CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Machine Learning

Machine Learning

Lecture: Model Selection

Ted Scully

Data Pre-processing for Scikit Learn

- Dealing with Outliers
- Dealing with Missing Values
- Handling Categorical Data
- Scaling Data
- Feature Selection
- Handling Imbalance

Machine Learning Process

- The machine learning process is much more involved than the high level work-flow depicted in the previous slide.

- The stages can be broadly defined as follows:

3. Building and Evaluating Models
   - Train many models from different categories (e.g., linear, naïve Bayes, SVM, kNN, decision trees, Random Forest, etc.) using standard parameters.
   - Measure and compare their performance.
   - Debug ML models and analyse the types of errors the models make.

4. Fine Tuning and Optimization
   - Perform hyper-parameter optimization
   - Incorporate transformation choices from part 2 as part of the hyper-parameter optimization
   - Try Ensemble methods
   - Finally assess the generalization capability of your model on the test set.

Machine Learning Process

- The machine learning process is much more involved than the high level work-flow depicted in the previous slide.

- The stages can be broadly defined as follows:

Model Selection

3. Building and Evaluating Models
   - Train many models from different categories (e.g., linear, naïve Bayes, SVM, kNN, decision trees, Random Forest, etc.) using standard parameters.
   - Measure and compare their performance.
   - Debug ML models and analyse the types of errors the models make.

4. Fine Tuning and Optimization
   - Perform hyper-parameter optimization
   - Incorporate transformation choices from part 2 as part of the hyper-parameter optimization
   - Try Ensemble methods
   - Finally assess the generalization capability of your model on the test set.

Machine Learning – Model Selection

- There are many **ML algorithms** that are candidates for running on our dataset.
- **Hyper-parameters** in machine learning algorithms are basically **configuration settings** for the algorithm, they are parameters whose values are set prior to the commencement of the learning process
- In additional there are a range of **pre-processing techniques** for configuring the data before it even reaches a ML algorithm.
- The process of finding the **best-performing model** from a range of different models, produced using different hyperparameter settings (and even pre-processing techniques) is called **model selection**.

f1	f2	f3	…	fn	class

Training

Holdout Set Configuration

f1	f2	f3	...	fn	class

Training

f1	f2	f3	...	fn	class

Test

The drawback of the holdout method

- As shown in the previous slide we are currently using a holdout set configuration. That is, we have a training dataset and we have a test dataset.

- However, if we **reuse the <u>same test dataset</u> over and over again** in order to optimize the hyper-parameters of the model we will **<u>overfit</u>** on the test data.
- The accuracy we get from the best model configuration may be overly optimistic.
- Information about the test set can indirectly "leak" into the model.

- This means that even if we obtain really strong accuracy from model selection using a holdout set the model may not obtain the same level of performance when deployed.

- Unfortunately, despite this very significant issue, many people still use the test set for model selection, which is **not good machine learning practice**.

Holdout Cross-Validation

- A better way of using the holdout method for model selection is to separate the data into three parts: a **training** set, a **validation** set, and a **test** set.
- The training set is used to fit the different models, and the performance on the validation set is then used for the model selection (see next slide).
- The advantage of having a **test set,** which the model hasn't seen before during the training and model selection steps, is that we can obtain a **less biased estimate** of its ability to generalize to new data.
- This method is referred to as **holdout cross validation**.

f1	f2	f3	…	fn	class

Training

f1	f2	f3	…	fn	class

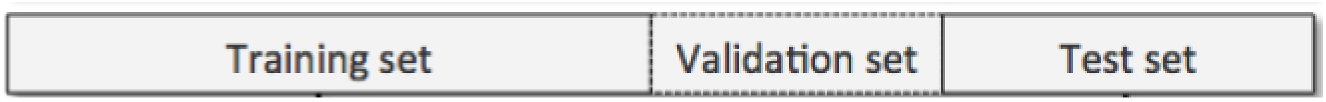
Validation

f1	f2	f3	…	fn	class

Test

The drawback of the holdout cross-validation

- A disadvantage of the holdout cross validation method is that the **performance estimate is sensitive** to how we partition the training set into the training and validation subsets.
- Different validation splits might produce different optimized hyper-parameter configurations and likewise different final accuracy values.
- In other words the final results may vary for different samples of the data. Notice we are continually using the same data sample to build our models and identifying the best hyper-parameter configuration.



- Another problem with holdout cv is that we also seriously **reduce the amount of data** we use for building the model (because we divide the data into three parts).
- A more robust technique for performance estimation, is **k-fold cross-validation**, where we repeat the holdout method k times on k subsets on the training data

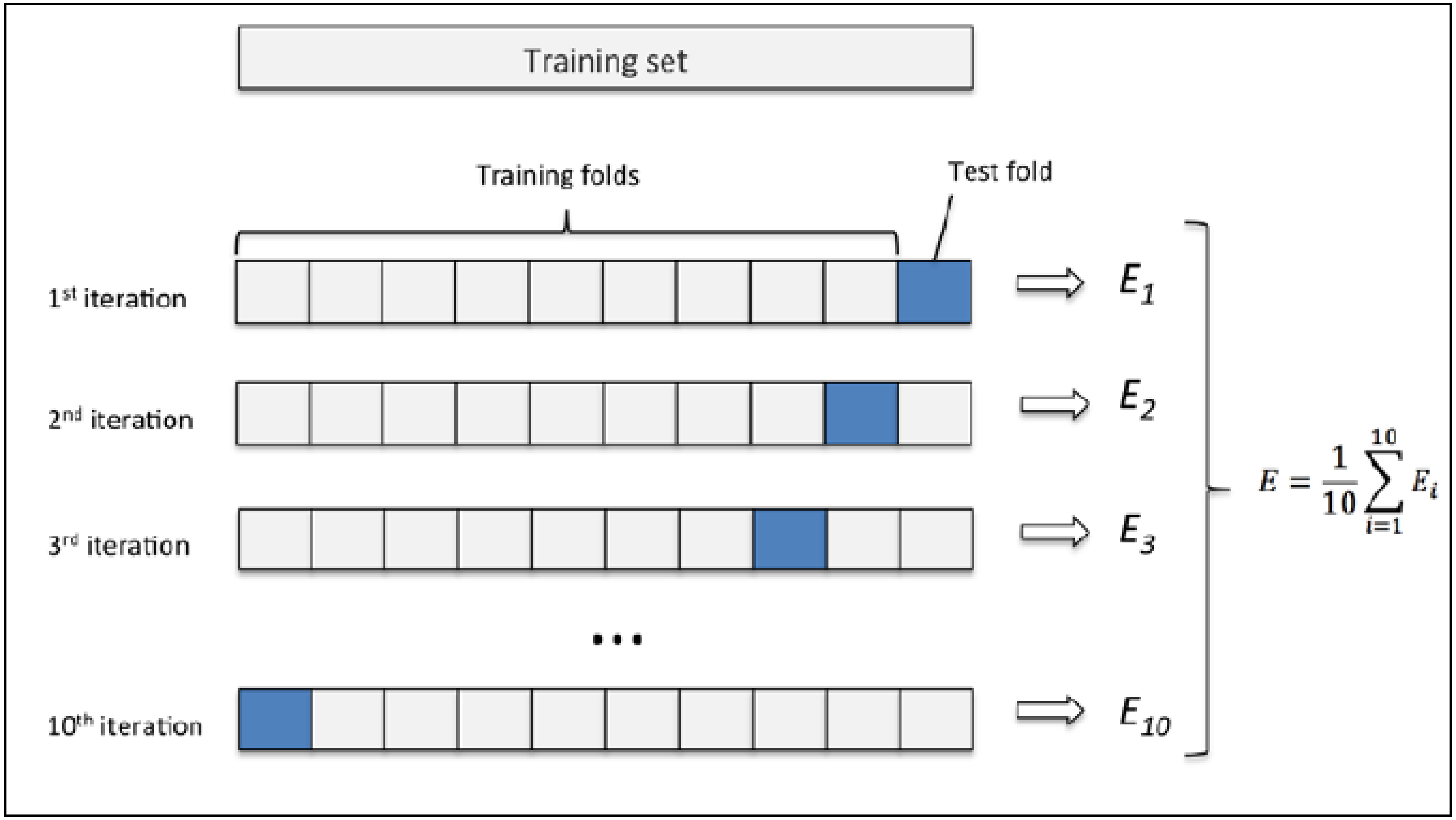
K Fold Cross Validation

- In k-fold cross-validation, we randomly split the training dataset into **k** folds without replacement, where **k –1** folds are used for the model training and one fold is used for testing. This procedure is repeated k times so that we obtain k models and performance estimates.

- We then calculate the **average performance** of the models based on the different, independent folds to obtain a performance estimate that is less sensitive to the sub-partitioning of the training data compared to the holdout method.

- Also useful to calculate the standard deviation of the model performance across folds.

K Fold Cross Validation

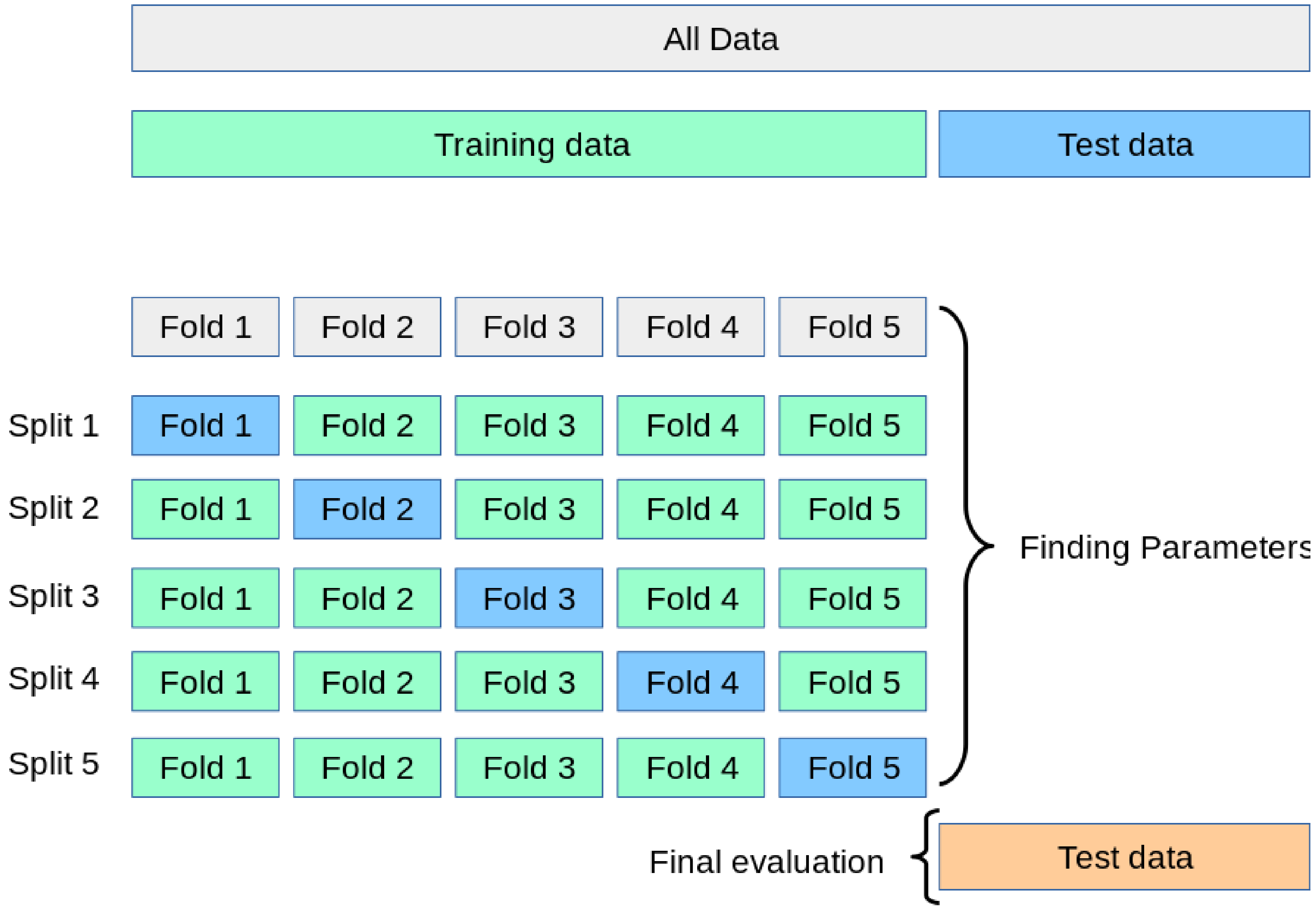


K Fold Cross Validation

- So while we have just describe the core cross validation process it is also important to understand that before performing k-fold cross validation we **subdivide** the original data into two parts (the **training** and **test** data).

- We then subsequently apply **k-fold cross-validation** using training data for model tuning, finding the optimal hyper-parameter values that yields a satisfactory generalization performance.

- Once we have found best hyper-parameter values, we can:
  - **Retrain the model on the complete training set**
  - Obtain a **final performance** estimate using the independent **test set**.



https://scikit-learn.org/stable/_images/grid_search_cross_validation.png

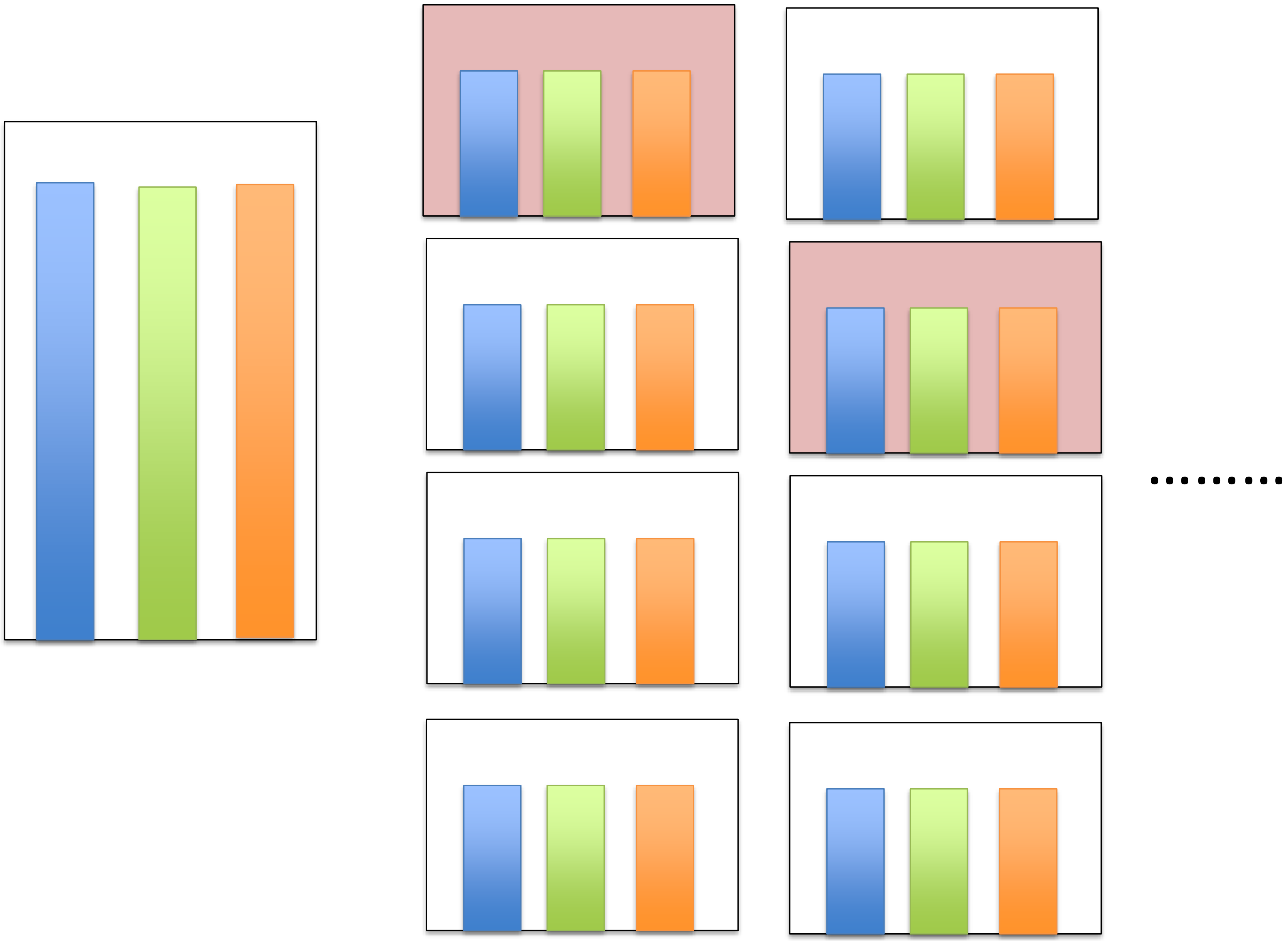
Stratified Cross Fold Validation

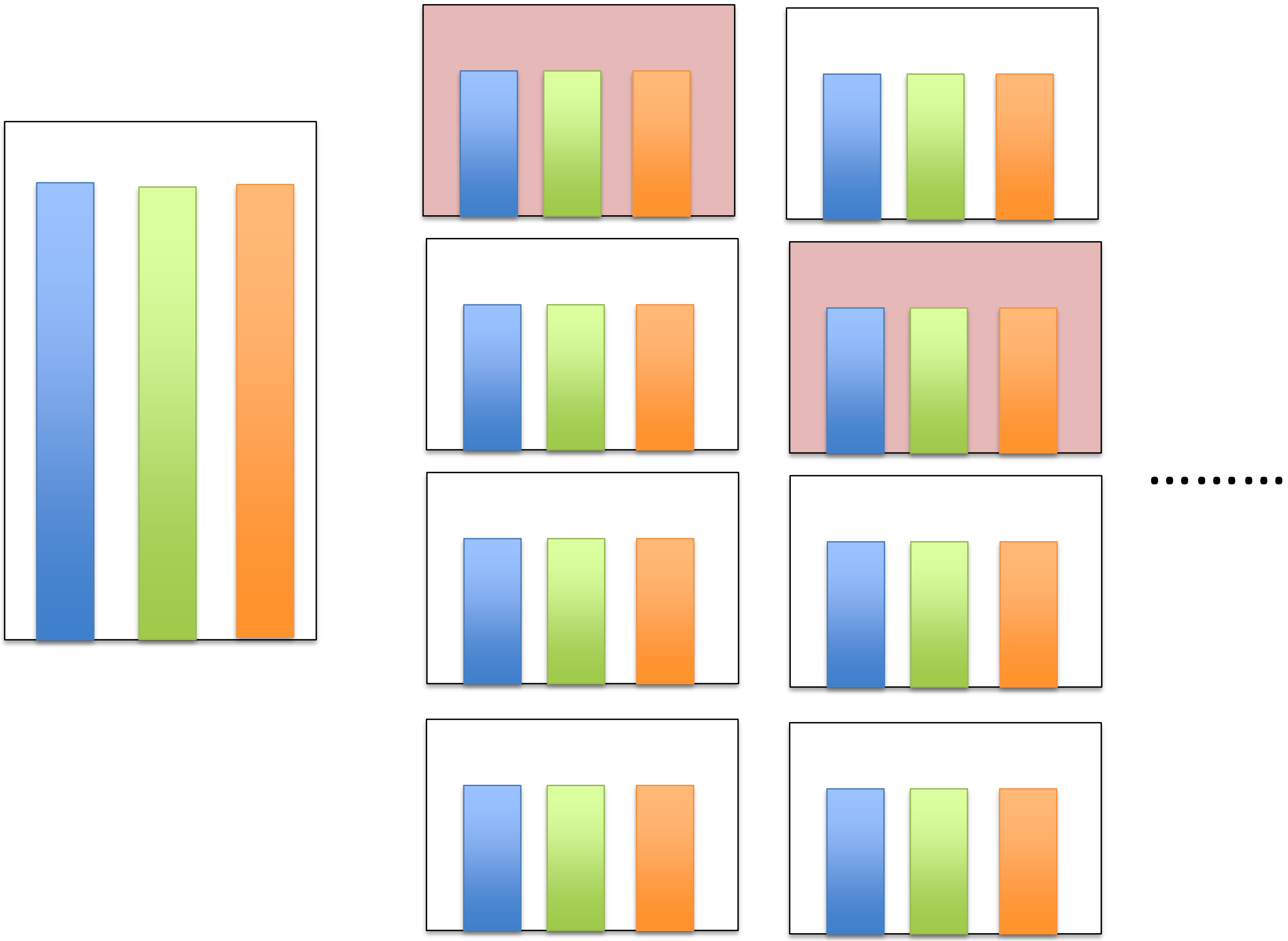
- There are many available variants of k-fold cross validation.
- One such variant is stratified k-fold cross-validation.

- In stratified CV the data for each fold is ordered so that each fold is a good **representation** of the class distribution in the original dataset.

- In stratified cross-validation, the **class proportions are preserved in each fold** to ensure that each fold is representative of the class proportions in the training dataset.

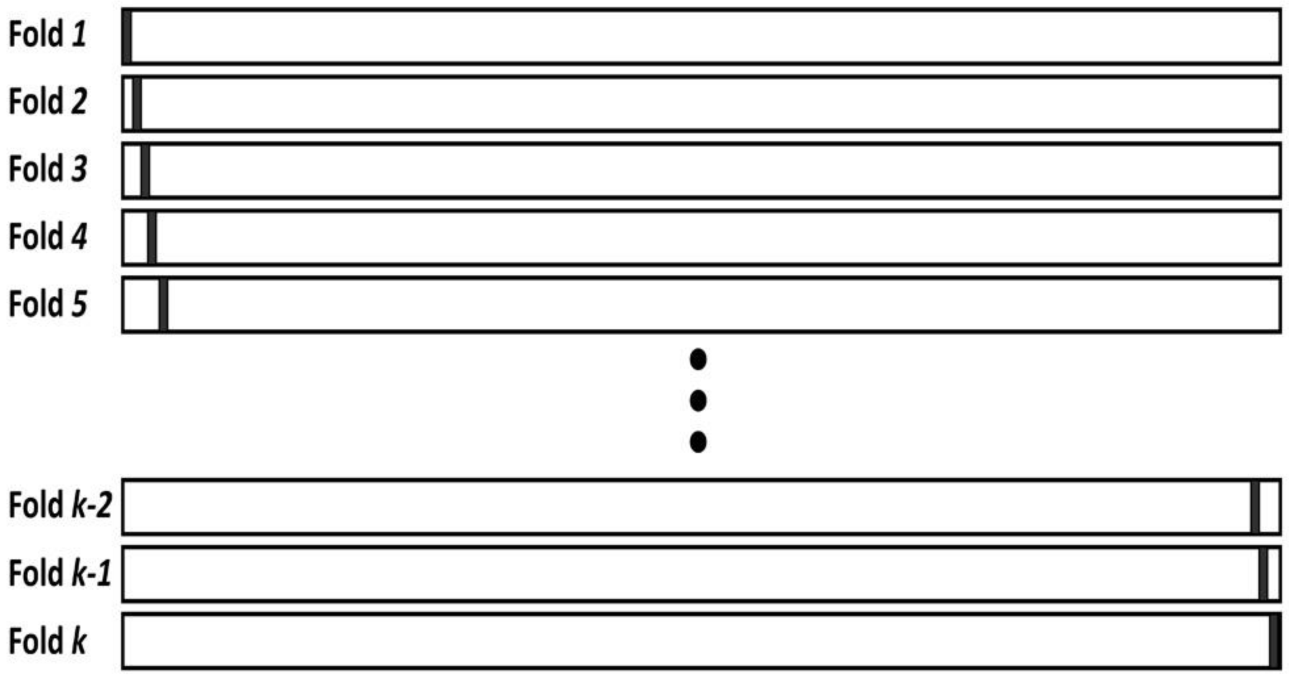
- This is good practice for classification problems and is the default for classification based cross-fold validation in Scikit Learn.





Leave-one-out Cross Fold Validation

- LOOCV is an extreme form of cross validation where k = number of training instances.
- Therefore, for each fold the test set just contains a single instance.
- LOOCV is useful when the **amount of data available is too small** to allow big enough training sets in a k fold cross validation. However, it can also be very time-consuming.



Assessing Accuracy (Cross Fold Validation)

- The simplest way to use cross-validation is to call the ***cross_val_score*** function on the classifer and the dataset. Please note we use stratified cross fold validation by default

In the example, we apply stratified cross fold validation.

Notice the scores variable holds the result after each fold. To get the final result we obtain the mean.

Also notice we don't have to directly call the fit function

```
from sklearn import model_selection
from sklearn import datasets
from sklearn import tree


iris = datasets.load_iris()

clf = tree.DecisionTreeClassifier()

scores = model_selection.cross_val_score(clf, iris.data, iris.target, cv=10)

print (scores.mean(), scores.std())
```

```
0.96   0.0442216638714
```

k Fold Cross Validation (sklearn.model_selection.Kfold)

- While the approach presented in the previous slides is simple, we may want to get access to not just the accuracy for each fold but also what classes were correctly or incorrectly predicted during each fold.

- Therefore, it can often be very useful to perform cross fold validation manually.
    - In the example that follows we use normal (non-stratified) k-fold cross validation.

- We can achieve this by using **sklearn.model_selection.Kfold** or by using **sklearn.model_selection.StratifiedKFold**.
    - 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.

Scikit Learn - k Fold Cross Validation

```
import numpy as np
from sklearn import datasets
from sklearn import tree
from sklearn import model_selection
from sklearn import metrics

iris = datasets.load_iris()
allResults = []

kf = model_selection.StratifiedKFold(n_splits=6, shuffle=True, random_state=1)

for train_index, val_index in kf.split(iris.data, iris.target):

    clf = tree.DecisionTreeClassifier()
    clf.fit( iris.data[train_index], iris.target[train_index] )

    results= clf.predict( iris.data[val_index] )

    print ( results [ results != iris.target[val_index] ])

    allResults.append ( metrics.accuracy_score(results, iris.target[val_index]) )

print ("Accuracy is ", np.mean(allResults))
```

When we create the Kfold object we specify the number of folds as 6. Each time we iterate we call the **split** function, which will divide the indices into training and test (5/6 for training and 1/6 for test). Each time the loop iterates it generates a different fold.

Scikit Learn - k Fold Cross Validation

```
import numpy as np
from sklearn import datasets
from sklearn import tree
from sklearn import model_selection
from sklearn import metrics

iris = datasets.load_iris()
allResults = []

kf = model_selection.StratifiedKFold(n_splits=6, shuffle=True, random_state=1)

for train_index, val_index in kf.split(iris.data, iris.target):

    clf = tree.DecisionTreeClassifier()
    clf.fit( iris.data[train_index], iris.target[train_index] )

    results= clf.predict( iris.data[val_index] )

    print ( results [ results != iris.target[val_index] ])

    allResults.append ( metrics.accuracy_score(results, iris.target[val_index]) )

print ("Accuracy is ", np.mean(allResults))
```

Notice that NumPy array results contains all the predictions made by our model. If I want to determine the classes the model got wrong for each I compare them to the actual results using array-based indexing.

Scikit Learn - k Fold Cross Validation

```
import numpy as np
from sklearn import datasets
from sklearn import tree
from sklearn import model_selection
from sklearn import metrics

iris = datasets.load_iris()
allResults = []

kf = model_selection.StratifiedKFold(n_splits=6, shuffle=True, rando

for train_index, val_index in kf.split(iris.data, iris.target):

    clf = tree.DecisionTreeClassifier()
    clf.fit( iris.data[train_index], iris.target[train_index] )

    results= clf.predict( iris.data[val_index] )

    print ( results [ results != iris.target[val_index] ])

    allResults.append ( metrics.accuracy_score(results, iris.target[val_index]) )

print ("Accuracy is ", np.mean(allResults))
```

The use of Kfold allows us a great degree of control and visability of the cross validation process. For example, if I insert the following line it prints the incorrect classifications made for each iteration of cross fold.

```
[2 1 1]
[2]
[1 1 1]
[2]
[2]
[1 1]
Accuracy is
0.926666666667
```

Cross Fold Validation with Unbalanced Data.

- You will remember that when we covered imbalanced data we said that you should apply the rebalancing technique (SMOTE, Tomek, etc) to the training data only (**not the test data**).

- This crates a problem if we are using the high level model_selection.**cross_val_score**

- The cross_val_score class will automatically partition the data into training and test data. There is no way for us to introduce a rebalancing technique on the train partition but not on the test partition.

- However, the low level control offered by **model_selection.KFold** and **model_selection. StratifiedKFold** gives us a method of achieving this.

```
from sklearn import datasets
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
from imblearn.datasets import make_imbalance
from imblearn.over_sampling import SMOTE
from sklearn import model_selection
import numpy as np
from sklearn import metrics


RANDOM_STATE = 42

# Generate a balanced dataset
X, y = datasets.make_classification(n_classes=2, n_features=20, n_samples=15000,
                                    random_state=RANDOM_STATE)

# We use this initial dataset and make it imbalanced
X, y = make_imbalance(X, y, sampling_strategy={0: 7400, 1:200},
random_state=RANDOM_STATE)

totalConfusionMatrix = np.zeros((2,2))
```

In this code we create an imbalanced dataset. In the final line we create a blank confusion matrix.

Aside: For simplicity we have only depicted the cross fold process (we have not included the separate test set, which is used for final evaluation)

```
kf = model_selection.StratifiedKFold(n_splits=6, shuffle=True, random_state=2)

for train_index, val_index in kf.split(X,y):

    # rebalance the training data for this split of cross fold
    sm = SMOTE(random_state=0)
    X_train, y_train = sm.fit_sample(X[train_index], y[train_index])

    # create out ML Model and train on the rebalanced data
    clf = LinearSVC(random_state=RANDOM_STATE, max_iter=2000)
    clf.fit(X_train, y_train)

    # test the model on the validation set (note the validation data has not undergone resampling)
    results= clf.predict( X[val_index] )


    confusionMatrix = metrics.confusion_matrix(y_true = y[val_index], y_pred =results )
    totalConfusionMatrix += confusionMatrix

print (totalConfusionMatrix)
```

Initially we create our StratifiedKFold object and then begin to iterate each of the 6 iterations.

```
kf = model_selection.StratifiedKFold(n_splits=6, shuffle=True, random_state=2)

for train_index, val_index in kf.split(X,y):

    # rebalance the training data for this split of cross fold
    sm = SMOTE(random_state=0)
    X_train, y_train = sm.fit_sample(X[train_index], y[train_index])

    # create out ML Model and train on the rebalanced data
    clf = LinearSVC(random_state=RANDOM_STATE, max_iter=2000)
    clf.fit(X_train, y_train)

    # test the model on the validation set (note the validation data has not undergone resampling)
    results= clf.predict( X[val_index] )


    confusionMatrix = metrics.confusion_matrix(y_
    totalConfusionMatrix += confusionMatrix

print (totalConfusionMatrix)
```

Each time inside the loop we rebalance the training data and build a model using the rebalanced data. Next we push the test data (which has not been rebalanced through the ML model and collect the results in the array results.)

```
kf = model_selection.StratifiedKFold(n_splits=6, shuffle=True, random_state=2)

for train_index, val_index in kf.split(X,y):

   # rebalance the training data for this split of d
   sm = SMOTE(random_state=0)
   X_train, y_train = sm.fit_sample(X[train_inde

   # create out ML Model and train on the rebal
   clf = LinearSVC(random_state=RANDOM_STATE, max_iter=2000)
   clf.fit(X_train, y_train)

   # test the model on the validation set (note the validation data has not undergone resampling)
   results= clf.predict( X[val_index] )


   confusionMatrix = metrics.confusion_matrix(y_true = y[val_index], y_pred =results )
   totalConfusionMatrix += confusionMatrix

print (totalConfusionMatrix)
```

Each time we iterate we generate a new confusion matrix and add it to the main confusion matrix called totalConfusionMatrix.

```
[[6752.  648.]
 [  30.  170.]]
```