Machine Learning

Machine Learning

Lecture: CV, Hyper-parameters and Nested CV

Ted Scully

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

```
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())
```

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

By default if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used.

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**.
  - 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.

Please note there is also a **sklearn.model_selection.StratifiedKFold**

Scikit Learn - k Fold Cross Validation

```
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.KFold(n_splits=6, shuffle=True, random_state=1)

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

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

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


    allResults.append(metrics.accuracy_score(results, iris.target[test_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

```
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.KFold(n_splits=6, shuffle=True, random_state=1)

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

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

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


    allResults.append(metrics.accuracy_score(results, iris.target[test_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

```
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.KFold(n_splits=6, shuffle=True, random_state=

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

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

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

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

    allResults.append( metrics.accuracy_score(results, iris.target[test_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
```

Scikit Learn – Manual Stratified k Fold Cross Validation

```
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, test_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[test_index] )

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

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

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

Notice the code for manual stratified k-fold is quite similar. The main difference is that when calling the split function we must pass both the training data and the class labels. We provide the target labels as the splits should reflect the distribution of classes.

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** 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, test_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 test set (note the test data has not undergone resampling)
    results= clf.predict( X[test_index] )


    confusionMatrix = metrics.confusion_matrix(y_true = y[test_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, test_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 test set (note the test data has not undergone resampling)
    results= clf.predict( X[test_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, test_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 test set (note the test data has not undergone resampling)
   results= clf.predict( X[test_index] )


   confusionMatrix = metrics.confusion_matrix(y_true = y[test_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.]]
```

Model Selection using Scikit Learn

- Using cross fold validation
- Hyper-parameter optimization
- Nested Cross Fold Validation
- Using Pipelines
- Evaluation

Hyper Parameter Optimization

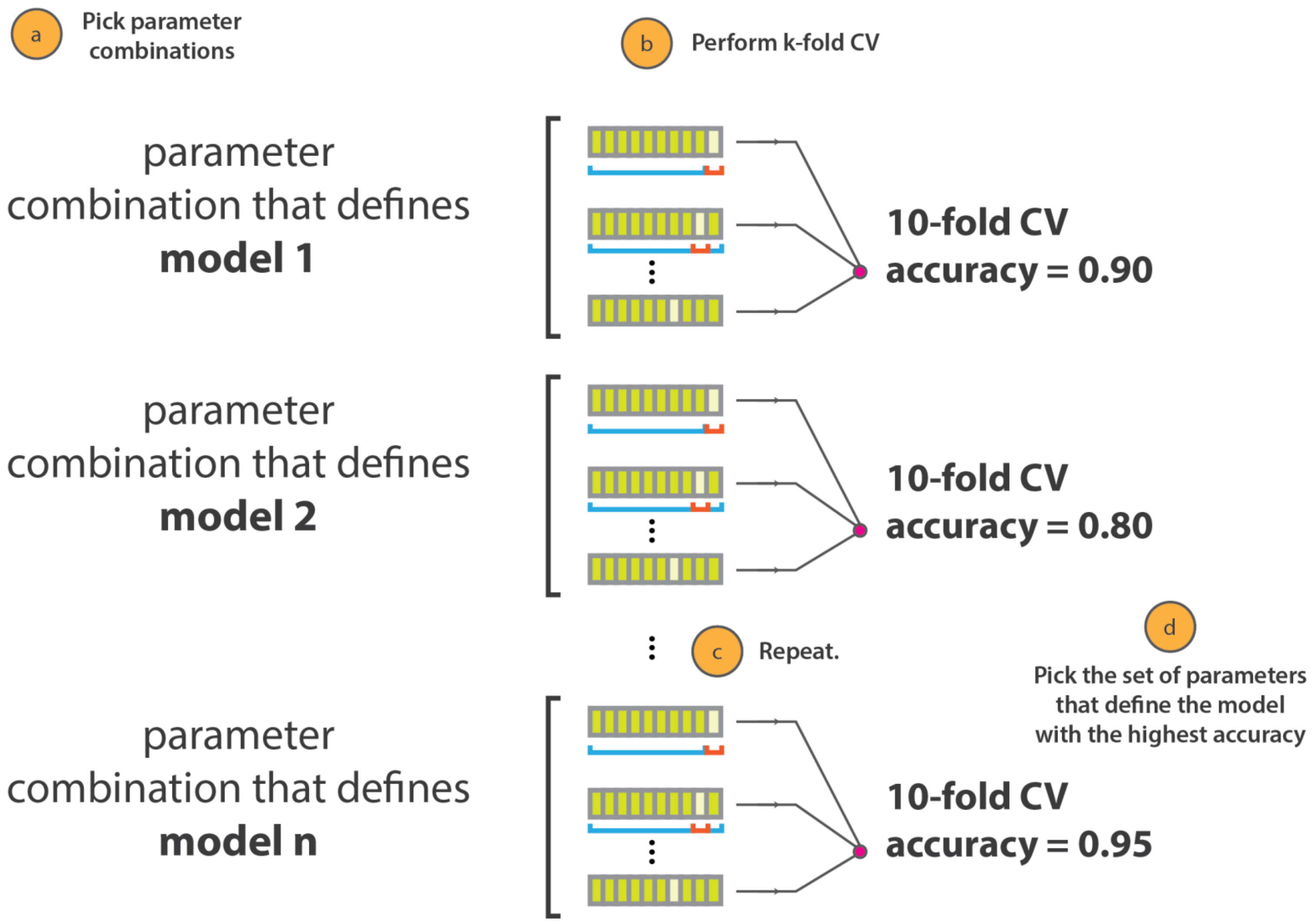
- Hyper parameters (which are those parameters not directly learnt by the machine learning algorithm) can be determined through experimentation, in other words searching the possible values for the parameter in order to obtain the best cross-validation value.

- Such independent parameters are often referred to as **hyper-parameters**.

- A search process for the values of hyper-parameters consists of:
  - The learning algorithm
  - A parameter space (range of values for the parameter(s))
  - A method for searching these values
  - Cross Validation

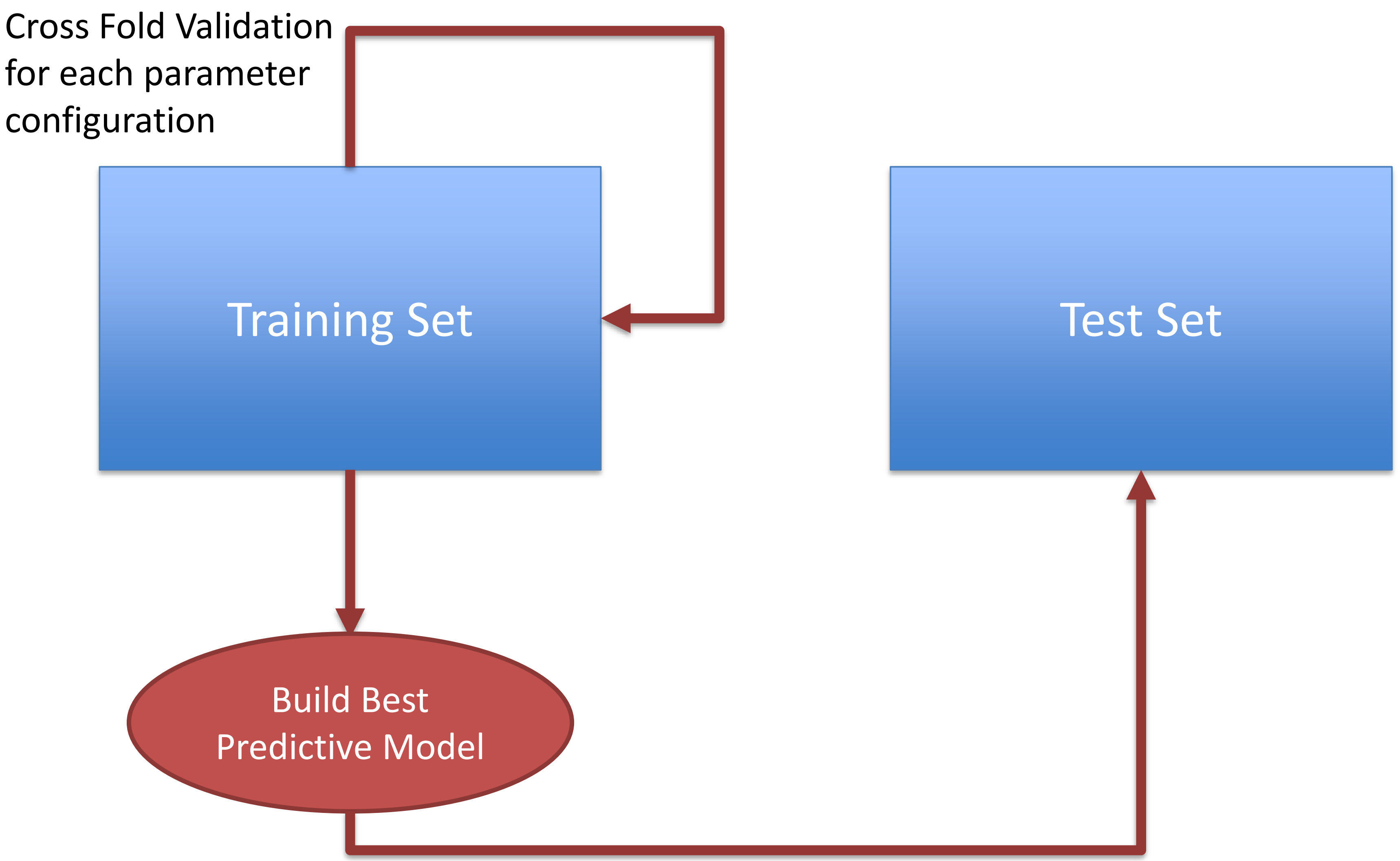
- To find the names and current values for all parameters for a given estimator (ML Algorithm), use: **algorithm.get_params()** or the API pages.

- The learning algorithm
- A parameter space (range of values for the parameter(s))


a
Pick parameter combinations
b
Perform k-fold CV
parameter combination that defines **model 1**
10-fold CV accuracy = 0.90
parameter combination that defines **model 2**
10-fold CV accuracy = 0.80
c
Repeat.
d
Pick the set of parameters that define the model with the highest accuracy
parameter combination that defines **model n**
10-fold CV accuracy = 0.95

https://cambridgecoding.wordpress.com/2016/04/03/scanning-hyperspace-how-to-tune-machine-learning-models/

High Level Hyper-Parameter Optimization with Cross Fold Validation


Cross Fold Validation for each parameter configuration
Training Set
Test Set
Build Best Predictive Model

Parameter Optimization

- The most commonly used method of parameter optimization in Scikit-learn is **GridSearchCV** (sklearn.grid_search.GridSearchCV).

- It performs exhaustive search over a specified range of parameter values for an estimator.

- The grid search provided by GridSearchCV exhaustively generates candidates from a grid of parameter values specified with the **param_grid** parameter.
- The main input parameter to GridSearchCV is **param_grid** which is **list of dictionaries**

- Each dictionary has:
  - Parameters names as keys
  - Lists of parameter settings to try as values.
- You can include multiple grids.

```
from sklearn import model_selection
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

iris = datasets.load_iris()

knn = KNeighborsClassifier()
scores = model_selection.cross_val_score(knn, iris.data, iris.target, cv=10)
print (scores.mean())


param_grid = [ {'n_neighbors': list(range(1, 80)),  'p':[1, 2, 3, 4, 5] }  ]
clf = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10)

clf.fit(iris.data, iris.target)

print("\n Best parameters set found on development set:")
print(clf.best_params_ , "with a score of ", clf.best_score_)
```

```
from sklearn import model_selectio
from sklearn import datasets
from sklearn.neighbors import KNeig
from sklearn.model_selection impor

iris = datasets.load_iris()

knn = KNeighborsClassifier()
scores = model_selection.cross_val_
print (scores.mean())


param_grid = [ {'n_neighbors': list(r
clf = GridSearchCV(KNeighborsClass

clf.fit(iris.data, iris.target)

print("\n Best parameters set found on development set:")
print(clf.best_params_ , "with a score of ", clf.best_score_)
```

0.966666666667

Best parameters set found on development set
(note it is stored as a dictionary):
{'n_neighbors': 6, 'p': 3} with a score of 0.98

Note you can also use
clf. best_estimator_ and it will return the instance of the model that produced the best accuracy.

Note by default gridSearch will use the best parameters (identified using CV) and refit these to the entire dataset (this is what is return for clf.best_estimator_).

```
……….

iris = datasets.load_iris()

knn = KNeighborsClassifier()
scores = cross_validation.cross_val_score(knn, iris.data, iris.target, cv=10)
print scores.mean()



param_grid = [ {'n_neighbors': range(1, 80), 'p':[1, 2, 3, 4, 5]} ,
                {'algorithm':['auto', 'ball_tree', 'kd_tree', 'brute'] } ]
```

Notice we can insert more than a single grid. In this example we have two separate grids. It is important to understand that each grid is searched separately

Parameter Optimization – Running jobs in parallel.

- **GridSearchCV** includes a parameter called **n_jobs**, which allows us to specify the number of jobs to run in **parallel**.
- By default the **n_jobs is set to 1**, which mean no joblib level parallelism is used at all.
- If set to -1, all available CPUs are used. Incorporating parallelism can significantly speed up your hyper-parameter optimization.
- For example, specifying the n_jobs=-1 below will provide a 3X speed-up on the computation search on my machine.

```
……….

param_grid = [ {'n_neighbors': list(range(1, 100)),  'p':[1, 2, 3, 4] }  ]

clf = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10, n_jobs=-1)

………
```

Parameter Optimization When Dataset is Imbalanced.

When your data is imbalanced and you are performing cross fold validation, you need to resample the training data but not the validation data during each iteration of cross fold. We solved this problem easily when just performing cross fold validation by using a manual cross fold validation operator such as kFold.

The question asked was how do **we perform GridSearchCV when we have imbalanced data?**

One option is we can just implement the search process ourselves manually. The drawback of this is that it will be slow and we can't take advantage of the parallel execution facilitated by GridSearchCV

An alternative that gets around this problem is to utilize the **Pipeline** class in the **ImbalancedLearn** contribution package.

The imbalanced-learn Pipeline class allows us to apply any sampling technique and it will apply that technique to the training data during each iteration of cross fold (more specifically it will only apply the sampling when the fit function is called, which will only be called in each iteration of cross fold for the training data (not the validation data)).

Parameter Optimization When Dataset is Imbalanced.

We can create an instance of GridSearchCV and pass it an Imbalanced learn pipeline.

The sampling technique in this pipeline will only be applied to the data used for training the model during each iteration of cross fold (it will not resample the validation fold).

```
RANDOM_STATE = 42
from imblearn.pipeline import Pipeline
from sklearn import datasets
from imblearn.datasets import make_imbalance
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# Initally we generate an artifical 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 using make_imbalance
# now the dataset has 7400 instances of class 0 and only 200 or class 1
X, y = make_imbalance(X, y, sampling_strategy={0: 7400, 1:200},
random_state=RANDOM_STATE)
```

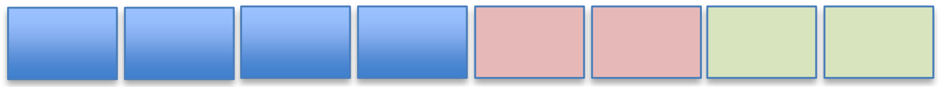
Parameter Optimization When Dataset is Imbalanced.

```
# The random forest will be the classifier that is part of the pipeline
randForest = RandomForestClassifier(random_state=RANDOM_STATE)

# We are going to use SMOTE resampling as part of the pipeline
sm = SMOTE(random_state=RANDOM_STATE)

# Create a search grid for the random forest
param_grid = [ {'model__n_estimators': [200, 300],  'model__max_depth':[8, 10] }  ]

# Create an imbalanced learn pipeline. The pipeline just consists of SMOTE and
# a random forst classifier. Smote will only be applied to the training set during
# each iteration of cross fold validation (not the test data)
pipeline = Pipeline( [('smt', sm), ('model', randForest)] )



# Create a normal instnace of grid search CV and pass it the pipeline object
clf = GridSearchCV(pipeline, param_grid, cv=5, n_jobs=-1)
clf.fit(X, y)

print(clf.best_params_ , "with a score of ", clf.best_score_)
```

Model Selection using Scikit Learn

- Using cross fold validation
- Hyper-parameter optimization
- Nested Cross Fold Validation
- Using Pipelines
- Evaluation

Nested Cross Fold Validation - Motivation

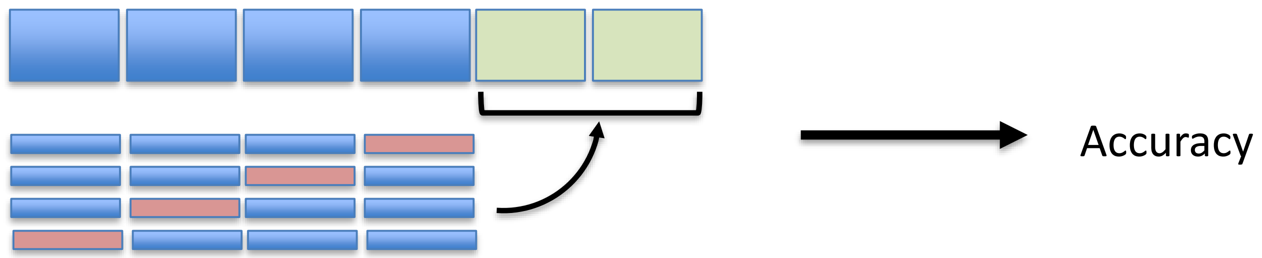


- You will remember with holdout cross validation we split the data into training, validation and test data. We train multiple models using the training set, evaluate them on the validation test and then test the best model on the separate test set.

- A drawback here is that the performance estimates we obtain are sensitive to the partitioning of the data. Depending on how we partition the data into train/validation and test we may end up getting a different final accuracy values.

- The solution was that we didn’t train on just one fixed set and test on one fixed validation set, we rotated the training and validation sets multiple times using cross fold validation.

Nested Cross Fold Validation - Motivation



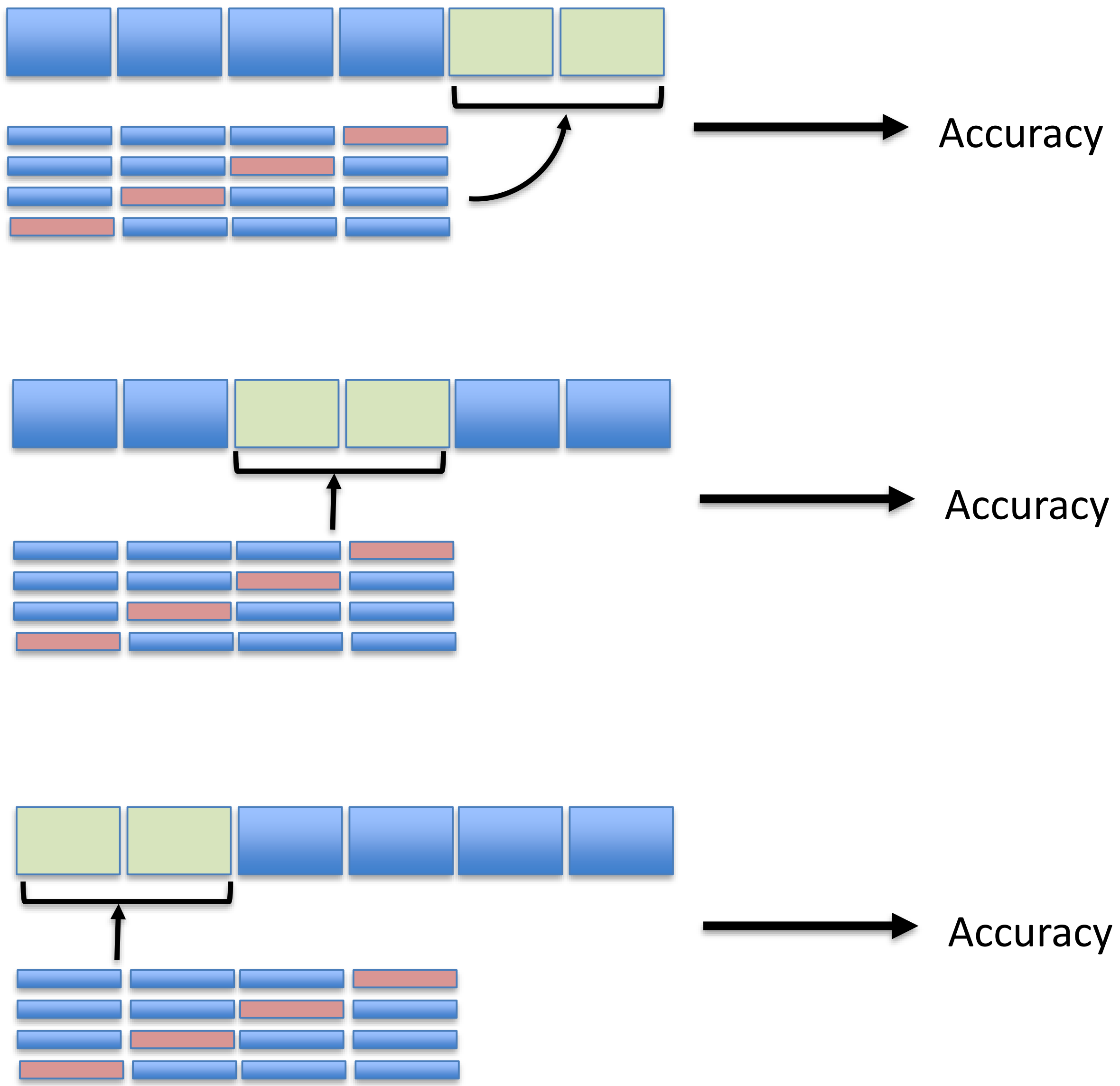
- A more robust methodology is using cross fold validation.

- We separate the data into training and test.
- We perform cross fold validation on the training data using different models and obtain a final performance of the best model on the separate test set.

- Of course the final accuracy value we obtain here may be sensitive to how we partition the data initially into training and test.

To overcome this problem perhaps we could adopt the same methodology again. Rotate the test set. In effect, cross fold validation multiple times … this is referred to as nest cross fold validation.

Nested Cross Fold Validation



This is an example of nested cross fold validation. With an outer loop of 3 and an inner loop of 4.

Nested Cross Fold Validation

- In nested cross-validation, we have an **outer** k-fold cross-validation loop to split the data into training and test folds.

- For each split we then have an **inner loop** that is used to perform grid search (perform hyper-parameter optimization) using k-fold cross-validation on the training data from the outer loop.

- After model selection, the test fold is then used to evaluate the model performance.

- The returned average cross-validation accuracy gives us a very strong estimate of the unbiased performance of our model.

Nested Cross Fold Validation

- Nested Cross Fold Validation
  - It provides us with an unbiased estimation of accuracy.
  - It is computationally expensive.
    - Consider a grid search that has 300 possible combinations. With GridSearchCV we would end up building 300*10 (3000) models
    - Now consider if we use a nest cross fold validation with an outer number of loops of 10. With nested cross fold validation we would end up building 10*300*10 (30000) models.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

X, y = make_classification(n_samples=2000, n_features=15, random_state=10, n_informative=10,
n_redundant=5)

# We will use 10 fold as the outer loop for nested CV
cv_outer = StratifiedKFold(n_splits=10, random_state=10, shuffle=True)

results = []
for train_indices, test_indices in cv_outer.split(X, y):

        # Seperate the data into test data and training data
        X_train, X_test = X[train_indices, :], X[test_indices, :]
        y_train, y_test = y[train_indices], y[test_indices ]

        # inner cross fold valdiation loop
        cv_inner = StratifiedKFold(n_splits=10, random_state=10 ,shuffle=True)

        model = RandomForestClassifier(random_state=0)
        param_grid = {'n_estimators':[10, 50, 100, 500], 'max_features':[2, 6, 10, 14]}
        grid_search = GridSearchCV(model, param_grid, scoring='accuracy', cv=cv_inner, refit=True)
        result = grid_search.fit(X_train, y_train)
```

Make a classification dataset. The outer loop of nested with be a 10 fold CV. For each of these iterations we will split the data into training and test data. We take the training data and perform GridSearch and we can then assess the performance of the best configuration on the test set.

```
impor
from s
from s
from s
from s

X, y = make_classification(n_samples=2000, n_features=15, random_state=10, n_informative=10,
n_redundant=5)

# We will use 10 fold as the outer loop for nested CV
cv_outer = StratifiedKFold(n_splits=10, random_state=10, shuffle=True)

results = []
for train_indices, test_indices in cv_outer.split(X, y):

        # Seperate the data into test data and training data
        X_train, X_test = X[train_indices, :], X[test_indices, :]
        y_train, y_test = y[train_indices], y[test_indices ]

        # inner cross fold valdiation loop
        cv_inner = StratifiedKFold(n_splits=10, random_state=10 ,shuffle=True)

        model = RandomForestClassifier(random_state=0)
        param_grid = {'n_estimators':[10, 50, 100, 500], 'max_features':[2, 6, 10, 14]}
        grid_search = GridSearchCV(model, param_grid, scoring='accuracy', cv=cv_inner, refit=True)
        result = grid_search.fit(X_train, y_train)
```

Notice we have an outer loop, which is controlled by Stratified Cross Fold Val. Each time the outer loop executes we generate a training split and a test split. We put the test split to one side. We perform GridSearchCV on the training split.

```
ve=10,

# We will use 10 fold as the outer loop for nested CV
cv_outer = StratifiedKFold(n_splits=10, random_state=10, shuffle=True)

results = []
for train_indices, test_indices in cv_outer.split(X, y):

    # Seperate the data into test data and training data
    X_train, X_test = X[train_indices, :], X[test_indices, :]
    y_train, y_test = y[train_indices], y[test_indices ]

    # inner cross fold valdiation loop
    cv_inner = StratifiedKFold(n_splits=10, random_state=10 ,shuffle=True)

    model = RandomForestClassifier(random_state=0)
    param_grid = {'n_estimators':[10, 50, 100, 500], 'max_features':[2, 6, 10, 14]}
    grid_search = GridSearchCV(model, param_grid, scoring='accuracy', cv=cv_inner, refit=True)
    result = grid_search.fit(X_train, y_train)
```

```
        # Continued from previous slide

        # At this point we have finished the inner cross validation loop
        # return the best performing model configuration fit on all training set
        best_model = result.best_estimator_

        # evaluate model on the original test set
        acc = best_model.score(X_test, y_test)
        results.append(acc)

        print("Best Result : ", acc, " with parameters ", result.best_params_ )

print('Overall Accuracy:' , np.mean(results), np.std(results))
```

Once the GridSearchCV finishes (for the current loop iteration) we identify the best model configuration. We then train a new model using this model configuration on the entire training dataset and finally test it on the test set put to one side.

```
# Continued from previous slide

# At this point we have finished the inner cross validation loop
# return the best performing model configuration fit on all training set
best_model = result.best_estimator

#
a
r

p

print(
```

```
Best Result :  0.925  with parameters  {'max_features': 2, 'n_estimators': 500}
Best Result :  0.91  with parameters  {'max_features': 6, 'n_estimators': 500}
Best Result :  0.935  with parameters  {'max_features': 2, 'n_estimators': 500}
Best Result :  0.945  with parameters  {'max_features': 2, 'n_estimators': 500}
Best Result :  0.95  with parameters  {'max_features': 2, 'n_estimators': 500}
Best Result :  0.93  with parameters  {'max_features': 2, 'n_estimators': 500}
Best Result :  0.93  with parameters  {'max_features': 6, 'n_estimators': 50}
Best Result :  0.925  with parameters  {'max_features': 2, 'n_estimators': 500}
Best Result :  0.91  with parameters  {'max_features': 10, 'n_estimators': 100}
Best Result :  0.89  with parameters  {'max_features': 6, 'n_estimators': 100}
Pverall Accuracy: 0.925 0.01688194301613412
```

Once the GridSearchCV finishes (for the current loop iteration) we identify the best model configuration. We then train a new model using this model configuration on the entire training dataset and finally test it on the test set put to one side.