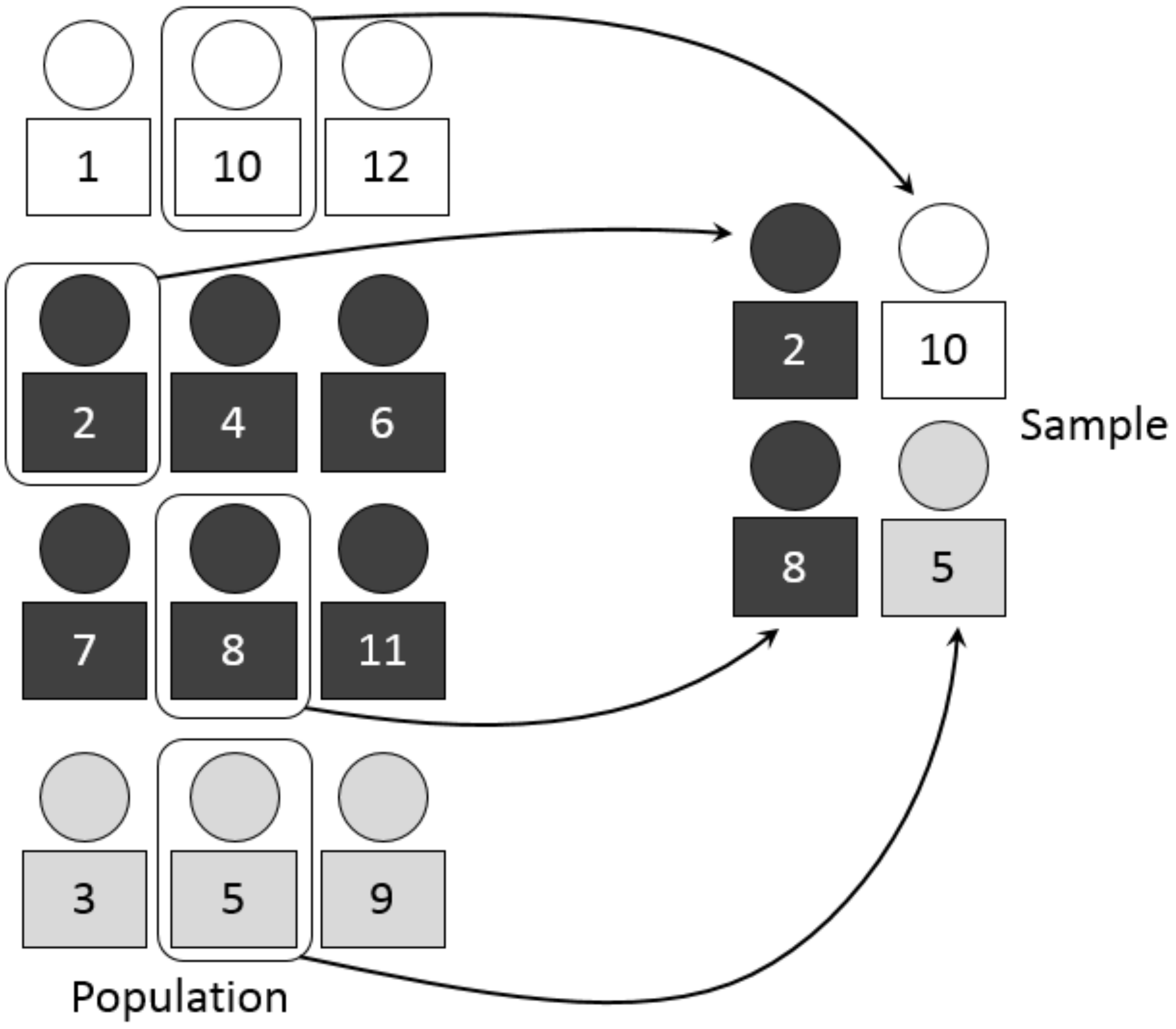
- Returns

splitting	list, length=2*len(arrays) of train and test sets

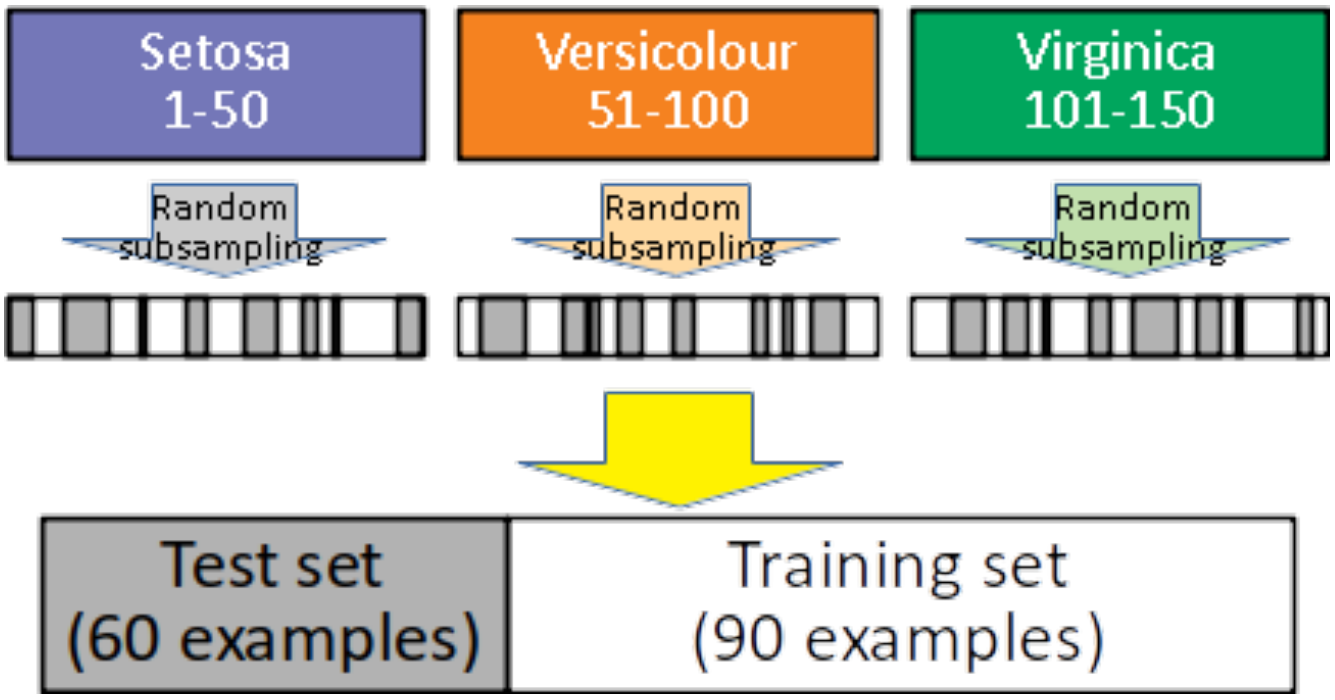
Stratified Random Sampling

From (Wikipedia)[https://en.wikipedia.org/wiki/Stratified_sampling (https://en.wikipedia.org/wiki/Stratified_sampling)]

- A method of sampling from a population which can be partitioned into subpopulations.
- When subpopulations within an overall population vary, it could be advantageous to sample each subpopulation (stratum) independently.
- Stratification is the process of dividing members of the population into homogeneous subgroups before sampling.
- The ratio of each category in the sample space is retained in the sub-sampled space.



HO5: Stratified Random Sampling



In [ ]:
```
# HO5: Stratified Random Sampling}
from sklearn.model_selection import train_test_split
X, y = datasets.load_iris(return_X_y=True)

# per-class random sampling by passing y to variable stratify,
Xtr,Xte,ytr,yte = train_test_split(X, y, test_size=0.4, shuffle=True, stra
tify=y)

# check number of samples of the individual classes
print('test: %d %d %d,  '%(sum(yte==0),sum(yte==1),sum(yte==2)),end='')
print('training: %d %d %d'%(sum(ytr==0),sum(ytr==1),sum(ytr==2)))

# due to the random initialization of the weights, the performance varies
# so we have to set the random seed for TwoLayerNet's initialization value
s
np.random.seed(len(y))

train_neuralnet_iris(Xtr,ytr,Xte,yte,
                     input_size=4, hidden_size=10, output_size=3,
                     iters_num = 1000, batch_size = 10, learning_rate = 0.
1)

"""# training and testing
y_pred = GaussianNB().fit(Xtr, ytr).predict(Xte)
nmisses = (yte != y_pred).sum()
print('Number of mislabeled out of a total %d samples : %d (%.2f%%)'
        % (len(yte), nmisses, float(nmisses)/len(yte)*100.0))

# [Multiple Executions]
# test: 20 20 20,  training: 30 30 30
# Number of mislabeled out of a total 60 samples : 6 (10.00%)
# test: 20 20 20,  training: 30 30 30
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
# test: 20 20 20,  training: 30 30 30
# Number of mislabeled out of a total 60 samples : 4 (6.67%)
# test: 20 20 20,  training: 30 30 30
# Number of mislabeled out of a total 60 samples : 3 (5.00%)
"""
```

Part 2. Cross Validation

```
1. k-fold Cross validation
2. Leave-one-out validation
3. Repeated random subsampling
```

The Limitation of Holdout Method

- The training and test data are fixed in the holdout, so the measured performance is highly dependent on the choice of the test set
- The limitations of the holdout, especially with the lack of training data, can be overcome with a family of resampling methods at the expense of higher computational cost
- Types of validation methods
  1. K-fold crossvalidation (KFCV)
  2. Leave-one-out cross-validation (LOOCV)
  3. Repeated random subsampling

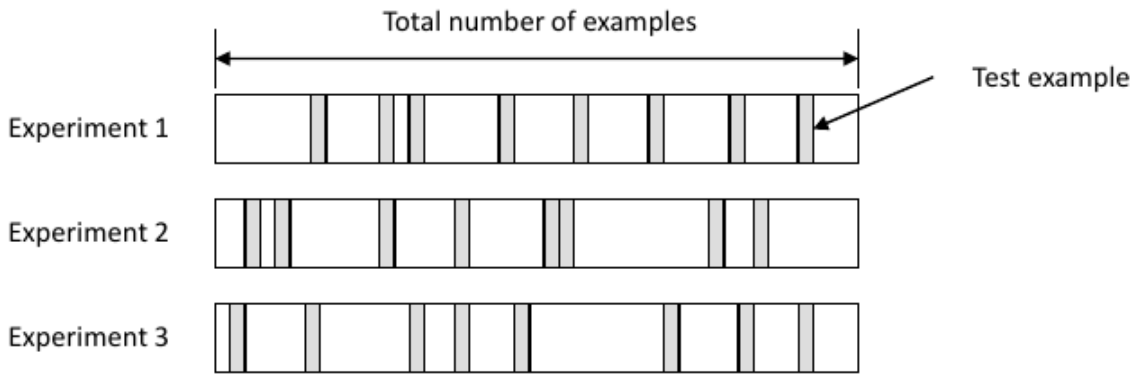
Random Subsampling: K Data Splits

Performing K data splits of the entire dataset

- Each data split randomly selects a (fixed) number of examples *without replacement* --- *each sample has only one chance to be selected*
- For each data split we retrain the classifier from scratch with the (unselected) training examples and then estimate the error, E_i, on the (selected) test examples
- The true error estimate is obtained as the average of the separate estimates

$$E_i\$:\$E = \frac{1}{K}\sum_{i=1}^{K} E_i$$

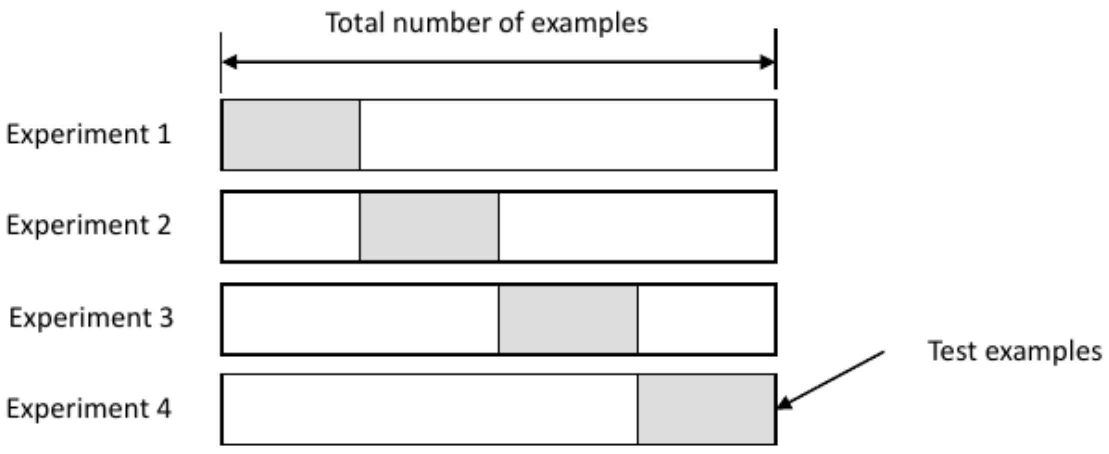
- This estimate is significantly better than the holdout estimate



In [ ]:
```
# Repeated Random Subsampling
# Repeating stratified random sampling K times

from sklearn.model_selection import train_test_split
X, y = datasets.load_iris(return_X_y=True)

# due to the random initialization of the weights, the performance varies
# so we have to set the random seed for TwoLayerNet's initialization value
s
np.random.seed(len(y))

K = 20
Acc = np.zeros([K,2], dtype=float)
for k in range(K):
    # stratified random sampling
    Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.4, shuffle=Tru
e, random_state=None, stratify=y)
    Acc[k,0], Acc[k,1] = train_neuralnet_iris(Xtr,ytr,Xte,yte,
                                  input_size=4, hidden_size=10, output_siz
e=3,
                                  iters_num = 1000, batch_size = 10, learn
ing_rate = 0.1,
                                  verbose = False)
    print('Trial %d: accuracy %.3f %.3f' % (k, Acc[k,0], Acc[k,1]))

# 20 trials, average mislabeled out of a total 60 samples : 2.6 (4.42%)
# 20 trials, average mislabeled out of a total 60 samples : 2.8 (4.67%)
# 20 trials, average mislabeled out of a total 60 samples : 2.6 (4.33%)
# 20 trials, average mislabeled out of a total 60 samples : 2.8 (4.58%)
# 20 trials, average mislabeled out of a total 60 samples : 3.0 (5.08%)
# 20 trials, average mislabeled out of a total 60 samples : 2.7 (4.50%)
```

K-fold Cross Validation (KFCV)

- Create a K-fold partition of the dataset
  - For each of K experiments, use $K-1$ folds for training and the remaining fold for testing
- K-Fold cross validation is similar to random subsampling
  - The advantage of KFCV is that all the examples in the dataset are eventually used for both training and testing
  - The true error is estimated as the average error rate on test examples: $E = \frac{1}{K}\sum_{i=1}^{K} E_i$



How many folds are needed?

- With a large number of folds
  - $(+)$ The bias of the true error rate estimator is small (accurate estimator)
  - $(-)$ The variance of the true error rate estimator is large
  - $(-)$ The computational time is very large as well (many experiments)
- With a small number of folds
  - $(+)$ The number of experiments and, therefore, computation time are reduced
  - $(+)$ The variance of the estimator is small
  - $(-)$ The bias of the estimator is large (larger than the true error rate) \end{itemize}
- In practice, the choice for K depends on the size of the dataset

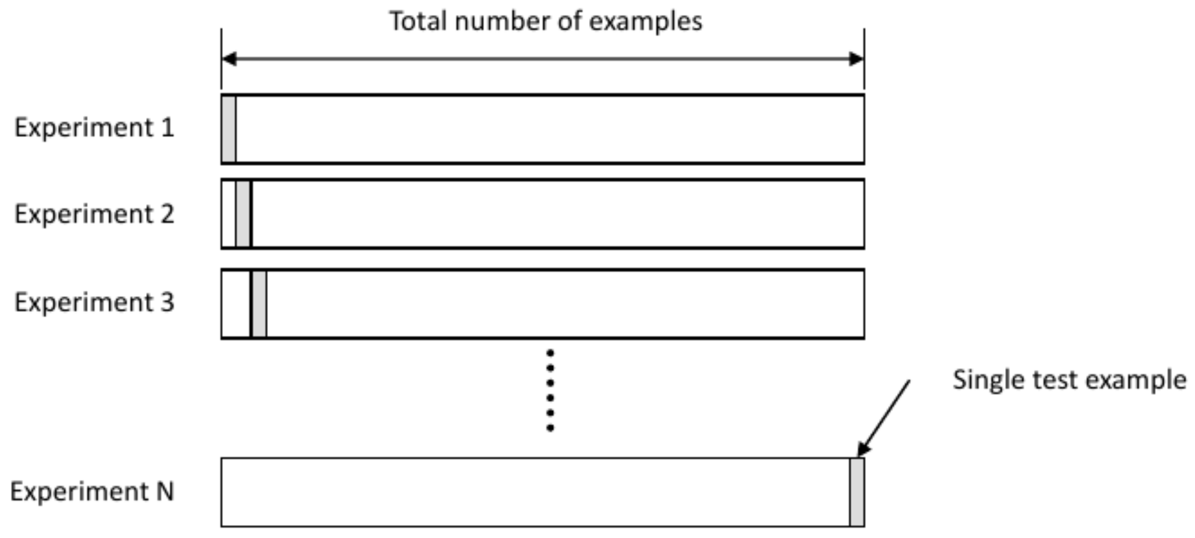
  1. For large datasets, even 3-fold cross validation will be quite accurate
  2. For very sparse datasets, we may have to use leave-one-out in order to train on as many examples as possible
- A common choise is $K = 10$ for moderately sized dataset, and $K = 5$ for small dataset

Leave-one-out Cross Validation

LOO is the degenerate case of KFCV, where K is chosen as the total number of examples

- For a dataset with N examples, perform N experiments
- For each experiment use $N - 1$ examples for training and the remaining single example for testing
- The true error is estimated as the average error rate on test examples:

$$E = \frac{1}{\underline{N}} \sum_{i=1}^{N} E_i$$



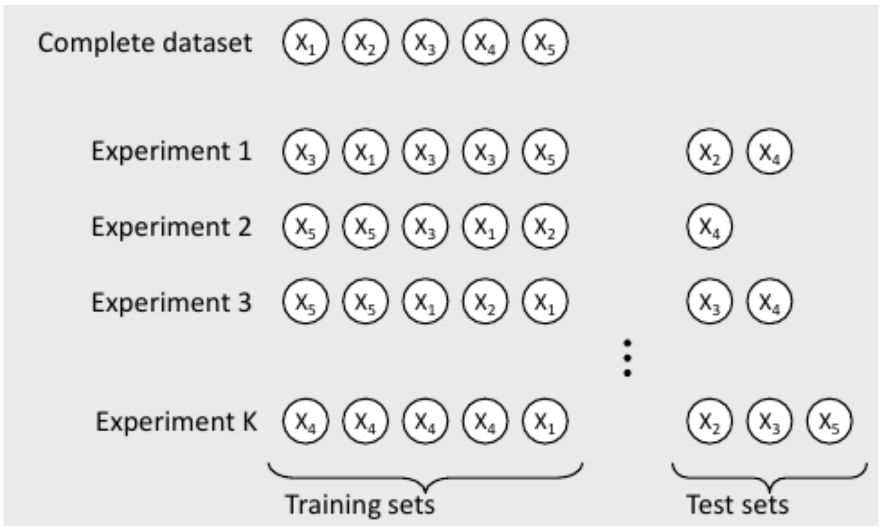
In [ ]:
```
# LOO is useful for $k$-NN, because no model training is required.
# File name: knn_iris_loo_skl.py
# Required Package(s): sklearn numpy
# Description: LOO (leave-one-out) cross validation, k nearest neighbors o
n IRIS

# modified from https://scikit-learn.org/stable/modules/generated/sklearn.
neighbors.KNeighborsClassifier.html

import numpy as np
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier

iris = datasets.load_iris()
for k in [1,3,5,7,9]:
    neigh = KNeighborsClassifier(n_neighbors=k)
    I = np.ones(iris.target.shape,dtype=bool)   # True index array
    y_pred = -np.ones(iris.target.shape,dtype=int)    # prediction, assign
ed -1 for initial values
    for n in range(len(iris.target)):
        I[n] = False    # unselect, leave one
        y_pred[n] = neigh.fit(iris.data[I,:], iris.target[I]).predict(iri
s.data[n,:].reshape(1,-1))
        I[n] = True     # select, for the next step

    nsamples = iris.data.shape[0]
    nmisses = (iris.target != y_pred).sum()
    print('kNN with k=%d' % k)
    print('Number of mislabeled out of a total %d samples : %d (%.2f%%)'
            % (nsamples, nmisses, float(nmisses)/float(nsamples)*100.0))

# 1-NN, mislabeled out of a 150 samples : 6 (4.00%)
# 3-NN, mislabeled out of a 150 samples : 6 (4.00%)
# 5-NN, mislabeled out of a 150 samples : 5 (3.33%)
# 7-NN, mislabeled out of a 150 samples : 5 (3.33%)
# 9-NN, mislabeled out of a 150 samples : 5 (3.33%)
```

Bootstrapping

- The Bootstrap: resampling *with replacement*
- From a dataset with N examples

1. Randomly select (*with replacement*) N examples and use this set for training
2. The remaining examples not selected for training are used for testing
3. This value is likely to change from fold to fold

- Repeat this process for a specified number of folds (K)
- The true error is estimated as the average error rate on test data



Comparison of CV and Boostrapping

- Compared to basic CV, the bootstrap increases the variance that can occur in each fold [Efron and Tibshirani, 1994]
  - This is a desirable property since it is a more realistic simulation of the real-life experiment from which our dataset was obtained
- Consider a classification problem with C classes, a total of N examples and N_c examples for each class c

1. The *a priori* probability of choosing an example from class c is N_c/N
   - Once we choose an example from class c, if we do not replace it for the next selection, then the \emph{a priori} probabilities will have changed since the probability of choosing an example from class c will now be $(N_c - 1)/N$
2. Thus, sampling with replacement preserves the *a priori* probabilities of the classes throughout the random selection process
3. An additional benefit is that the bootstrap can provide accurate measures of BOTH the bias and variance of the true error estimate

---

Part 3 Three-way Partitioning

Two issues arise for machine learning system design

- **Selection:** How do we select the *optimal* parameter(s) for a given problem?
- **Validation:** Once we have chosen a model, how to estimate its *TRUE* error rate?
  - The true error rate is the error rate of the classifier when tested on the *ENTIRE POPULATION*

Problem: Limited Number of Examples

- If we had access to an unlimited number of examples, these questions would have a straightforward answer
  1. Choose the model that provides the lowest error rate on the entire population
  2. And, of course, that error rate is the true error rate
- However, in real applications only a finite set of examples is available
  1. This number is usually smaller than that we would hope for
  2. Data collection is a very expensive process
- The model from the real data is an estimate of the true one
  - As a compromise, find a model as close as possible given the limited amount of data

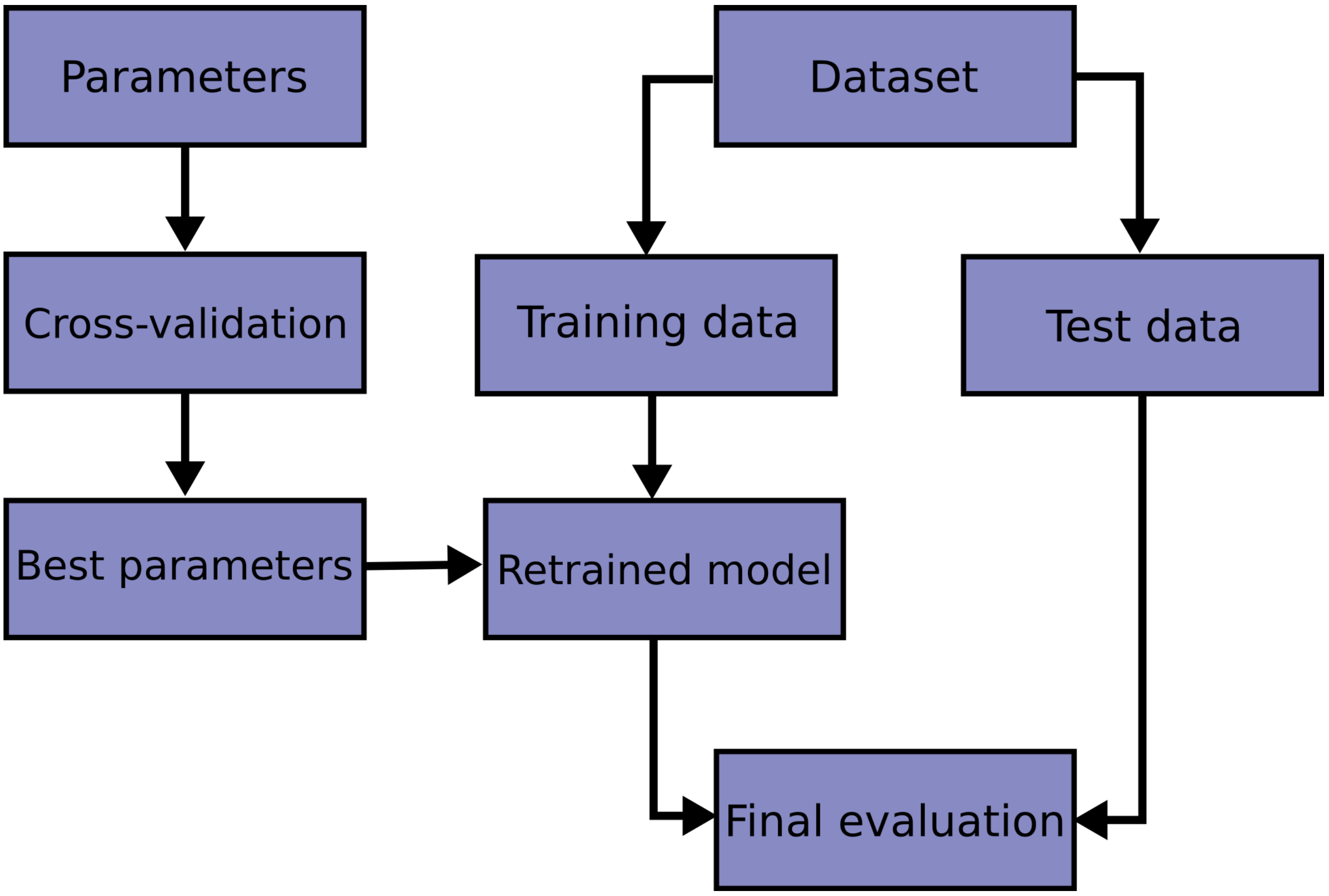
Three-way data splits

- If model selection **and** true error estimates are to be computed simultaneously, the data should be divided into three disjoint sets [Ripley, 1996]
  - **Training set** used for learning, e.g., to fit the parameters of the classifier
    - In an MLP, use the training set to find the **optimal** weights with the back-propagation learning rule
  - **Validation set** used to select among several trained classifiers
    - In an MLP, use the validation set to find the **optimal** number of hidden units or determine a stopping point for the back-propagation algorithm
  - **Test set** used only to assess the performance of a fully-trained classifier
    - In an MLP, use the test to estimate the error rate after we have chosen the final model (MLP size and actual weights)
- Why separate test and validation sets?
  - The error rate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is used to select the final model
  - After assessing the final model on the test set, YOU MUST NOT tune the model any further!

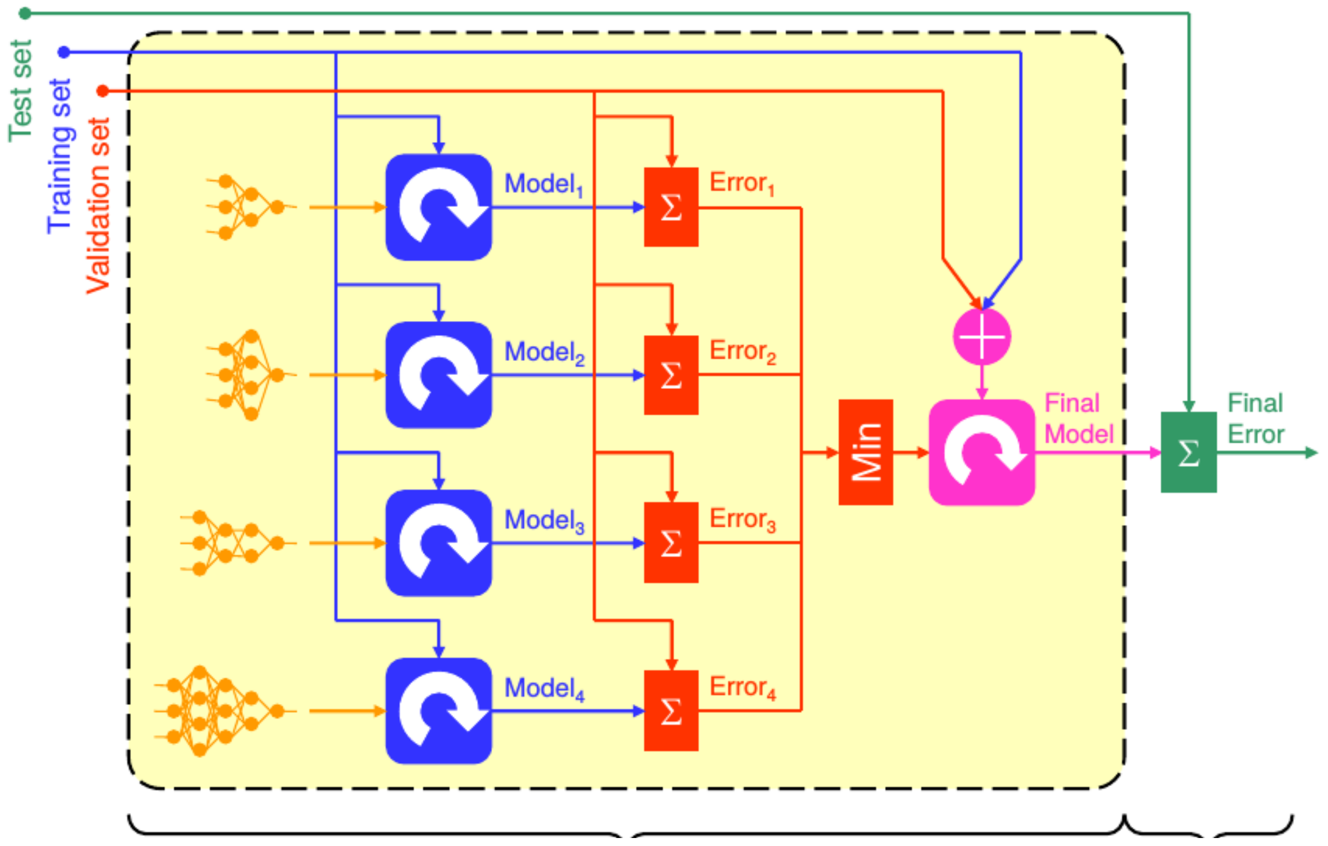
Three-way data splitting procedure

1. Divide the available data into training, validation and test set
2. Select architecture and training parameters
3. Train the model using the training set
4. Evaluate the model using the validation set
5. Repeat steps 2 through 4 using different architectures and training parameters
6. Select the best model and train it using data
7. the training and validation sets
8. Assess this final model using the test set
9. This outline assumes a holdout method
10. If CV or bootstrap are used, steps 3 and 4 have to be repeated for each of the K folds

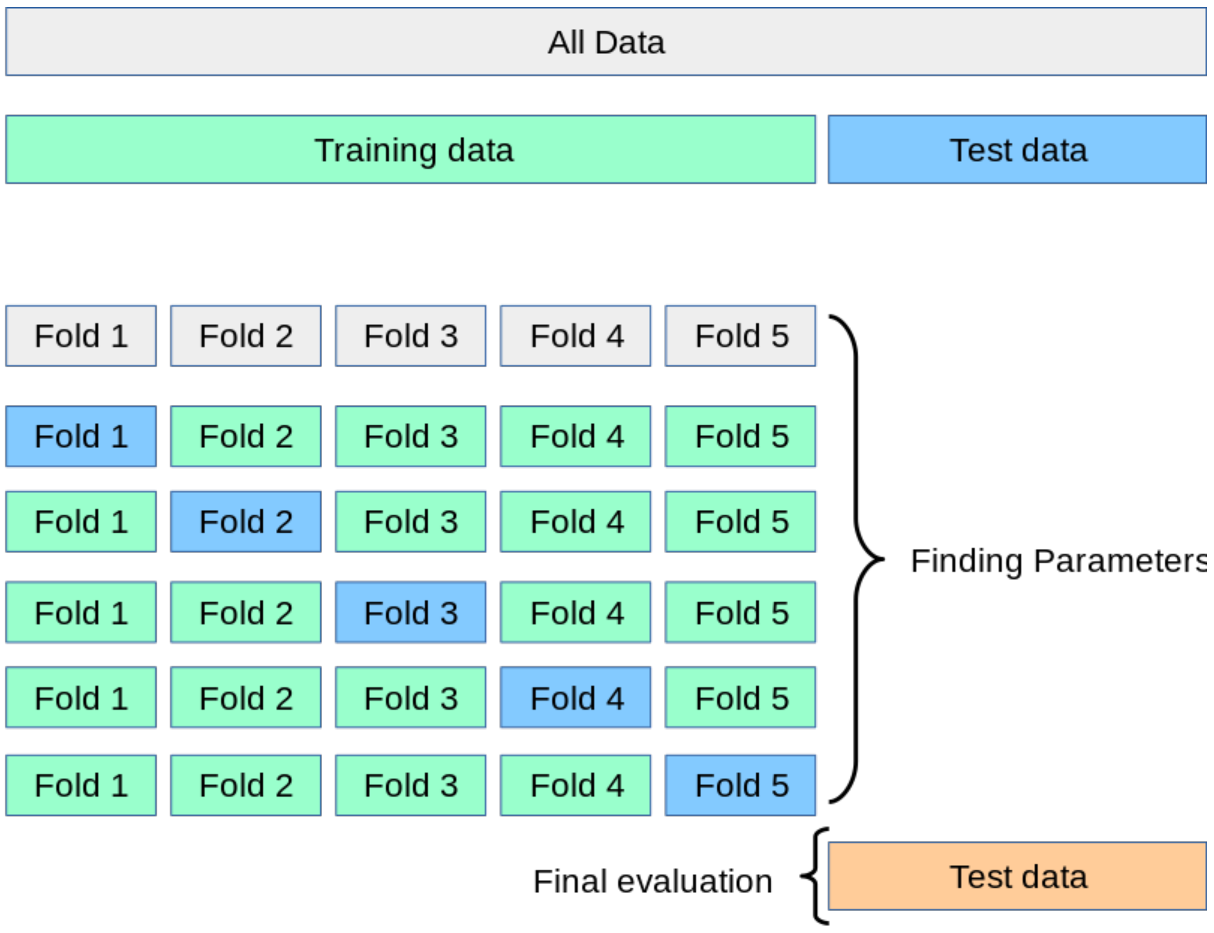
Hyperparameter Estimation Workflow



Dataflow in Three-Way Splits



Three-Way Split with Cross Validation


All Data
Training data
Test data
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Split 1
Split 2
Split 3
Split 4
Split 5
Finding Parameters
Final evaluation
Test data