CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

scikit learn

Machine Learning

Machine Learning

Lecture: Model Selection

Ted Scully

Model Selection using Scikit Learn

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

Replication Crisis in Science

- The replication crisis in the scientific community is an ongoing issue of concern. It has been found that many scientific papers and finding cannot be replicated or reproduced.

- For example, in 2015 an attempt to reproduce **100 psychology studies** was only able to replicate only **39** of them.

- In 2018 an international effort to reproduce prominent studies could only reproduce **14 of the 28** replicated, and an attempt to replicate studies from top journals Nature and Science found that **13 of the 21 results** looked at could be reproduced.
    - Falsifying results
    - Cherry-picking data/results
    - Poor methodology

Essay

Why Most Published Research Findings Are False

John P. A. Ioannidis

Summary

There is increasing concern that most current published research findings are false. The probability that a research claim is true may depend on study power and bias, the number of other studies on the same question, and, importantly, the ratio of true to no relationships among the relationships probed in each scientific field. In this framework, a research finding is less likely to be true when the studies conducted in a field are smaller; when effect sizes are smaller; when there is a greater number and lesser preselection of tested relationships; where there is greater flexibility in designs, definitions, outcomes, and analytical modes; when there is greater financial and other interest and prejudice; and when more teams are involved in a scientific field in chase of statistical significance. Simulations show that for most study designs and settings, it is more likely for

factors that influence this problem and some corollaries thereof.

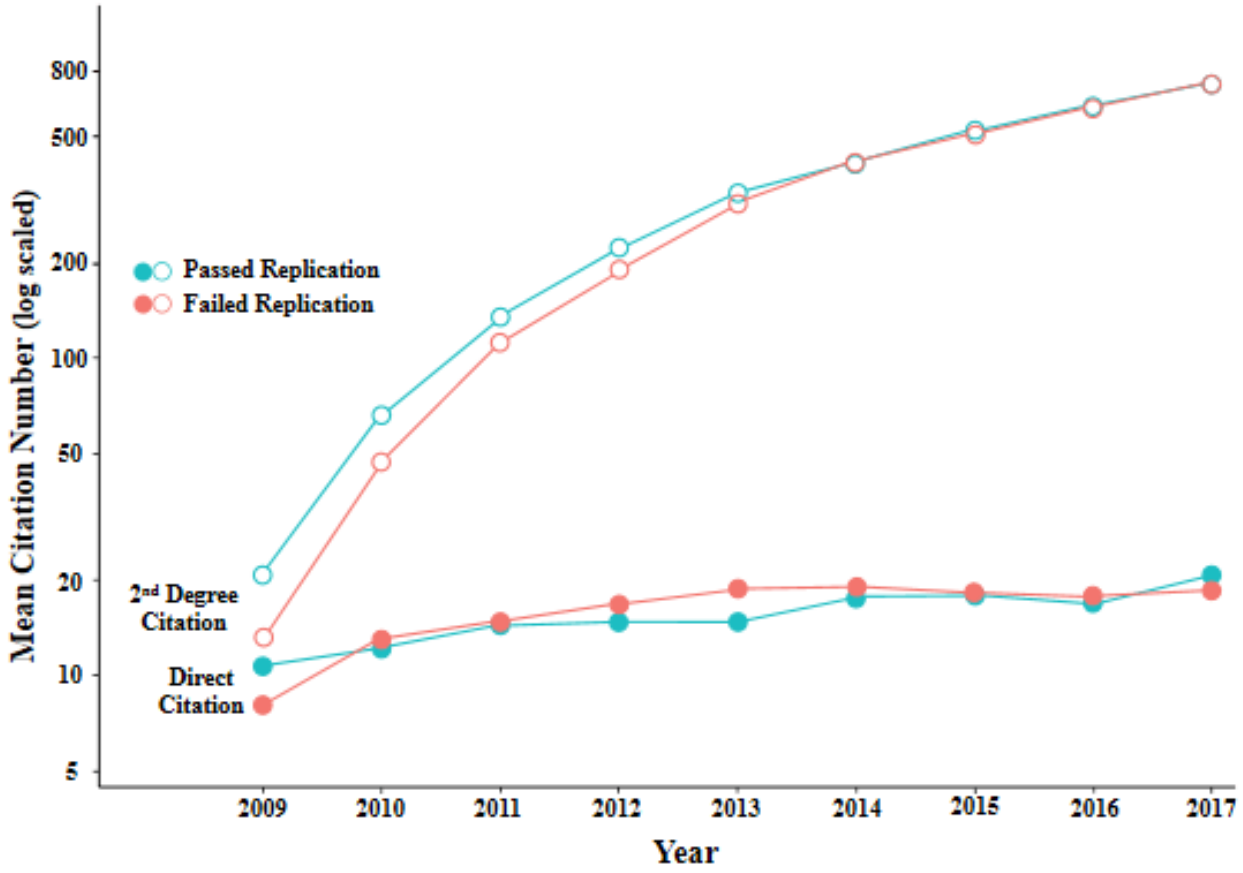
Modeling the Framework for False Positive Findings

Several methodologists have pointed out [9–11] that the high rate of nonreplication (lack of confirmation) of research discoveries is a consequence of the convenient, yet ill-founded strategy of claiming conclusive research findings solely on the basis of a single study assessed by formal statistical significance, typically for a *p*-value less than 0.05. Research is not most appropriately represented and summarized by *p*-values, but, unfortunately, there is a widespread notion that medical research articles

It can be proven that most claimed research findings are false.

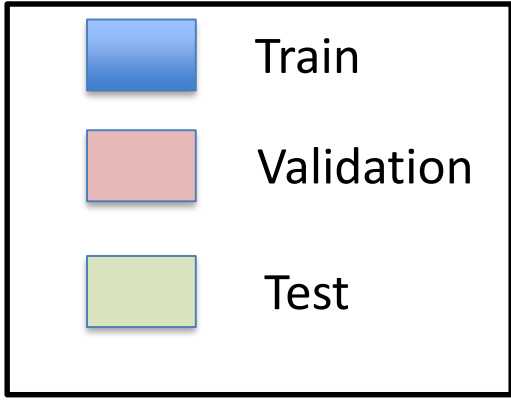
is characteristic of the field vary a lot depending on wh field targets highly likely rel or searches for only one or true relationships among th and millions of hypotheses be postulated. Let us also co for computational simplicity circumscribed fields where is only one true relationshi many that can be hypothesi the power is similar to find several existing true relatio pre-study probability of a re being true is $R/(R+1)$. The of a study finding a true rel reflects the power $1-\beta$ (on the Type II error rate). The of claiming a relationship w truly exists reflects the Type rate, α. Assuming that c rela are being probed in the fiel expected values of the 2×2 given in Table 1. After a res finding has been claimed ba

Consequently many scientific areas where significant theories are grounded on unreproducible experimental work are in turn influencing the direction of future work.



https://www.pnas.org/content/117/20/10762

Holdout Cross Fold Validation

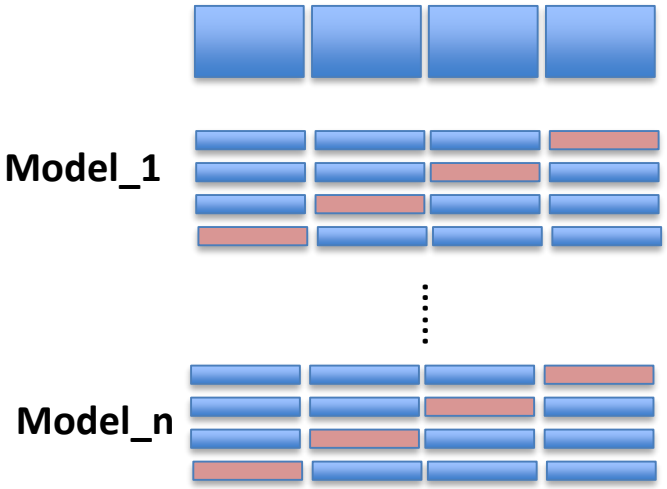


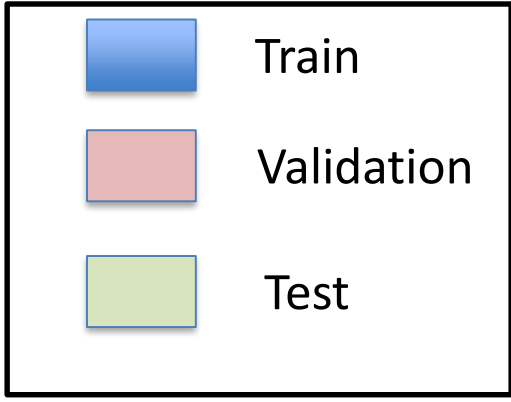
- You will remember with holdout cross validation we split the data into training, validation and test data. We train multiple models using the training set, evaluate them on the validation test and then test the best model on the separate test set.

- 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.

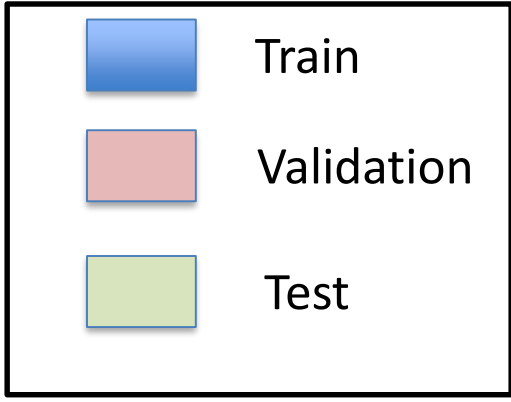
K-Fold Cross Validation

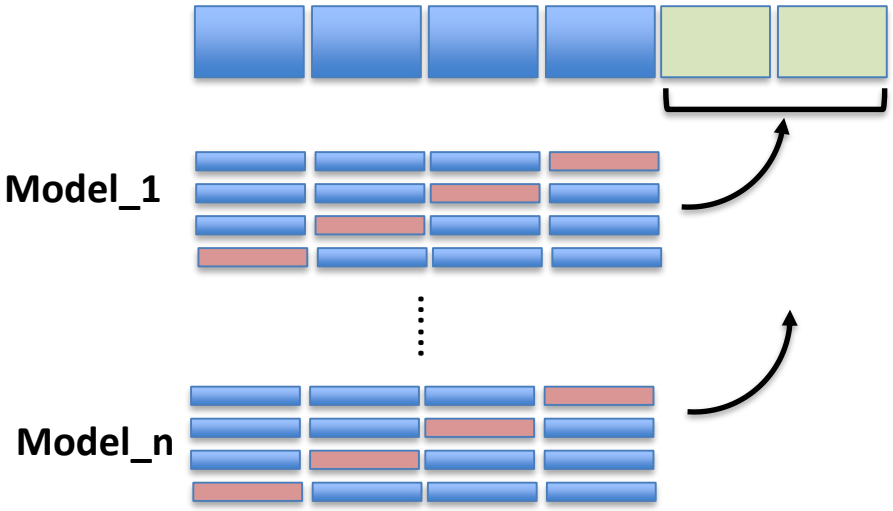




- Let's just for the moment assume that we don't use all the data for cross validation (which is not good practice)
- The problem with just using cross fold validation by itself is that it has been shown to overfit. The accuracy value produced by each model is optimistically biased.
- Therefore, it is recommend that we initially split the data into training and test and use the test set to obtain a final unbiased estimation of the model performance.

Nested Cross Fold Validation - Motivation

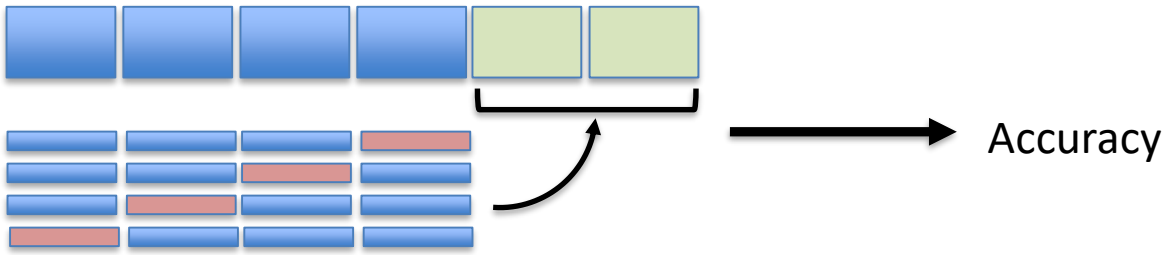


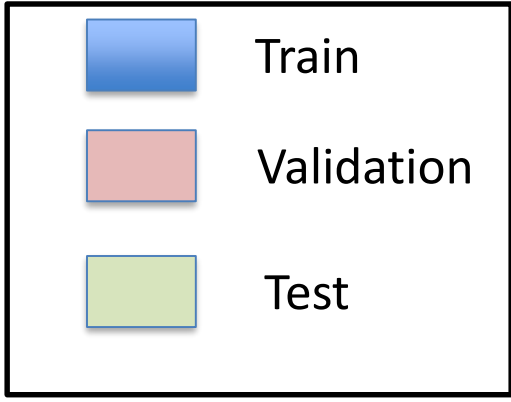


- Of course the final accuracy value we obtain here may be sensitive to how we **partition the data initially into training and test**.

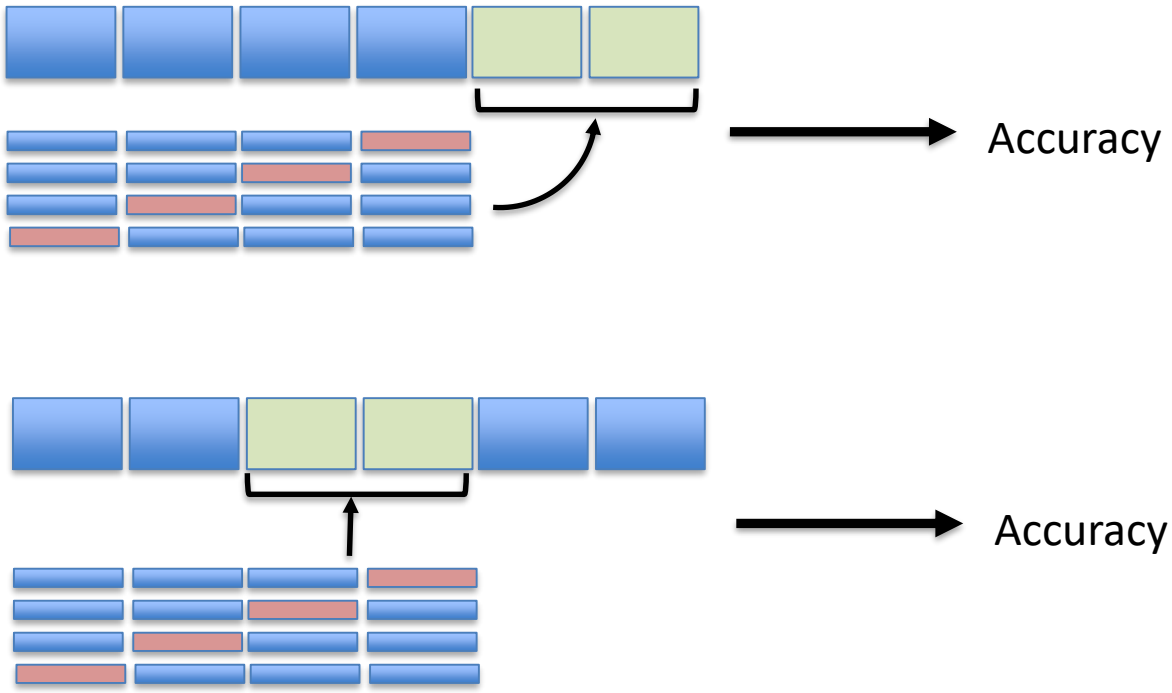
To overcome this problem perhaps we could rotate the test set. In effect, we perform cross fold validation multiple times … this is referred to as **nested cross fold validation**.

Nested Cross Fold Validation

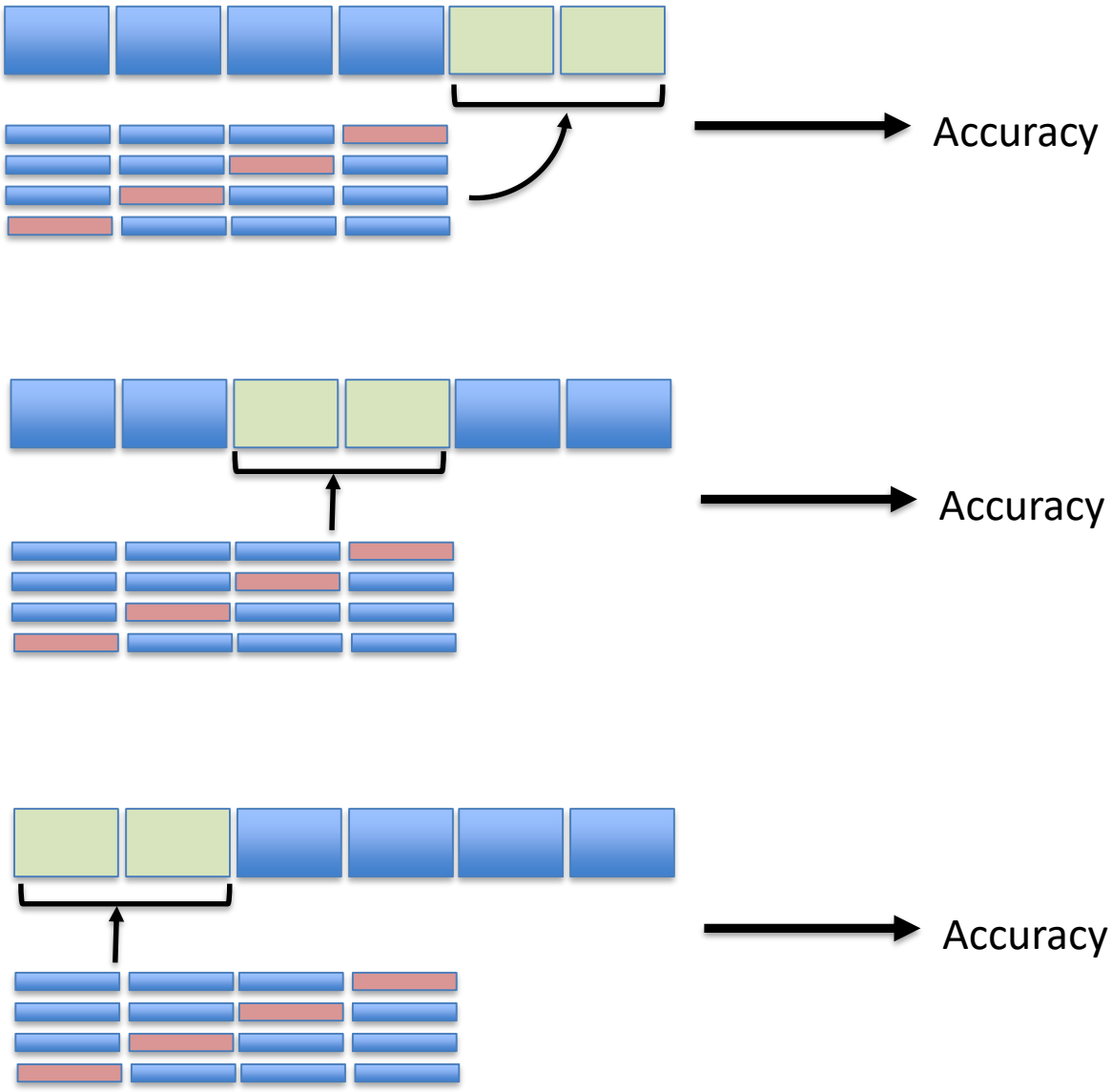




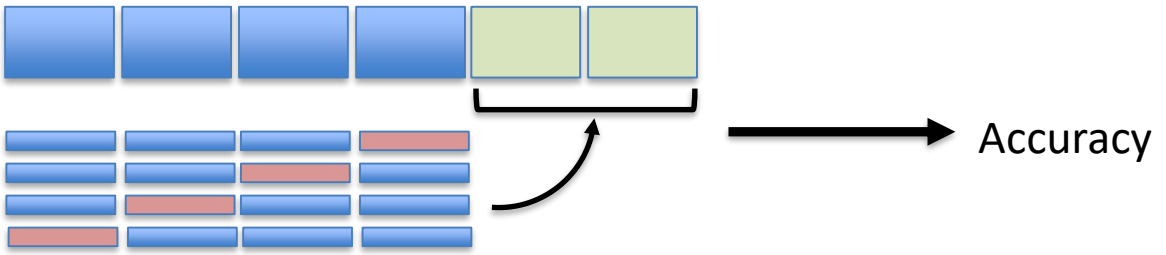
Nested Cross Fold Validation

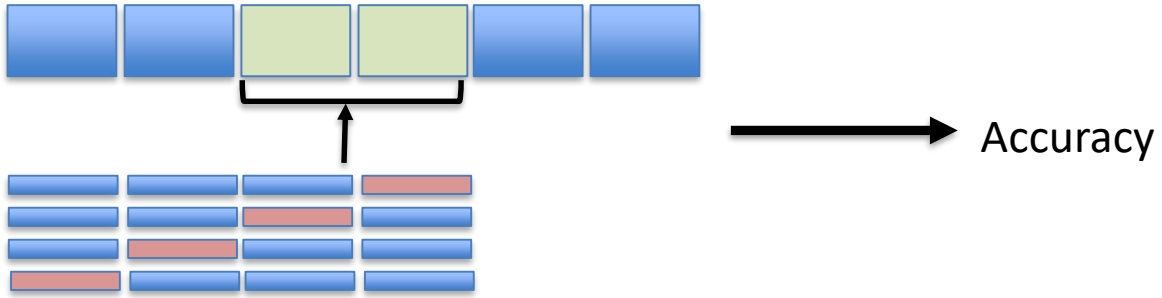


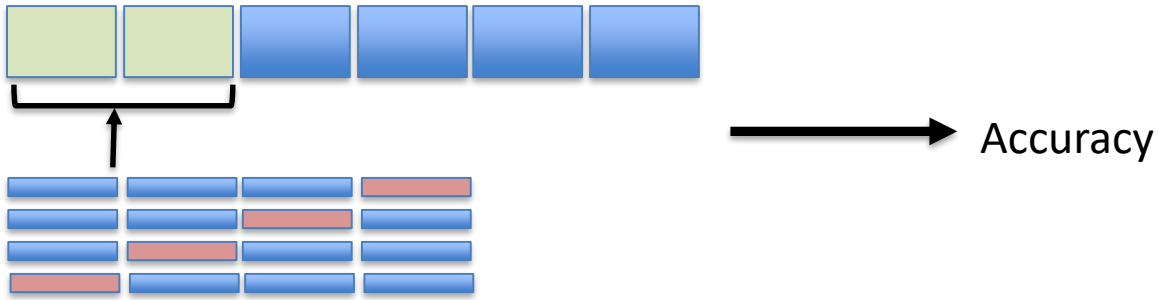
Nested Cross Fold Validation

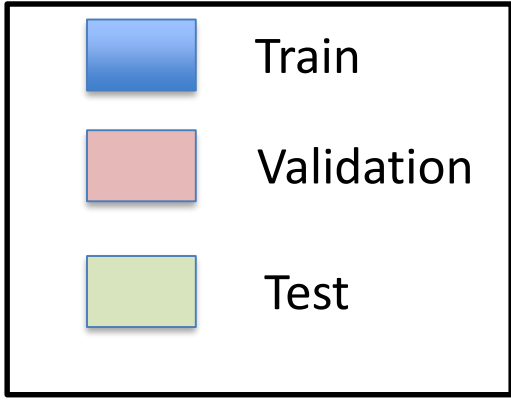


Nested Cross Fold Validation









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 (average the accuracy report from each outer fold)** gives us a **very strong estimate of the unbiased performance of our tuned model**.

Nested Cross Fold Validation

- It provides us with a very strong 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.

Nested Cross Fold Validation

- The reality is that we may not be able to use Nested CV in every situation.
- **Small Dataset**: You should always used nested CV for small data (for example a dataset that contains just a **few thousand instances**).
- **Medium Dataset**: Use Nested CV if at all practical/feasible (for example a dataset that contains **100K/200K rows**). Certain models can take a long time to train such as deep learning models in which case Nested CV is just not practical (but you should used k-fold CV)
- **Large Dataset**: Holdout CV (single training, validation and test set). Less of an issue when dealing with very large datasets.

```
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, n_jobs=-1)
        result = grid_search.fit(X_train, y_train)
```

```
import n
from skle
from skle
from skle
from skle

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, n_jobs=-1)
    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.

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.

```
im
f
f
f
f

X                                                                    =10,
r

# 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, n_jobs=-1)
    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(
```

```
Acc for Iteration:  0.96  with parameters  {'max_features': 2, 'n_estimators': 500}
Acc for Iteration:  0.965  with parameters  {'max_features': 2, 'n_estimators': 500}
Acc for Iteration:  0.955  with parameters  {'max_features': 2, 'n_estimators': 100}
Acc for Iteration:  0.945  with parameters  {'max_features': 6, 'n_estimators': 50}
Acc for Iteration:  0.935  with parameters  {'max_features': 2, 'n_estimators': 500}
Acc for Iteration:  0.975  with parameters  {'max_features': 2, 'n_estimators': 500}
Acc for Iteration:  0.925  with parameters  {'max_features': 2, 'n_estimators': 500}
Acc for Iteration:  0.95  with parameters  {'max_features': 2, 'n_estimators': 500}
Acc for Iteration:  0.975  with parameters  {'max_features': 2, 'n_estimators': 500}
Acc for Iteration:  0.97  with parameters  {'max_features': 2, 'n_estimators': 500}
Overall Accuracy: 0.9555 0.016039014932345 42
```

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.

Model Selection using Scikit Learn

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

Using Pipelines in Scikit Learn

- As we have seen we often have to perform various pre-processing techniques on a machine learning algorithm such as standardization, encoding etc.

- Scikit contains a useful tool called a **Pipeline** class that facilitates this flow of operation by allowing us to **chain together multiple transformative steps** in sequence (it is worth noting that a Pipeline object is itself an transformer object).

- When creating a pipeline we pass it a **list of tuples**. Each tuple specifies:
  - A **string identifier** that we can use to refer to the element of the pipeline
  - A **transformer** (estimator object only if the last tuple in the list)

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import accuracy_score

X, y = load_breast_cancer(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=1)

pipe_lr = Pipeline( [ ('scl', StandardScaler()), ('pca', PCA(n_components=2)),
          ('clf', KNeighborsClassifier()) ])

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

```
from sklearn.model_selection im
from sklearn.preprocessing impo
from sklearn.decomposition imp
from sklearn.neighbors import K
from sklearn.pipeline import Pip
from sklearn.datasets import loa
from sklearn.metrics import accu

X, y = load_breast_cancer(return

X_train, X_test, y_train, y_test = train_te                , y, test_size=0.20,
random_state=1)

pipe_lr = Pipeline( [ ('scl', StandardScaler()), ('pca', PCA(n_components=2)),
          ('clf', KNeighborsClassifier()) ])

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

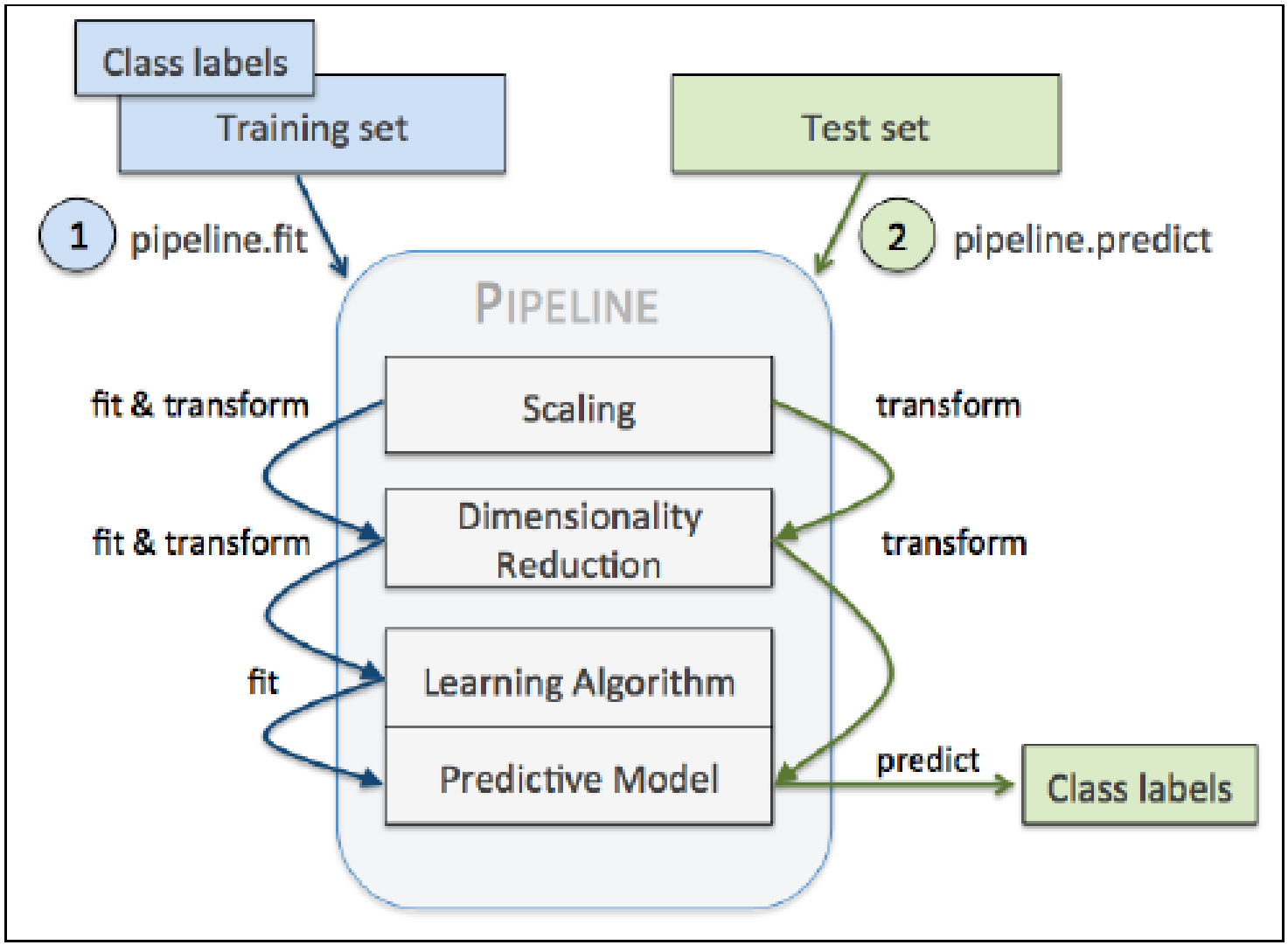
The `Pipeline` object takes a list of **tuples** as input, where the first value in each tuple is an arbitrary **identifier string** that we can use to access the individual elements in the pipeline. The second element in every tuple is a scikit-learn **transformer** or **estimator**.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import ac

X, y = load_breast_cancer(retu

X_train, X_test, y_train, y_test
random_state=1)

pipe_lr = Pipeline( [ ('scl', Sta      caler()), ('pca', PCA(n_components=2)),
          ('clf', KNeighborsC      ifier()) ])

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

Fit all the transformations one after the other and transform the training data, then fit the transformed data using the final estimator (in other words build the ML model).

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors i
from sklearn.pipeline in
from sklearn.datasets ir
from sklearn.metrics im

X, y = load_breast_canc

X_train, X_test, y_train,
random_state=1)

pipe_lr = Pipeline( [ ('scl', StandardS                pca', PCA(n_components=2)),
            ('clf', KNeighborsClassifier())

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

Applies the transformations to the data, followed by the predict method of the final estimator in the pipeline. In other words the pipeline takes in the input data and transforms the data using each of the components in the pipeline. It then uses inputs this transformed data to the model built by the final estimator and it returns a set of predicted classes.


Class labels
Training set
Test set
1
pipeline.fit
2
pipeline.predict
PIPELINE
fit & transform
Scaling
transform
fit & transform
Dimensionality Reduction
transform
fit
Learning Algorithm
Predictive Model
predict
Class labels

Using Pipelines

- The initial steps in a pipeline constitute scikit-learn **transformers**, and the last step is an **estimator (or another transformer)**.

- In the example code, we built a pipeline that consisted of two intermediate steps, (i) a **StandardScaler** and (ii) a **PCA** transformer, and finally a **nearest neighbour** classifier as a final estimator.

- The following actions take place when we executed the fit method on the pipeline **pipe_lr**:
  - The StandardScaler performed fit and transform on the training data, and the transformed training data was then passed onto the next object in the pipeline, the PCA.
  - Similar to the previous step, PCA also executed fit and transform on the scaled input data and passed it to the final element of the pipeline, the estimator.
- There isn't any upper limit to the number of intermediate steps in this pipeline

Using Pipelines for Cross Fold Validation

- (Notice in the previous example, we had test data to test the model produced from the pipeline)
- In many cases a pipeline is used to assemble several steps that can be cross-validated together while setting different parameters.

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import cross_val_score

X, y = load_breast_cancer(return_X_y=True)


pipe_lr = Pipeline([('scl', StandardScaler()), ('pca', PCA(n_components=2)),
            ('clf', KNeighborsClassifier())])


results = cross_val_score(pipe_lr, X, y, cv=10)

print(results.mean())
```

Using Pipelines with GridSearch

- We can also easily use a pipeline with grid search for hyper-parameter optimization.
- One of the benefits of this approach is that we can now incorporate **parameters of the transformers** in the search process.
- For this, a pipeline enables setting parameters of the various steps using **their names** and the **parameter name** separated by a **'__',** as in the example below
  - **transformerName__parameterName**.

```
from sklearn.model_selection import GridSearchCV

X, y = load_breast_cancer(return_X_y=True)


pipe_lr = Pipeline([('scl', StandardScaler()), ('pca', PCA(n_components=2)),
            ('clf', KNeighborsClassifier())])


param_grid = {pca__n_components:[2, 3, 4, 5], clf__n_neighbors: list(range(1, 30, 2))

grid_search = GridSearchCV(pipe_lr, param_grid=param_grid)
grid_search.fit(X, y)
print(grid_search.best_estimator_, grid_search.best_score_)
```

Model Persistence

- After tuning the performance of your model you will most likely want to persist the model for future use.
- It is possible to save and load a scikitlearn model using **joblib** (joblib provide a replacement for pickle Python objects containing large data, in particular large NumPy arrays).
- The following example shows how we can save an SVM model that we have tuned for the titanic dataset and then reload.

```
from sklearn.externals import joblib

param_grid = [ {'kernel': ['rbf', 'poly', 'linear'],  'C':range(1,15)}  ]
clf = GridSearchCV(SVC(), param_grid, cv=10)
clf.fit(data, target)


joblib.dump(clf.best_estimator_, 'titanic_svm.joblib')

loadedSVM = joblib.load('titanic_svm.joblib')
```