
# Fourth Industrial Summer School

## Module 4: ML

## Supervised Learning: Classification

Friday, July 5, 2024

# Session Objectives

- ✓ Performance Evaluation
  - ✓ Classification Accuracy
  - ✓ Resampling for robust modeling
  - ✓ Confusion Matrix (CM)
  - ✓ Model Selection & cross validation

# Evaluation

- Accuracy is one metric that aggregates the model performance as one value, but it can be broken down into its ingredients.
- Depending on accuracy alone:
  - may mislead the model evaluation and selection, and
  - will not pinpoint what went wrong during the modeling.

# Accuracy

- Recall that:

$$Accuracy = \frac{\#correctPredictions}{\#total\ instances}$$

- Suppose having a tumor classification problem of two classes:
  - Positive class (P) (or Malignant tumors)
  - Negative class (N) (or Benign tumors)

- Using **1000** testing data, **the trained model reported 99.0%** classification testing **accuracy**!

***Is that an amazing performance??***

# Facts on the problem

- Out of 1000 randomly selected patients, on average
  - 10 patients are relevant and a **Positive class**
  - The rest of the patients are **Negative class**.
- Suppose now, we have another model (2) which is **blindly predicting** the most frequent class (i.e. **Negative class**).
- So, with **model2,** the expected accuracy of 1000 samples testing set is:

$$Accuracy = \frac{990}{1000}(\%) = \mathbf{99.0\%}$$

- It is also known as ***null accuracy***.
- **Model2** is ignoring the positive -class completely, and yet it shows high **accuracy!**
- Usually, the **null accuracy** is used as a **baseline, and the target is to build a model that has an improved performance**.

# How to compute accuracy using Sklearn?

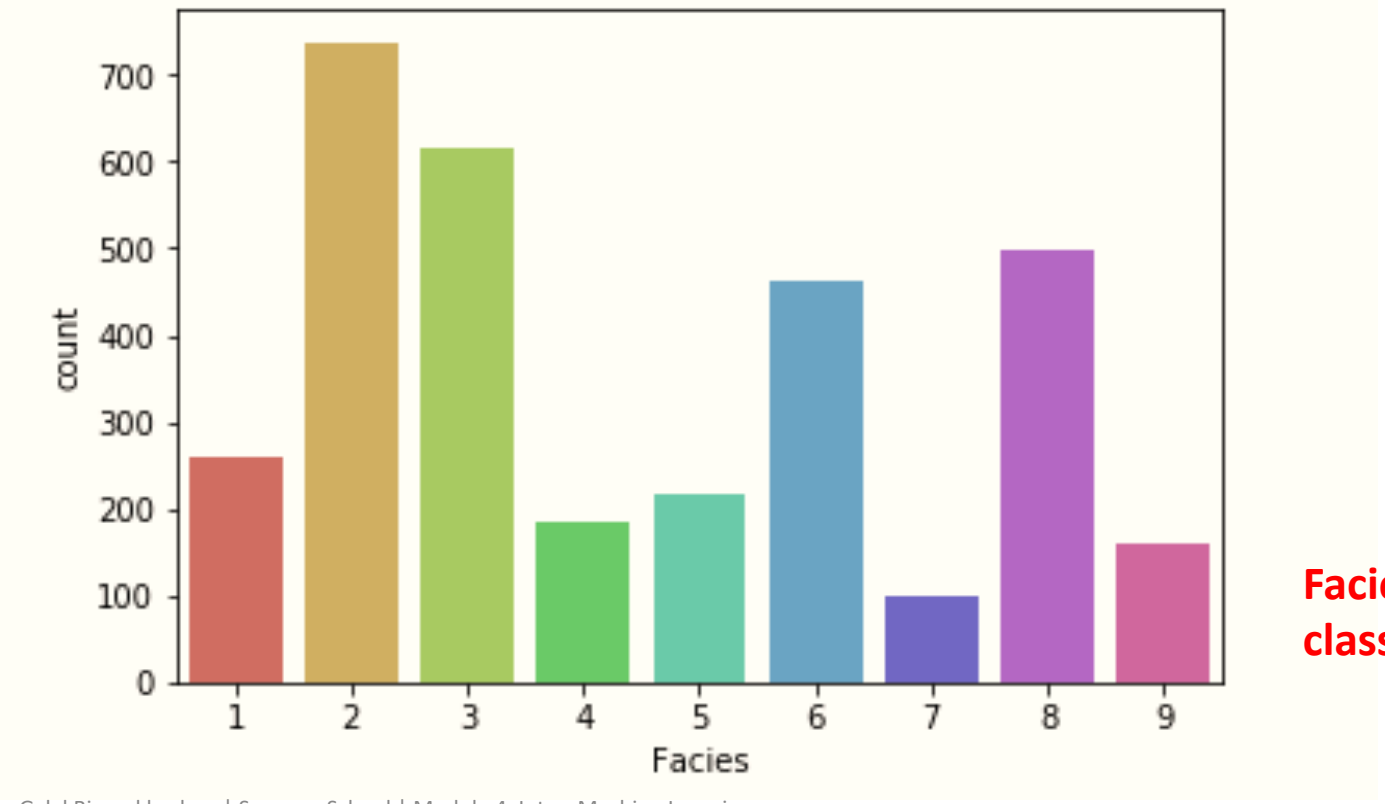
- **Usage:**

1. Import 'accuracy_score' from **metrics** package
2. Pass the genuine test labels as the first argument
3. Pass the predicted test labels as the second argument
4. accuracy_score will compute and return the accuracy score.

```
# Compute accuracy
from sklearn import metrics
metrics.accuracy_score(y_test, y_pred)
```

# Example: compute the baseline accuracy

```
# is there any imbalanced among classes
import seaborn as sns
sns.countplot(x='Facies', data = data, palette='hls')
plt.show()
```




**Facies classification**

# Compute the null (baseline) accuracy

- Find the most frequent class

```
# Since there is class imbalance, let's check null accuracy
nullaccuracy = np.max(data['Facies'].value_counts()/data['Facies'].value_counts().sum())
print("Null accuracy:%.2f" % nullaccuracy)
```

```
Null accuracy:0.23
```

# Dummy classifier: baseline accuracy score

- Also, we can build a **dummy classifier** with most **frequent strategy** to report the base accuracy

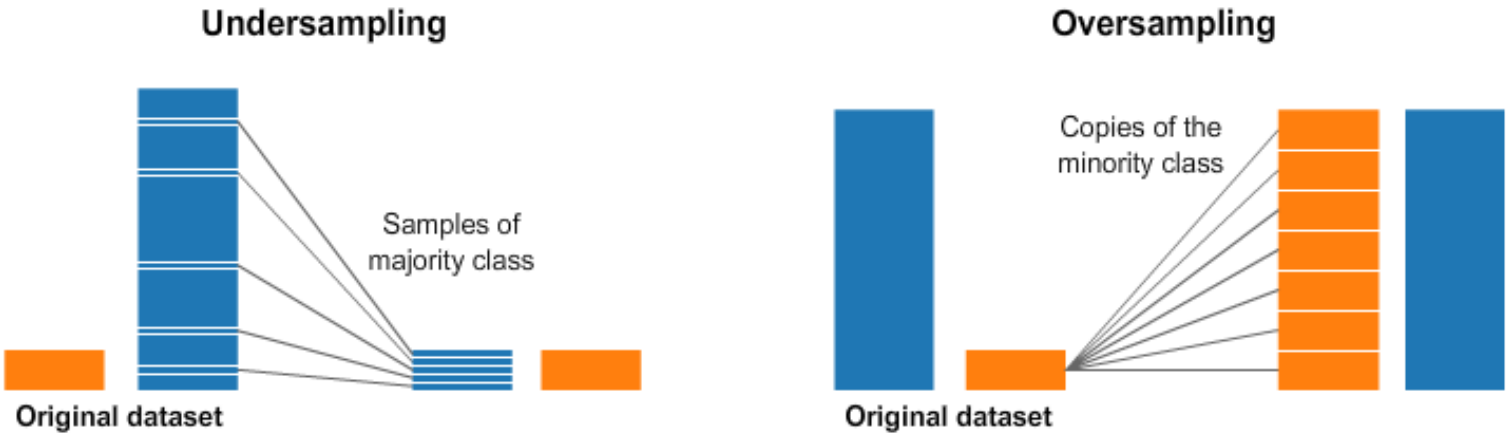
```
1 from sklearn.dummy import DummyClassifier
2 basemodel = DummyClassifier(strategy='most_frequent')
3 basemodel.fit(data, labels)
4 basemodel.score(data, labels)
```

- If performance of developed model is close to baseline accuracy, it should warn you of an issue:
  - Large class imbalance exists in the data that affects the learning!
  - Ineffective, erroneous or missing features
  - Poor choice of hyperparameters or feature transformation

# Balancing the Training Data Set

- As we have seen in the medical problem, the data usually skewed or imbalanced!
- We may need to balance our training data to avoid building biased models.
- The purpose is to provide the classification algorithms with a better balance of records for each category.
- We should balance **the training set!** As an example, the portion of positive class should be increased by ***resampling with replacement***.

# Balancing the Training Data Set




- Balancing the training data can be done using oversampling or under-sampling.
  - The oversampling may cause model overfitting as of duplicating samples of the minor class for the sake of building unbiased model
  - The under-sampling may cause information loss and model underfitting
  - But either methods may lead to better modeling than the one created with the significant unbalanced dataset


Source: https://www.kaggle.com/code/rafjaa/resampling-strategies-for-imbalanced-datasets

# Balancing the Training Data Set

- Resampling is the process of sampling at random and usually we use it with replacement to balance the training data.
- Let's suppose our dataset has **1k** (**P**) class, and **100k** (**N**) class instances (k=1000).
- The **P** samples represents **1%** of the dataset, Suppose we want to increase this to **25%**.
- In this case, we need to add **32000** samples. We can do that by resampling **P** class instances. So, in total we'll have ***33000*** samples.

***Note: the test data should never be balanced!***

# Python – Pandas sampling

- Instances per class

```
1 Medical_train['Cases'].value_counts()
```

```
1    357
0    212
Name: Cases, dtype: int64
```

- Find how many samples we need to increase the other class to reach 50%

```
2 sapmles  = int((0.5*(569) - 212) / (1-0.5))
3 sapmles
```

```
145
```

- Perform resampling and increase the rare class instances

```
1 # Using Pandas DataFrame
2 #1. get the rare class data
3 to_resample = Medical_train.loc[Medical_train['Cases'] == 0]
4
5 #2. perform the sampling with replacement
6 our_resample = to_resample.sample(n = 145, replace = True)
7
8 #3. concatonate back the new sampled data to the dataset
9 Medical_train_upsampled = pd.concat([Medical_train, our_resample])
```

- Check the new training dataset

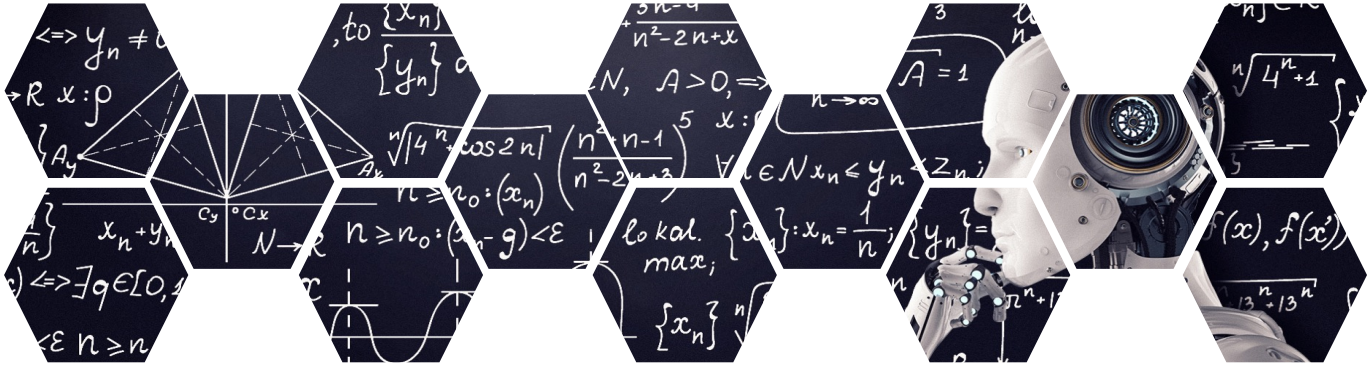
```
1 Medical_train_upsampled['Cases'].value_counts()
```

```
0    357
1    357
Name: Cases, dtype: int64
```

There is a library that can help in such issue **imbalanced-learn, reach it here**

# Exercises (3 – 5)

- Null accuracy using dummy classifier
- Simple model Selection based on accuracy
- Resampling data for better modeling



# Model Performance Evaluation

An application of **KNN** Method

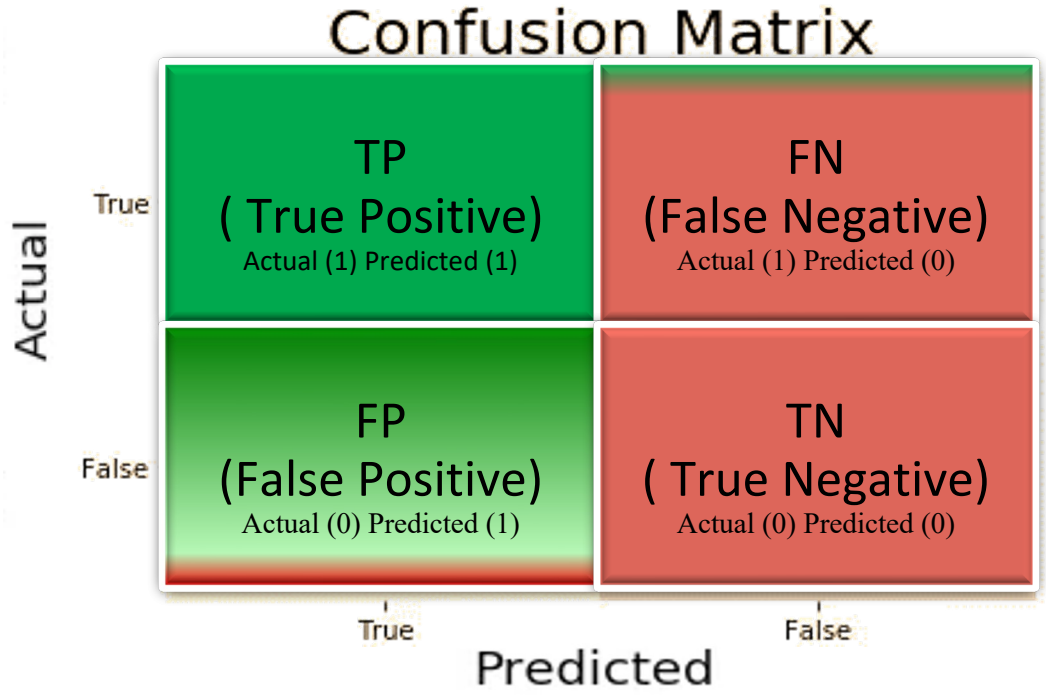
- Metrics
- Strategies
- Modeling selection

# Model performance: Binary Setting

- Instead of using only accuracy for our model evaluation,
- We can break-down the evaluation into its ingredients
  - What makes that accuracy and how to improve?
- One way to do that is to keep track on the model decision on each sample, then analyzing its decisions (correct or incorrect) that will help in improving/understanding the model performance.


Confusion matrix

# Confusion Matrix




Label 1 = positive class (class of interest)

Label 0 = negative class (everything else)

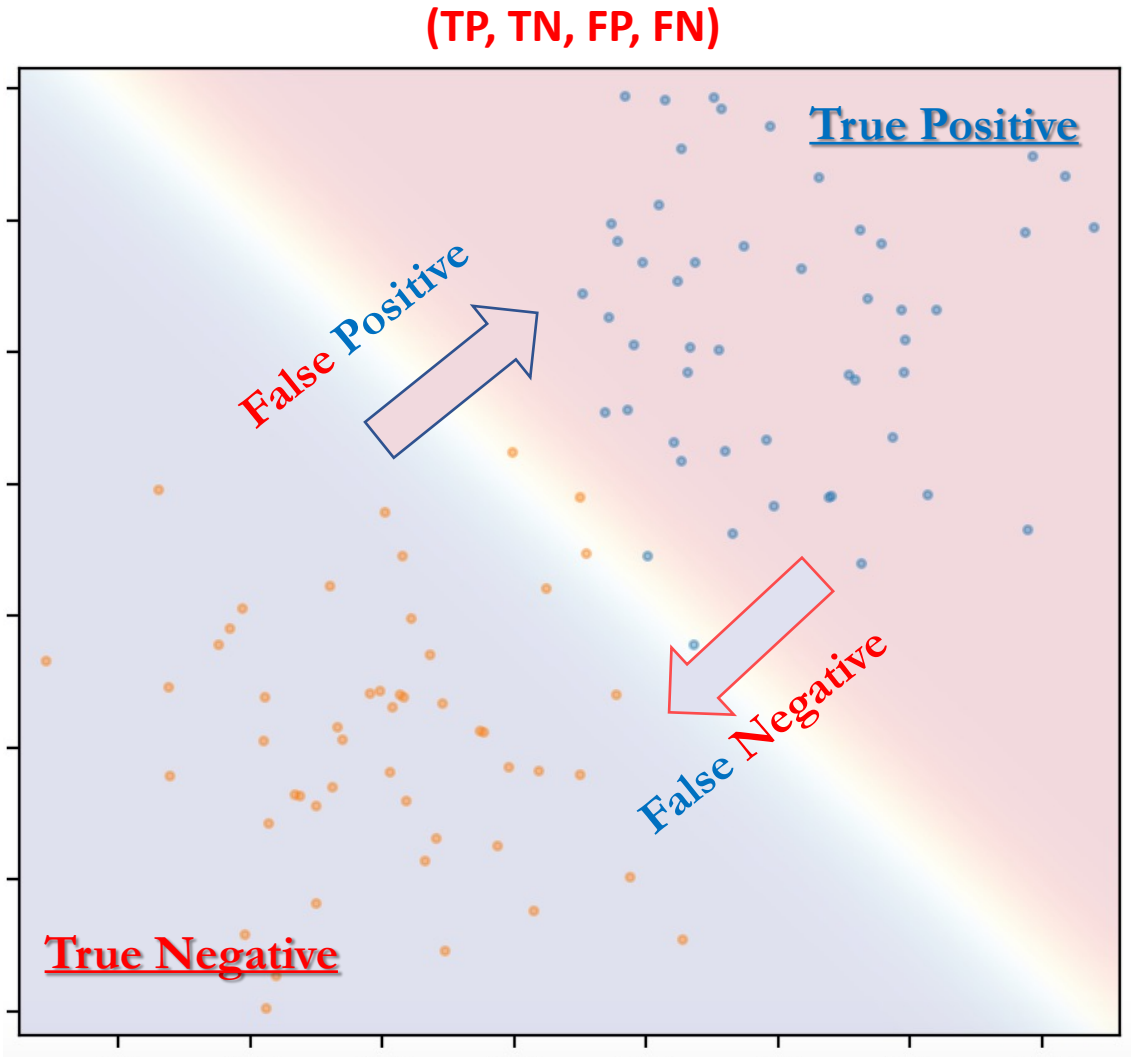
TP = true positive,  FP = false positive

TN = true negative, FN = false negative

# Confusion Matrix

- Every test instance is landing in exactly one box.
- CM reveals the model errors ***False positive*** (samples that are similar to positive class, but they are not) and ***False negative*** (samples that are positive class, but the model fails to recognize as one) .
- CM provides more information than simple accuracy.
- Helps us investigate data samples deeply, looking at the erroneous samples.
- Allow us to deriving multiple measurements rather than accuracy only, which shows the model strengths or weakness.

# Confusion matrix: Decision Space



# Summarize CM: Accuracy

- It is obvious that we can compute the accuracy of the model using CM as:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$accuracy = \frac{40 + 82}{40 + 82 + 15 + 4} = 86.5$$

# Summarize CM: Errors

- How about the classification errors (**1 - accuracy**). For what fraction our classifier predictions were incorrect.

$$Error\_Frac = \frac{FP + FN}{TP + TN + FP + FN}$$

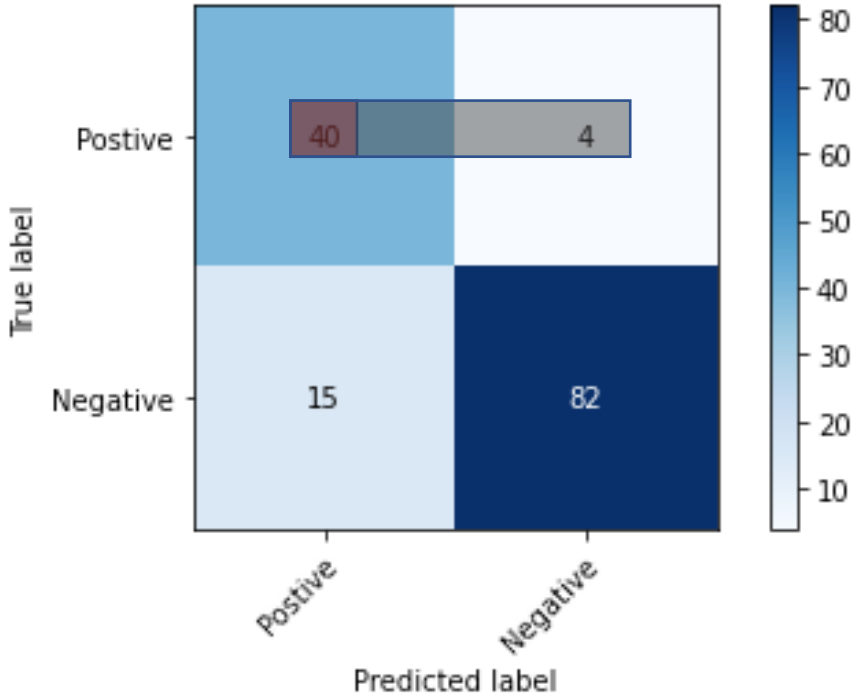
| | | |
|---|---|---|
| True Positive | TP=40 | FN =4 |
| True Negative | FP = 15 | TN=82 |
| | Predicted Positive | Predicted Negative |

$$Error\_Frac = \frac{15 + 4}{40 + 82 + 15 + 4} = 13.5$$

# Summarize CM: Model Recall

- **Recall** interprets the fraction of all positive samples; our classifier has correctly identified as positive.

  - $Recall = \frac{TP}{TP+FN}$
  - $Recall = \frac{40}{40+4} = 90.9$

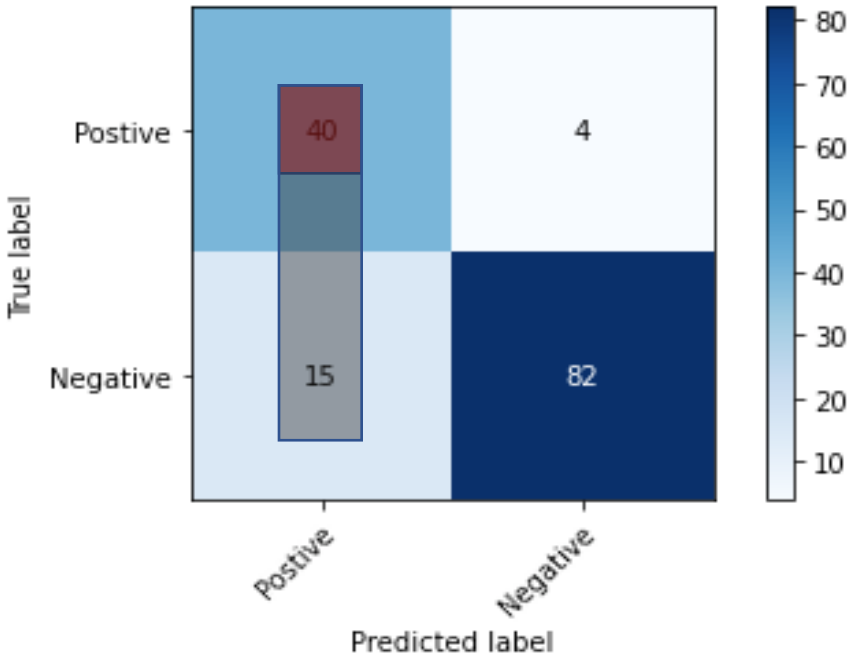



- Also, it is called True Positive Rate (**TPR**), Sensitivity, and Probability of detection.

# Summarize CM: Precision

- Precision interprets the fraction of positive predictions that are correct.

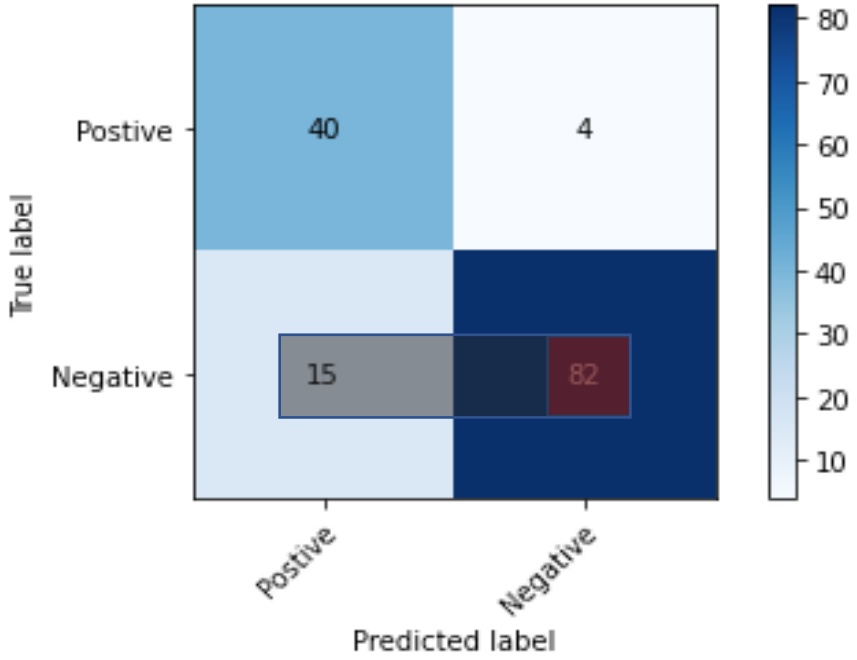
  - $Precision = \frac{TP}{TP+FP}$
  - $Precision = \frac{40}{40+15} = 0.73$



# Summarize CM: Specificity

- Specificity measures the ability to classify a negative samples

  - $Specificity = \frac{TN}{TN+FP}$
  - $Specificity = \frac{82}{82+15} = 0.85$



# Summarize CM: F1-Score

- A combination of **Precision** and **Recall** is called **F1-score** or **F-measure**.

- It represents a harmonic mean of the two metrics, and is computed as:

$$F1_{score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

```
# Compute F1-score
from sklearn import metrics
metrics.f1_score(y_test, y_pred, average='macro')
```

# Example Binary classification

| Y | 0 | 1 | 0 | 1 |
|---|---|---|---|---|
| Yhat | 1 | 1 | 1 | 0 |

| Confusion matrix | Actual | P | 1 | 1 |
|---|---|---|---|---|
| | | N | 2 | 0 |
| | | | P | N |
| | | | Predicttion | |

- Accuracy = 25%
- Error = 75%
- Sensitivity = 50%
- Specificity = 0
- Recall = 50%
- Precision = 33%
- F1score = 39.7%

# Multiclassification

- Sklearn supports two styles binary or multi-class evaluation. In multi-class, we need to select an average type to report the metric:

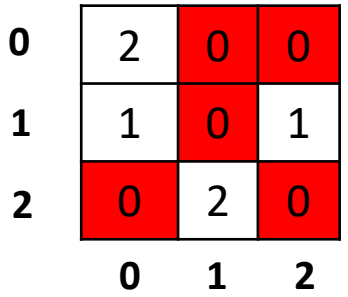
$$macro_{precision} = \frac{\sum_i Precision_i}{\sum_{i \in c} i}$$

$$weighted_{precision} = \sum_{i \in C} w_i \times Precision_i \qquad w_i = \frac{samples(class_i)}{samples(Dataset)}$$

- Macro option is computing an arithmetic mean of a metric ( precision, recall)
- Weighted option is a macro but multiplied by a class weight. Usually, it is used when we encounter imbalanced dataset

# Example

- y_true = [0, 1, 2, 0, 1, 2]
- y_pred = [0, 2, 1, 0, 0, 1]




- Instances per class: **class0** = 2, **class1** = 2, **class2** = 2
- Performance macro precision $= \frac{(0.66 + 0 + 0)}{3} = 0.22$
- Weighted precision $= \frac{2}{6} \times 0.66 + \frac{2}{6} \times 0 + \frac{2}{6} \times 0 = 0.22$

# Sklearn: Confusion matrix

- From Facies classification exercise

```
from sklearn.metrics import confusion_matrix
# Can be computed as below
mtConf = confusion_matrix(yf_ts, ypred)
print(mtConf)
```

```
[[ 50   9   2   3   0   0   0   0   0]
 [ 15 106  10   9   3   1   3   1   0]
 [  3  22  71   6   3   2   0   2   0]
 [  0   4   0  31   3   1   2   1   0]
 [  1   2   1   5  28   9   0   1   0]
 [  2   3   0  13  14  57   1   8   2]
 [  0   0   0   0   2   2  11   2   2]
 [  2   3   1   8   8  15   2  42   6]
 [  0   0   0   1   0   0   0   0  30]]
```

# To visualize CM

- Load the confusion matrix display module from ***sklearn.metrics***

```
from sklearn.metrics import ConfusionMatrixDisplay
```

- Three Possible ways:
  1. Confusion Matrix and labels: Pass the computed confusion matrix and labels

```
>>> disp = ConfusionMatrixDisplay(confusion_matrix=cm,
...                               display_labels=clf.classes_)
>>> disp.plot()
<...>
>>> plt.show()
```

  2. From Estimator: Pass the classifier, and test data

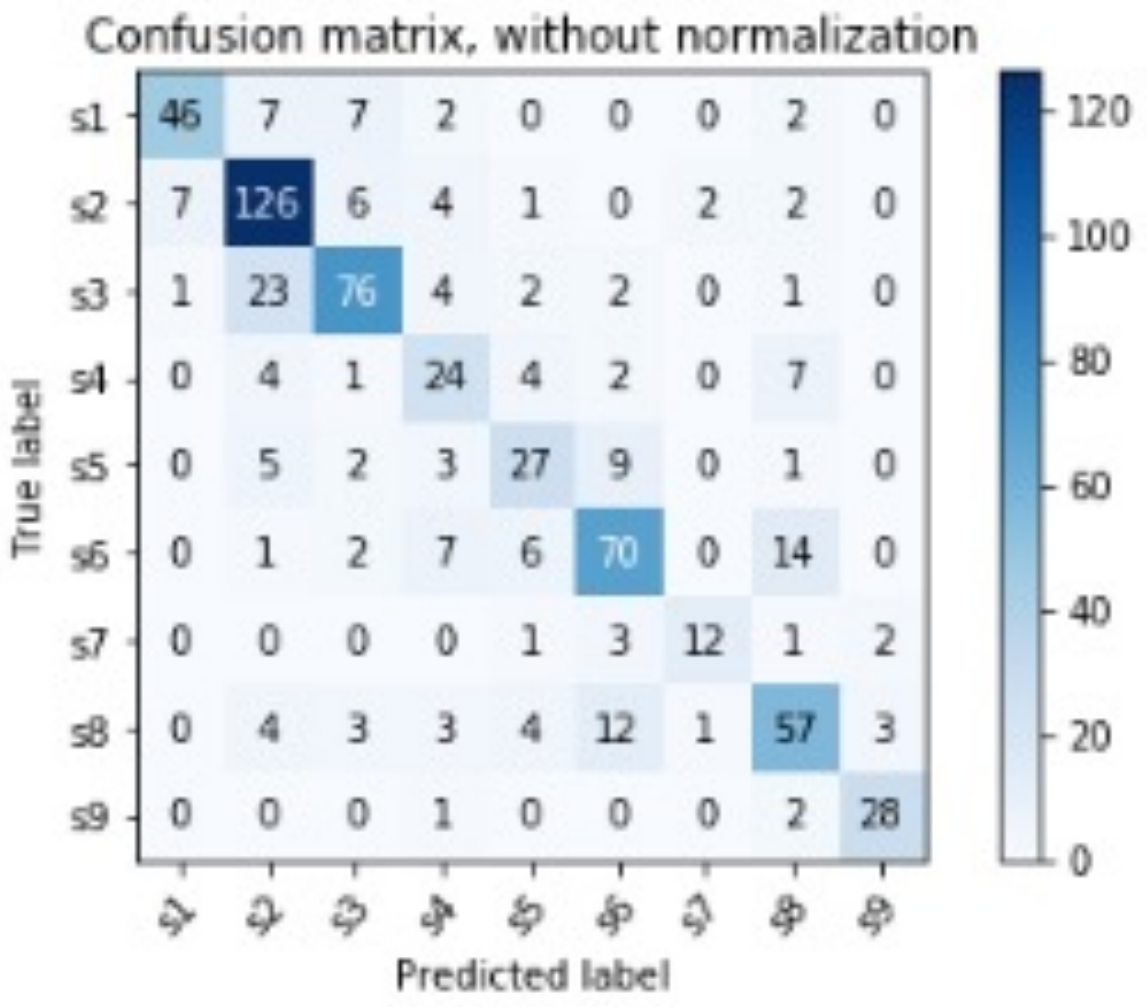
```
>>> ConfusionMatrixDisplay.from_estimator(
...     clf, X_test, y_test)
<...>
>>> plt.show()
```

  3. From prediction: Pass the actual and predicted labels

```
>>> ConfusionMatrixDisplay.from_predictions(
...    y_test, y_pred)
<...>
>>> plt.show()
```

Plot code: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ConfusionMatrixDisplay.html?highlight=confusionmatrixdisplay#sklearn.metrics.ConfusionMatrixDisplay

# Visualizing CM



# Precision-Recall Tradeoff

- There are a tradeoff between Precision and Recall in most applications
- If both metrics are high, the better the performance.
- Recall-oriented machine learning tasks:
  - Often associated with a human expert to filter out false negative, the model recall is low!
    - In tumor detection (ML will predicted an **N** and ask for more examination to make sure!)
    - In forensics (ML will predict a fraudulent transaction and ask for more investigation)
- Precision-oriented machine learning tasks:
  - No need for a human expert to filter out false positives. In other words, it is okay to have incorrect positive samples in the results, the model precision is low!
    - Example: Search engine ranking, query suggestion: as the queries may return some **unrelated pages**, but not harmful to have

# Precision / Recall

- If we have binary classification

**Binary** classification

```
# Compute precision
from sklearn import metrics
metrics.precision_score(y_test, y_pred)
```

**Multi-class** classification

```
# Compute precision
from sklearn import metrics
metrics.precision_score(y_test, y_pred, average='macro')
```

**Binary** classification

```
# Compute Recall
from sklearn import metrics
metrics.recall_score(y_test, y_pred)
```

**Multi-class** classification

```
# Compute Recall
from sklearn import metrics
metrics.recall_score(y_test, y_pred, average='macro')
```

# Classification Report

```
from sklearn.metrics import classification_report
labels = ['SS', 'CSiS', 'FSiS', 'SiSh','MS', 'WS', 'D','PS', 'BS']
print(classification_report(y_test, y_pred, target_names=labels ))
```

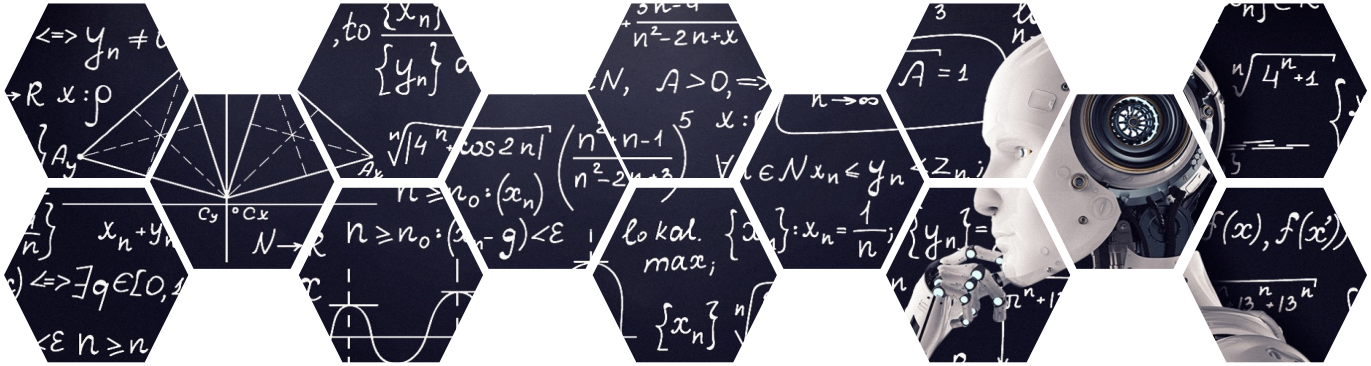
```
              precision    recall  f1-score   support

          SS       0.74      0.62      0.67        50
        CSiS       0.67      0.79      0.73       141
        FSiS       0.78      0.71      0.74       136
        SiSh       0.74      0.63      0.68        49
          MS       0.69      0.48      0.56        46
          WS       0.57      0.73      0.64        70
           D       0.88      0.58      0.70        24
          PS       0.65      0.72      0.68        97
          BS       0.93      0.76      0.84        34

   macro avg       0.74      0.67      0.69       647
weighted avg       0.71      0.70      0.70       647
```

# Exercises (6 , 7)

- Confusion matrix
- Understand model performance



# Model Performance Evaluation

An application of **KNN** Method

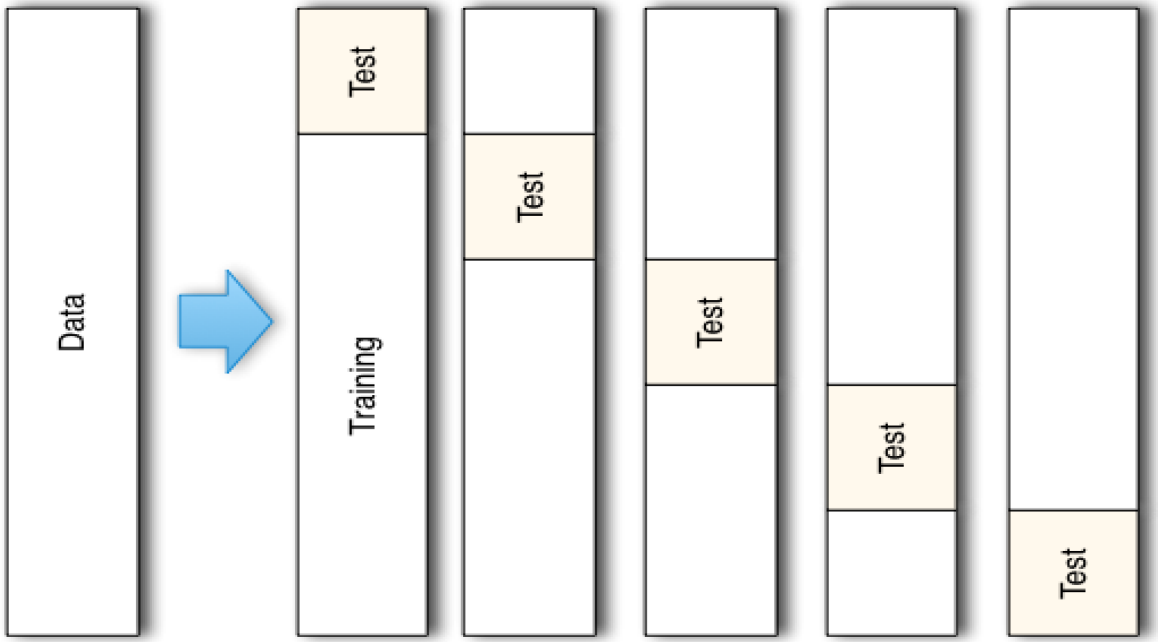
- Metrics
- Strategies
- Modeling selection

# Model Evaluation Strategies

- **Split the data into training /testing sets**
  - It allows the model to be trained and tested on different data samples
  - Useful due to its speed and simplicity.
  - All we need is to split the data into two sets, build the model on training and test it directly on testing dataset.
  - However, we may have a cost of manual tuning the parameters and sample the best training and testing spaces
  - Less robust modeling, as the model might be performed well by chance of picking the lucky training and testing samples! **And not due to generalization**

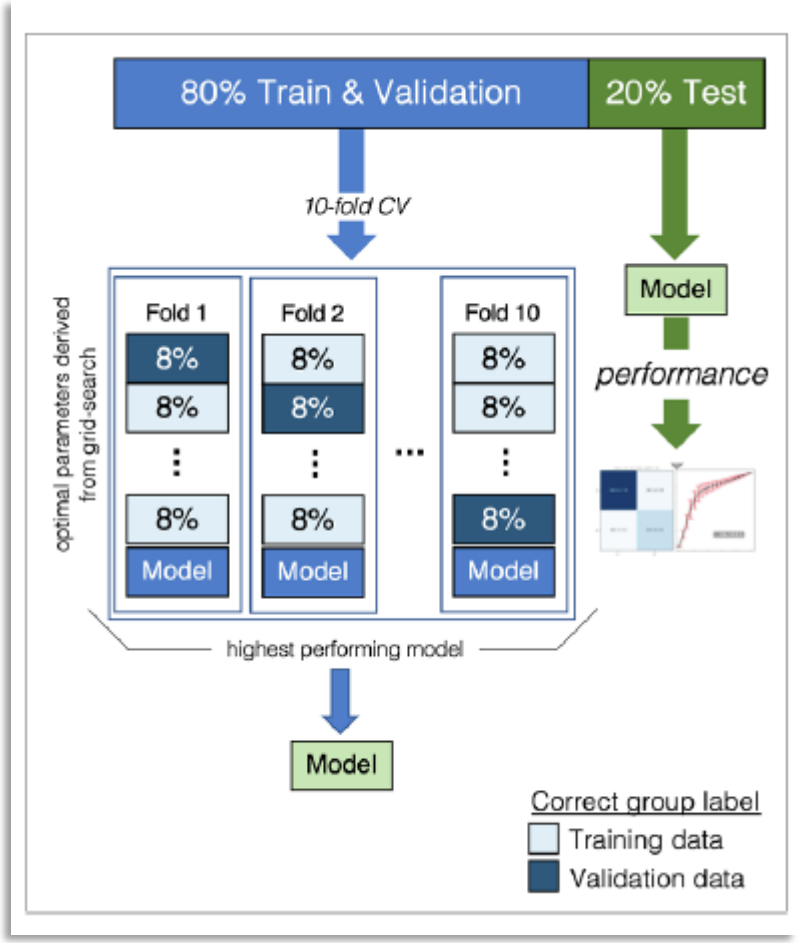
# K-fold cross validation

- K-fold cross validation
  - Systematically creates *K* train/test (validate) splits
  - Each time a model is trained on a larger part and tested on smaller part
  - We can report the average of the test results of all *K* splits
  - We can exploit it for model selection as well



# Cross validation: Model Selection

- In a research paper on temporal lobe epilepsy classification using MRI data, the authors used two strategies as depicted on the left to build and select the optimal model
- First the data divided into 80-20 training and testing respectively.
- Then, several models has been evaluated using cross-validation
- Remember: any data processing has to be done on training only, then, build the model, and finally use the training preprocessing factors on validation data before prediction
- Once you collected the models, pick the best one and use it on the hold out (testing) dataset (includes the preprocessing factors) to report how good is your model.

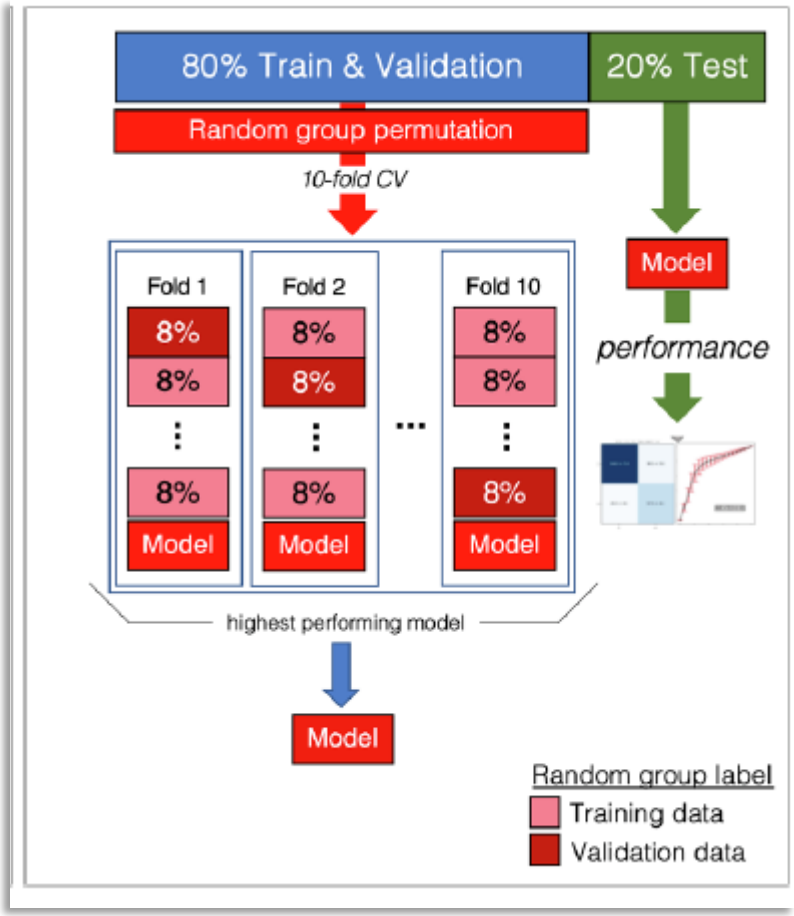




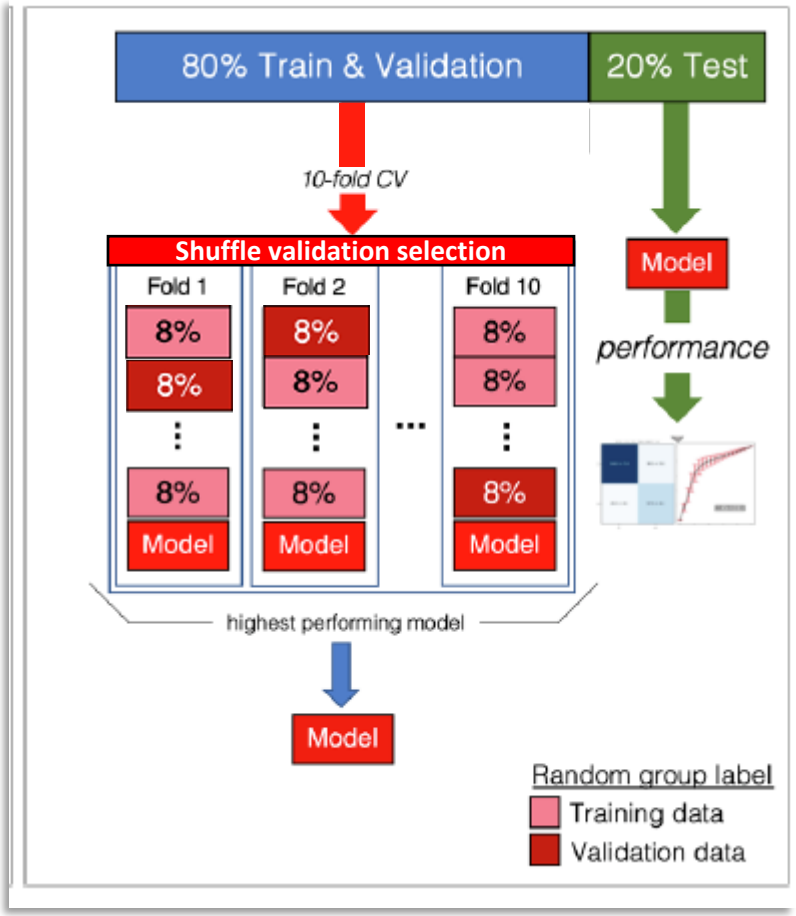
Source: https://www.sciencedirect.com/science/article/pii/S2213158221002096

# Contd..

**Shuffle training data to overcome sample order issue**




**Shuffle validation data to impose random selection**



# Sklearn: KFold

- Let us see how sklearn arranges our folds ( case 1)

```
1 from sklearn.model_selection import KFold
2 import numpy as np
3
4 data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
5                  11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
6 kfolds = KFold(n_splits =10, shuffle = False)
7 for train, val in kfolds.split(data):
8   print('train:%s, test: %s'% (data[train], data[val]))
```

```
train:[ 3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20], test: [1 2]
train:[ 1  2  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20], test: [3 4]
train:[ 1  2  3  4  7  8  9 10 11 12 13 14 15 16 17 18 19 20], test: [5 6]
train:[ 1  2  3  4  5  6  9 10 11 12 13 14 15 16 17 18 19 20], test: [7 8]
train:[ 1  2  3  4  5  6  7  8 11 12 13 14 15 16 17 18 19 20], test: [ 9 10]
train:[ 1  2  3  4  5  6  7  8  9 10 13 14 15 16 17 18 19 20], test: [11 12]
train:[ 1  2  3  4  5  6  7  8  9 10 11 12 15 16 17 18 19 20], test: [13 14]
train:[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 17 18 19 20], test: [15 16]
train:[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 19 20], test: [17 18]
train:[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18], test: [19 20]
```

# Sklearn: KFold

- Let us see how sklearn arranges our folds ( case 2)

```
1 from sklearn.model_selection import KFold
2 import numpy as np
3
4 data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
5                  11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
6 kfolds = KFold(n_splits =10, shuffle = True)
7 for train, val in kfolds.split(data):
8   print('train:%s, test: %s'% (data[train], data[val]))
```

```
train:[ 2  3  4  5  6  7  8  9 10 11 13 14 15 16 17 18 19 20], test: [ 1 12]
train:[ 1  2  3  4  5  6  7  8  9 11 12 14 15 16 17 18 19 20], test: [10 13]
train:[ 1  2  3  4  5  6  7  9 10 11 12 13 14 15 16 17 18 19], test: [ 8 20]
train:[ 1  2  3  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20], test: [4 5]
train:[ 1  2  3  4  5  6  8  9 10 11 12 13 15 16 17 18 19 20], test: [ 7 14]
train:[ 1  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 19 20], test: [ 2 18]
train:[ 1  2  4  5  7  8  9 10 11 12 13 14 15 16 17 18 19 20], test: [3 6]
train:[ 1  2  3  4  5  6  7  8  9 10 12 13 14 15 16 17 18 20], test: [11 19]
train:[ 1  2  3  4  5  6  7  8 10 11 12 13 14 15 17 18 19 20], test: [ 9 16]
train:[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 16 18 19 20], test: [15 17]
```

# Sklearn: GroupKFold

- Let us see how sklearn arranges our folds ( case 3)

```python
from sklearn.model_selection import GroupKFold
import numpy as np
import random
# data samples
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]).reshape(-1,2)

# group indecies, need to be shuffled
groups = [0,1,2,3,4,5,6,7,8,9];  random.shuffle (groups)

# instentiated the kfolding groups into 10 splits
group_kfold = GroupKFold(n_splits=10)

# show the indcies of training and testing
for train, val in group_kfold.split(data, groups=groups):
  print('train:%s, test: %s'% (data[train].reshape(1,-1), data[val]))
```

```
train:[[ 1  2  3  4  5  6  7  8  9 10 11 12 15 16 17 18 19 20]], test: [[13 14]]
train:[[ 1  2  3  4  5  6  7  8  9 10 13 14 15 16 17 18 19 20]], test: [[11 12]]
train:[[ 1  2  3  4  7  8  9 10 11 12 13 14 15 16 17 18 19 20]], test: [[5 6]]
train:[[ 1  2  3  4  5  6  7  8 11 12 13 14 15 16 17 18 19 20]], test: [[ 9 10]]
train:[[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18]], test: [[19 20]]
train:[[ 1  2  3  4  5  6  9 10 11 12 13 14 15 16 17 18 19 20]], test: [[7 8]]
train:[[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 19 20]], test: [[17 18]]
train:[[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 17 18 19 20]], test: [[15 16]]
train:[[ 3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]], test: [[1 2]]
train:[[ 1  2  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]], test: [[3 4]]
```

# Steps to cross validate a model

- Inside the loop (line 9), we can perform:
  1. Data preprocessing,
  2. Train and build a model
  3. Perform data preprocessing on validation part data
  4. classifying the validation data using the current trained model
  5. Keep track of validation scores for model selection

```
1 from sklearn.model_selection import KFold
2 import numpy as np
3
4 data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
5                 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
6 np.random.shuffle(data)
7 kfolds = KFold(n_splits =10, shuffle = False)
8 for train, val in kfolds.split(data):
9   print('train:%s, test: %s'% (data[train], data[val]))
```

# Cross validation score

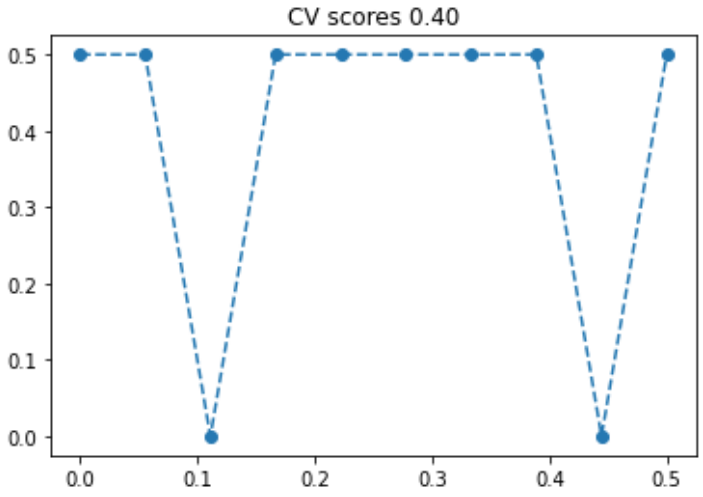
- Keep track of indices is prone to human errors
- We can perform cross validation and compute the evaluation, all at once using **cross_val_score** method
- ***cross_val_score*** parameters are

| Parameter | value |
|---|---|
| Estimator | Any object implements fit function |
| X | Dataset |
| y | Labels for each sample in the dataset |
| scoring | A string with scoring method such as ‘accuracy’ |
| cv | a number for how many splits |

# cross_val_score Usage

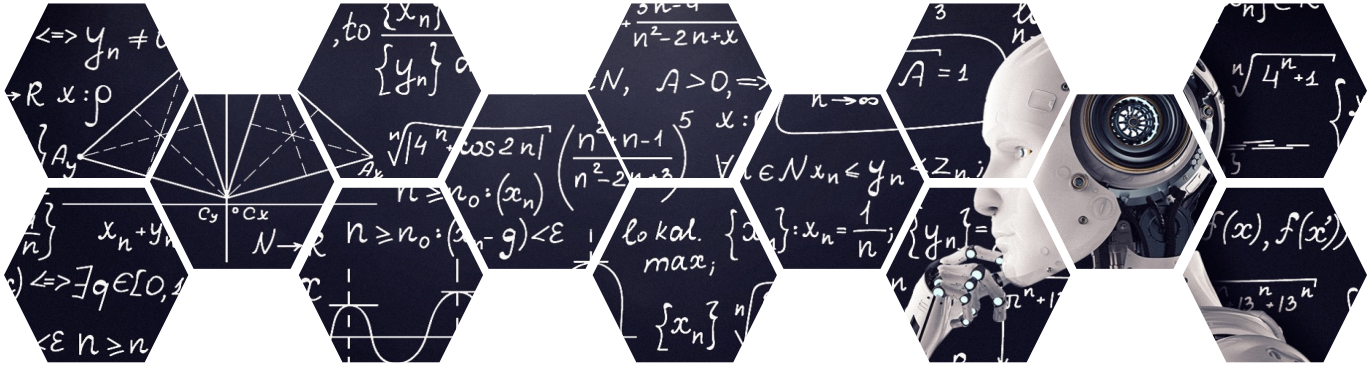
- Fortunately, Sklearn provides functionalities that help reducing coding efforts such as cross_val_score

```
1 from sklearn.model_selection import cross_val_score
2 import matplotlib.pyplot as plt
3 from sklearn.neighbors import KNeighborsClassifier
4 knn = KNeighborsClassifier(n_neighbors=2)
5 # returns only scores
6 scores = cross_val_score(knn, X=data.reshape(-1,1), y=labels,scoring='accuracy', cv=10, verbose=False)
7 plt.plot(np.linspace(0,max(scores), len(scores)), scores, 'o--')
8 plt.title('CV scores %.2f'% np.mean(scores))
9 plt.show()
```




There are other variations of this method:

- ***cross_validate*** : can return fit_time, score_time and test_score
- ***cross_val_predict*** : return predictions per iteration



# Model Performance Evaluation

An application of **KNN** Method

- Metrics
- Strategies
- Modeling selection

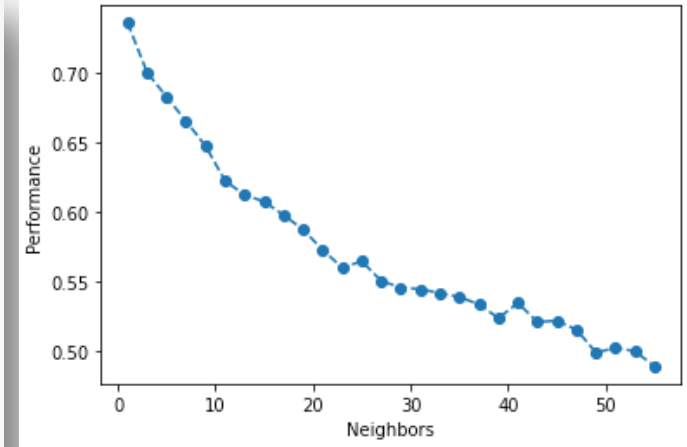
# Model Selection:

- We can select a model by tuning some parameters and pick the one that produced the highest results

```
# creating odd list of K for KNN
myList = np.arange(1,np.sqrt(X.shape[0]).astype(int) )
neighbors = [num for num in myList if num % 2 == 1]

cv_scores = []

data1 = np.hstack((X,y.reshape(-1,1)))
for k in neighbors:
    np.random.shuffle(data1)
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn,
                             data1[:,:8],
                             data1[:,8],
                             cv=5,
                             scoring='accuracy')
    cv_scores.append(scores.mean())
```



# Model selection: GridSearchCV

- This is a good tool to **tune** ML algorithms **hyperparameters**. An example is shown in the code snippt

```
1 from sklearn.model_selection import GridSearchCV
2 from sklearn.neighbors import KNeighborsClassifier
3 # Construct a search space dictionary
4 n_neighbors = [1,2,3,5]
5 param = {'n_neighbors': n_neighbors}
6
7 # configure the grid search
8 model = GridSearchCV(KNeighborsClassifier(), param_grid= param, cv = 5, scoring = 'accuracy')
9
10 # train
11 model.fit(data.reshape(-1,1), labels)
12
13 # check the best, or use predict directly on the testing
14 print(model.best_estimator_)
```

```
KNeighborsClassifier(n_neighbors=1)
```

- Slowness is the issue of GridSearchCV method!

# Summary

- A raw data should be divided into 2 splits in general:
  - **Training set (for model building)**
  - **Test set (for final evaluation)**
- Computing data preprocessing factors such as the scaling (mean/std) must be done at training stage only.
- Preprocessing operations should be computed on the training dataset, and we should be careful to avoid data leakage
- Cross validation strategies can be used to select an optimal model

# Exercises (8 – 10)

- Cross validation
- Model selection
- Grid Search