ABDELGHAFOR'S VIRTUAL INTERNSHIP

MACHINE LEARNING PROGRAM

SESSION (3)

PREPARED BY : MARK KOSTANTINE

LECTURE OVERVIEW

- Train - Test Split
- Train - Val - Test Split
- Confusion Matrix
- ROC Curve and AUC
- Mean Squared Error (MSE) and Mean Absolute Error (MAE)
- Cross Validation
- Questions

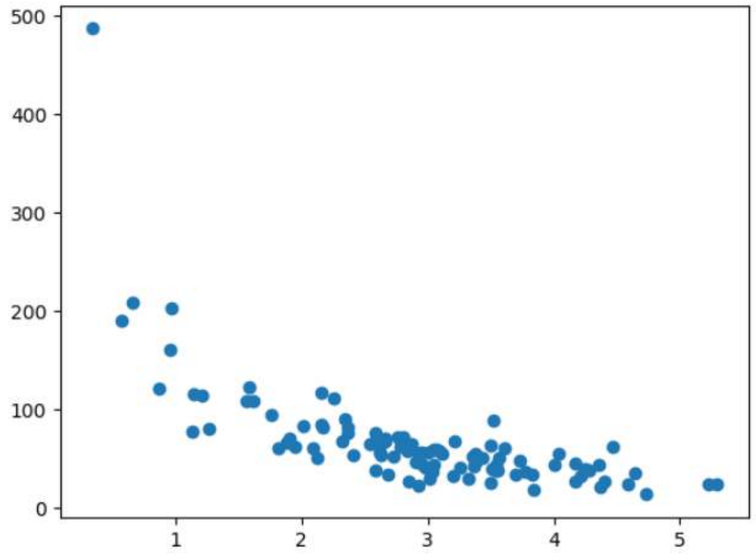
TRAIN/TEST SPLIT

What is Train/Test?

- **Train/Test** is a method to measure the accuracy of your model.
- It is called Train/Test because you split the data set into two sets : a training set and a testing set.
- **80%** for training, and **20%** for testing.
- Train the model means **create** the model.
- Test the model means **test** the accuracy of the model.

Start With a Data Set

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

plt.scatter(x, y)
plt.show()
```



TRAIN/TEST SPLIT

Split Into Train/Test

- The training set should be a random selection of **80%** of the original data.
- The testing set should be the remaining **20%**

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]
```

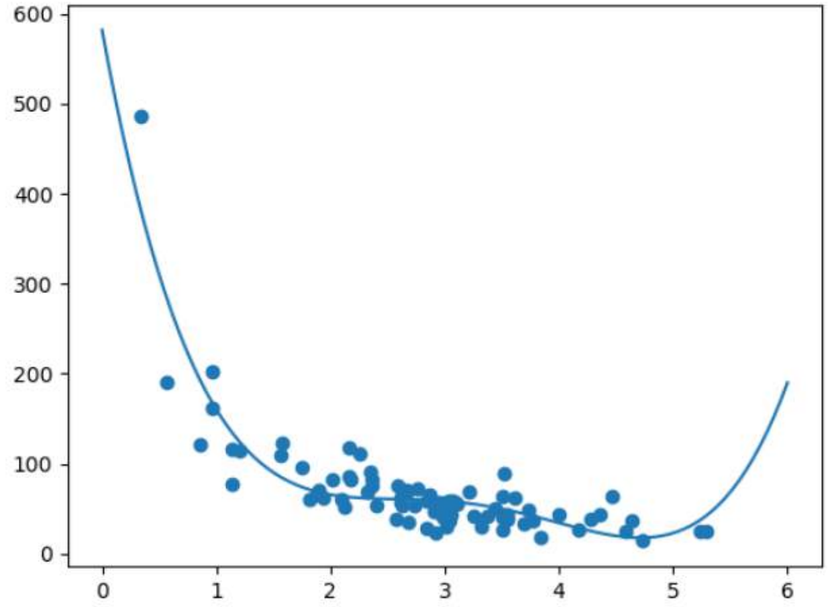
Note: using **80:20** split isn't fixed, we could use different splits like **75:25** or **70:30**

TRAIN/TEST SPLIT

Fit the Data Set

To draw a line through the data points, we use the **plot( )** method of the matplotlib module

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

myline = numpy.linspace(0, 6, 100)

plt.scatter(train_x, train_y)
plt.plot(myline, mymodel(myline))
plt.show()
```



TRAIN/TEST SPLIT

R-squared

It measures the relationship between the x axis and the y axis, and the value ranges from 0 to 1, where 0 means no relationship, and 1 means totally related.

```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(train_y, mymodel(train_x))

print(r2)
```

Note: The result **0.799** shows that there is a OK relationship.

TRAIN/TEST SPLIT

Bring in the Testing Set

test the model with the testing data as well, to see if gives us the same result.

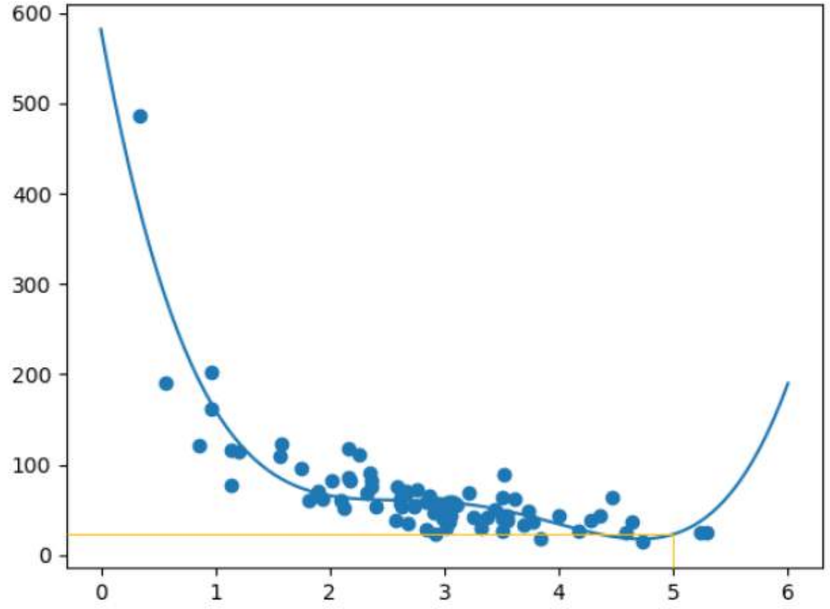
```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(test_y, mymodel(test_x))

print(r2)
```

Note: The result **0.809** shows that the model fits the testing set as well, and we are confident that we can use the model to predict future values.

TRAIN/TEST SPLIT

Predict Values

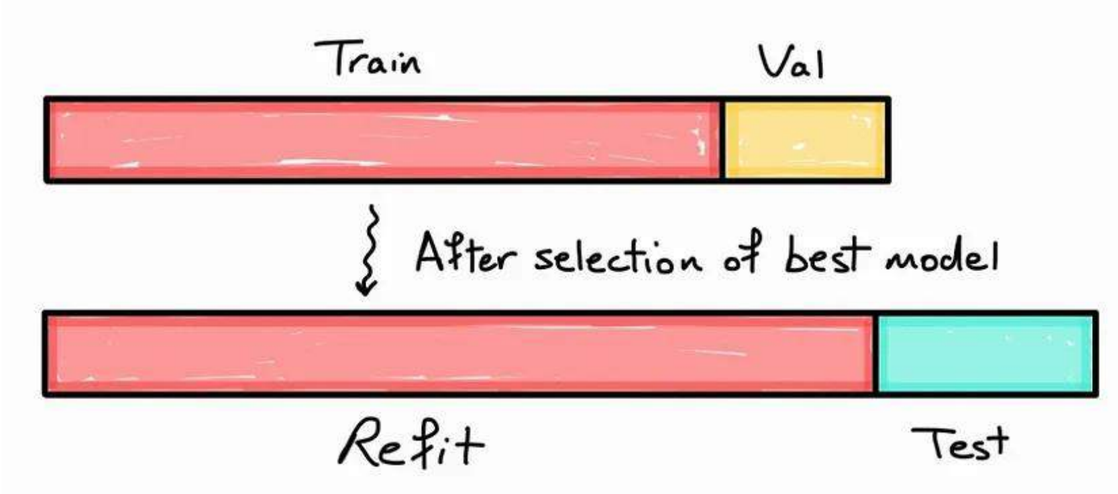
```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

print(mymodel(5))
```



TRAIN, TEST, AND VALIDATION SPLIT

In this data-splitting approach, we perform **two splits** and create an additional set known as the validation set. It is crucial to ensure that the validation set closely resembles the test set in all respects, including data sources, variable distributions, outcome distribution, and so forth. In this method, we:

- split the data into three sets: train, validation, and test sets **(e.g., 70% — 15% — 15%)**
- train the baseline form (simplest, vanilla form) of each model on the train set
- measure classification metrics in the validation set
- perform hyperparameter optimization
- select the model **(architecture + hyperparameters)** that achieved the best performance in the **validation** set
- commonly, retrain the optimal model on the combined train and validation sets
- measure the classification metrics of the resulting model in the **test set**
- report the optimal model architecture and hyperparameters and the metrics obtained in the **test set**.



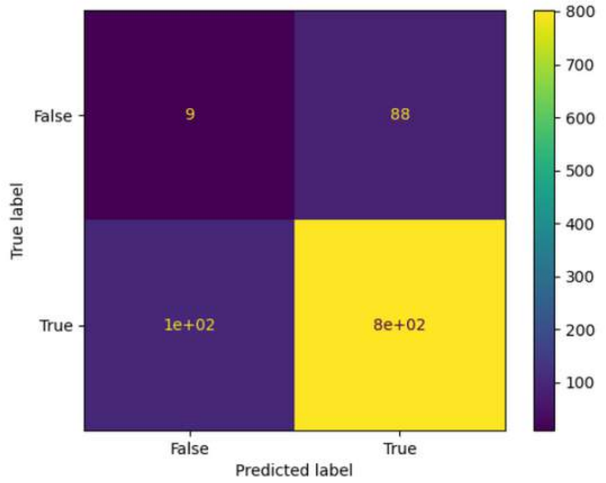
CONFUSION MATRIX

What is a confusion matrix?

- It is a table that is used in classification problems to assess where errors in the model were made.
- The rows represent the actual classes the outcomes should have been. While the columns represent the predictions we have made. Using this table it is easy to see which predictions are wrong.

Creating a Confusion Matrix

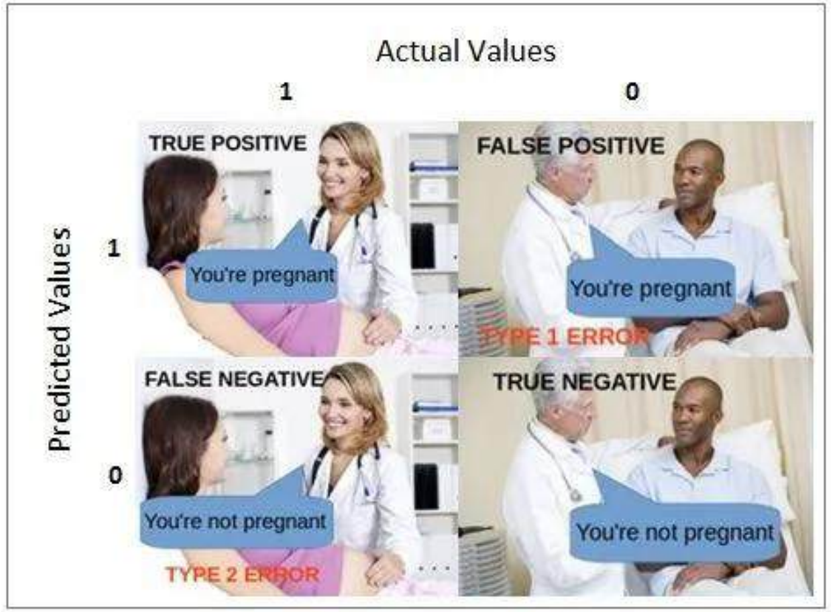
```
import matplotlib.pyplot as plt
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)

confusion_matrix = metrics.confusion_matrix(actual, predicted)

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [False, True])

cm_display.plot()
plt.show()
```



CONFUSION MATRIX

Predicted Values \ Actual Values	Positive (1)	Negative (0)
Positive (1)	TP	FP
Negative (0)	FN	TN

1. **Predicted:** Negative & **Actual Value**: Positive Your predicted False (FN)
2. **Predicted:** Negative & **Actual Value**: Negative Your predicted True(TN)
3. **Predicted:** Positive & **Actual Value**: Positive Your predicted True (TP)
4. **Predicted:** Positive & **Actual Value**: Negative Your predicted False (FP)



CONFUSION MATRIX

Created Metrics

- The matrix provides us with many useful metrics that help us to evaluate out classification model.
- The different measures include: **Accuracy, Precision, Sensitivity (Recall), Specificity, and the F-score**

Accuracy

- Accuracy measures how often the model is correct
- To calculate we use the equation : **(True Positive + True Negative) / Total Predictions**

```
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)

Accuracy = metrics.accuracy_score(actual, predicted)

print(Accuracy)
```

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

CONFUSION MATRIX

Precision

- Of the positives predicted, what percentage is truly positive ?
- To calculate we use the equation : **True Positive / (True Positive + False Positive)**

```
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)

Precision = metrics.precision_score(actual, predicted)

print(Precision)
```

Note: Precision does not evaluate the correctly predicted negative cases

$$\text{Precision} = \frac{TP}{TP + FP}$$

CONFUSION MATRIX

Sensitivity (Recall)

- Sensitivity **(sometimes called Recall)** measures how good the model is at predicting positives.
- To calculate we use the equation : **True Positive / (True Positive + False Negative)**

```
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)

Sensitivity_recall = metrics.recall_score(actual, predicted)

print(Sensitivity_recall)
```

Note: Sensitivity is good at understanding how well the model predicts something is positive

$$\text{Recall} = \frac{TP}{TP + FN}$$

CONFUSION MATRIX

Specificity

- Specificity is similar to sensitivity, but looks at it from the persepctive of **negative results.**
- To calculate we use the equation : **True Negative / (True Negative + False Positive)**

```
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)

Specificity = metrics.recall_score(actual, predicted, pos_label=0)

print(Specificity)
```

CONFUSION MATRIX

F1 Score

- F1-score is the **"harmonic mean"** of precision and sensitivity.
- To calculate we use the equation : **2 * ((Precision * Sensitivity) / (Precision + Sensitivity))**

```
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)

F1_score = metrics.f1_score(actual, predicted)

print(F1_score)
```

Note: It considers both false positive and false negative cases and is good for imbalanced datasets.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

ROC CURVE AND AUC

What is it?

- In classification, there are many different evaluation metrics. The most popular is accuracy, which measures how often the model is correct. This is a great metric because it is easy to understand and getting the most correct guesses is often desired. There are some cases where you might consider using another evaluation metric.
- Another common metric is AUC, area under the receiver operating characteristic (ROC) curve. The Reciever operating characteristic curve plots the true positive (TP) rate versus the false positive (FP) rate at different classification thresholds. The thresholds are different probability cutoffs that separate the two classes in binary classification. It uses probability to tell us how well a model separates the classes.

$$FPR = \frac{FP}{FP + TN}$$

ROC CURVE AND AUC

Imbalanced Data

- Suppose we have an imbalanced data set where the majority of our data is of one value. We can obtain high accuracy for the model by predicting the majority class

```
import numpy as np
from sklearn.metrics import accuracy_score, confusion_matrix, roc_auc_score, roc_curve

n = 10000
ratio = .95
n_0 = int((1-ratio) * n)
n_1 = int(ratio * n)

y = np.array([0] * n_0 + [1] * n_1)
# below are the probabilities obtained from a hypothetical model that always predicts the majority class
# probability of predicting class 1 is going to be 100%
y_proba = np.array([1]*n)
y_pred = y_proba > .5

print(f'accuracy score: {accuracy_score(y, y_pred)}')
cf_mat = confusion_matrix(y, y_pred)
print('Confusion matrix')
print(cf_mat)
print(f'class 0 accuracy: {cf_mat[0][0]/n_0}')
print(f'class 1 accuracy: {cf_mat[1][1]/n_1}')
```

ROC CURVE AND AUC

Imbalanced Data

- Although we obtain a very high accuracy, the model provided no information about the data so it's not useful. We accurately predict class 1 **100%** of the time while inaccurately predict class 0 **0%** of the time. At the expense of accuracy, it might be better to have a model that can somewhat separate the two classes.

```
# below are the probabilities obtained from a hypothetical model that doesn't always predict the mode
y_proba_2 = np.array(
    np.random.uniform(0, .7, n_0).tolist() +
    np.random.uniform(.3, 1, n_1).tolist()
)
y_pred_2 = y_proba_2 > .5

print(f'accuracy score: {accuracy_score(y, y_pred_2)}')
cf_mat = confusion_matrix(y, y_pred_2)
print('Confusion matrix')
print(cf_mat)
print(f'class 0 accuracy: {cf_mat[0][0]/n_0}')
print(f'class 1 accuracy: {cf_mat[1][1]/n_1}')
```

ROC CURVE AND AUC

Imbalanced Data

- For the second set of predictions, we do not have as high of an accuracy score as the first but the accuracy for each class is more balanced. Using accuracy as an evaluation metric we would rate the first model higher than the second even though it doesn't tell us anything about the data.
- In cases like this, using another evaluation metric like AUC would be preferred.

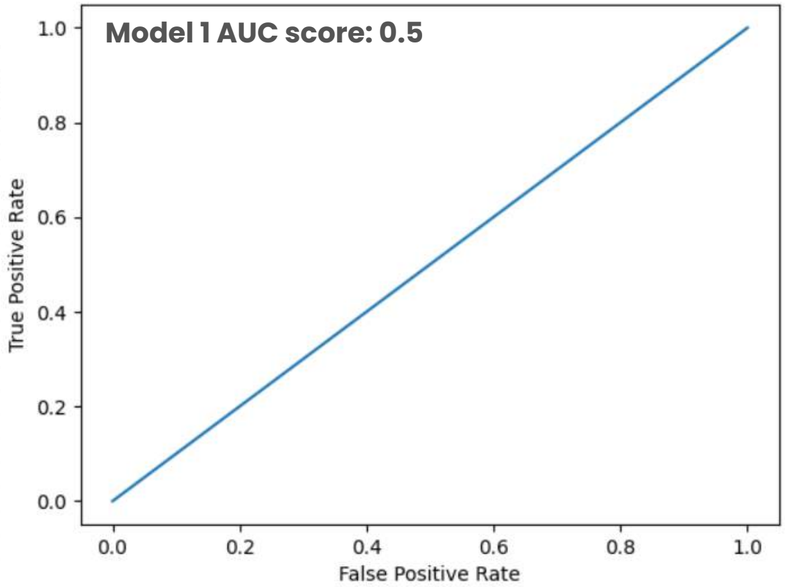
```
def plot_roc_curve(true_y, y_prob):
    """
    plots the roc curve based of the probabilities
    """

    fpr, tpr, thresholds = roc_curve(true_y, y_prob)
    plt.plot(fpr, tpr)
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(y, y_proba)
print(f'model 1 AUC score: {roc_auc_score(y, y_proba)}')
```

ROC CURVE AND AUC

Imbalanced Data

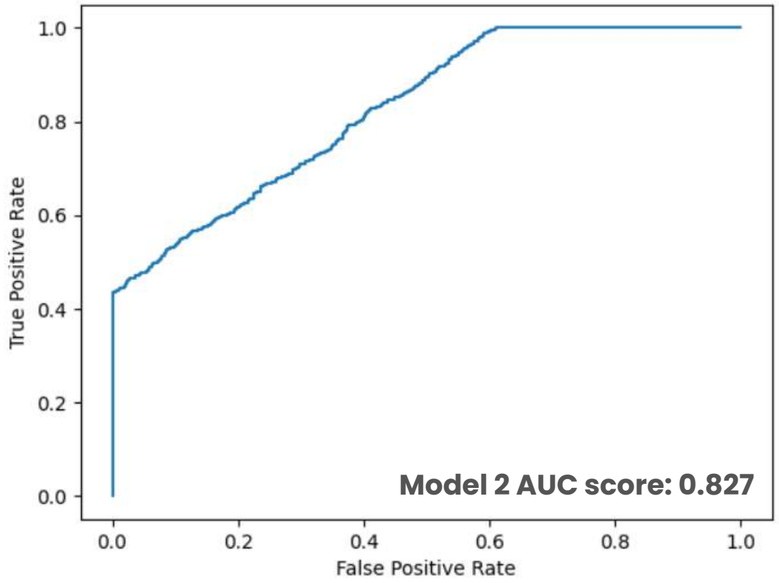
- **Model 1 Plot :**

```
plot_roc_curve(y, y_proba)
print(f'model 1 AUC score: {roc_auc_score(y, y_proba)}')
```



- **Model 2 Plot :**

```
plot_roc_curve(y, y_proba_2)
print(f'model 2 AUC score: {roc_auc_score(y, y_proba_2)}')
```



An AUC score of around **0.5** would mean that the model is unable to make a distinction between the two classes and the curve would look like a line with a slope of 1. An AUC score closer to 1 means that the model has the ability to **separate** the two classes and the curve would come closer to the **top left corner** of the graph.

ROC CURVE AND AUC

Probabilities

- Because AUC is a metric that utilizes probabilities of the class predictions, we can be more confident in a model that has a higher AUC score than one with a lower score even if they have similar accuracies.

```
import numpy as np

n = 10000
y = np.array([0] * n + [1] * n)
#
y_prob_1 = np.array(
    np.random.uniform(.25, .5, n//2).tolist() +
    np.random.uniform(.3, .7, n).tolist() +
    np.random.uniform(.5, .75, n//2).tolist()
)
y_prob_2 = np.array(
    np.random.uniform(0, .4, n//2).tolist() +
    np.random.uniform(.3, .7, n).tolist() +
    np.random.uniform(.6, 1, n//2).tolist()
)

print(f'model 1 accuracy score: {accuracy_score(y, y_prob_1>.5)}')
print(f'model 2 accuracy score: {accuracy_score(y, y_prob_2>.5)}')

print(f'model 1 AUC score: {roc_auc_score(y, y_prob_1)}')
print(f'model 2 AUC score: {roc_auc_score(y, y_prob_2)}')
```

MEAN ABSOLUTE ERROR (MAE)

- MAE is the most straightforward and intuitive metric among the three. It is calculated by taking the average of the absolute values of the errors, which are the differences between the predicted and actual values. For example, if you have a dataset with five observations and your model predicts [12, 13, 14, 15, 16] while the actual values are [10, 11, 12, 13, 14], then the MAE is (|10-12| + |11-13| + |12-14| + |13-15| + |14-16|) / 5 = 2. MAE measures how close the predictions are to the actual values on average, regardless of the direction of the error. It is advantageous when the outliers in the dataset can significantly impact the model's performance. This is because MAE is **less sensitive to outliers than MSE or RMSE.** In addition, because MAE is calculated based on absolute differences, it measures how far off the predictions are on average, which can help interpret the model's performance.
- Mean Absolute Error(MAE) is the mean size of the mistakes in collected predictions. We know that an error basically is the absolute difference between the actual or true values and the values that are predicted. The absolute difference means that if the result has a negative sign, it is ignored.
- Hence, **MAE = True values – Predicted values**

MEAN ABSOLUTE ERROR (MAE)

- **The MAE has some nice properties:**
  - It is easy to understand and interpret.
  - It is robust to outliers, meaning that it is not affected by extreme errors.
  - It has the same unit as the target variable, making it easy to compare. The MAE tells us that, on average, our predictions are off by two units from the actual values.
- **However, the MAE also has some drawbacks:**
  - It does not penalize large errors as much as small errors, meaning that it might not reflect the true accuracy of the model.
  - It is not differentiable at zero, meaning optimizing using gradient-based methods is harder.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

MEAN SQUARED ERROR (MSE)

- MSE stands for **Mean Squared Error**, and it is calculated as the average of the squared differences between the actual and predicted values. Using the same example as above, the MSE is ((10-12)^2 + (11-13)^2 + (12-14)^2 + (13-15)^2 + (14-16)^2) / 5 = 4. MSE measures how close the predictions are to the actual values on average, but it gives more weight to large errors than small ones. This means that MSE is **more sensitive to outliers** and can be useful in identifying models that make large mistakes.

- Hence, MSE = $\frac{1}{N}\sum_{i=1}^{n}(\text{actual values} - \text{predicted values})^2$

Here **N** is the total number of observations/rows in the dataset. The **sigma** symbol denotes the difference between actual and predicted values taken on every **i** value ranging from 1 to n.

MEAN SQUARED ERROR (MSE)

- **The MSE has some advantages over the MAE:**
  - It penalizes large errors more than small ones, reflecting the true accuracy of the model better.
  - It is differentiable everywhere, meaning optimizing using gradient-based methods is easier.
- **However, the MSE also has some disadvantages:**
  - It is sensitive to outliers, meaning that extreme errors can skew it.
  - It has a different unit than the target variable, making it harder to compare.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

CROSS VALIDATION

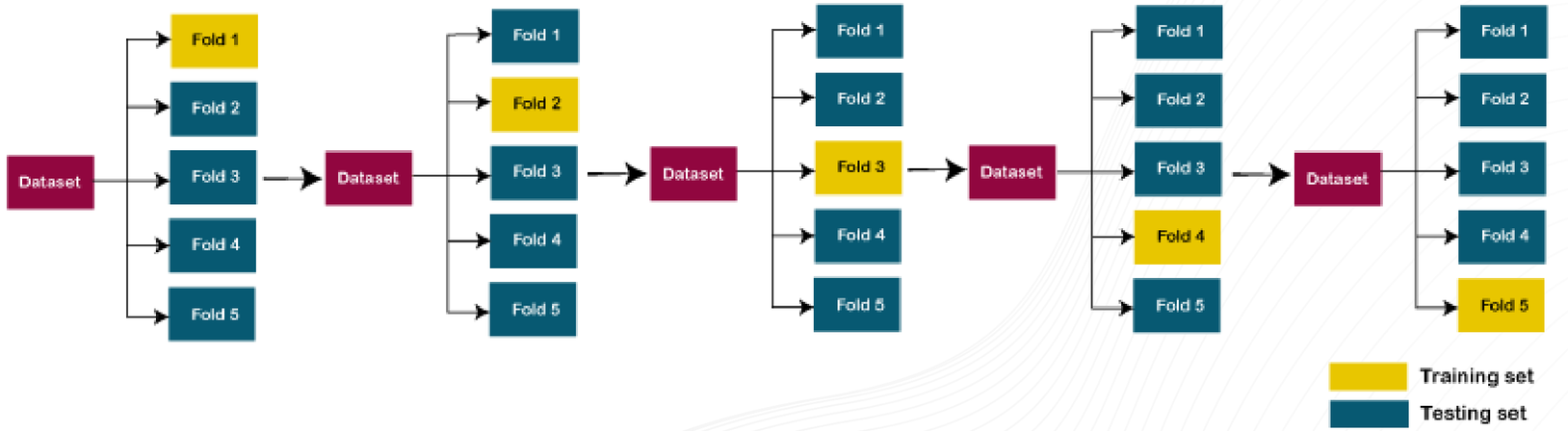
- Cross-validation is a technique for validating the model efficiency by training it on the subset of input data and testing on previously unseen subset of the input data. **We can also say that it is a technique to check how a statistical model generalizes to an independent dataset.**
- In machine learning, there is always the need to test the stability of the model. It means based only on the training dataset; we can't fit our model on the training dataset. For this purpose, we reserve a particular sample of the dataset, which was not part of the training dataset. After that, we test our model on that sample before deployment, and this complete process comes under cross-validation. This is something different from the general train-test split.
- Hence the basic steps of cross-validations are:
  - Reserve a subset of the dataset as a validation set.
  - Provide the training to the model using the training dataset.
  - Now, evaluate model performance using the validation set. If the model performs well with the validation set, perform the further step, else check for the issues.

K-FOLD CROSS VALIDATION

K-fold cross-validation approach divides the input dataset into K groups of samples of equal sizes. These samples are called folds. For each learning set, the prediction function uses k-1 folds, and the rest of the folds are used for the test set. This approach is a very popular CV approach because it is easy to understand, and the output is less biased than other methods.

The steps for k-fold cross-validation are:

- Split the input dataset into K groups
- For each group:
  - Take one group as the reserve or test data set.
  - Use remaining groups as the training dataset
  - Fit the model on the training set and evaluate the performance of the model using the test set.



STRATIFIED K-FOLD CROSS VALIDATION

- This technique is similar to k-fold cross-validation with some little changes. This approach works on stratification concept, it is a process of rearranging the data to ensure that each fold or group is a good representative of the complete dataset. To deal with the bias and variance, it is one of the best approaches.
- It can be understood with an example of housing prices, such that the price of some houses can be much high than other houses. To tackle such situations, a stratified k-fold cross-validation technique is useful.
- In cases where classes are imbalanced we need a way to account for the imbalance in both the train and validation sets. To do so we can stratify the target classes, meaning that both sets will have an equal proportion of all classes.

LEAVE-P-OUT CROSS VALIDATION

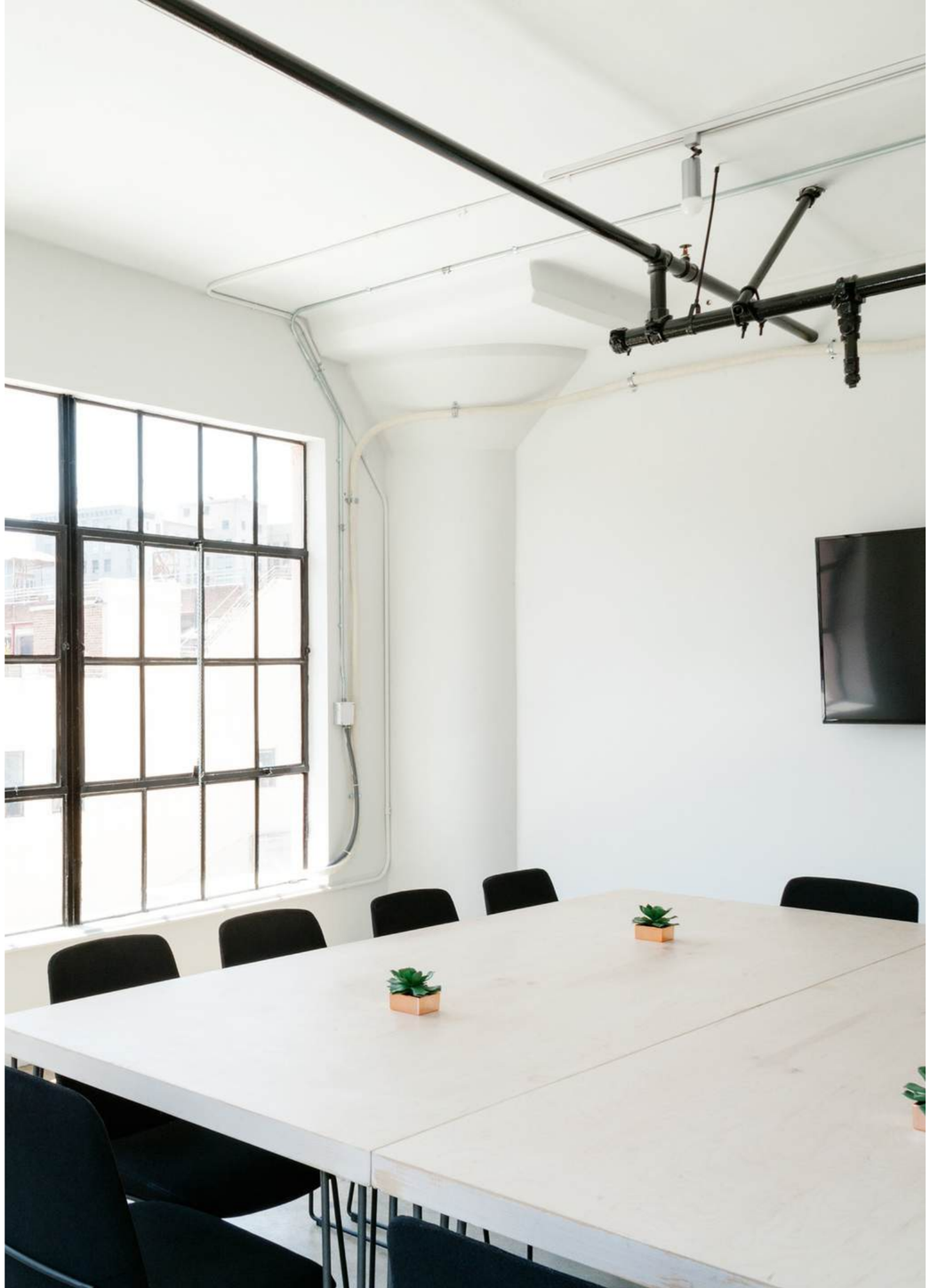
- In this approach, the p datasets are left out of the training data. It means, if there are total n datapoints in the original input dataset, then n-p data points will be used as the training dataset and the p data points as the validation set. This complete process is repeated for all the samples, and the average error is calculated to know the effectiveness of the model.
- There is a **disadvantage** of this technique; that is, it can be computationally difficult for the **large p.**

LEAVE ONE OUT CROSS VALIDATION

- This method is similar to the leave-p-out cross-validation, but instead of p, we need to take 1 dataset out of training. It means, in this approach, for each learning set, **only one datapoint is reserved**, and the remaining dataset is used to train the model. This process repeats for each datapoint. Hence for n samples, we get n different training set and n test set. It has the following features:
  - In this approach, the bias is minimum as all the data points are used.
  - The process is executed for n times; hence execution time is high.
  - This approach leads to high variation in testing the effectiveness of the model as we iteratively check against one data point.

ANY QUESTIONS?

Feel free to ask



MACHINE LEARNING PROGRAM

THANK YOU

UPCOMING NEXT WEEK : SESSION (4)