Chapter 5

Model Evaluation and Validation

5. Model Evaluation and Validation [8 hours]

5.1 Need of Model Evaluation in ML

5.2 Model Evaluation Metrics

1.2.1 Classification Metrics

Accuracy, Precision, Recall and F1 score

Confusion matrix

ROC and PR-Curve

1.2.2 Regression Metrics:

Mean Absolute Error (MAE)

Mean-Squared Error (MSE)

Root Mean Squared Error (RMSE)

R-Squared

5.3 Model Validation Techniques:

Train-Test Split

Cross-validation: K-fold Cross Validation

5.4 Hyper-parameter Tuning: Grid Search, Random Search

5.1 Model Evaluation and its Need in ML

Model evaluation is the process of assessing how well a machine learning model performs on unseen data.

Model evaluation tells us **how good** a model is at making predictions and **helps choose the best model** for deployment.

Purpose:

- To **measure accuracy**, **reliability**, and **generalization** of the model
- To **compare models** and select the best one

Why it is important:

- Prevents **overfitting** (model memorizing training data) or Underfitting
- Helps choose the best model among many options (e.g., decision tree vs. random forest).
- Helps ensure model works well in real-world scenarios
- Guides improvements in data, features, and algorithms
- Helps tune parameters (hyperparameters) to improve results.
- Ensures model performs well on **unseen/test data**, not just training data.
- A well-evaluated model builds confidence before deployment in real applications.

5.2 Model Evaluation Metrics

Common Evaluation Metrics:

For Classification:

- **Accuracy** – Correct predictions / Total predictions
- **Precision** – True positives / (True positives + False positives)
- **Recall** – True positives / (True positives + False negatives)
- **F1 Score** – Harmonic mean of precision and recall

- **Confusion Matrix** – Summary of prediction results

For Regression:

- **Mean Squared Error (MSE)**
- **Mean Absolute Error (MAE)**
- **R-squared (R^2)** – Measures how well predictions fit actual data

5.2.1. Classification Metrics

Confusion matrix

- The Confusion Matrix is a 2x2 table (for binary classification) that show the following:
    1. **True Positive (TP):** The count of positive examples that were correctly predicted.
    2. **True Negative (TN):** The count of negative examples that were correctly predicted.
    3. **False Positive(FP):** The count of negative examples that were incorrectly predicted as Positive. *(i.e actually negative but predicted positive).* (Type I error).
    4. **False Negative (FN):** The count of positive examples that were incorrectly predicted as Negative. *(i.e actually positive but predicted negative).(*Type II error).

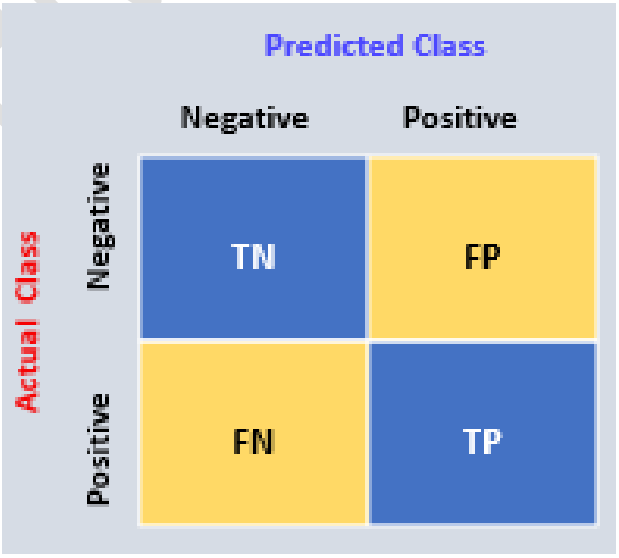


Figure: Confusion Matrix

Accuracy, Precision, Recall and F1 score

1. Accuracy

- Definition: The ratio of correctly predicted instances to the total instances.
- Use Case: Useful when the classes are balanced. Can be misleading if classes are imbalanced.
- Formula:

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

2. Recall (Sensitivity or True Positive Rate)

- Definition: The proportion of actual positives that were correctly identified.
- Use Case: Important in cases where identifying positives is crucial (e.g., disease detection).
- Formula:

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

3. Precision

- Definition: The proportion of positive predictions that were actually correct.
- Use Case: Important when false positives are costly (e.g., spam email detection).
- Formula:

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

4. F1-Score

- Definition: The harmonic mean of precision and recall, balancing both metrics.
- Use Case: Useful when there is an imbalance between classes and when both precision and recall are important. Use when you need a balance between precision and recall.
- Formula:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

5. Specificity (True Negative Rate)

- It measures the proportion of actual negative cases that the model correctly identifies as negative.
- A high specificity (close to 1 or 100%) means the model is good at identifying actual negatives (healthy people).
- A low specificity means the model is making many false positive errors, incorrectly labeling healthy people as sick
- Use Case: Crucial in scenarios where minimizing false positives is more important than minimizing false negatives.

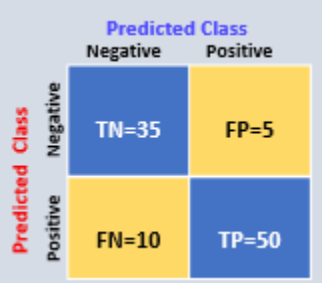
$$\text{Specificity} = \frac{TN}{TN + FP}$$

6. Fall-out (*False Positive rate)*

- It is the fraction of negative examples incorrectly classified.
- FPR is the percentage of negatives wrongly classified as positive.
- FPR = $\frac{\text{FP}}{\text{FP+TN}}$
- Also, FPR = 1 – Specificity

Numerical Example:

Let's consider an example of a binary classification model with the following confusion matrix. Compute the accuracy, precision, recall and F1 score.



- Accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{50 + 35}{50 + 35 + 5 + 10} = 0.85 = 85\%$$

- Recall:

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{50}{50 + 10} = 0.833 = 83.3\%$$

- Precision:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{50}{50 + 5} = 0.909 = 90.9\%$$

- F1-Score:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = 2 \cdot \frac{0.909 \cdot 0.833}{0.909 + 0.833} = 0.87 = 87\%$$

Python code implementation:

```
from sklearn.metrics import confusion_matrix

# Generate confusion matrix

conf_matrix = confusion_matrix(y_test, y_pred)

# Display results
```

```
print("Confusion Matrix:")
print(pd.DataFrame(conf_matrix, index=iris.target_names,   columns=iris.target_names))
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print("\nEvaluation Metrics:")
print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")

from sklearn.metrics import classification_report
# Display classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

Output:

```
Confusion Matrix:
            setosa  versicolor  virginica
setosa          19           0          0
versicolor       0          13          0
virginica        0           0         13


Evaluation Metrics:
Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1-Score: 1.00


Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

ROC Curve:

The Receiver Operating Characteristic (ROC) curve is a graphical representation to evaluate the performance of a binary classification model at different classification threshold values.

It plots the trade-off between True Positive Rate (Sensitivity/Recall) and False Positive Rate (Fall-out).

- X-axis: False Positive Rate (FPR)
- Y-axis: True Positive Rate (TPR)
- However, we sometimes put Specificity in X-axis. (Specificity= 1- FPR)

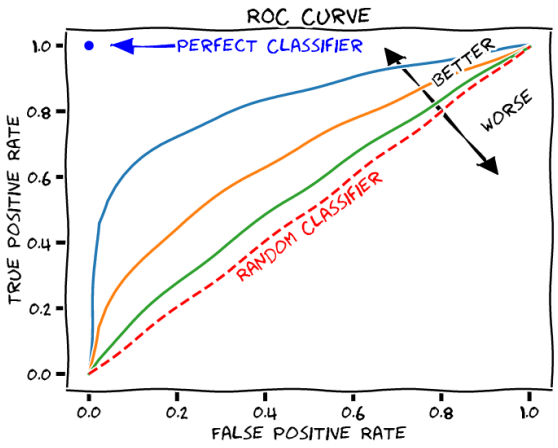


Figure: ROC

Why is the ROC Curve Useful?

- Evaluates classifier performance across all classification thresholds.
- Helps choose the best decision threshold for classification models.
- Compares multiple models' performance using AUC (Area Under Curve).

Interpreting the ROC Curve

- A perfect model: ROC curve hugs the top-left corner (TPR = 1, FPR = 0).
- A random model: ROC curve is a diagonal line (AUC = 0.5).
- A poor model: ROC curve is below the diagonal (AUC < 0.5).

Numerical:

A binary classification model assigns the following probability scores to 10 instances. The actual class labels (1 for positive, 0 for negative) are given in the table below.

a) Compute the True Positive Rate (TPR) and False Positive Rate (FPR) at different threshold values.

b) Plot the ROC curve using the computed TPR and FPR values.

Sample	Actual Class	Predicted Probability
1	1	0.95
2	0	0.85
3	1	0.80
4	1	0.70
5	0	0.60
6	1	0.55
7	0	0.50
8	0	0.40
9	1	0.30
10	0	0.20

Solution:

There are 5 positive (1) samples and 5 negative (0) samples in the dataset. Total Positives (P) = 5 (Samples: 1, 3, 4, 6, 9)

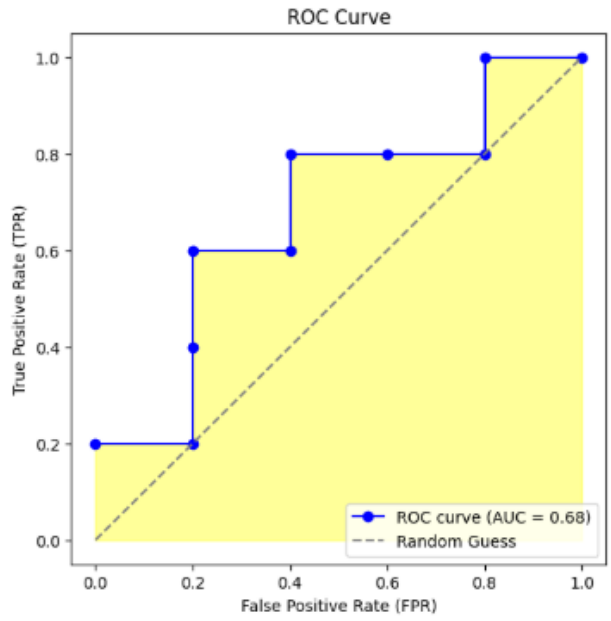
Total Negatives (N) = 5 (Samples: 2, 5, 7, 8, 10)

True Positive (TP): Predicted 1, actual 1.

False Positive (FP): Predicted 1, actual 0.

Threshold	TP	FP	TPR (TP / 5)	FPR (FP / 5)
≥ 0.95	1	0	0.20	0.00
≥ 0.85	1	1	0.20	0.20
≥ 0.80	2	1	0.40	0.20
≥ 0.70	3	1	0.60	0.20
≥ 0.60	3	2	0.60	0.40
≥ 0.55	4	2	0.80	0.40
≥ 0.50	4	3	0.80	0.60
≥ 0.40	4	4	0.80	0.80
≥ 0.30	5	4	1.00	0.80
≥ 0.20	5	5	1.00	1.00

We plot FPR (x-axis) vs. TPR (y-axis) using the computed values.



PR-Curve

The PR curve (Precision-Recall curve) is another important tool used to evaluate the performance of a binary classification model, especially when dealing with imbalanced datasets.
The PR curve plots Precision (y-axis) against Recall (x-axis) at different threshold values.
Unlike the ROC curve which looks at both positives and negatives the PR curve focuses only on how well the model handles the positive class.

Why it's useful:

- Better suited for **imbalanced datasets** (e.g., fraud detection, rare disease diagnosis).
- Focuses only on the **positive class**.
- Helps assess the trade-off between precision and recall.

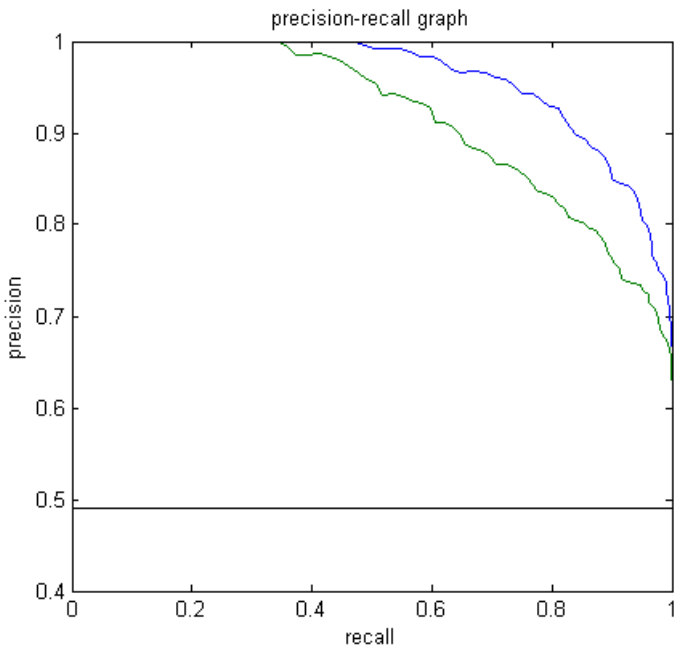


Figure: PR curve

5.2.2. Regression Metrics:

Regression Metrics helps to evaluate how well the model predicts compared to actual values.

Sometimes it is also known as **loss function**. It is used when the output is a continuous value.

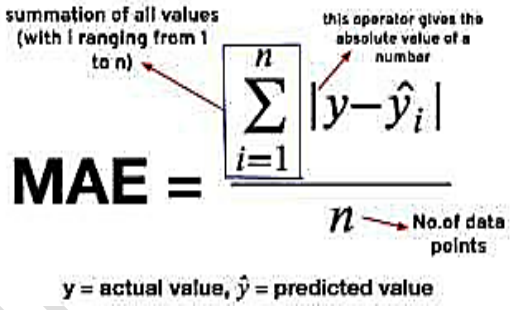
i.e. These metrics are used to evaluate the performance of regression models.

Some of the evaluation metrics are:

i) Mean Absolute Error (MAE)
ii) Mean Squared Error (MSE)
iii) Root Mean Squared Error (RMSE)
iv) R-Squared (R^2)

i) Mean Absolute Error (MAE)

- It's defined as the average of the absolute difference between actual and predicted values.



- **Interpretation:**
  - How much the predictions deviate from the true values, on average.
  - Lower MAE means better predictions.
- **Example:** Suppose the actual values are y = [3, 5, 2, 7] and the predicted values are [2.5, 5, 2, 8].
  - Absolute errors are: $|3-2.5|=0.5$, $|5-5|=0$, $|2-2|=0$, $|7-8|=1$
  - Sum the absolute errors: $0.5+0+0+1=1.5$
  - $MAE=1/4 * 1.5 = 0.375$

ii) Mean Squared Error (MSE)

- The average of the squared differences between the predicted values and actual values.

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

- **Interpretation:**
  - Measures how far predictions are from the actual values.
  - Lower MSE indicates better model performance.
- **Example:** Suppose the actual values are y = [3,5, 2, 7] and the predicted values are [2.5, 5, 2, 8].
  - Absolute errors are: $(3-2.5)^2=0.25$, $(5-5)^2=0$, $(2-2)^2=0$, $(7-8)^2=1$
  - Sum the absolute errors: $0.25+0+0+1=1.25$
  - MAE=1/4 * 1.25 = 0.125

iii) Root Mean Squared Error (RMSE)

- The **square root** of the MSE.

$$\text{RMSE} = \sqrt{\text{MSE}}$$

- **Interpretation:**
  - Brings the error back to the same units as the output (y).
  - Easy to interpret like MAE.
  - Lower values indicate better performance
- **Example:** Using the MSE from the previous example:

$$\text{RMSE} = \sqrt{0.3125} \approx 0.559$$

iv) R-Squared (R^2)

- It measures the proportion of variance in the dependent variable that is predictable from the independent variables.

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

- **Interpretation:**
  - $R^2=1$ Perfect fit.
  - $R^2=0$: Model performs as badly as the mean.
  - $R^2<0$: Worse than the mean.

Example:

Compute the MAE, MSE, RMSE and R2 a model that performed below prediction as in the report.

Observation	Actual Value (y)	Predicted Value (ŷ)
1	3	2.5
2	5	5.1
3	7	6.8
4	9	9.3

Actual values: y = [3,5,7,9]

Predicted values: ŷ = [2.5,5.1,6.8,9.3]

$$\text{MAE} = \frac{1}{n}\sum |y_i - \hat{y}_i|$$

$$= \frac{|3-2.5|+|5-5.1|+|7-6.8|+|9-9.3}{4}$$

$$= \frac{0.5+0.1+0.2+0.3}{4}$$

$$= 0.275$$

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

$$= \frac{1}{4} \text{ x } (3\text{-}2.5)^2 + (5\text{-}5.1)^2 + (7\text{-}6.8)^2 + (9\text{-}9.3)^2$$

$$= \frac{1}{4} \text{ x } (0.25+0.01+0.04+0.09)$$

$$= \frac{1}{4} \text{ x } 0.39$$

$$= 0.0975$$

$$\text{RMSE} = \sqrt{MSE}$$

$$= \sqrt{0.0975}$$

$$= 0.312$$

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}$$

Where:

- SSE = Sum of Squared Errors (already calculated = 0.39)
- SST = Total Sum of Squares = Sum of $(y_i - \bar{y})^2$
- $\bar{y}$ = mean of actual y values

$$\bar{y} = \frac{3+5+7+9}{4} = 6$$

We have:

i	y_i	$\hat{y}$	$(y_i - \hat{y})^2$
1	3	2.5	0.25
2	5	5.1	0.01
3	7	6.8	0.04
4	9	9.3	0.09

i	yi	$y_i - \bar{y}$	$(y_i - \bar{y})^2$
1	3	-3	9
2	5	-1	1
3	7	1	1
4	9	3	9

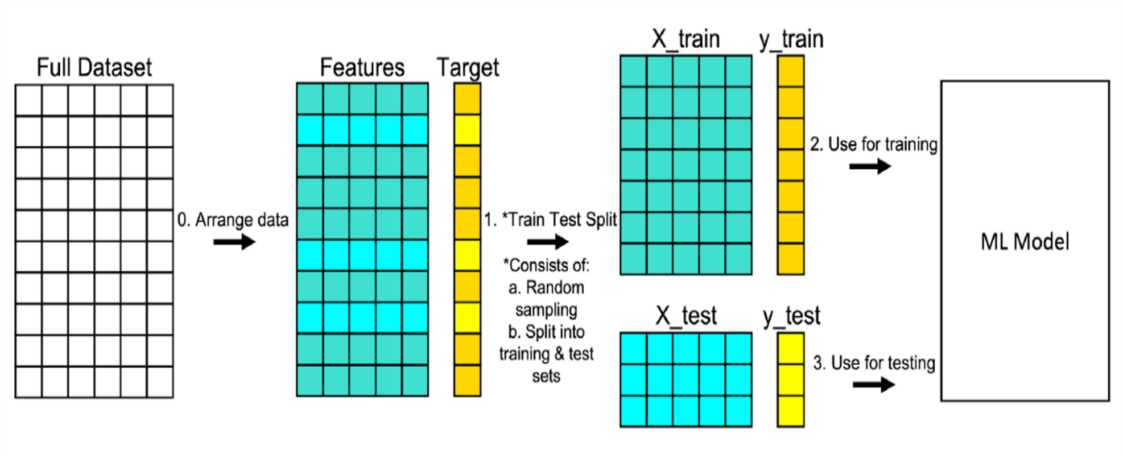
Hence $R^2 = 1 - \frac{0.39}{20} = 0.9805$

5.3. Model Validation Techniques:

In **Machine Learning**, **model validation techniques** are used to evaluate how well your model generalizes to **unseen data**. The goal is to estimate performance and prevent problems like **overfitting** and **underfitting**.

Train-Test Split

Train-test split is one of the **simplest and most commonly used model validation techniques** in machine learning.



It involves splitting the original dataset into two parts:

- **Training set**: Used to **train** the model (learn patterns).
- **Test set**: Used to **evaluate** the model's performance on **unseen data**.

We perform train-test split to:

- **Train** the model on one part of the data.
- **Test** it on different data to estimate how well it generalizes.
- Prevent **overfitting** (memorizing training data).

Typical Split Ratios:

Split	Training Set	Test Set
80-20 split	80%	20%
70-30 split	70%	30%
60-40 split	60%	40%

Example:

If you have 1000 samples, a typical split might be:

- 80% (800 samples) → **Training set**
- 20% (200 samples) → **Test set**

Or sometimes:

- 60% training, 20% validation, 20% test.

Python code Implementation:

Given a dataset as: "Info.csv", perform Train-Test split with 80:20 ratio.

Store	Date	Sales	Visitors	Region	Class
A	12/1/2024	1000	300	North	Medium
B	12/1/2024	1200	350	South	High
C	12/1/2024	950	280	West	Low
D	12/1/2024	1100	320	East	Medium
E	12/1/2024	800	250	North	Low
A	12/2/2024	1050	310	North	Medium
B	12/2/2024	1300	370	South	High
C	12/2/2024	980	290	West	Low
D	12/2/2024	1150	330	East	Medium
E	12/2/2024	850	260	North	Low
A	12/3/2024	1100	320	North	Medium
B	12/3/2024	1250	360	South	High
C	12/3/2024	1000	300	West	Medium
D	12/3/2024	1200	340	East	High
E	12/3/2024	900	270	North	Low
A	12/4/2024	1150	330	North	Medium
B	12/4/2024	1350	380	South	High
C	12/4/2024	1050	310	West	Medium
D	12/4/2024	1250	350	East	High
E	12/4/2024	950	280	North	Low
A	12/5/2024	1200	340	North	High
B	12/5/2024	1400	390	South	High
C	12/5/2024	1100	320	West	Medium
D	12/5/2024	1300	360	East	High
E	12/5/2024	1000	300	North	Medium

```
import pandas as pd

from sklearn.model_selection import train_test_split

# Read Dataset

data = pd.read_csv("Info.csv")

df = pd.DataFrame(data)

# Dataset

X = df[['Store', 'Date', 'Sales', 'Visitors', 'Region']]  # Features

y = df['Class']  # Target

# Split data into 80% train and 20% test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Displaying training and testing dataset

print("Training Dataset:\n", X_train)

print("Testing Dataset:\n", X_test)
```

Output:

Training Dataset:

	Store	Date	Sales	Visitors	Region
9	E	2024-12-02	850	260	North
13	D	2024-12-03	1200	340	East
1	B	2024-12-01	1200	350	South
22	C	2024-12-05	1100	320	West
5	A	2024-12-02	1050	310	North
2	C	2024-12-01	950	280	West
12	C	2024-12-03	1000	300	West
15	A	2024-12-04	1150	330	North
3	D	2024-12-01	1100	320	East
4	E	2024-12-01	800	250	North
20	A	2024-12-05	1200	340	North
17	C	2024-12-04	1050	310	West
21	B	2024-12-05	1400	390	South
18	D	2024-12-04	1250	350	East
24	E	2024-12-05	1000	300	North
7	C	2024-12-02	980	290	West
10	A	2024-12-03	1100	320	North
14	E	2024-12-03	900	270	North
19	E	2024-12-04	950	280	North
6	B	2024-12-02	1300	370	South

Testing Dataset:

	Store	Date	Sales	Visitors	Region
8	D	2024-12-02	1150	330	East
16	B	2024-12-04	1350	380	South
0	A	2024-12-01	1000	300	North
23	D	2024-12-05	1300	360	East
11	B	2024-12-03	1250	360	South

Approaches for Train-test split:

i) Simple Train-Test Split (Random split)

- The dataset is randomly split into two parts: one for training and one for testing.
- The data is shuffled randomly before splitting.
- A specified percentage (e.g., 80%) of the data is used for training, and the remaining (e.g., 20%) is used for testing.
- **When to use:** When the dataset is relatively large and there is no concern about class imbalances.
- **Python code implementation:**

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

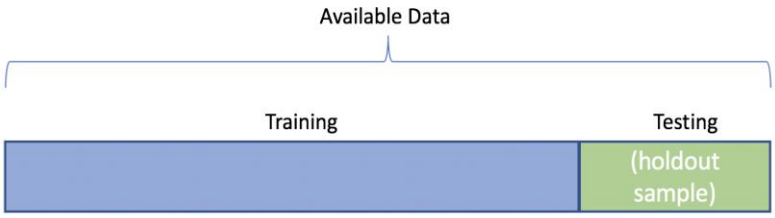


Figure: Simple Train-test split

ii) Holdout Method

- The data is split into three sets: training, validation, and test.
- The model is trained on the training set, tuned on the validation set, and then tested on the test set.
- When to use: Use when you want to have an additional dataset (validation set) to fine-tune hyperparameters before final testing on the test set.
- Python code Implementation:

```
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)

X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

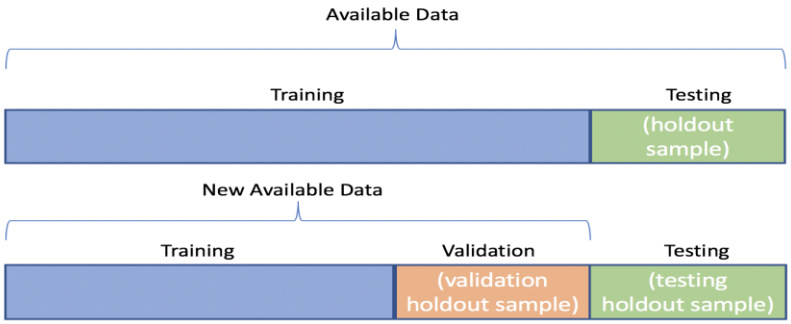


Figure: Hold Out Method

iii) Stratified Train-Test Split

- The data is split while maintaining the proportion of each class in both the training and testing sets, so that both sets represent the original class distribution.
- It is particularly useful for imbalanced datasets in classification problems.
- Python code Implementation:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

Figure: Startified Train-Test Split Method

iv) K-Fold Cross-Validation

- The dataset is divided into k equal (or nearly equal) folds.
- The model is trained and tested k times, with each fold being used once as a test set and the remaining k-1 folds as the training set.
- The performance is then averaged over all k runs.

- It helps in utilizing the entire dataset for both training and testing, reducing bias and providing a more reliable estimate of model performance.
- When to use: When you want a more robust evaluation, and you don't want to waste data by leaving a test set out. Common choices are 5-fold or 10-fold cross-validation.

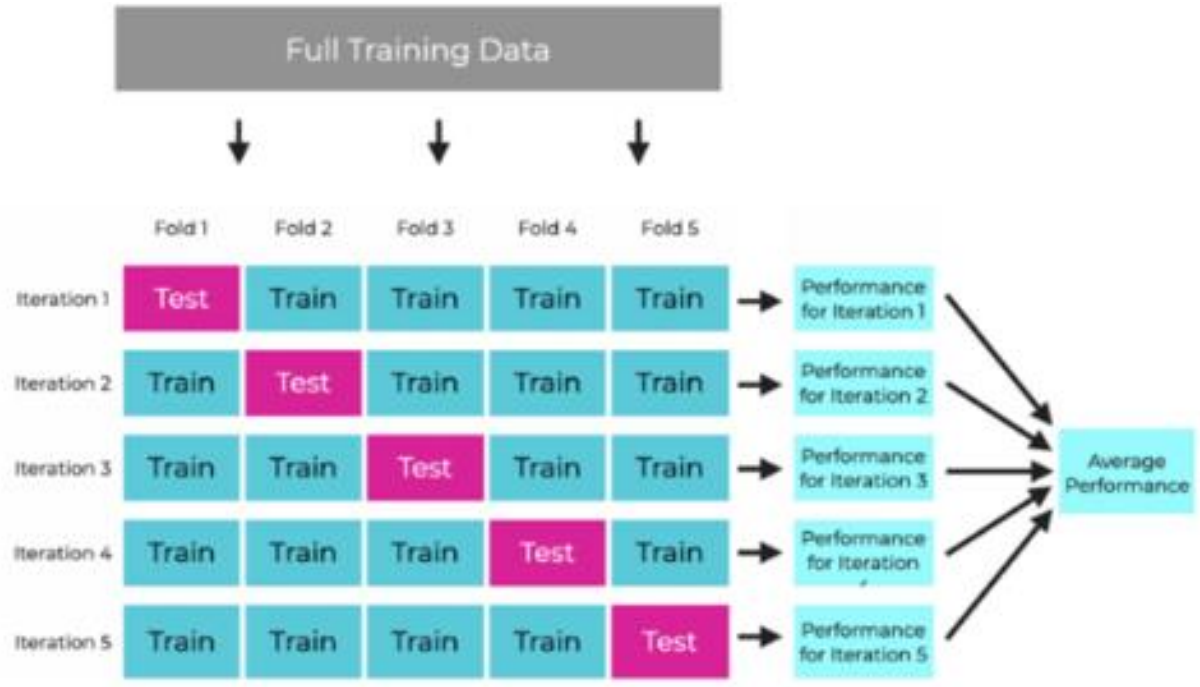


Figure: K-fold Cross Validation

Python code Implementation:

```
from sklearn.model_selection import cross_val_score

model = RandomForestClassifier()

kf = KFold(n_splits=10, shuffle=True, random_state=42)

scores = cross_val_score(model, X, y, cv=kf)  # 5-fold cross-validation

print(scores)

print(f"Average Accuracy: {np.mean(scores):.4f}\n")
```

a) Stratified K-fold Cross Validation

- This is similar to k-fold cross validation.
- The difference is that this approach preserves the distribution of target classes in each folds.

Figure: Stratified K-fold Cross Validation

b) Leave-One-Out Cross-Validation (LOOCV)

- This is a special case of K-fold cross-validation where k is set to the total number of data points in the dataset.
- Each individual data point is used once as a test set, while the rest of the data points are used for training.
- LOOCV is particularly useful for small datasets, as it allows the model to be evaluated on all available data points. However, it can be computationally expensive for large datasets.

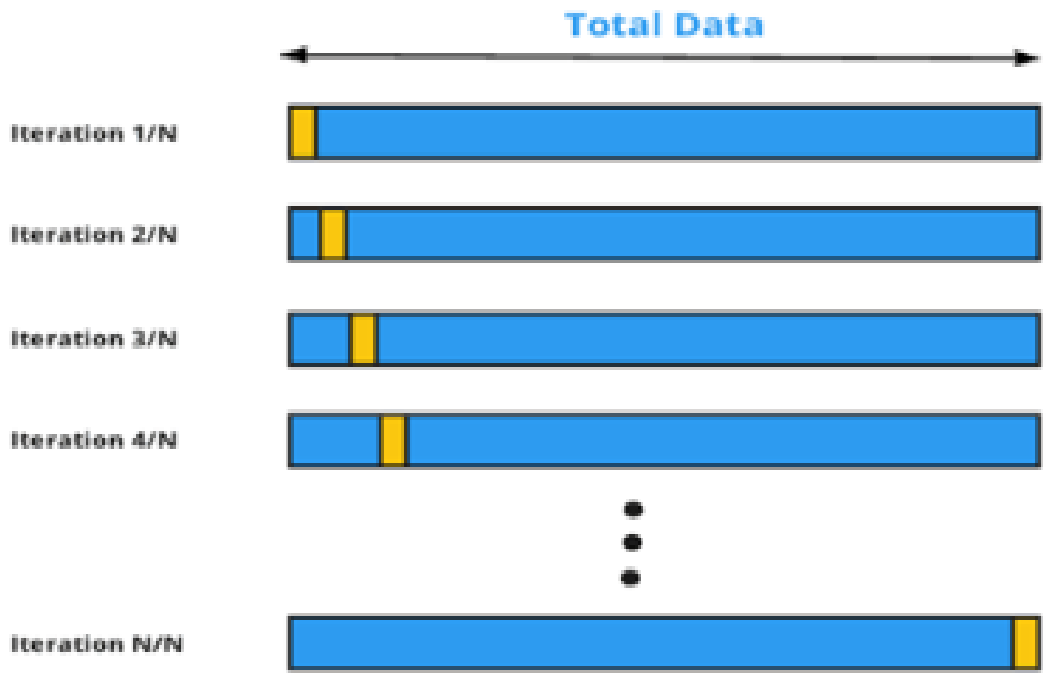


Figure: Leave One Out Cross Validation (LOOCV)

5.4. Hyper-parameter Tuning: Grid Search, Random Search

Hyperparameter tuning is the process of **finding the best combination of settings** (hyperparameters) that result in the **best performance** of a machine learning model.

Unlike model parameters (like weights in linear regression), **hyperparameters are set before training** (e.g., number of trees in Random Forest, learning rate in gradient descent, etc.)

What Are Hyperparameters?

Examples of hyperparameters include:

- k in k-NN
- max_depth, n_estimators in decision trees / random forest
- C, gamma in SVM
- learning_rate, batch_size in neural networks

Why Tune Hyperparameters?

- Different hyperparameters can produce **very different performance**.
- Goal: Find the settings that **maximize model performance** (e.g., accuracy, F1 score) on **validation data**.

Because **choosing hyperparameters randomly or by guesswork** might not give you the best result. So, we need to find some methods for hyperparameter tuning.

Common Hyperparameter Tuning Methods:

a) Grid Search (GridSearchCV)

Grid Search is a method used in machine learning to **find the best settings** (called **hyperparameters**) for your model.

Because **choosing hyperparameters randomly or by guesswork** might not give you the best result.

Imagine you're baking a cake and trying different combinations of:

- Oven temperature (150°C, 180°C, 200°C)
- Baking time (30, 40, 50 minutes)

You try **every possible combination** of temperature and time to see which one gives the best cake. It **tries all combinations** to find the one that works best.

In Machine Learning Terms:

Let's say you're using a **Random Forest** model. It has these hyperparameters:

- n_estimators: number of trees (e.g., 100, 150, 200)
- max_depth: how deep the trees can go (e.g., 3, 5, 7)

With Grid Search:

- It will try:
  (100, 3), (100, 5), (100, 7),
  (150, 3), (150, 5), (150, 7),
  (200, 3), (200, 5), (200, 7) → total **9 combinations**

Pros:

- Tries all combinations → ensures best in that grid.

Cons:

- **Computationally expensive** (slow for large grids or many hyperparameters).

Sample Python code:

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
params = {
   'n_estimators': [100, 150, 200],
   'max_depth': [3, 5, 7]
}
grid = GridSearchCV(RandomForestClassifier(), params, cv=5)
grid.fit(X_train, y_train)

print("Best parameters:", grid.best_params_)
```

b) Random Search (RandomizedSearchCV)

Random Search is a method used to find the **best hyperparameters** for a machine learning model — just like **Grid Search** — but instead of checking **every possible combination**, it tries a **random selection** of combinations.

Let's say you want to tune a **Random Forest** model.

You have two hyperparameters:

- n_estimators: [100 to 200]
- max_depth: [3 to 10]

With **Random Search**, you just randomly pick (for example) 10 combinations like:

- (120, 5)
- (180, 8)
- (150, 3)
- … and so on (up to 10 total)

Then it finds the best one out of those.

Pros:

- **Faster** than grid search (only evaluates n_iter random combinations).
- More efficient for **high-dimensional spaces**.

Cons:

- Might miss the best combination (not exhaustive).

Python Example Using RandomizedSearchCV

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint

# Define hyperparameter distributions
param_dist = {
    'n_estimators': randint(100, 200),
    'max_depth': randint(3, 10)
}
```

```
# Random Search with 10 random combinations
random_search = RandomizedSearchCV(RandomForestClassifier(),
param_distributions=param_dist, n_iter=10, cv=5)
random_search.fit(X_train, y_train)

print("Best parameters found:", random_search.best_params_)
```

***