In this module, you will learn how to assess the effectiveness of machine learning models using industry-standard evaluation and validation techniques. You'll explain key classification and regression metrics, evaluate models using real-world data, and interpret results with tools like confusion matrices and feature importance charts. You'll explore how to assess clustering quality in unsupervised learning and apply cross-validation to reduce overfitting. The module also introduces regularisation methods to improve model generalisation and reduce feature complexity. Finally, you'll build complete machine learning pipelines and optimise them with GridSearchCV, while identifying common pitfalls like data leakage. To support your learning, you'll receive a Cheat Sheet: Evaluating and Validating Machine Learning Models covering key metrics, techniques, and model tuning strategies.

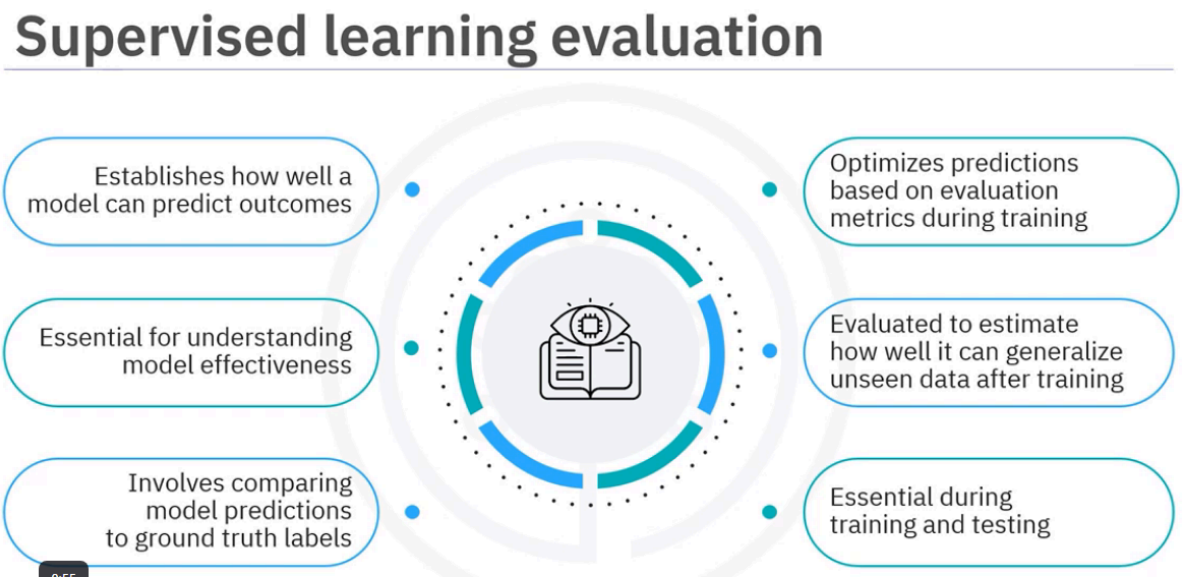
Learning Objectives

- Explain common evaluation metrics for classification models and their use in assessing model performance
- Evaluate classification models using confusion matrices and performance metrics
- Describe regression model metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R^2 score
- Assess the performance and feature importance of a Random Forest regression model
- Interpret evaluation strategies used in unsupervised learning, including heuristics for clustering quality
- Apply cross-validation techniques to validate models and reduce overfitting
- Compare the impact of regularization techniques on regression performance and feature selection
- Build optimized models using pipelines and hyperparameter tuning with GridSearchCV
- Identify risks such as data leakage and explain their impact on model validity

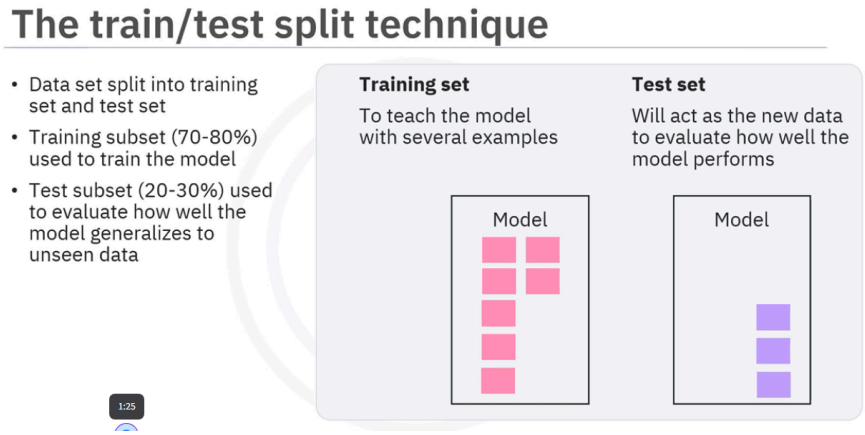
Here's a **well-organised note** from your transcript on **Classification Metrics and Evaluation Techniques**:

📌 Classification Metrics & Evaluation Techniques

🔹 **Purpose**



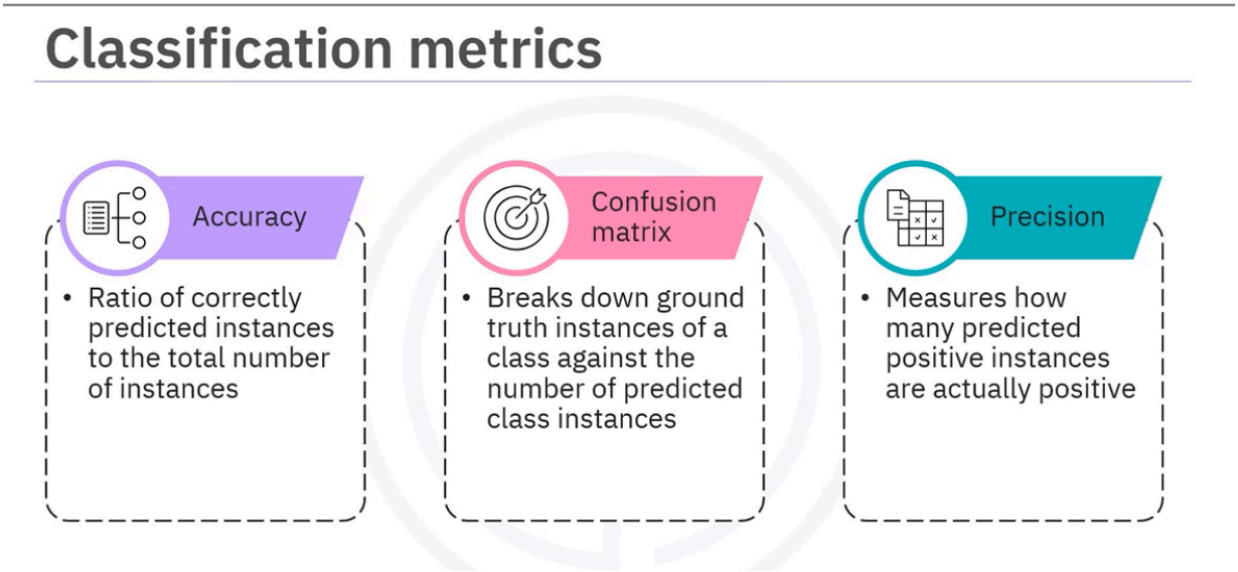
- Evaluate how well a **supervised learning model** predicts outcomes on **unseen data**.
- Helps measure **effectiveness, accuracy, and generalisation** of models.

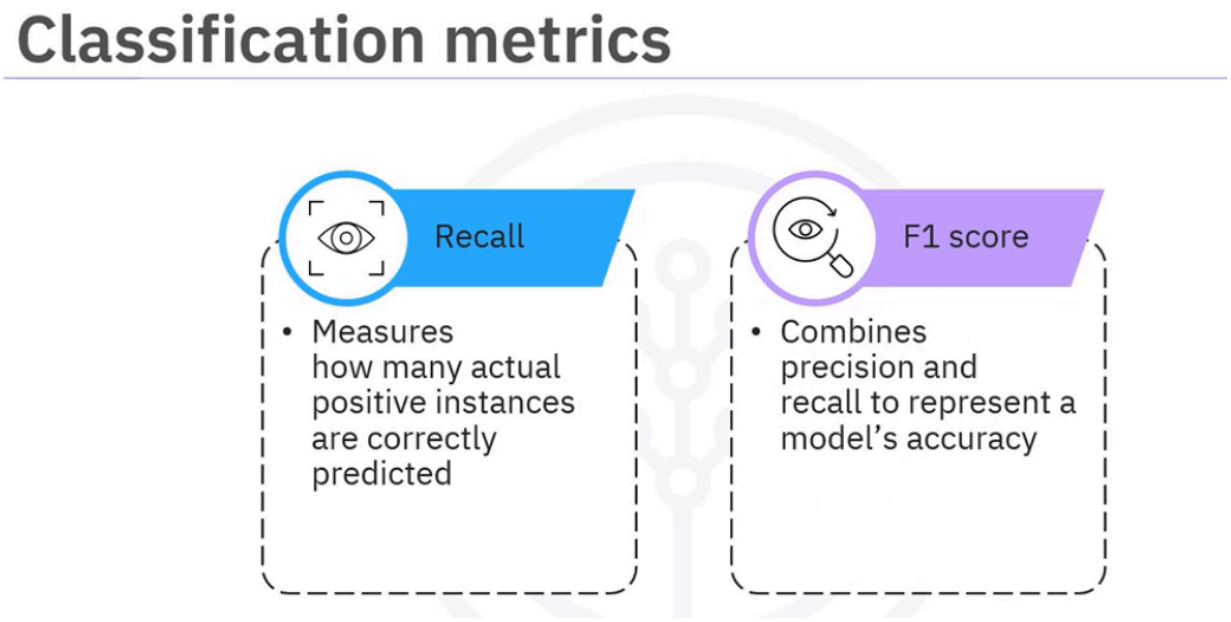


1️⃣ Train-Test Split

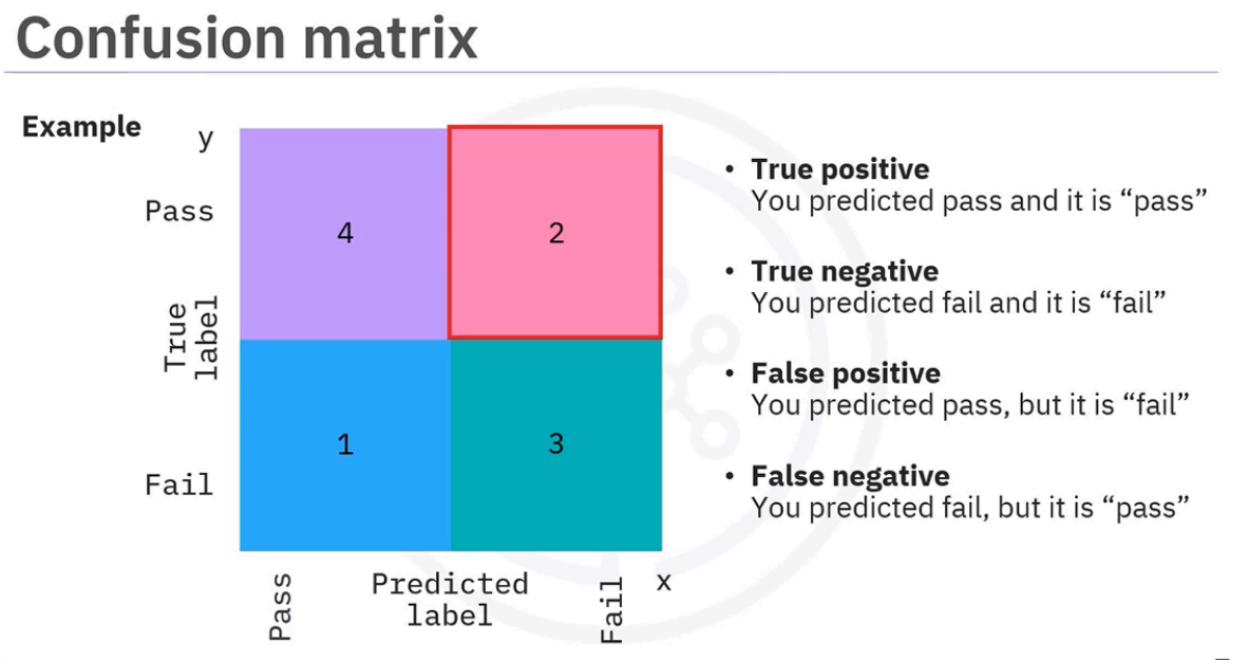
- Dataset is divided into:

- **Training set (70–80%)** → train the model.
- **Test set (20–30%)** → evaluate generalization on unseen data.



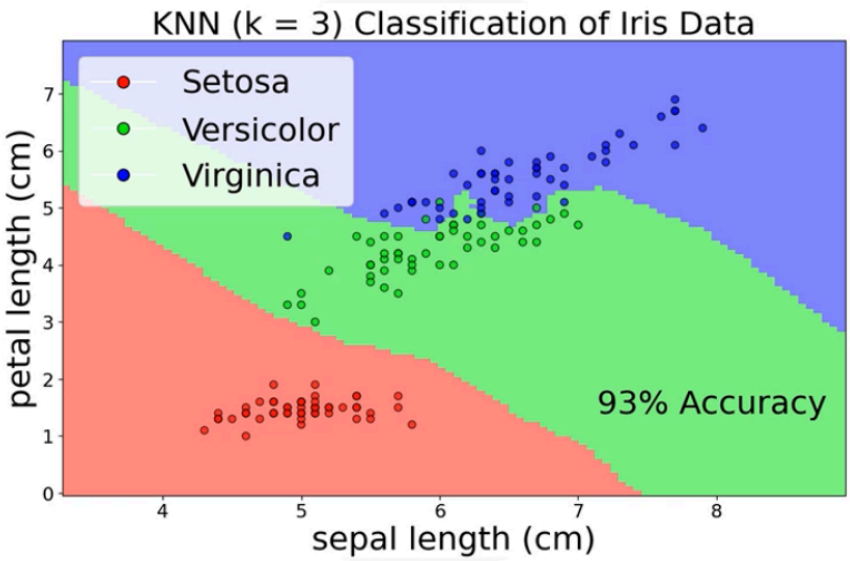


2 Confusion Matrix

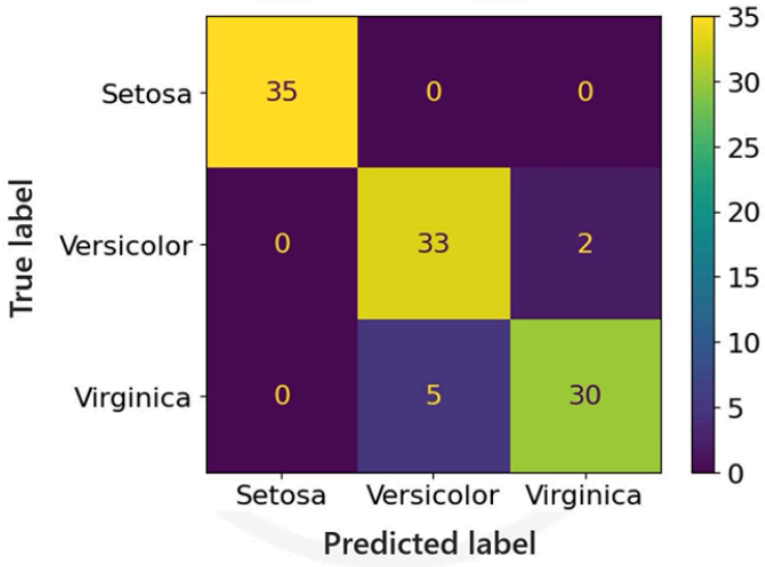


- A table showing **actual vs predicted labels**.
- Terms:
    - **True Positive (TP):** Predicted positive & actual positive.
    - **True Negative (TN):** Predicted negative & actual negative.
    - **False Positive (FP):** Predicted positive but actually negative.
    - **False Negative (FN):** Predicted negative but actually positive.

Iris classifier decision boundary



Iris confusion matrix



📊 Structure:

	Predicted Positive	**Predicted Negative**

Actual Positive	TP	FN
Actual Negative	FP	TN

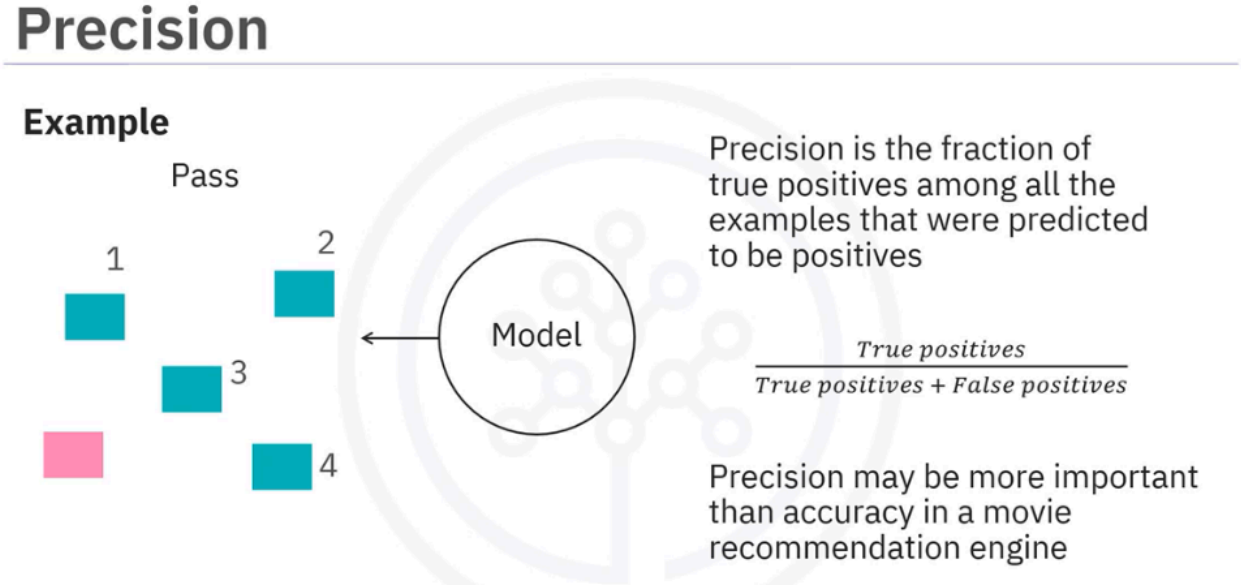
3️⃣ Evaluation Metrics

✅ Accuracy

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

- Fraction of **correct predictions**.
- Example: Correctly predicted pass/fail outcomes ÷ total predictions.
- ⚠️ Can be misleading with **imbalanced datasets**.

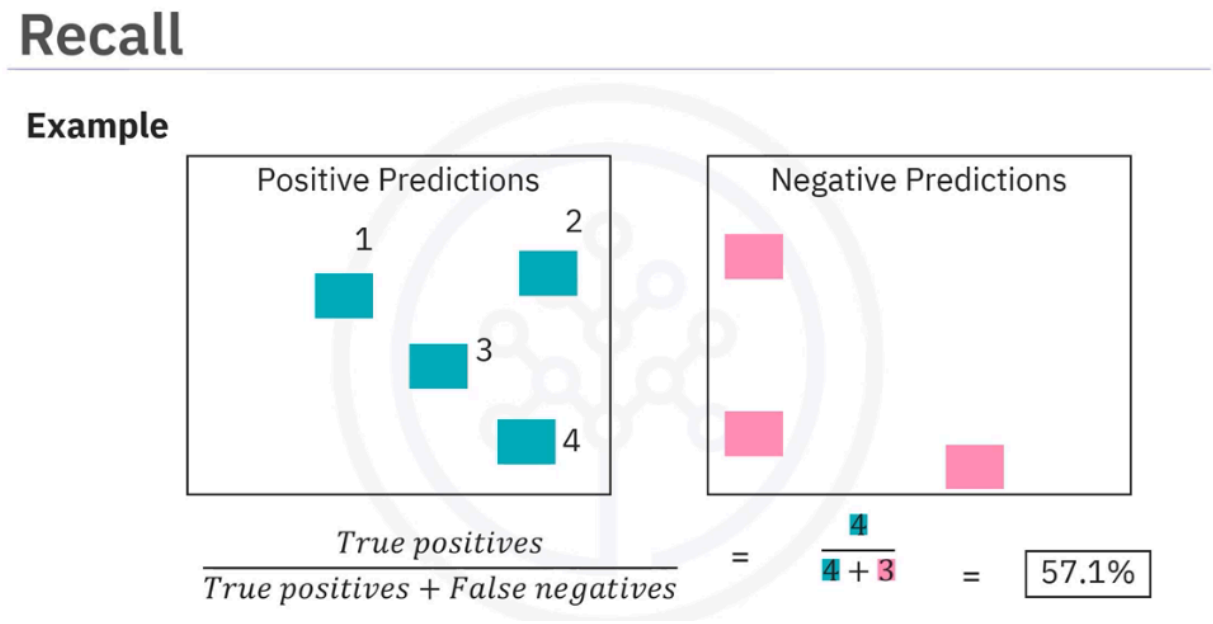
✅ Precision



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

- Out of all predicted positives, how many are actually positive?
- **Use case:** Movie recommendation system (avoid false positives = wrong costly suggestions).

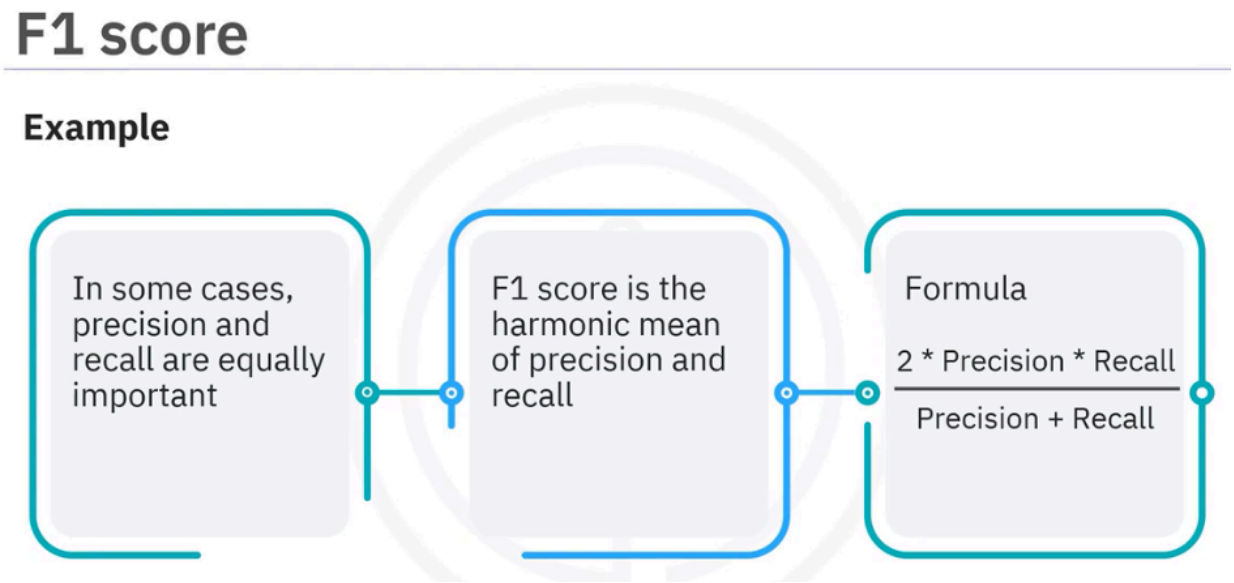
✅ Recall (Sensitivity)



[
Recall = \frac{TP}{TP + FN}
]

- Out of all actual positives, how many did the model correctly identify?
- **Use case:** Medical diagnosis (avoid false negatives = missed patients).

✅ F1 Score



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

- **Harmonic mean** of Precision and Recall.
- Balances both metrics when **both false positives & false negatives matter**.
- **Use case:** Medical applications (diagnosis accuracy).

F1 score

	Precision	Recall	F1 score	Support
Setosa	1.00	1.00	1.00	35
Versicolor	0.87	0.94	0.90	35
Virginica	0.94	0.86	0.90	35
Accuracy	-	-	0.93	105
Weighted avg	0.94	0.93	0.93	105

4️⃣ Example Applications

- **High Precision needed:** Recommendation engines, spam detection.
- **High Recall needed:** Disease detection, fraud detection.
- **F1 Score needed:** When **precision & recall are equally important**.

✅ Key Takeaways

- **Train-Test-Split** → ensures fair evaluation of model performance.
- **Confusion Matrix** → shows detailed classification outcomes.
- **Accuracy** → overall correctness, but limited with imbalanced data.
- **Precision & Recall** → focus on false positives and false negatives respectively.
- **F1 Score** → balances precision and recall.

📘 Notes: Evaluating Classification Models

🎯 Objectives

After completing this, you'll be able to:

- Implement and evaluate classification models on real-world data.
- Interpret and compare evaluation metrics (accuracy, precision, recall, F1-score).
- Use a **confusion matrix** to analyze performance.
- Understand the effect of **noise** on model performance.

📝 Introduction

In this lab, we:

- Use the **Breast Cancer dataset** from `scikit-learn`.
- Build **two classifiers** (KNN & SVM).
- Evaluate them with performance metrics + confusion matrix.
- Add **Gaussian noise** to simulate measurement errors.
- Focus is on **interpretation of metrics**, not on finding the best classifier.

⚙️ Step 1: Install Dependencies

```
!pip install numpy==2.2.0 pandas==2.2.3 scikit-learn==1.6.0 matplotlib==3.9.3 seaborn==0.13.2
```

📦 Step 2: Import Required Libraries

```
import numpy as np

import pandas as pd

from sklearn.datasets import load_breast_cancer

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

import matplotlib.pyplot as plt

import seaborn as sns
```

📊 Step 3: Load Breast Cancer Dataset

```
# Load dataset

data = load_breast_cancer()

X, y = data.data, data.target

labels = data.target_names

feature_names = data.feature_names


print("Target classes:", labels)
```

```
print("Shape of X:", X.shape)
```

✅ Dataset details:

- **Instances:** 569
- **Features:** 30 numeric
- **Classes:**
    - Malignant (212)
    - Benign (357)

⚖️ Step 4: Standardize Features

```
scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)
```

🎲 Step 5: Add Gaussian Noise

```
# Add Gaussian noise

np.random.seed(42)

noise_factor = 0.5

X_noisy = X_scaled + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X.shape)


# Compare original vs noisy

df = pd.DataFrame(X_scaled, columns=feature_names)
```

```
df_noisy = pd.DataFrame(X_noisy, columns=feature_names)


print("Original Data (first 5 rows):")
print(df.head())
print("\nNoisy Data (first 5 rows):")
print(df_noisy.head())
```

🧑‍🤖 Step 6: Train-Test Split

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

🤖 Step 7: Train Two Models

(a) K-Nearest Neighbors

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
```

(b) Support Vector Classifier

```
svm = SVC(kernel='linear', random_state=42)
svm.fit(X_train, y_train)
```

```
y_pred_svm = svm.predict(X_test)
```

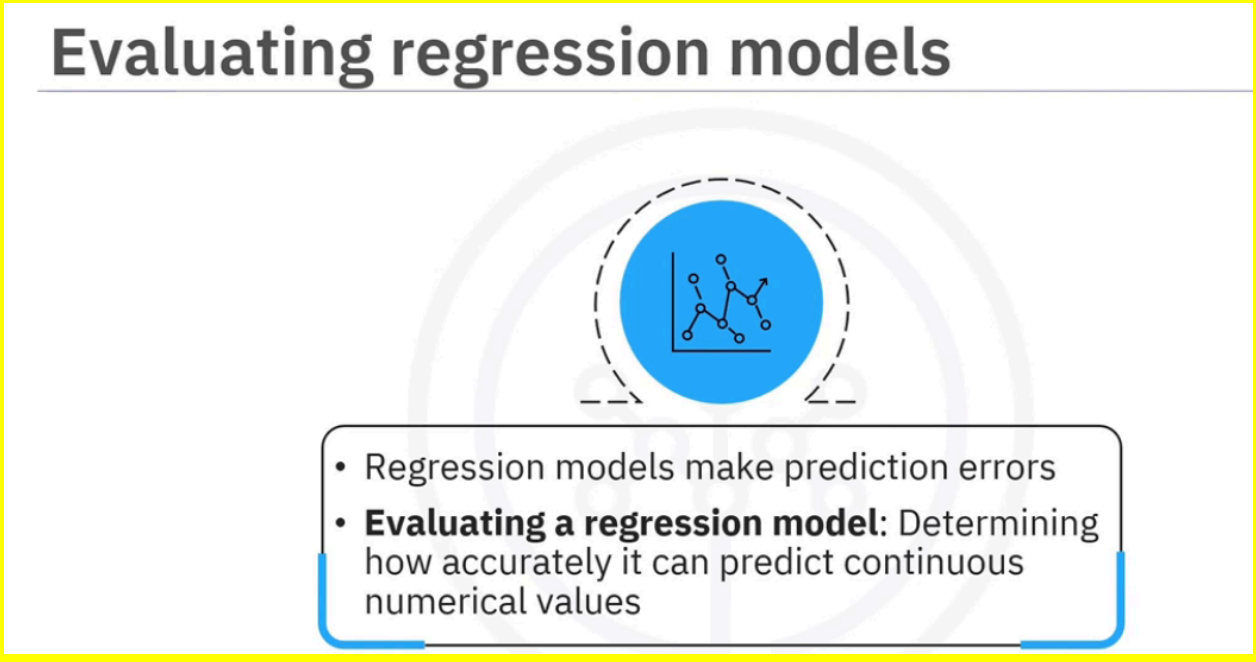
📈 Step 8: Evaluate Models

```
def evaluate_model(y_true, y_pred, model_name):
    print(f"\n--- {model_name} ---")
    print("Accuracy:", accuracy_score(y_true, y_pred))
    print("Classification Report:\n", classification_report(y_true, y_pred, target_names=labels))

    # Confusion Matrix
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(5,4))
    sns.heatmap(cm, annot=True, fmt='d', cmap="Blues", xticklabels=labels, yticklabels=labels)
    plt.title(f"Confusion Matrix: {model_name}")
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

# Evaluate both models
evaluate_model(y_test, y_pred_knn, "KNN Classifier")
evaluate_model(y_test, y_pred_svm, "SVM Classifier")
```

🔍 Key Insights

- **Accuracy** alone can be misleading (e.g., if data is imbalanced).
- **Precision** (how many predicted positives are actually positive).
- **Recall** (how many real positives were captured).
- **F1-score** (harmonic mean of precision & recall).
- **Confusion Matrix** provides detailed error distribution.
- Adding **noise** often lowers performance → simulates real-world uncertainty.

📌 Regression Metrics & Evaluation Techniques



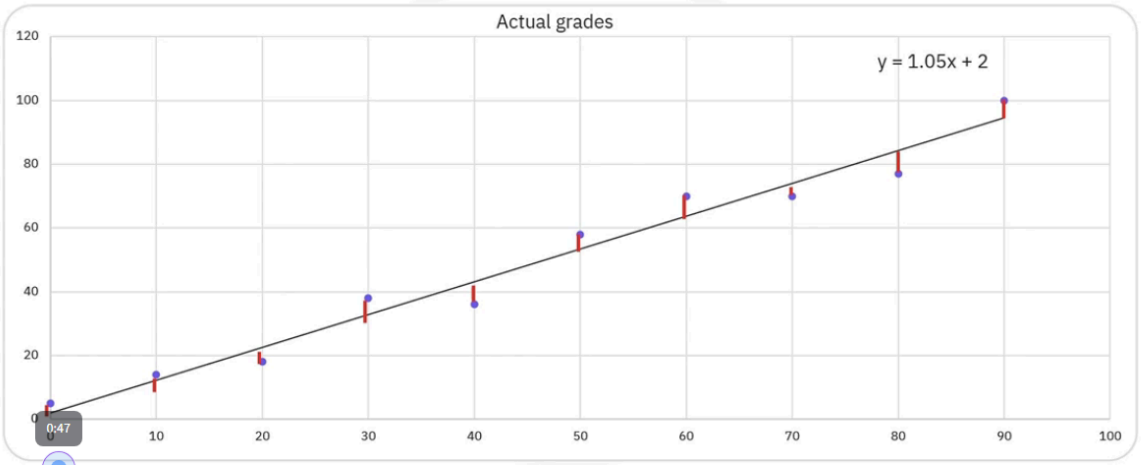
🔹 Why Evaluate Regression Models?

- Regression models predict **continuous values** (e.g., exam grades, prices, sales).

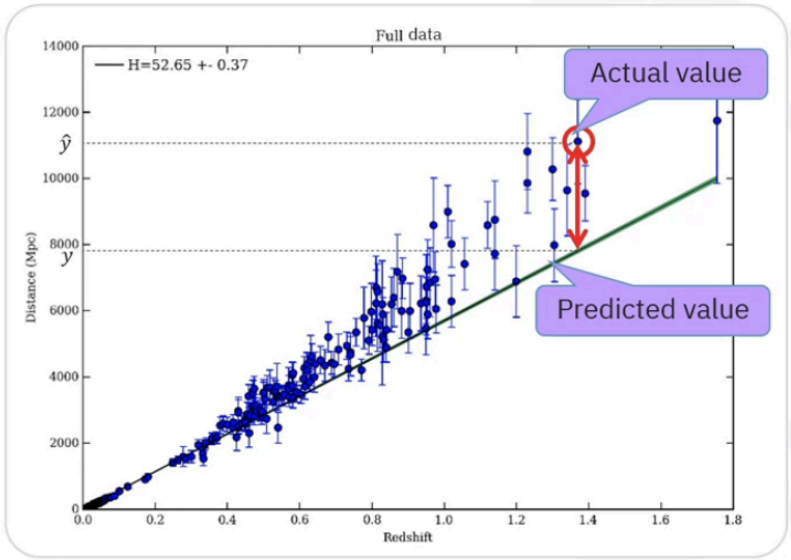
- Models are not perfect → they make **errors** (difference between predicted & actual values).

Evaluating regression models

Scenario



Error of the model



Error: Measure of the difference between the data points and the trend line generated by the algorithm

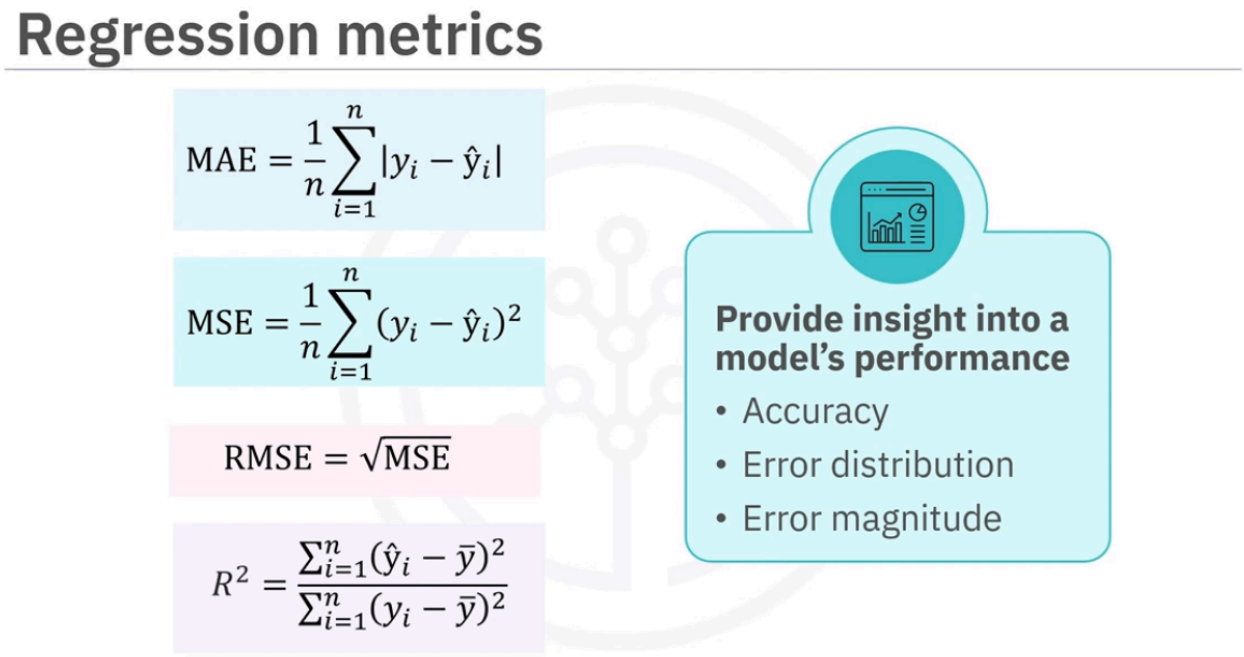
- Evaluation metrics measure:
    - **Accuracy** of predictions
    - **Error magnitude**

- **Error distribution**
- **Goodness of fit**

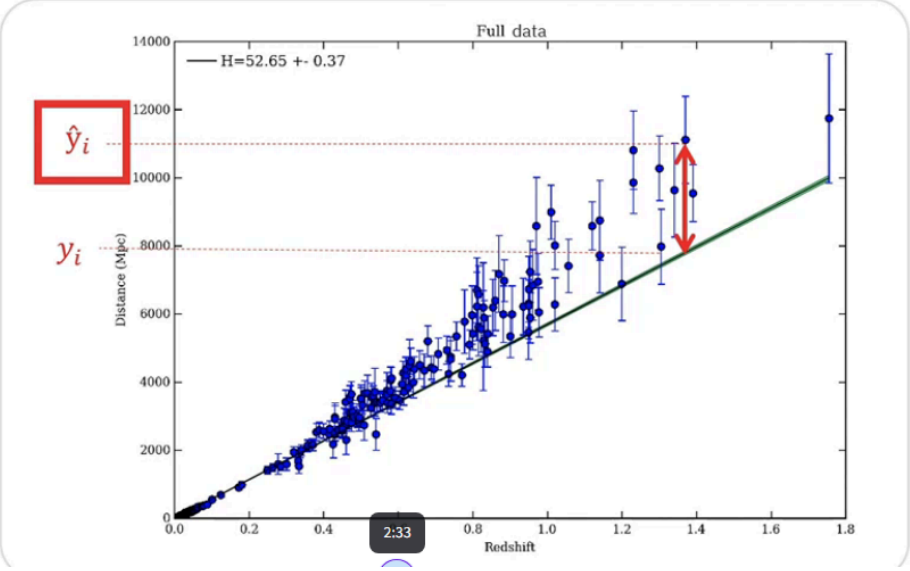
1️⃣ Error in Regression

- **Error = Predicted Value – Actual Value**
- In regression, errors are measured across all data points.
- Different metrics capture error in **different ways**.

2️⃣ Common Regression Metrics



Evaluation metrics



Explained variance and R^2

$$\text{Explained Variance} = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$$

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

$$\text{Total Variance} = \sum_{i=1}^{n} (y_i - \bar{y})^2 = \sigma^2$$

$$R^2 = \frac{\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} = 1 - \frac{\text{Unexplained Variance}}{\text{Total Variance}}$$

✅ MAE (Mean Absolute Error)

[
MAE = \frac{1}{n} \sum |y_i - \hat{y}_i|
]

- Average **absolute** difference between predicted & actual values.
- **Easy to interpret**, but treats all errors equally.

✅ MSE (Mean Squared Error)

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

- Squares the errors → **penalizes larger errors** more.
- Useful when you want to **avoid big mistakes**.

✅ RMSE (Root Mean Squared Error)

$$
RMSE = \sqrt{MSE}
$$

- Same unit as the target variable (e.g., grades, price).
- Easier to interpret than MSE.
- Popular metric for regression tasks.

✅ R^2 (Coefficient of Determination)

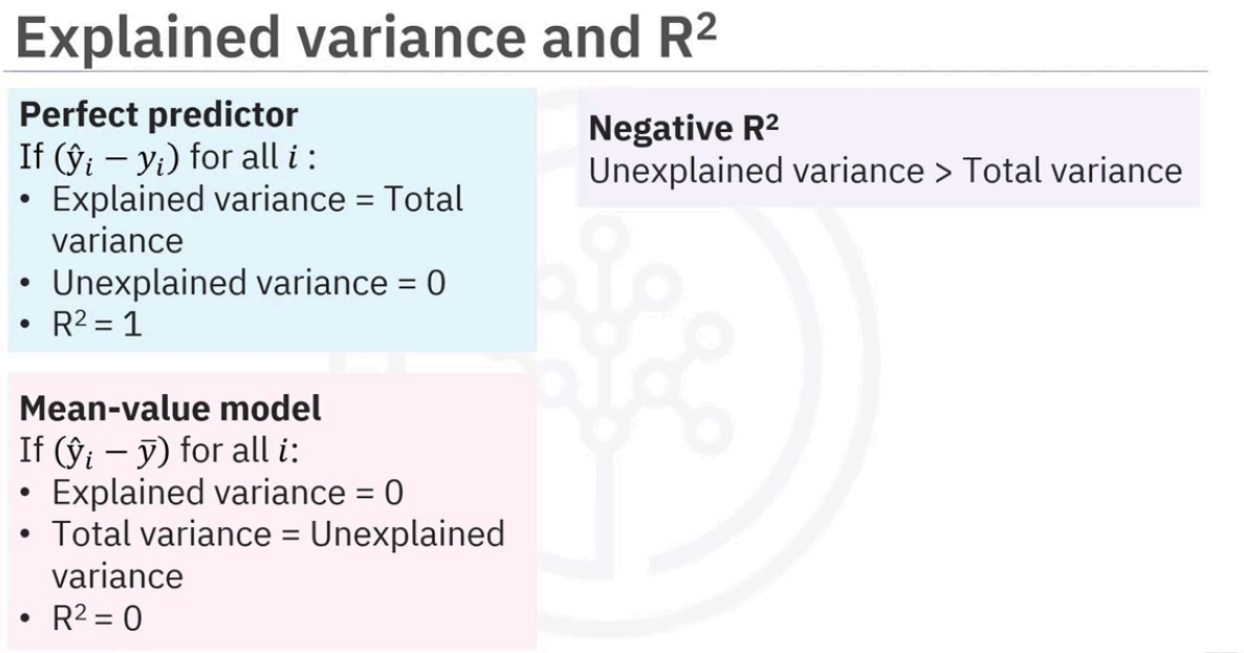
$$
R^2 = 1 - \frac{Unexplained\ Variance}{Total\ Variance}
$$

- Measures **goodness of fit**.
- Range:

- **1** → perfect model
- **0** → model predicts only mean (no variance explained)
- **< 0** → model worse than just predicting the mean

- ⚠️ Works best for **linear relationships**, may mislead for nonlinear models.

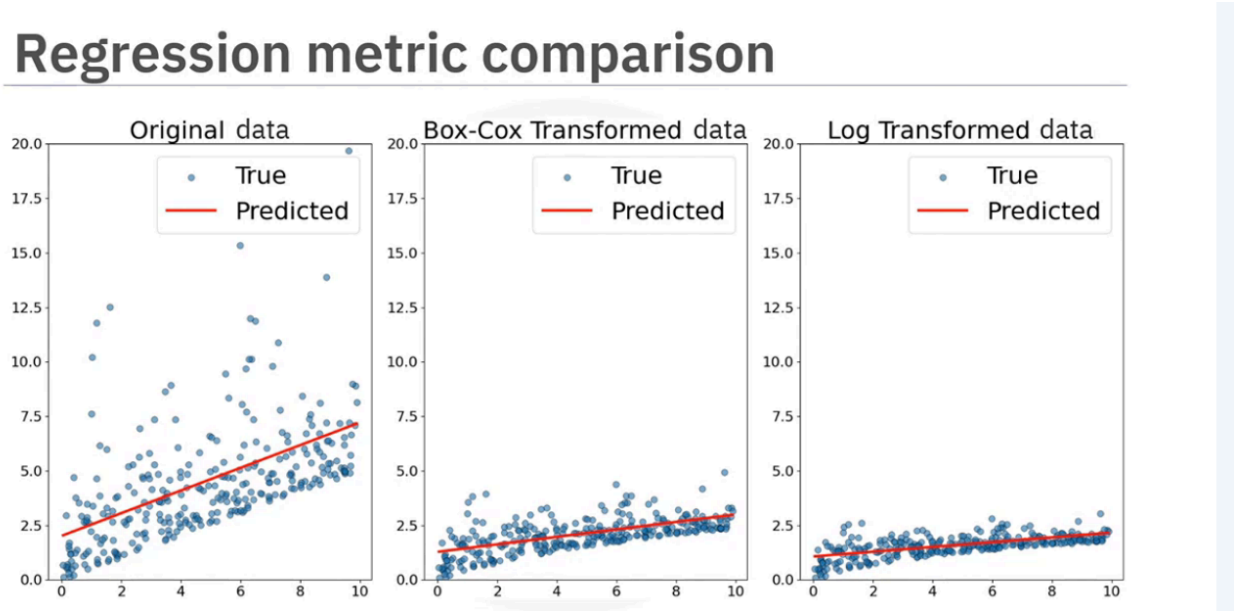
3️⃣ Explained vs. Unexplained Variance

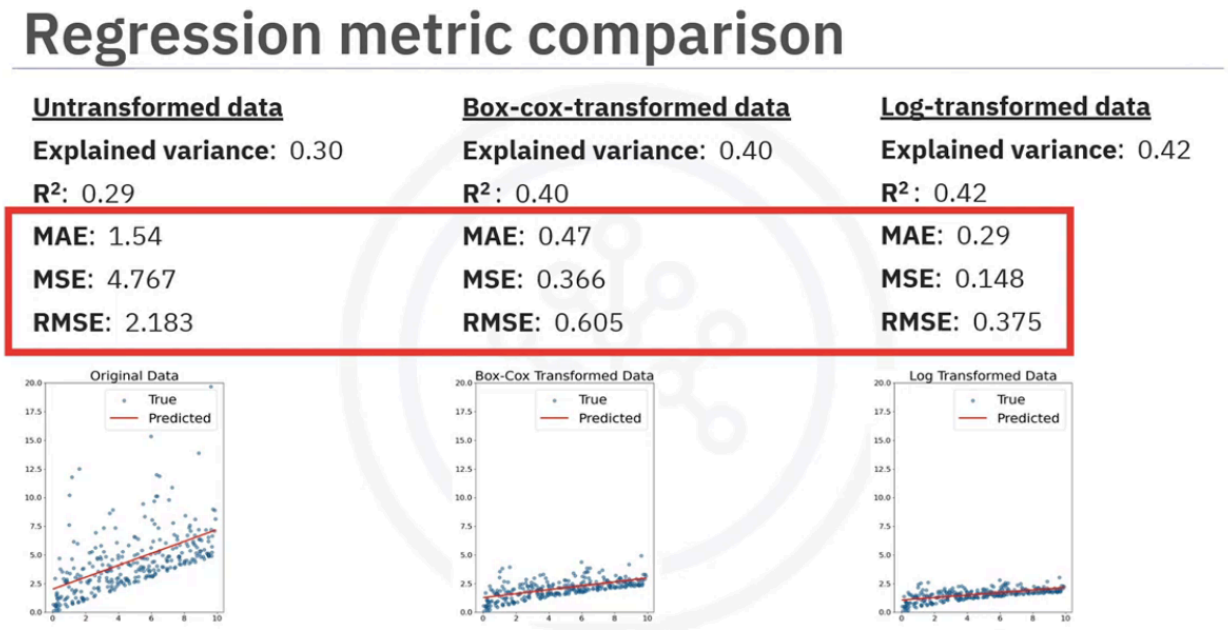


- **Explained Variance:** Portion of total variance captured by model.
- **Unexplained Variance:** Portion model fails to capture.
- **R²** compares explained vs unexplained variance.

---

4 Example Insights





- **Box-Cox & log transformations** improve regression fit for skewed (e.g., log-normal) data.
- Metrics like **R²** ↑ and **MAE/MSE/RMSE** ↓ confirm better performance.
- Always **visualize predictions vs actuals** to check fit.

✅ Key Takeaways

Recap

- **Evaluating a regression model**: Determine how accurately it can predict continuous numerical values
- **Model error**: Measure of the difference between the data points and the trend line
- **Essential regression metrics**: MAE, MSE, RMSE, and R2
- **Explained variance**: Sum of squared differences between the predictions and the average value of the actual target data
- **R2**: Measures the proportion of variance in target variable predictable from input variables

- Regression models must be evaluated because predictions always have **errors**.
- **MAE** → average size of errors.
- **MSE** → penalizes big errors more.
- **RMSE** → interpretable, same unit as target.
- **R^2** → variance explained, measures fit.
- No single metric is enough → combine metrics + visual plots.

🧠 Evaluating Random Forest Performance

Objective:
Learn how to:

- Implement and evaluate **Random Forest Regression** on real-world data.
- Interpret model evaluation metrics & plots.
- Understand **feature importance**.

📦 Step 1. Install Required Libraries

```
!pip install numpy==2.2.0 pandas==2.2.3 scikit-learn==1.6.0 matplotlib==3.9.3 scipy==1.14.1
```

🧩 Step 2. Import Libraries

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt


from sklearn.datasets import fetch_california_housing

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score,
root_mean_squared_error

from scipy.stats import skew
```

🏡 Step 3. Load the California Housing Dataset

```
data = fetch_california_housing()

X, y = data.data, data.target
```

Dataset Info:

- Instances: 20,640
- Features: 8 numeric (predictive attributes)
- Target: Median house value ($100,000 units)
- Source: 1990 U.S. Census (California districts)

✂️ Step 4. Split Data into Train & Test Sets

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

🔍 Step 5. Exploratory Data Analysis (EDA)

```
eda = pd.DataFrame(data=X_train, columns=data.feature_names)

eda["MedHouseVal"] = y_train

eda.describe()
```

Question: What range are most house prices in?
➡ Most median house prices fall between **$119,300 and $265,000**.

📊 Step 6. Distribution of Median House Prices

```
plt.hist(1e5 * y_train, bins=30, color='lightblue', edgecolor='black')

plt.title(f"Median House Value Distribution\nSkewness: {skew(y_train):.2f}")

plt.xlabel("Median House Value ($)")

plt.ylabel("Frequency")

plt.show()
```

📈 **Insight:**
Distribution is **right-skewed**, with values capped near **$500,000**.

🌲 Step 7. Train the Random Forest Model

```
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)

rf_regressor.fit(X_train, y_train)

y_pred_test = rf_regressor.predict(X_test)
```

📏 Step 8. Evaluate Model Performance

```
mae = mean_absolute_error(y_test, y_pred_test)

mse = mean_squared_error(y_test, y_pred_test)

rmse = root_mean_squared_error(y_test, y_pred_test)

r2 = r2_score(y_test, y_pred_test)


print(f"Mean Absolute Error (MAE): {mae:.4f}")

print(f"Mean Squared Error (MSE): {mse:.4f}")
```

```
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")

print(f"R² Score: {r2:.4f}")
```

Output:

MAE: 0.3276

MSE: 0.2556

RMSE: 0.5055

R²: 0.8050

Interpretation:

- Average error ≈ **$33,220**
- RMSE ≈ **$50,630**
- R² = **0.80**, meaning model explains ~80% variance.

⚖️ Step 9. Actual vs Predicted Plot

```
plt.scatter(y_test, y_pred_test, alpha=0.5, color="blue")

plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=2)

plt.xlabel("Actual Values")

plt.ylabel("Predicted Values")

plt.title("Random Forest Regression - Actual vs Predicted")

plt.show()
```

✅ **Observation:**
Most predictions align well but some deviation at extreme price ranges.

🪟 Step 10. Residual Analysis

(a) Histogram of Residuals

```
residuals = 1e5 * (y_test - y_pred_test)

plt.hist(residuals, bins=30, color="lightblue", edgecolor="black")

plt.title("Median House Value Prediction Residuals")

plt.xlabel("Prediction Error ($)")

plt.ylabel("Frequency")

print("Average error =", int(np.mean(residuals)))

print("Standard deviation of error =", int(np.std(residuals)))
```

Result:

Average error = -1215

Std deviation = 50537

📊 Residuals roughly follow a **normal distribution**, centered near zero.

(b) Residuals vs Actual Prices

```
residuals_df = pd.DataFrame({
    'Actual': 1e5 * y_test,
    'Residuals': residuals
}).sort_values(by='Actual')

plt.scatter(residuals_df['Actual'], residuals_df['Residuals'], marker='o', alpha=0.4, ec='k')
```

```
plt.title("Residuals Ordered by Actual Median Prices")
plt.xlabel("Actual Median House Value ($)")
plt.ylabel("Residuals ($)")
plt.grid(True)
plt.show()
```

Interpretation:

- Low-value houses are **overpredicted**.
- High-value houses are **underpredicted**.
- Indicates mild bias along price scale.

🌟 Step 11. Feature Importance Visualization

```
importances = rf_regressor.feature_importances_
indices = np.argsort(importances)[::-1]
features = data.feature_names

plt.bar(range(X.shape[1]), importances[indices], align="center")
plt.xticks(range(X.shape[1]), [features[i] for i in indices], rotation=45)
plt.xlabel("Feature")
plt.ylabel("Importance")
plt.title("Feature Importances in Random Forest Regression")
plt.show()
```

Top Features:

1. Median Income (`MedInc`)
2. Average Rooms (`AveRooms`)
3. House Age (`HouseAge`)

💡 **Insight:**
Income strongly correlates with house price — expected.
Some correlated features (e.g., rooms & bedrooms) may share importance.

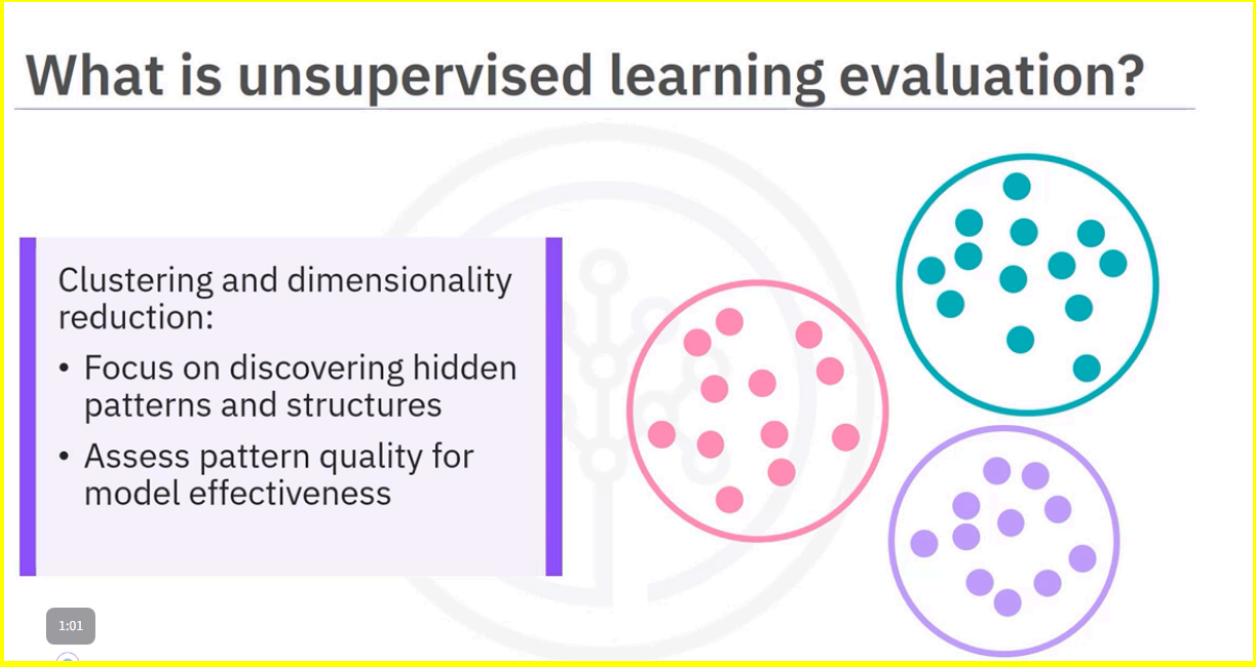
🧾 Summary

Metric	Value	Interpretation
MAE	0.3276	Avg error ≈ $33k
RMSE	0.5055	Typical deviation ≈ $50k
R² Score	0.805	80% variance explained
Bias	Low	Slight underprediction for expensive homes
Top Feature	`MedInc`	Highest influence

🧭 Key Takeaways

- Random Forest performs **well** on housing data but not perfect.
- Income, location, and room count are dominant predictors.
- Residual analysis helps detect bias in predictions.
- Always visualize results to **interpret model reliability**.

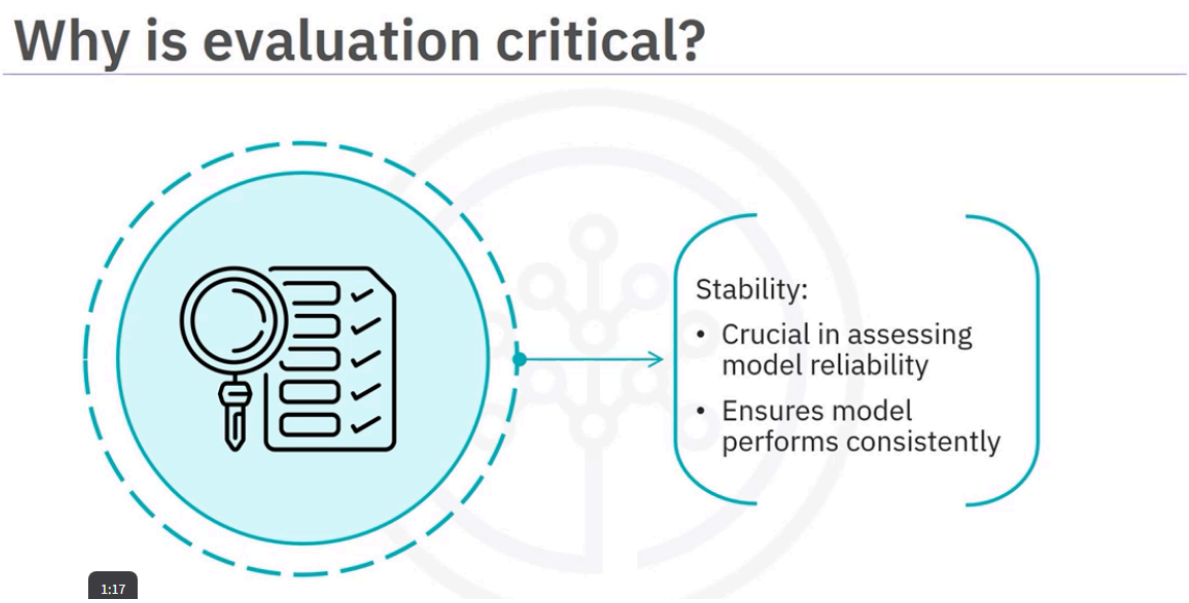
🧠 Evaluating Unsupervised Learning Models: Heuristics & Techniques



🔹 1. Purpose of Evaluation

- Unsupervised learning has **no predefined labels or ground truth**.
- The goal is to **discover hidden patterns and structures** in data.

- Evaluation checks:

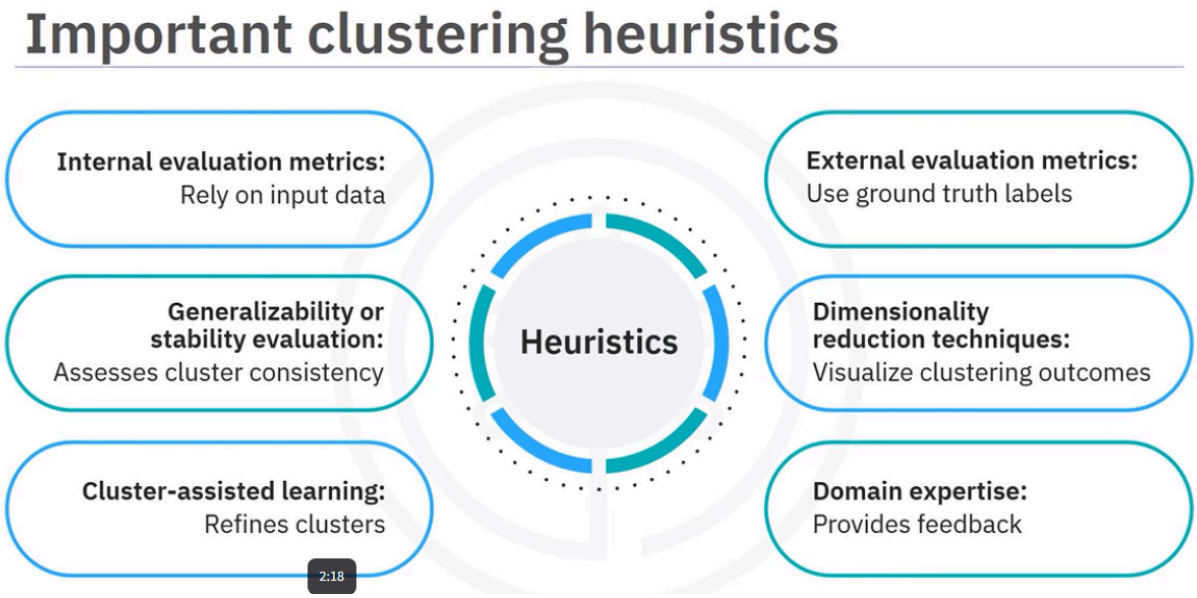


  - **Quality of discovered patterns**
  - **Model stability** (consistency across data variations)
  - **How effectively data points are grouped**

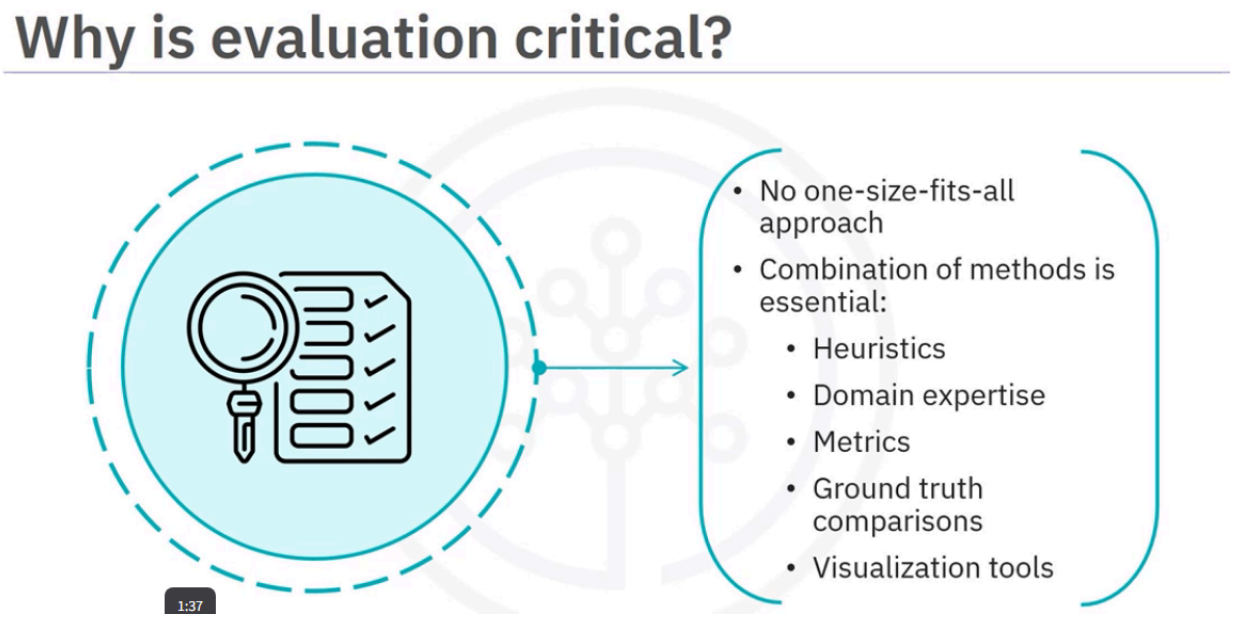
⚡ Example: A stable clustering model gives similar clusters even when data slightly changes.

🔹 2. Importance of Stability

- Ensures **model reliability**.
- Stable models produce **consistent results** on different subsets or versions of data.



🔹 **3. Evaluation Approaches**



There is **no single correct method** — multiple techniques are combined:

- **Heuristics**
- **Domain expertise**

- **Quantitative metrics**
- **Ground truth (when available)**
- **Visualization tools** (e.g., scatter plots, PCA, t-SNE)

🧩 4. Clustering Evaluation

Internal clustering evaluation metrics

Assess the quality of clustering

Focuses on input data

Internal clustering evaluation metrics

Silhouette score:

- Compares cohesion within each cluster
- Assesses separation from other clusters
- Ranges from -1 to 1
- Higher values indicate better-defined clusters

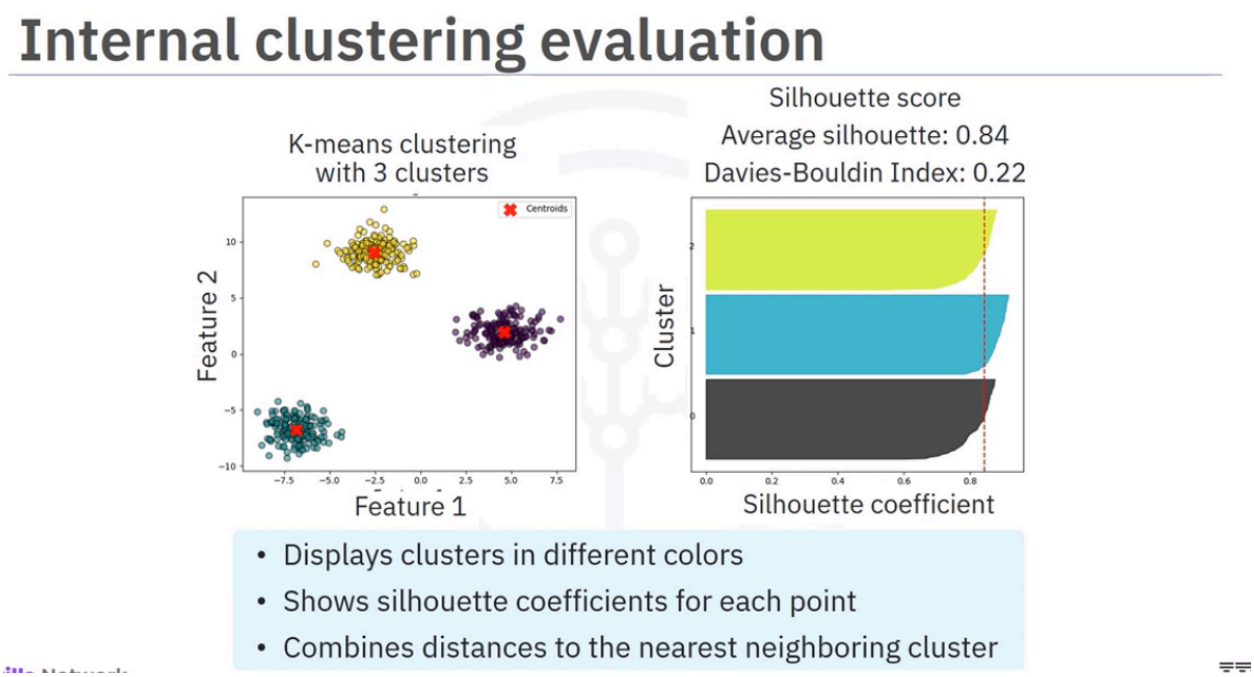
Davies-Bouldin Index:

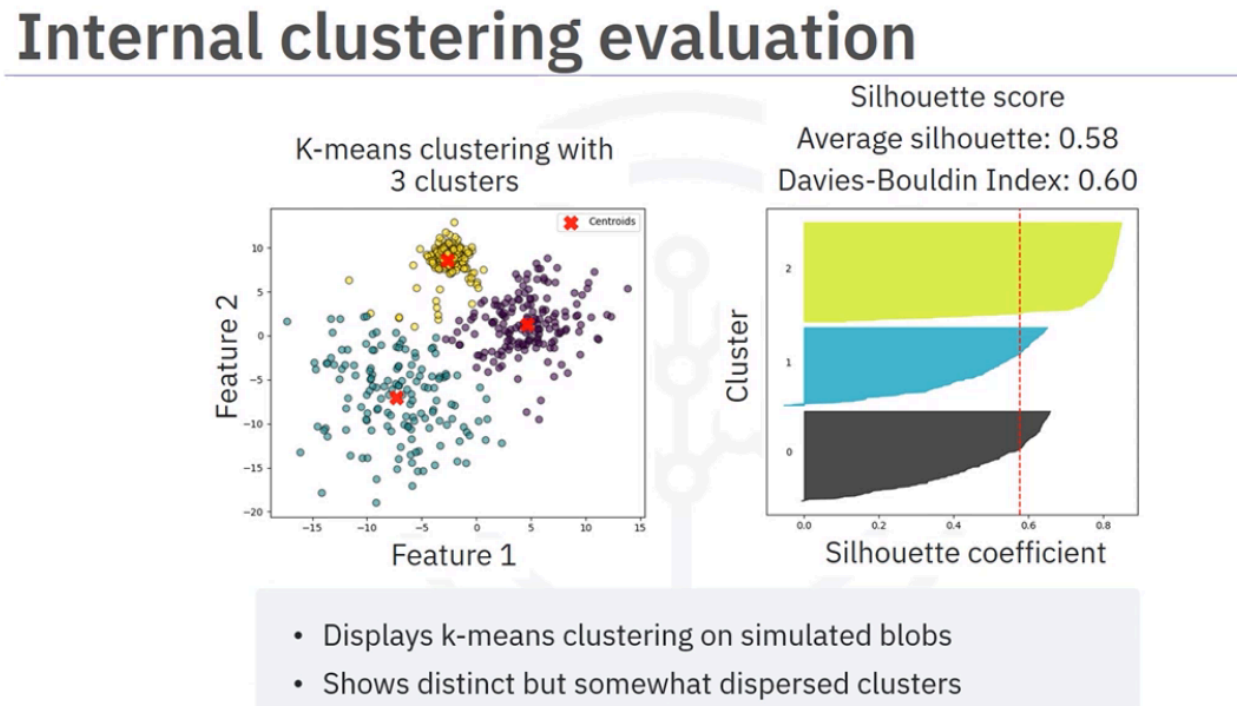
- Measures cluster compactness ratio
- Assesses separation from the nearest cluster
- Lower values indicate more distinct clusters
- Indicates more compact clusters

3:00

Inertia:

- Used in k-means clustering
- Calculates the sum of variances within clusters
- Lower values suggest more compact clusters
- Increasing clusters reduces variance





A. Internal Evaluation Metrics

→ Based only on the input data (no true labels).

Metric	Description	Goal

Silhouette Score	Measures cohesion within a cluster vs. separation from others. Range: -1 to +1	Higher = better-defined clusters
Davies–Bouldin Index (DBI)	Average ratio of cluster compactness to separation	Lower = better
Inertia (K-Means)	Sum of within-cluster variances	Lower = more compact clusters

🧪 **Example Results:**

Case	Silhouette	DBI	Interpretation
Distinct & Dense Clusters	0.84	0.22	Excellent clustering
Spread-out Clusters	0.58	0.6	Reasonable clustering, some overlap

---

B. External Evaluation Metrics

External clustering evaluation metrics

Adjusted Rand Index

- Measures the similarity between true labels and outcomes
- Ranges from -1-1 for scoring
- Score of 1 indicates perfect alignment
- Score of 0 indicates random clustering
- Negative values suggest worse-than-random performance

Normalized Mutual Information

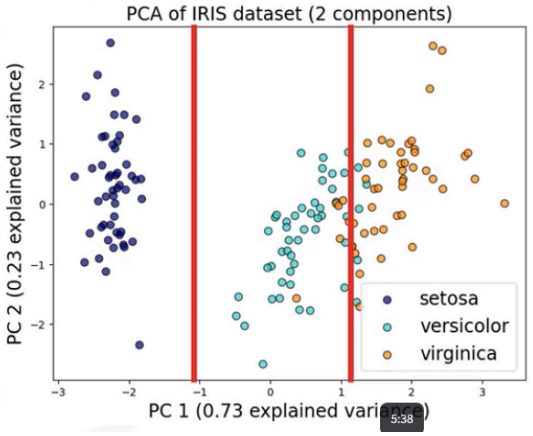
- Quantifies shared information between cluster assignments
- Scale ranges from 0 to 1
- Score of 1 indicates perfect agreement
- Score of 0 indicates no shared information

Fowlkes-Mallows Index

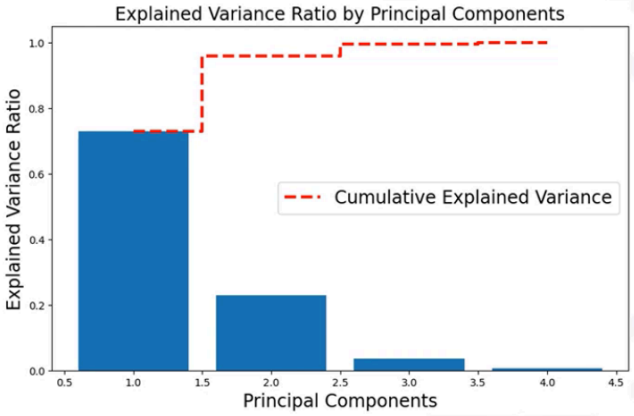
- Measures clustering performance
- Calculates the geometric mean of precision
- Calculates the geometric mean of recall
- Higher score indicates better clustering results

Explained variance ratio for PCA

- Scatterplot displays the first two principal components
- Points are color-coded by iris species
- Species include setosa, versicolor, and virginica



Explained variance ratio for PCA



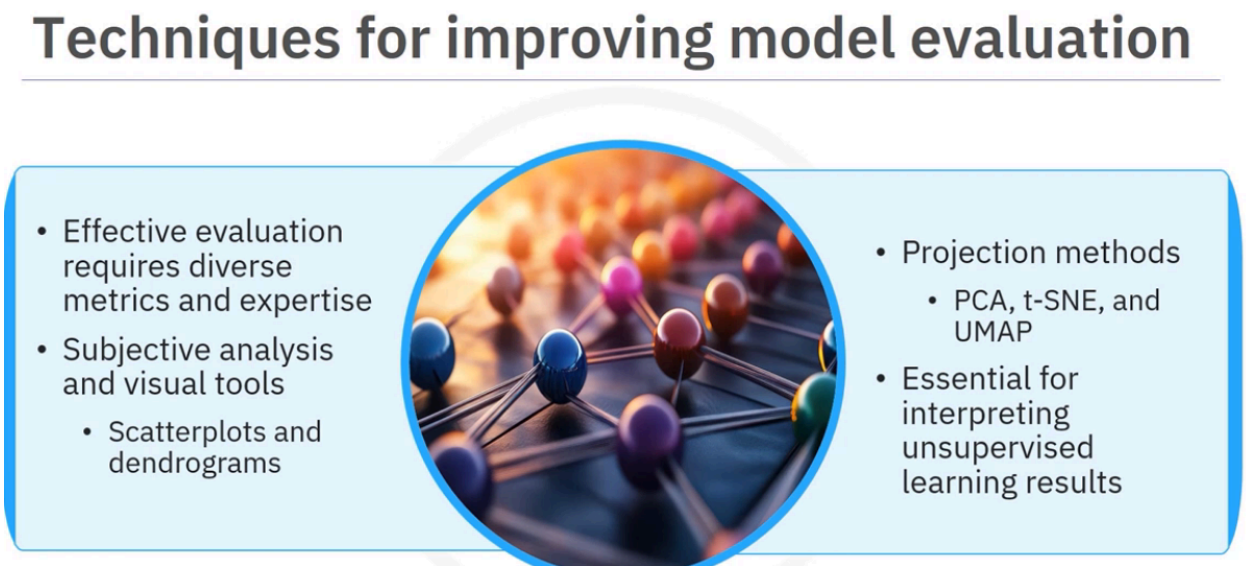
- Red dashed line shows the cumulative explained variance
- First two components account for most variance
- Additional components add little to explain variance

→ Used **when true labels are available**.

Metric	Description	Range / Ideal Value
Adjusted Rand Index (ARI)	Measures similarity between true and predicted labels	-1 to 1 → 1 = perfect match
Normalized Mutual Information (NMI)	Measures shared information between clusters and labels	0 to 1 → 1 = perfect alignment
Fowlkes–Mallows Index (FMI)	Geometric mean of precision & recall	0 to 1 → higher = better clustering

---

🌀 5. Dimensionality Reduction Evaluation



Used for **visualizing and validating clustering** results (e.g., PCA, t-SNE, UMAP).

Metric	Description	Goal
Explained Variance Ratio (PCA)	Measures variance captured by principal components	Higher = better representation
Reconstruction Error	Difference between original and reconstructed data	Lower = better preservation
Neighborhood Preservation	How well local data relationships are maintained	Higher = better for visualization methods (t-SNE, UMAP)

📊 Example:

- PCA on **Iris dataset**:
    - PC1 nearly separates species.
    - First 2 PCs capture most variance.
    - Bar plot shows cumulative explained variance (red dashed line).

🎯 6. Visualization & Interpretation Tools

- **Scatter plots** – visualize cluster separation.
- **Silhouette plots** – show cohesion & separation.
- **Dendrograms** – visualize hierarchical clustering.
- **Projection plots (PCA, t-SNE, UMAP)** – visualize high-dimensional data in 2D.

🧩 7. Key Takeaways

✅ Evaluate unsupervised models using multiple methods.
✅ Stability ensures reliability across data variations.
✅ Combine internal & external metrics when possible.
✅ Dimensionality reduction helps visualise and interpret clusters.
✅ Domain expertise is crucial for understanding patterns and anomalies.

Here's a **clean, concise, and complete note set with code** for **Evaluating K-Means Clustering**, written in a structured and easy-to-revise format.
All unnecessary logs are removed, and the notes are organized topic-wise with explanations and runnable code.

📘 Evaluating K-Means Clustering

Date: October 11, 2025

Goal: Learn to implement and evaluate K-Means clustering using synthetic data.

🧠 1. Objectives

After completing this lab, you'll be able to:

- Implement and evaluate **K-Means clustering** on synthetic data.
- Interpret **evaluation metrics** and **visualizations**.
- Compare clustering results against known classes using synthetic data.

⚙️ 2. Introduction

In this lab, you will:

- Generate **synthetic datasets** using `scikit-learn`.
- Create and compare **K-Means models**.
- Analyze **evaluation metrics** like:
    - **Silhouette Score**
    - **Davies-Bouldin Index**
    - **Inertia (Elbow Method)**

The goal is to build **intuition** around identifying *good clustering solutions*.

🧩 3. Import Required Libraries

```
# Install dependencies (run once if needed)
!pip install numpy==2.2.0 pandas==2.2.3 scikit-learn==1.6.0 matplotlib==3.9.3 scipy==1.14.1


# Importing essential libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score, silhouette_samples, davies_bouldin_score
from matplotlib import cm
```

🧮 4. Define Clustering Evaluation Function

```
def evaluate_clustering(X, labels, n_clusters, ax=None, title_suffix=''):
    """
    Evaluate clustering using Silhouette Score & Davies-Bouldin Index.
    Displays a silhouette plot for visual analysis.
    """
    if ax is None:
        ax = plt.gca()

    # Calculate silhouette metrics
    silhouette_avg = silhouette_score(X, labels)
    sample_silhouette_values = silhouette_samples(X, labels)

    # Create plot
    unique_labels = np.unique(labels)
    colormap = cm.tab10
    y_lower = 10

    for i in unique_labels:
        cluster_values = sample_silhouette_values[labels == i]
        cluster_values.sort()
        size = cluster_values.shape[0]
        y_upper = y_lower + size
        color = colormap(float(i) / n_clusters)
```

```
        ax.fill_betweenx(np.arange(y_lower, y_upper),
                         0, cluster_values,
                         facecolor=color, edgecolor=color, alpha=0.7)
        ax.text(-0.05, y_lower + 0.5 * size, str(i))
        y_lower = y_upper + 10

    ax.axvline(x=silhouette_avg, color="red", linestyle="--")
    ax.set_title(f'Silhouette Plot {title_suffix}\nAvg Silhouette = {silhouette_avg:.2f}')
    ax.set_xlabel("Silhouette Coefficient")
    ax.set_yticks([])
```

🌐 5. Create Synthetic Data (4 Blobs)

```
from sklearn.datasets import make_blobs

X, y = make_blobs(
    n_samples=500, n_features=2,
    centers=4, cluster_std=[1.0, 3, 5, 2],
    random_state=42
)
```

Apply and Visualize K-Means

```
n_clusters = 4
```

```
kmeans = KMeans(n_clusters=n_clusters, random_state=42)

y_kmeans = kmeans.fit_predict(X)

colormap = cm.tab10


plt.figure(figsize=(18, 6))


# Original data

plt.subplot(1, 3, 1)

plt.scatter(X[:, 0], X[:, 1], s=50, alpha=0.6, edgecolor='k')

plt.title("Original Synthetic Data")


# Clustered data

plt.subplot(1, 3, 2)

plt.scatter(X[:, 0], X[:, 1], c=colormap(y_kmeans.astype(float)/n_clusters), s=50, edgecolor='k')

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='red', s=200,
marker='X')

plt.title("K-Means Clustering Result")


# Silhouette plot

plt.subplot(1, 3, 3)

evaluate_clustering(X, y_kmeans, n_clusters, title_suffix='(k=4)')

plt.tight_layout()

plt.show()
```

📊 6. Cluster Stability (Different Random Seeds)

```
n_runs = 8

inertia_values = []


plt.figure(figsize=(16, 16))

for i in range(n_runs):

    kmeans = KMeans(n_clusters=4, random_state=None)

    kmeans.fit(X)

    inertia_values.append(kmeans.inertia_)


    plt.subplot(4, 2, i + 1)

    plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='tab10', alpha=0.6, edgecolor='k')

    plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],

                c='red', s=200, marker='x')

    plt.title(f'Run {i + 1}')


plt.tight_layout()

plt.show()


# Display inertia scores

for i, inertia in enumerate(inertia_values, start=1):

    print(f'Run {i}: Inertia = {inertia:.2f}')
```

📍 **Observation:**

- Inertia values differ between runs when centroids start randomly.
- Stable models have **consistent inertia** across runs.
- Variability indicates sensitivity to initialization.

📈 7. Optimal Number of Clusters (Elbow, Silhouette & DB Index)

```
k_values = range(2, 11)

inertia_values = []

silhouette_scores = []

db_indices = []


for k in k_values:

    kmeans = KMeans(n_clusters=k, random_state=42)

    y_kmeans = kmeans.fit_predict(X)


    inertia_values.append(kmeans.inertia_)

    silhouette_scores.append(silhouette_score(X, y_kmeans))

    db_indices.append(davies_bouldin_score(X, y_kmeans))


plt.figure(figsize=(18, 6))


# Inertia

plt.subplot(1, 3, 1)
```

```
plt.plot(k_values, inertia_values, marker='o')
plt.title("Elbow Method (Inertia vs k)")
plt.xlabel("Number of Clusters")
plt.ylabel("Inertia")

# Silhouette
plt.subplot(1, 3, 2)
plt.plot(k_values, silhouette_scores, marker='o')
plt.title("Silhouette Score vs k")

# Davies-Bouldin Index
plt.subplot(1, 3, 3)
plt.plot(k_values, db_indices, marker='o')
plt.title("Davies-Bouldin Index vs k")

plt.tight_layout()
plt.show()
```

📍 **Interpretation:**

- **Elbow Method:** Best around *k = 3 or 4*.
- **Silhouette Score:** Peaks at *k = 3*.
- **Davies-Bouldin Index:** Lowest between *k = 3 and 4*.
  ➡️ **Optimal clusters ≈ 3 or 4**

🎯 8. Compare Cluster Results (k = 2, 3, 4)

```
plt.figure(figsize=(18, 12))

colormap = cm.tab10


for i, k in enumerate([2, 3, 4]):

    kmeans = KMeans(n_clusters=k, random_state=42)

    y_kmeans = kmeans.fit_predict(X)

    colors = colormap(y_kmeans.astype(float)/k)


    # Scatter plot

    ax1 = plt.subplot(2, 3, i + 1)

    ax1.scatter(X[:, 0], X[:, 1], c=colors, s=50, alpha=0.6, edgecolor='k')

    ax1.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],

                c='red', s=200, marker='X', label='Centroids')

    ax1.set_title(f"K-Means Results (k={k})")

    ax1.legend()


    # Silhouette plot

    ax2 = plt.subplot(2, 3, i + 4)

    evaluate_clustering(X, y_kmeans, k, ax=ax2, title_suffix=f'(k={k})')


plt.tight_layout()

plt.show()
```

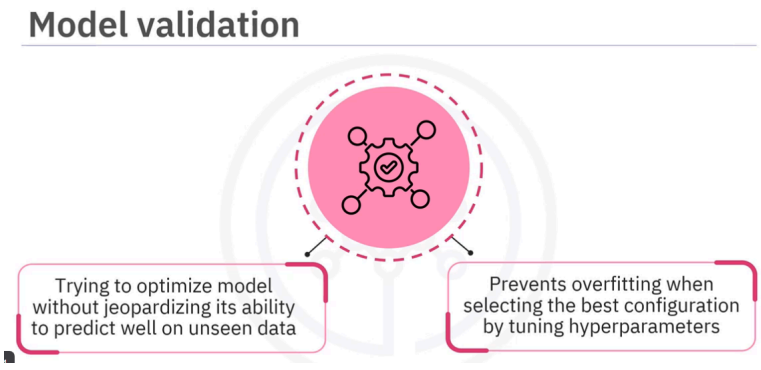
🗒️ Summary

Metric	Meaning	Goal
Inertia	Compactness of clusters	Lower = Better
Silhouette Score	How well points fit their cluster	Higher = Better
Davies-Bouldin Index	Similarity between clusters	Lower = Better

📍 **Best Practice:** Use all three metrics together for reliable cluster evaluation.

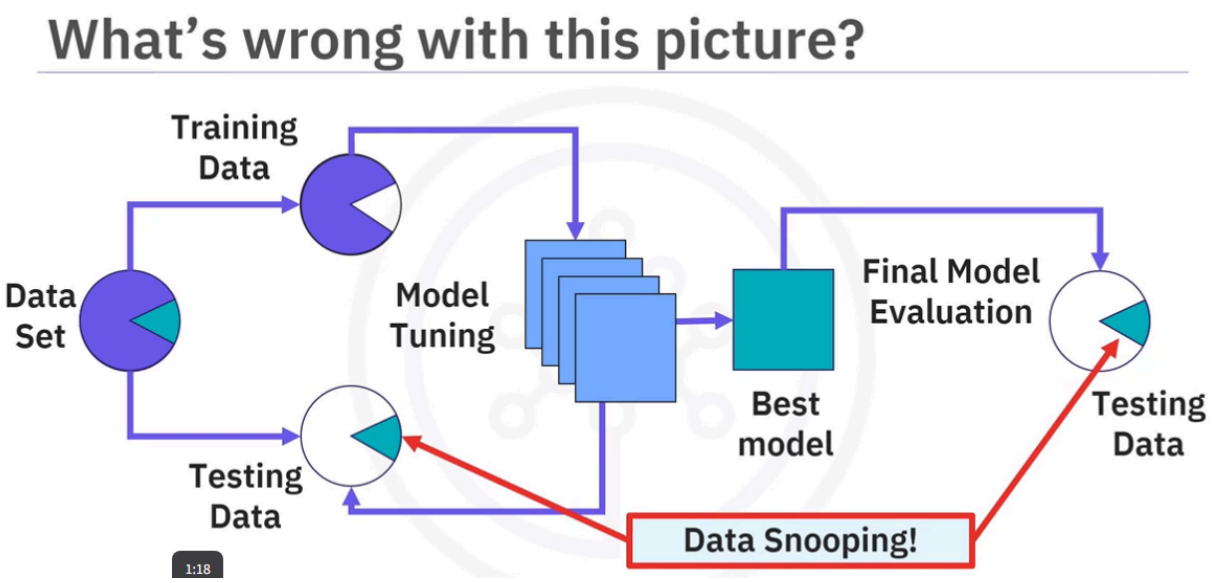
🎯 Cross-Validation and Advanced Model Validation Techniques

🧠 1. Model Validation



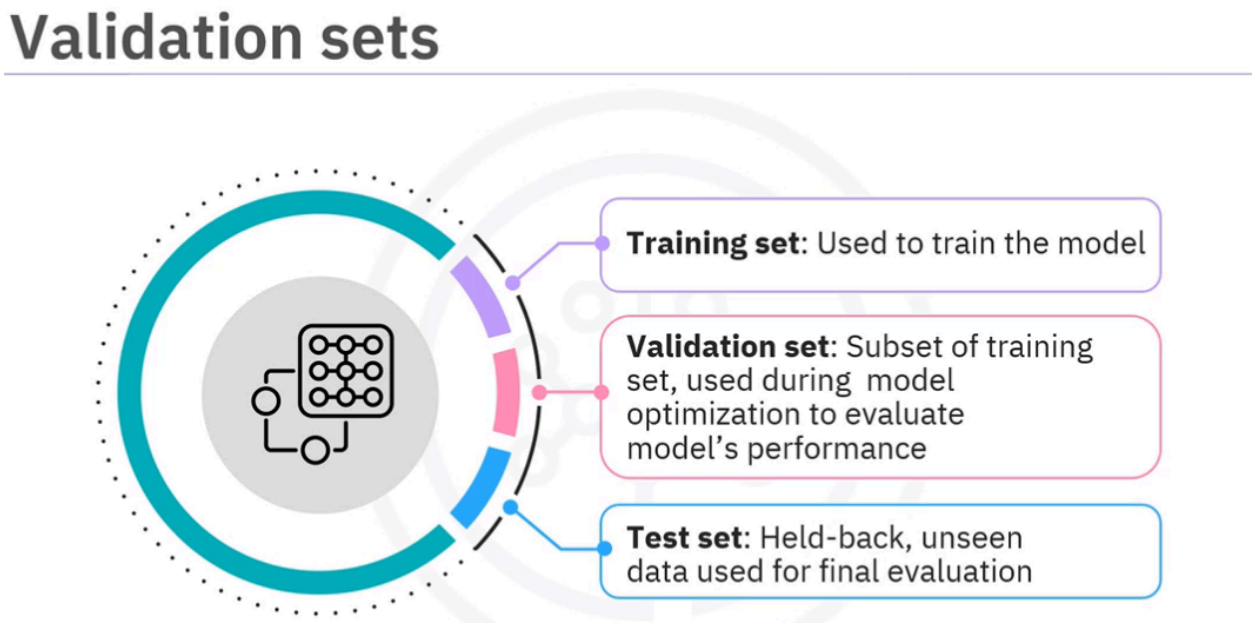
- **Definition:** Process of evaluating how well a machine learning model generalizes to **unseen data**.
- **Goal:** Prevent **overfitting** when selecting model hyperparameters.
- **Purpose:**
    - Ensure model generalization
    - Optimize hyperparameters safely
    - Provide reliable performance estimation

⚠️ 2. Data Snooping (Data Leakage)



- **Definition:** Using test data during model training or hyperparameter tuning.
- **Result:** Model learns test data patterns → **overfitting** → poor generalization.
- **Avoid by:**
    - Separating data into **training**, **validation**, and **test** sets.
    - Only using the **test set once** (final evaluation).

🧩 3. Three-Set Strategy

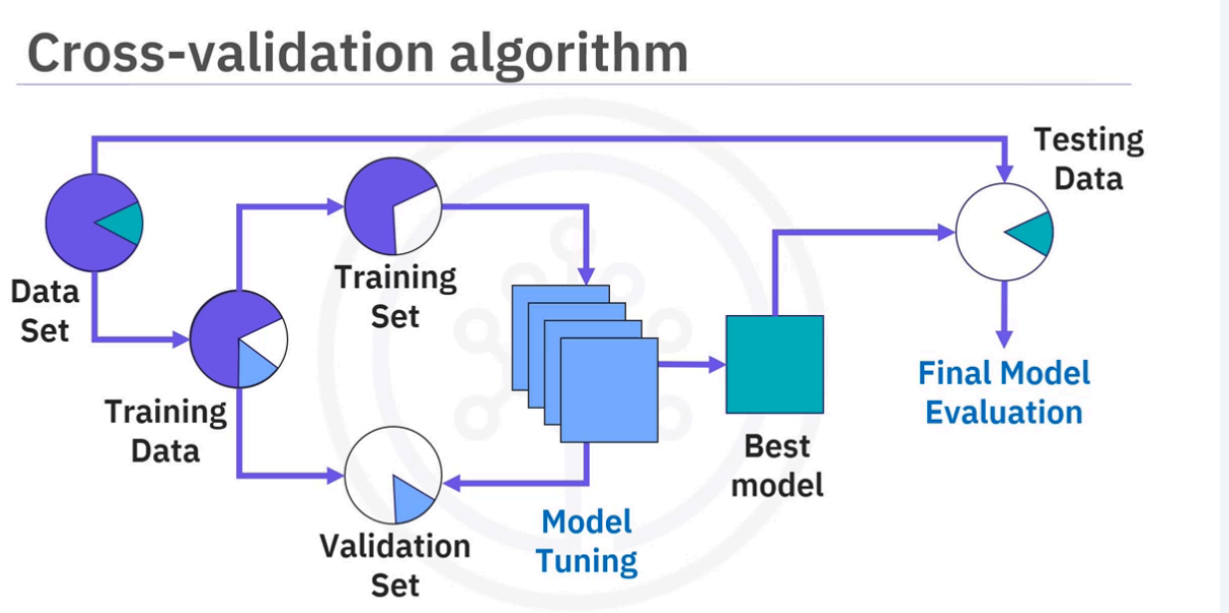


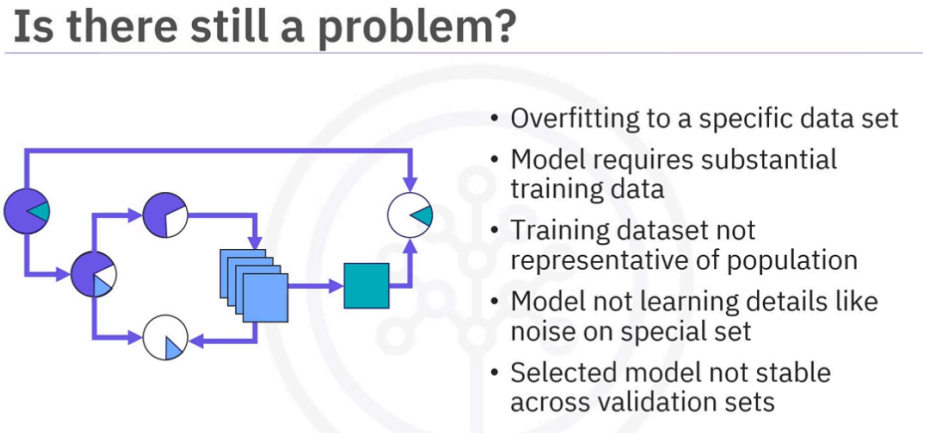
Dataset	Purpose
Training Set	Train model + tune hyperparameters

Validation Set	Evaluate model during tuning
Test Set	Final performance check (unseen data)

✅ Ensures model tuning and final evaluation are **decoupled**.

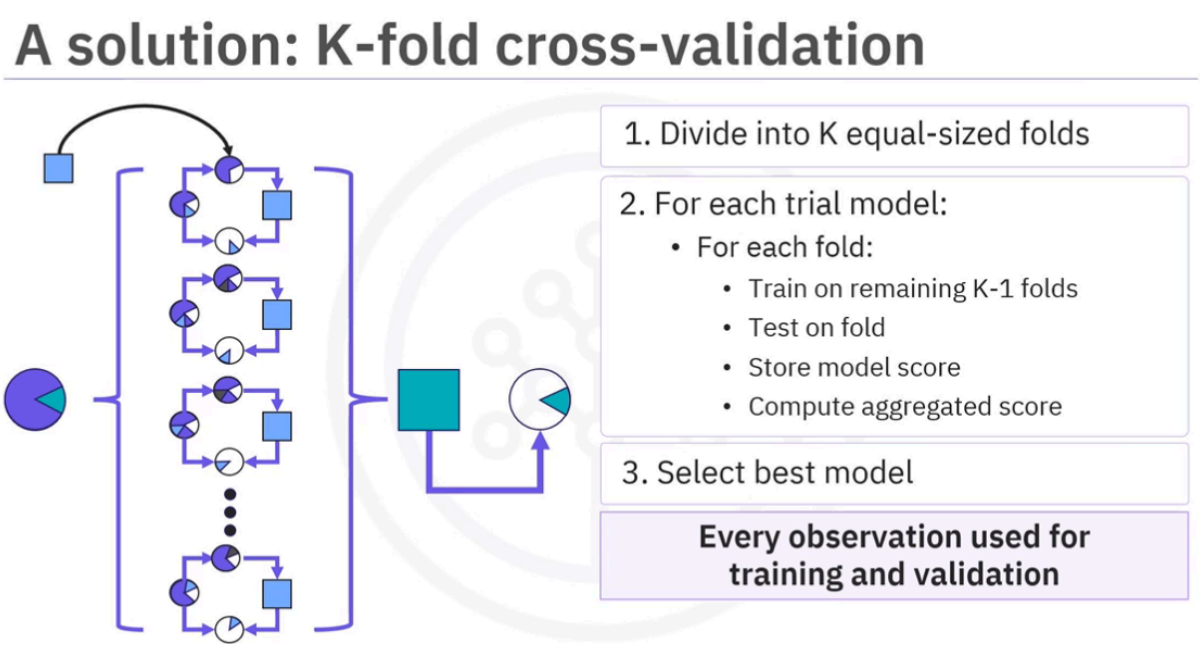
🔁 4. Cross-Validation (CV)





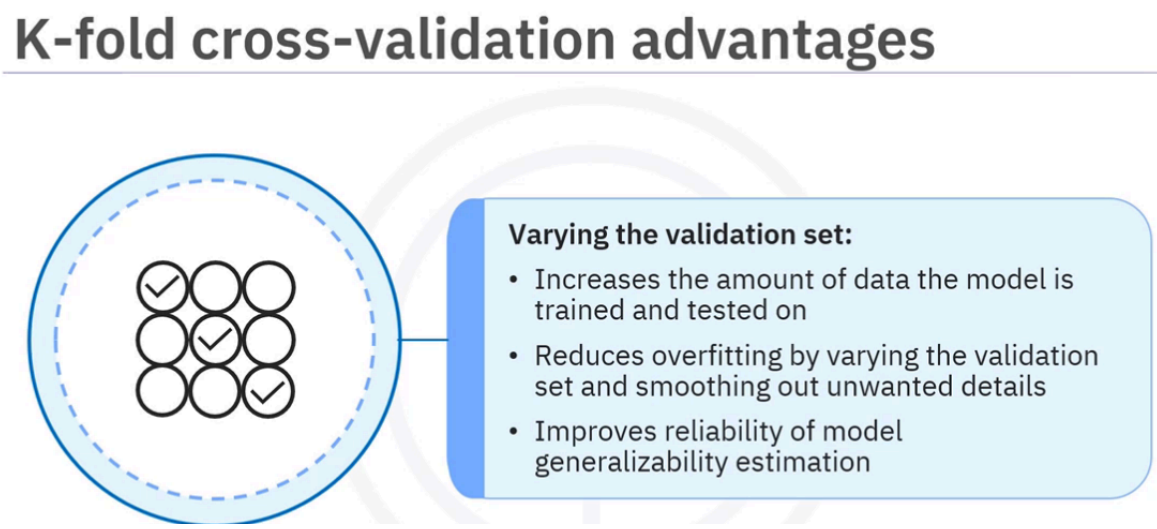
- **Purpose:** Provides a more reliable estimate of model performance.
- **Process:**
    1. Split data into **K equal folds**.
    2. Train the model on **K–1 folds**, test on the remaining one.
    3. Repeat for all folds.
    4. Average results → **overall performance**.

🔹 5. K-Fold Cross-Validation

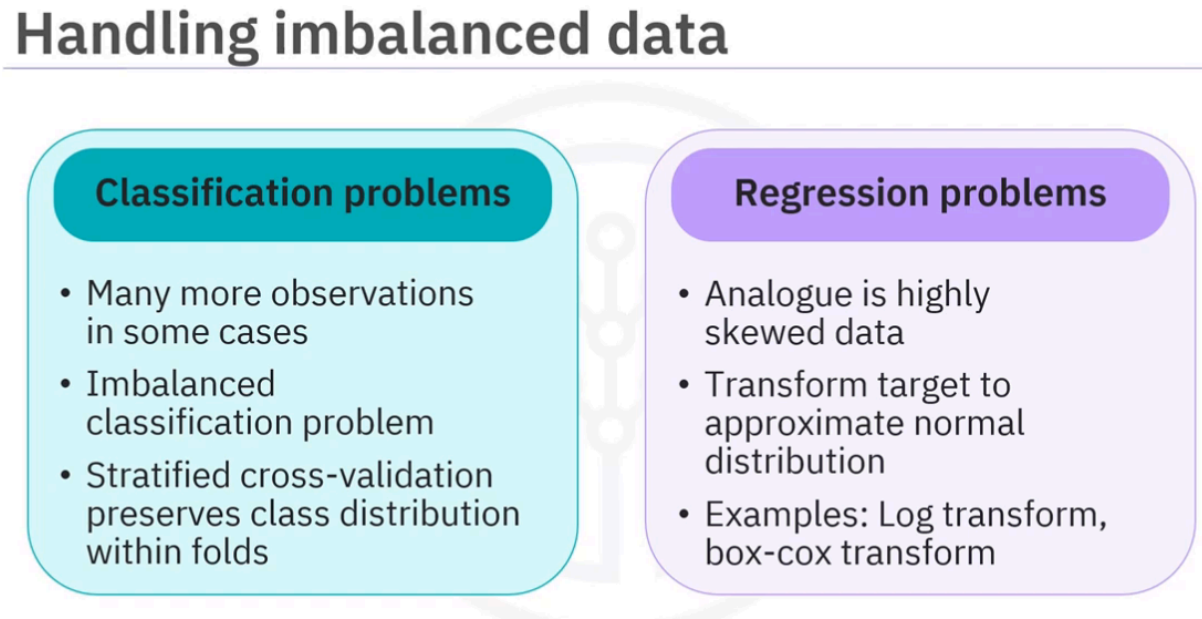


- Typical values: **K = 5 or 10**
- **Advantages:**
    - Every data point is used for both training & validation.
    - Reduces overfitting.

- Maximises data utilisation.
- More stable and reliable evaluation.



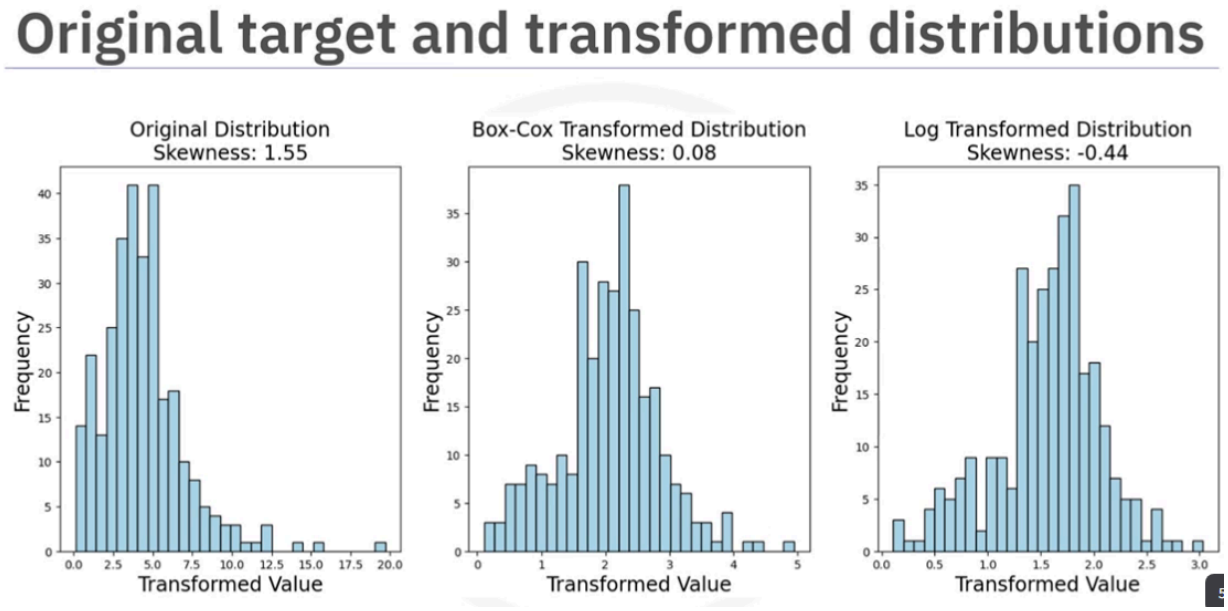
📊 6. Stratified Cross-Validation



- Used in **classification problems** with **imbalanced classes**.

- Ensures **class distribution** in each fold matches the overall dataset.
- Prevents **bias** in evaluation.

📉 7. Handling Skewed Target Variables (Regression)



- **Problem:** The Target variable is not normally distributed.
- **Solution:** Apply **transformations** to reduce skewness.
    - **Log transformation**
    - **Box-Cox transformation**
- Results in smoother distributions → better model fit.

🪜 8. Example Process

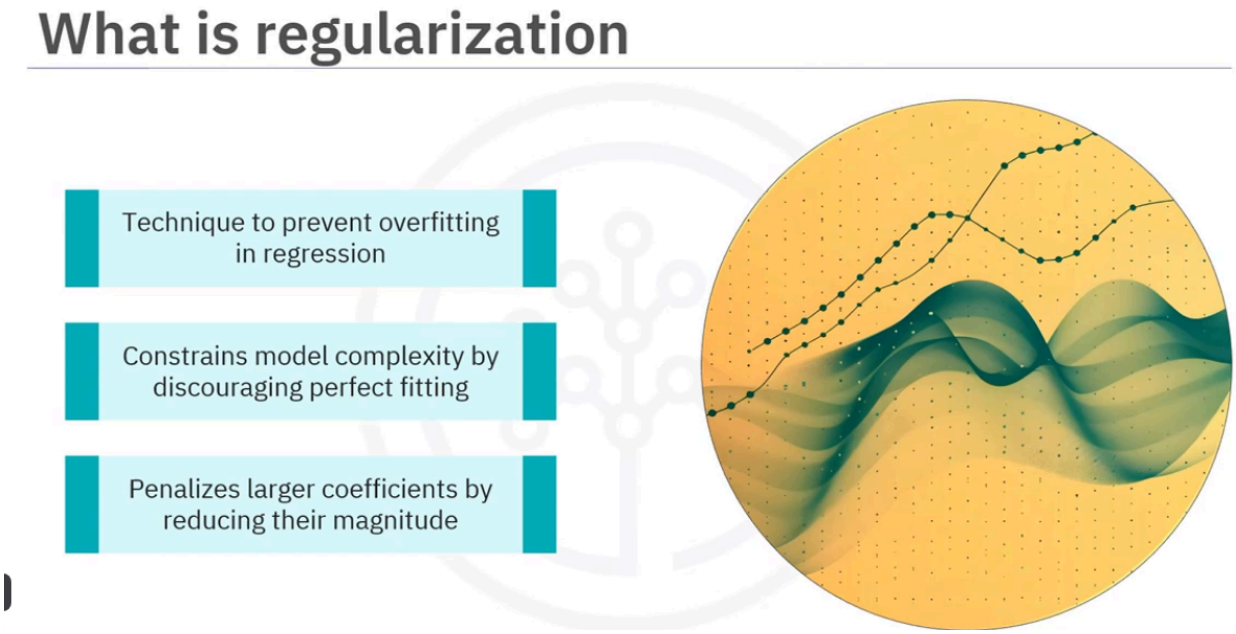
1. Split data → Training, Validation, Test
2. Use **K-Fold CV** for tuning hyperparameters.
3. Evaluate using **Validation folds**.
4. Select best hyperparameters.
5. Finally, test on **unseen Test data**.

🏁 9. Key Takeaways

- Model validation helps **prevent overfitting** and improves **generalization**.
- **Data Snooping** = using test data during tuning → must avoid.
- Use **Training**, **Validation**, and **Test** sets properly.
- **K-Fold CV** ensures balanced, fair, and efficient evaluation.
- **Stratified CV** prevents bias in classification.
- **Transforms** (log, Box-Cox) fix skewness in regression targets.

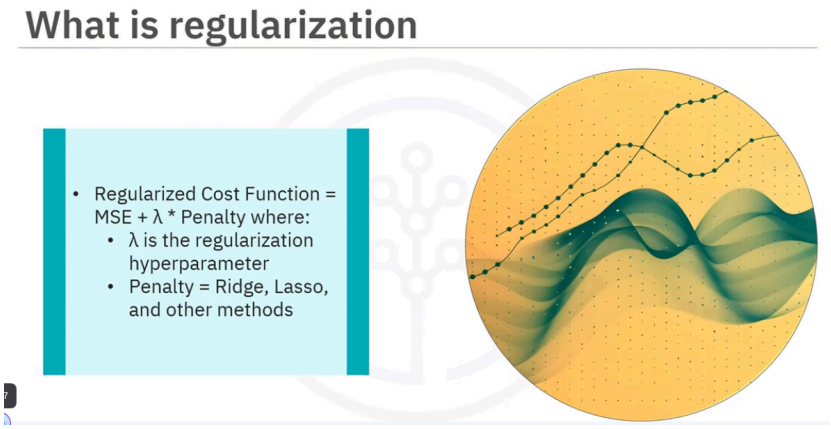
🧠 Regularisation in Linear Regression – Complete Notes

- **Definition**



Regularization is a **technique used to prevent overfitting** in regression models.
It works by **adding a penalty term** to the cost function to control the size of model coefficients (weights).

⚙️ Why Regularization?



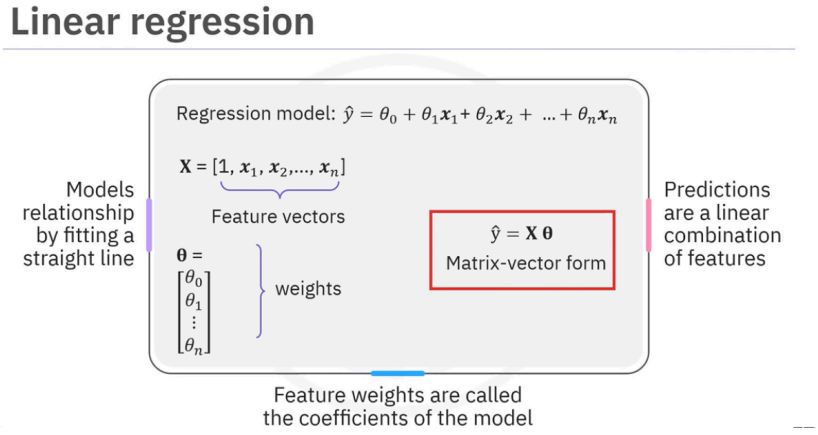
- Prevents the model from learning noise in training data.
- Keeps coefficients small and stable.
- Improves **generalization** on unseen data.
- Helps handle **multicollinearity** and **irrelevant features**.

🔸 Modified Cost Function

[
\text{Regularized Cost} = \text{MSE} + \lambda \times \text{Penalty Term}
]

Where:

- **MSE** = Mean Squared Error (original loss function).
- **λ (lambda)** = Regularization parameter (controls penalty strength).
- **Penalty term** = Measures coefficient size.
- If λ = 0 → behaves like **Linear Regression**.
- If λ is large → more shrinkage (simpler model).



Ridge and lasso regression

Ridge regression adds an L_2 penalty: $\lambda\|\theta\|_2 = \lambda \sum_{i=1}^{N} \theta_i^2$

Lasso regression adds an L_1 penalty: $\lambda\|\theta\|_1 = \lambda \sum_{i=1}^{N} |\theta_i|$

🔹 Types of Regression Models

Type	Penalty	Equation	Key Feature	Effect
Linear Regression (OLS)	None	MSE only	No shrinkage	Can overfit on noisy data
Ridge Regression	L2 = $\Sigma(\theta_i^2)$	MSE + $\lambda\Sigma(\theta_i^2)$	Shrinks all coefficients	Reduces variance, keeps all features
Lasso Regression	L1 = Σ	θ_i		MSE + $\lambda\Sigma$

🧮 Mathematical Representation

[
\hat{y} = \theta_0 + \theta_1x_1 + \theta_2x_2 + ... + \theta_nx_n
]

Where:

- **θ_0** = bias (intercept)

- $\theta_1, \theta_2 \ldots \theta_n$ = coefficients (weights)
- **X** = feature matrix

🔹 Sparse vs Non-Sparse Coefficients

- **Sparse coefficients** → only a few features have significant impact.
- **Non-sparse coefficients** → many features contribute meaningfully.

📊 Performance Comparisons (Based on SNR and Sparsity)

Scenario	Best Method	Observation
Sparse + High SNR	**Lasso**	Accurately finds zero coefficients
Sparse + Low SNR	**Lasso**	Handles noise, best feature selector
Non-Sparse + High SNR	**All perform well**	Ridge slightly less accurate
Non-Sparse + Low SNR	**Ridge**	Performs better on noisy data

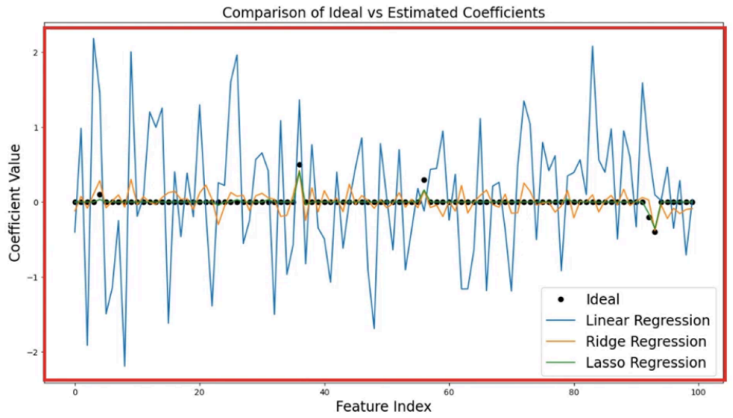
🔹 Performance Insights

- **Linear Regression** → Overfits, sensitive to noise and outliers.
- **Ridge Regression** → Smooths coefficients, handles multicollinearity.

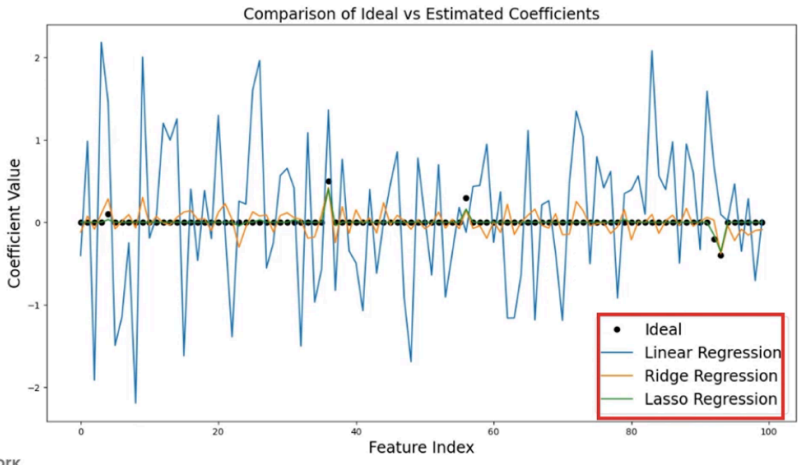
- **Lasso Regression** → Shrinks some coefficients to **zero**, ideal for **feature selection**.
- **MSE for Lasso** is about **30× lower** than Ridge or Linear regression in experiments.

📈 Model Behavior

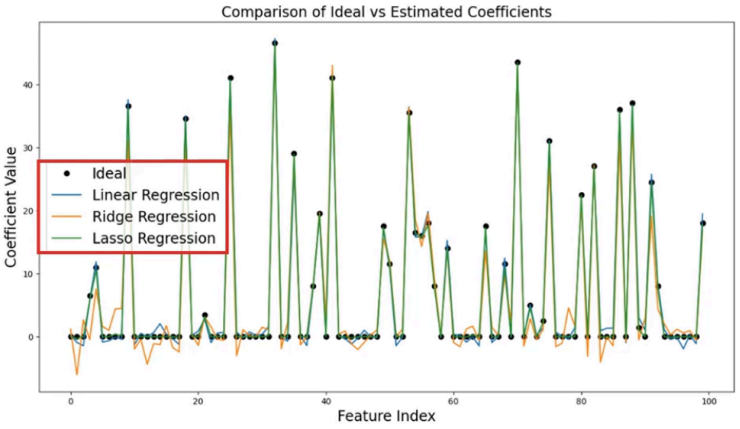
Sparse coefficients, low SNR



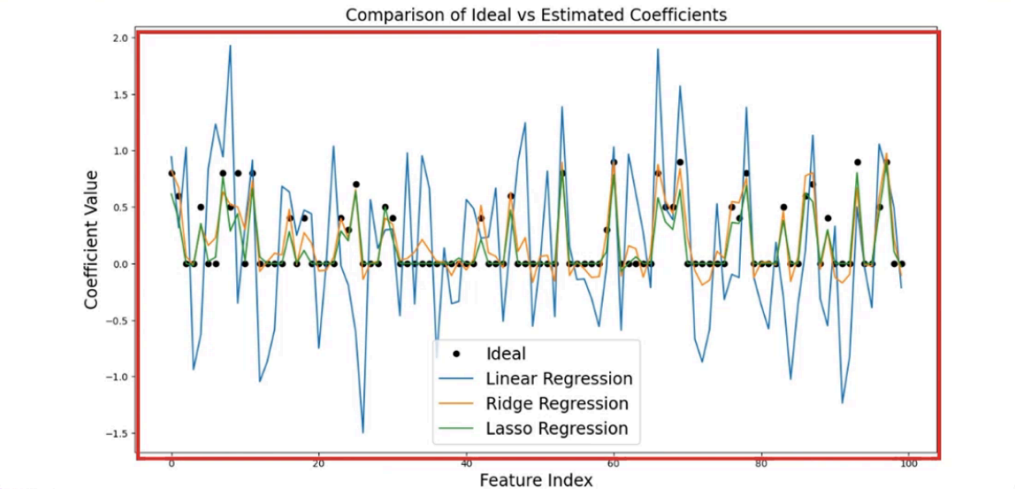
Sparse coefficients, low SNR



Non-sparse coefficients, high SNR

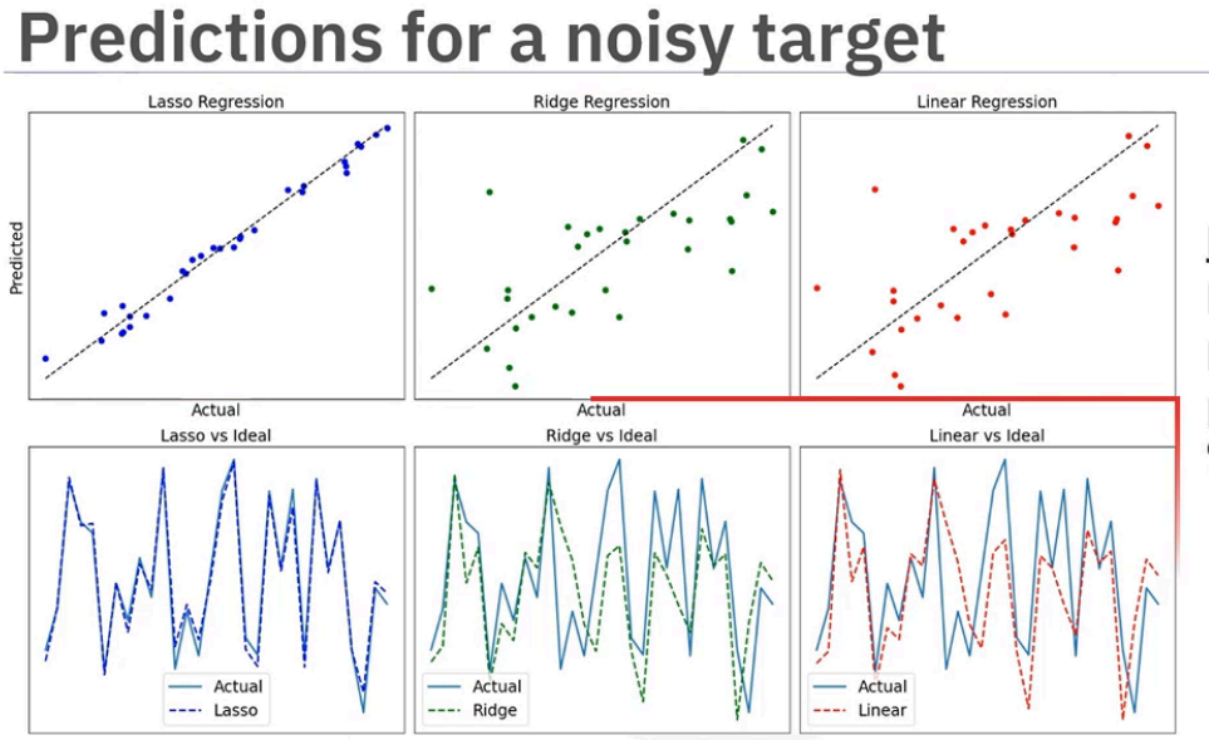


Non-sparse coefficients, low SNR



- **High SNR:**
    - All models perform well.
    - Ridge shows slight error, Lasso finds zero coefficients perfectly.
- **Low SNR:**
    - Linear Regression → overestimates, unstable.
    - Ridge → stable but keeps all features.
    - Lasso → finds zero coefficients, robust to noise.

🔹 Choosing the Right Model



Goal	Recommended Method
Avoid overfitting (general)	Ridge
Feature selection (sparse data)	Lasso
Combine both (balanced)	Elastic Net (L1 + L2)

Regularization performances

Linear Regression	Sparse Coefficients		Non-sparse Coefficients	
	High SNR	Low SNR	High SNR	Low SNR
Regular	****	*	****	*
Ridge	***	***	***	***
Lasso	*****	****	*****	****

⚖️ Key Takeaways

- Regularization adds a **penalty term** to control coefficient size.
- **λ (lambda)** decides how strongly coefficients are penalized.
- **Ridge (L2)**: Shrinks coefficients → no zeros.
- **Lasso (L1)**: Shrinks coefficients → some become zero.
- **Regularization = less overfitting + better generalization.**

🧩 In Summary

Aspect	Linear	Ridge	Lasso
Penalty	None	L2	L1
Coefficient Shrinkage	No	Yes	Yes (some = 0)

Feature Selection	❌	❌	✅
Handles Noise	Poorly	Good	Excellent
Best For	Clean data	Noisy data	Sparse features

📘 Regularization in Linear Regression — Notes with Code

🎯 Objectives

After this topic, you'll be able to:

- Implement, evaluate, and compare **Ordinary**, **Ridge**, and **Lasso** regression.
- Analyze the effect of regularization on noisy and outlier data.
- Use **Lasso** for **feature selection** in multiple regression.

⚙️ Step 1: Install & Import Libraries

```
!pip install numpy pandas scikit-learn matplotlib
```

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression, Ridge, Lasso

from sklearn.metrics import explained_variance_score, mean_absolute_error,
mean_squared_error, r2_score
```

🧱 Step 2: Define Evaluation Function

```
def regression_results(y_true, y_pred, model_name):
    ev = explained_variance_score(y_true, y_pred)
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)

    print(f"Evaluation metrics for {model_name} Linear Regression:")
    print(f"Explained Variance: {ev:.4f}")
    print(f"R² Score: {r2:.4f}")
    print(f"MAE: {mae:.4f}")
    print(f"MSE: {mse:.4f}")
    print(f"RMSE: {np.sqrt(mse):.4f}\n")
```

🧠 Step 3: Generate Synthetic Data

We create a dataset:
[
y = 4 + 3X + \text{noise}
]

```
np.random.seed(42)
noise = 1
X = 2 * np.random.rand(1000, 1)
y = 4 + 3 * X + noise * np.random.randn(1000, 1)
y_ideal = 4 + 3 * X
```

➕ Add Outliers

```
y_outlier = pd.Series(y.reshape(-1).copy())

threshold = 1.5

outlier_indices = np.where(X.flatten() > threshold)[0]

selected_indices = np.random.choice(outlier_indices, 5, replace=False)

y_outlier[selected_indices] += np.random.uniform(50, 100, 5)
```

📊 Step 4: Visualize Data

With Outliers

```
plt.figure(figsize=(12,6))

plt.scatter(X, y_outlier, alpha=0.4, label="Data with Outliers")

plt.plot(X, y_ideal, color='g', lw=3, label="Ideal Fit")

plt.xlabel("Feature (X)")

plt.ylabel("Target (y)")

plt.legend()

plt.show()
```

Without Outliers

```
plt.figure(figsize=(12,6))

plt.scatter(X, y, alpha=0.4, label="Data without Outliers")

plt.plot(X, y_ideal, color='g', lw=3, label="Ideal Fit")

plt.legend()
```

```
plt.show()
```

🧩 Step 5: Fit Models on Outlier Data

```
lin_reg = LinearRegression().fit(X, y_outlier)

ridge_reg = Ridge(alpha=1).fit(X, y_outlier)

lasso_reg = Lasso(alpha=0.2).fit(X, y_outlier)


y_pred_lin = lin_reg.predict(X)

y_pred_ridge = ridge_reg.predict(X)

y_pred_lasso = lasso_reg.predict(X)
```

Evaluation

```
regression_results(y, y_pred_lin, 'Ordinary')

regression_results(y, y_pred_ridge, 'Ridge')

regression_results(y, y_pred_lasso, 'Lasso')
```

Visualization

```
plt.figure(figsize=(12,6))

plt.scatter(X, y, alpha=0.4, label="Actual Data")

plt.plot(X, y_ideal, color='k', lw=2, label="Ideal Line")

plt.plot(X, y_pred_lin, lw=3, label="Linear")

plt.plot(X, y_pred_ridge, '--', lw=2, label="Ridge")

plt.plot(X, y_pred_lasso, lw=2, label="Lasso")
```

```
plt.legend()

plt.show()
```

Insight:

- **Lasso** resists the influence of outliers better than Ridge and Ordinary.
- Ordinary and Ridge both are pulled up by extreme points.

🧱 Step 6: Fit Models Without Outliers

Repeat the process using `y` instead of `y_outlier`.

```
lin_reg.fit(X, y)

ridge_reg.fit(X, y)

lasso_reg.fit(X, y)


y_pred_lin = lin_reg.predict(X)

y_pred_ridge = ridge_reg.predict(X)

y_pred_lasso = lasso_reg.predict(X)


regression_results(y, y_pred_lin, 'Ordinary')

regression_results(y, y_pred_ridge, 'Ridge')

regression_results(y, y_pred_lasso, 'Lasso')
```

🧠 Step 7: Multiple Regression with Feature Selection

Generate High-Dimensional Data

```
from sklearn.datasets import make_regression


X, y, ideal_coef = make_regression(
    n_samples=100, n_features=100, n_informative=10,
    noise=10, random_state=42, coef=True
)


ideal_predictions = X @ ideal_coef


X_train, X_test, y_train, y_test, ideal_train, ideal_test = train_test_split(
    X, y, ideal_predictions, test_size=0.3, random_state=42
)
```

Train All Models

```
lasso = Lasso(alpha=0.1)

ridge = Ridge(alpha=1.0)

linear = LinearRegression()


for model, name in zip([linear, ridge, lasso], ["Ordinary", "Ridge", "Lasso"]):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    regression_results(y_test, y_pred, name)
```

Result:

- Lasso achieves much higher R^2 (~0.98), meaning better generalization and feature selection.

📊 Step 8: Compare Coefficients

```
plt.figure(figsize=(12,6))

plt.bar(range(100), linear.coef_, label="Linear", alpha=0.6)

plt.bar(range(100), ridge.coef_, label="Ridge", alpha=0.6)

plt.bar(range(100), lasso.coef_, label="Lasso", alpha=0.6)

plt.plot(range(100), ideal_coef, 'k--', label="Ideal")

plt.legend()

plt.title("Comparison of Coefficients")

plt.show()
```

🔍 **Observation:**

- **Lasso** zeros out many coefficients → performs **feature selection**.
- **Ridge** reduces large weights but doesn't remove features.

🧭 Step 9: Feature Selection Using Lasso

```
threshold = 5

feature_importance_df = pd.DataFrame({

    'Lasso Coefficient': lasso.coef_,

    'Ideal Coefficient': ideal_coef
```

```
})

feature_importance_df['Feature Selected'] = feature_importance_df['Lasso Coefficient'].abs() > threshold

print(feature_importance_df[feature_importance_df['Feature Selected']])
```

Result:
Lasso successfully identified ~9 out of 10 truly informative features.

📈 Step 10: Refit Models Using Selected Features

```
important_features = feature_importance_df[feature_importance_df['Feature Selected']].index

X_filtered = X[:, important_features]


X_train, X_test, y_train, y_test = train_test_split(X_filtered, y, test_size=0.3, random_state=42)


for model, name in zip([LinearRegression(), Ridge(alpha=1), Lasso(alpha=0.1)], ["Ordinary", "Ridge", "Lasso"]):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    regression_results(y_test, y_pred, name)
```

🧩 Key Takeaways

Method	Regularization Type	Effect	Use Case

Ordinary	None	May overfit noisy data	Simple regression
Ridge (L2)	Penalizes large coefficients	Handles multicollinearity well	When all features important
Lasso (L1)	Shrinks and zeroes coefficients	Performs feature selection	When you need sparse model

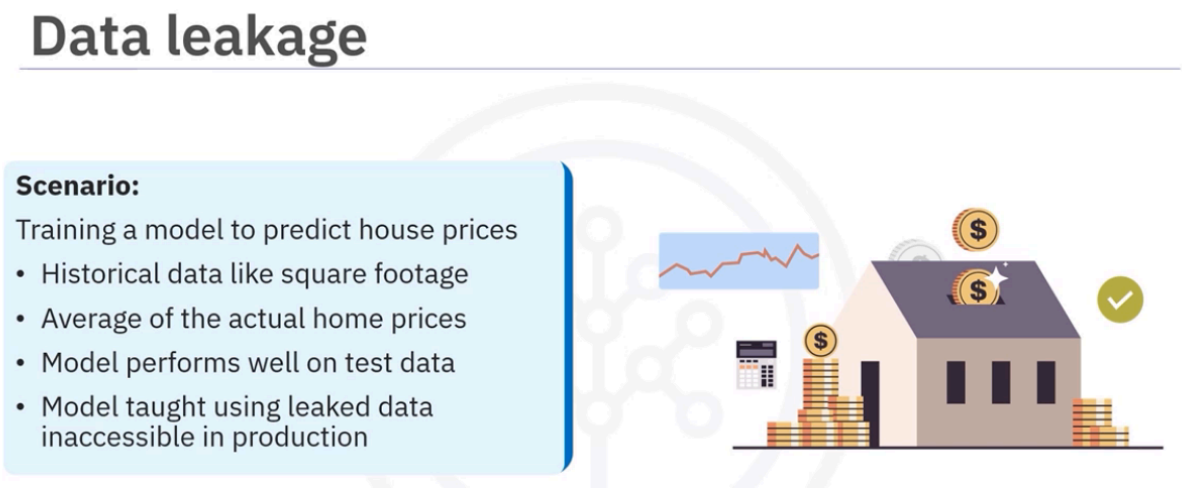
---

✅ Summary

- **Regularization** adds penalty to control overfitting.
- **Ridge** keeps all features but with smaller weights.
- **Lasso** eliminates weak features (sparse solution).
- **Lasso** performs best when only a few features are truly informative.

🧠 Data Leakage and Other Pitfalls — Complete Notes

🔹 What is Data Leakage?



Data leakage occurs when **training data includes information** that would **not be available in real-world scenarios** (i.e., after deployment).
➡️ It makes the model look accurate during training and validation but fail in real use.

⚙️ Example:

Predicting house prices using a feature that includes the **average of all home prices** in the dataset.
→ The model “learns” from future data → performs unrealistically well during testing → fails in production.

🔸 Causes of Data Leakage

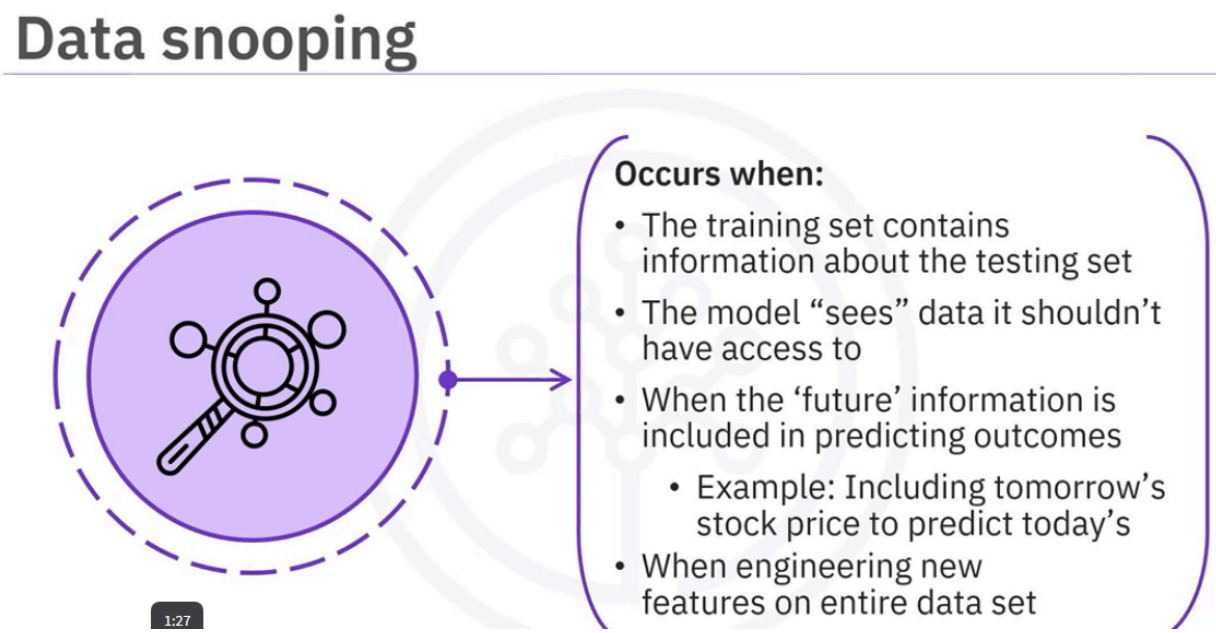
1. **Future information** used when predicting current/future outcomes.
   Example: Using tomorrow’s stock price to predict today’s.

2. **Feature engineering on the entire dataset** (instead of training set only).

3. **Overlapping or contaminated datasets** (train/test not properly separated).

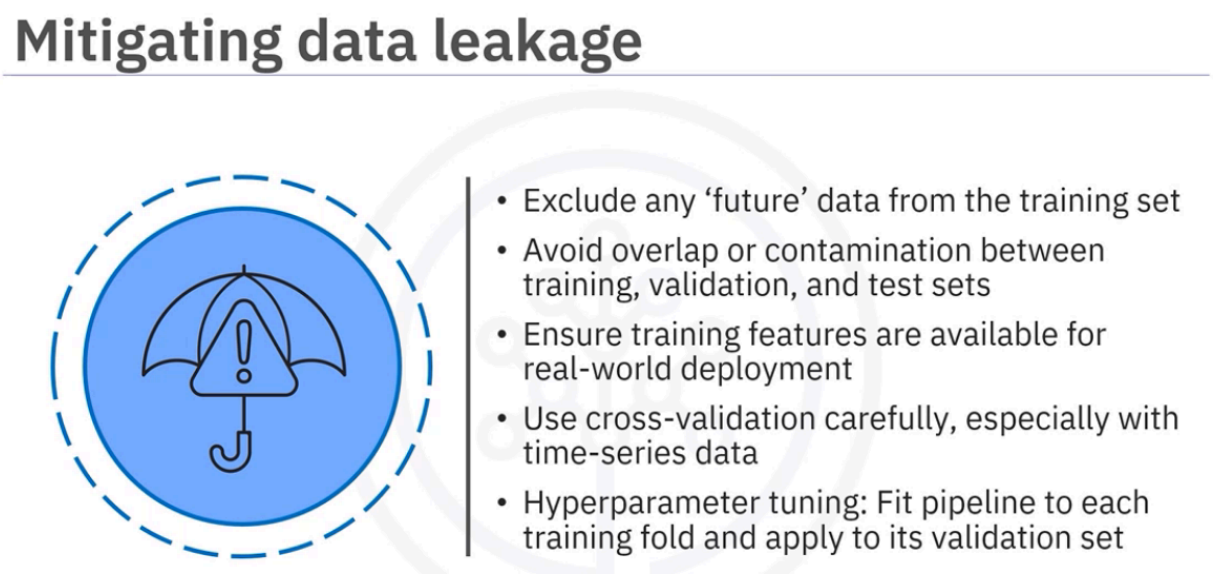
4. **Improper cross-validation**, especially in **time-series data**.

🧩 Data Snooping



A form of data leakage where **the model gains information about the test set** during training (directly or indirectly).

✅ How to Mitigate Data Leakage



Mitigation Strategy	Explanation
🚫 Avoid global features	Don't use averages or statistics derived from the full dataset.
🧱 Proper data splitting	Ensure clear separation between **training**, **validation**, and **test** data.
🕑 Handle time-based data carefully	Use **time-series split**, not random split. Train on past → test on future.
🔄 Correct cross-validation	Each fold should train and validate independently.
⚙️ Use pipelines correctly	Fit the **pipeline on each training fold** before validation (prevents leakage in scaling, PCA, etc.).

- **Pipeline Example (Python)**

Handling cross-validation data leakage

```
_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
stratify=y)
# Pipeline: StandardScaler, PCA, and KNN
pipeline = Pipeline([('scaler', StandardScaler()),
                     ('pca', PCA()),
                     ('knn', KNeighborsClassifier())
                     ])
# Hyperparameter search grid for PCA components, KNN neighbors
param_grid = {'pca__n_components': [2, 3],
              'knn__n_neighbors': [3, 5, 7]
             }
# Cross-validation method
cv = StratifiedKFold(n_splits=5, shuffle=True)
```

Handling cross-validation data leakage

```
# Get best parameters
best_model = GridSearchCV(pipeline, param_grid, cv=cv, scoring='accuracy')

# Fit GridSearchCV to training data
best_model.fit(X_train, y_train)

# Evaluate the best model on the test set
test_score = grid_search.score(X_test, y_test)

# Get the best parameters and score from grid search
print("Best Parameters:", best_model.best_params_)
print("Best Cross-Validation Accuracy:", best_model.best_score_)
# Evaluate the best model on the test setprint("Test Set Accuracy with
Best Parameters:", best_model.score(X_test, y_test))
Best Parameters: {'knn__n_neighbors': 3, 'pca__n_components': 3}
Best Cross-Validation Accuracy: 0.962
Test Set Accuracy with Best Parameters: 0.933
```

A pipeline might include:

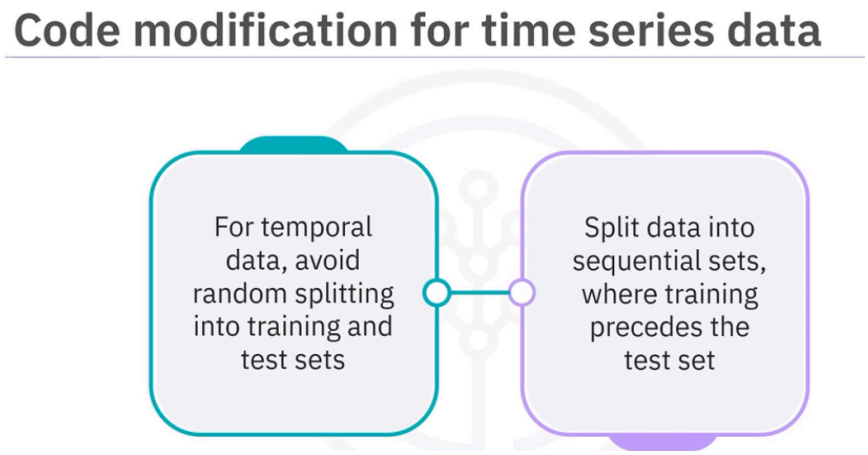
- **Scaler** → for normalization
- **PCA** → for dimensionality reduction

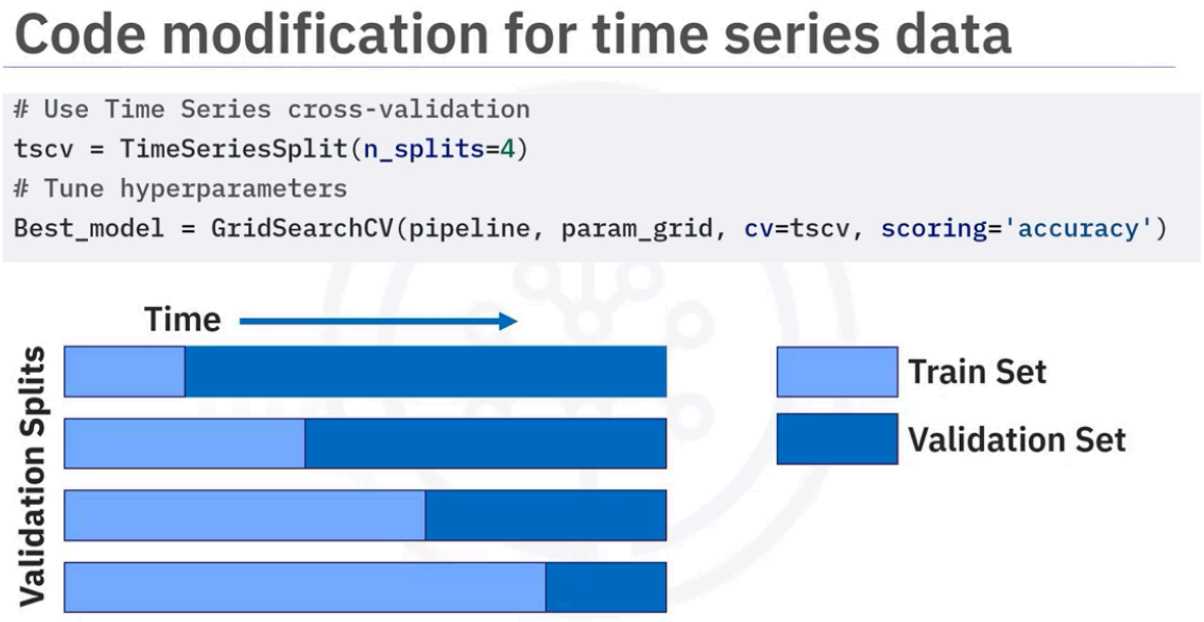
- **KNN Classifier** → for prediction

During **GridSearchCV**:

- Input the **pipeline** to the grid search.
- Ensures that **preprocessing is fitted only on training folds**, not on validation folds.

⏱️ Time-Series Cross-Validation





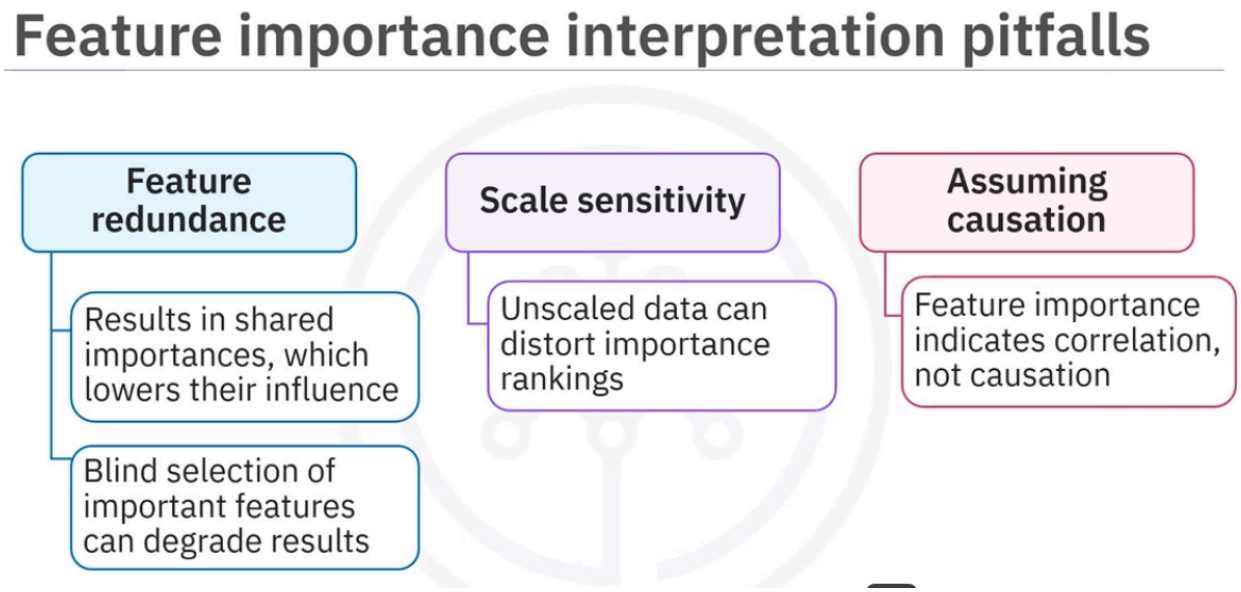
When data is **temporal**, use **TimeSeriesSplit** instead of random splitting.

- Each split uses **past data for training** and **future data for validation**.
- As training set expands, test set shrinks.
- Maintains **chronological order** → avoids leakage from future to past.

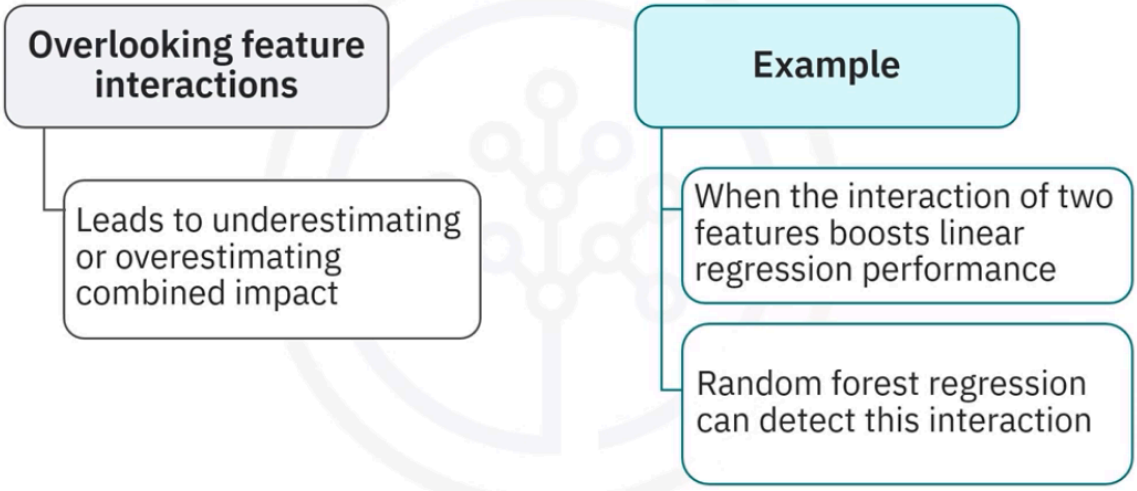
🔸 Code Adjustment for Time-Series CV

```
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV

tscv = TimeSeriesSplit(n_splits=4)

grid = GridSearchCV(pipeline, param_grid, cv=tscv)
```

🧠 Feature Importance & Interpretation Pitfalls



Feature importance interpretation pitfalls



🔹 Key Concepts

- **Feature importance** measures correlation, not causation.
- High importance ≠ causes target changes.
- Importance can be **shared** among correlated or redundant features.

⚠️ Common Pitfalls in Feature Importance

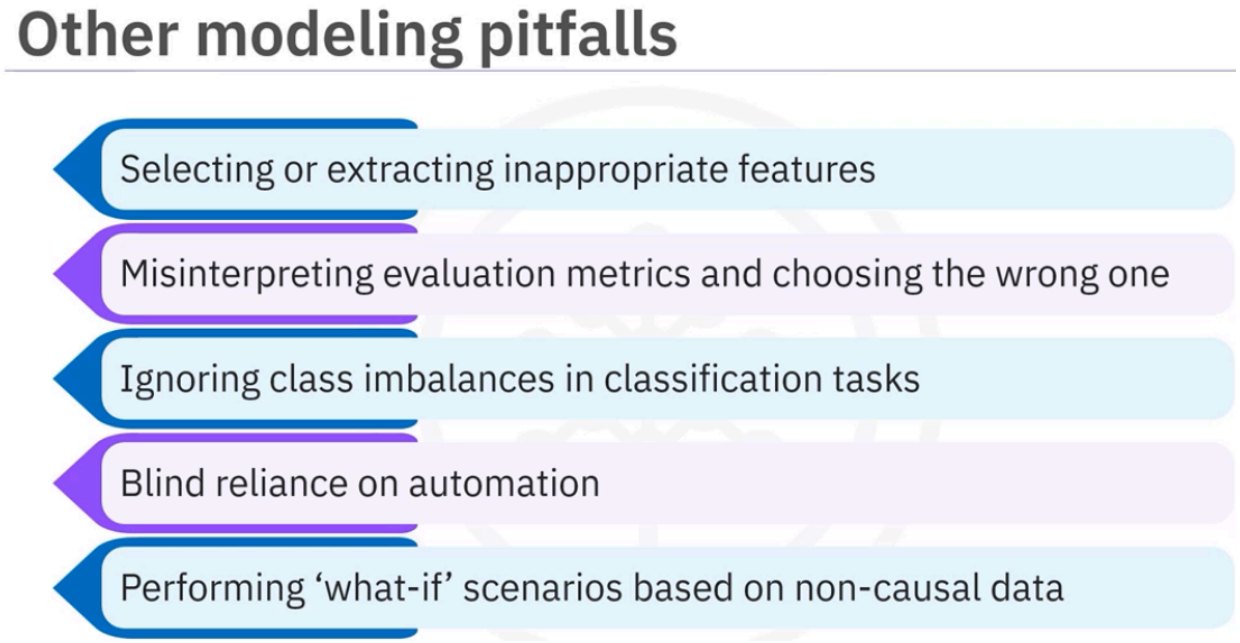
Pitfall	Description
🔁 **Feature Redundancy**	Highly correlated features share importance → appear less significant individually.
📏 **Scale Sensitivity**	Algorithms like Linear Regression are sensitive to unscaled data.

❌ **Causation Assumption**	Important features may correlate but not cause outcomes.
⚡ **Ignoring Feature Interactions**	Some models (like linear regression) can't detect interactions; nonlinear ones (like Random Forest) can.

📊 Example

- Two weak features → poor linear regression results.
- But their **interaction/product** may be powerful.
- Random Forest can capture this automatically; linear regression cannot.

⚠️ Other Common Modeling Pitfalls



Pitfall	Description
🔢 **No Feature Selection/Transformation**	Using raw data prevents finding the best model.
📉 **Wrong Evaluation Metric**	Choosing or interpreting metrics incorrectly can mislead results.
⚖️ **Ignoring Class Imbalance**	Causes bias toward majority classes in classification.
🤖 **Over-Reliance on AutoML**	Tools are useful, but you must still understand the data and model.
🧠 **Lack of Causal Features**	Without causal variables, "what-if" analyses may produce misleading results.

🧾 Summary

Concept	Key Points
Data Leakage	Training data includes unavailable real-world info → overestimation of performance
Data Snooping	Model indirectly learns from test data
Prevention	Proper splits, time-series handling, separate pipelines per fold

Feature Importance Issues	Correlation ≠ causation; redundant features share importance
Other Pitfalls	Wrong metrics, imbalance, ignoring feature scaling, blind AutoML use

---

🧠 Final Takeaways

- Data leakage = *future info leaks into training* → false high accuracy.
- Always separate preprocessing for train/test.
- Use **time-series split** for sequential data.
- Interpret feature importance with caution.
- Understand your data, don't rely blindly on tools.
- Ensure features have causal links to avoid misleading predictions.

Perfect 👍 Here's a **clean, complete, and easy-to-revise note with code examples** for your topic —
"Machine Learning Pipelines and GridSearchCV".
I've structured it for **clarity + learning + quick revision** 👇

🧠 Machine Learning Pipelines and GridSearchCV

📅 Date: October 13, 2025

🎯 Objectives

After completing this lab, you will be able to:

- Build and evaluate a Machine Learning **Pipeline**
- Implement **GridSearchCV** for hyperparameter tuning with cross-validation
- Optimize a **complex classification pipeline**
- Extract **feature importances** from a trained pipeline

📘 1. Introduction

In **Scikit-Learn**, a **Pipeline** allows you to combine preprocessing steps and model training into a single workflow.

✅ Advantages of Using Pipelines

- Keeps preprocessing and modeling steps consistent
- Prevents data leakage between training and test data
- Enables full model tuning with **GridSearchCV**
- Ensures reproducibility and cleaner code

⚙️ Pipeline Structure

```
Pipeline([
    ('step1_name', Transformer()),  # Must implement fit & transform
    ('step2_name', Transformer()),  # Optional
    ('model', Estimator())          # Must implement fit
])
```

You can tune parameters using:

stepname__parameter = value

Example:

imputer__strategy = 'median'

You can also **bypass** a step using `'passthrough'` or `None`.

🧩 2. Install Required Libraries

!pip install scikit-learn==1.6.0 matplotlib==3.9.3 seaborn==0.13.2

📚 3. Import Libraries

```
import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import Pipeline

from sklearn.metrics import confusion_matrix
```

🌸 4. Train a Model Using a Pipeline

4.1 Load the Iris Dataset

```
data = load_iris()

X, y = data.data, data.target

labels = data.target_names
```

4.2 Build the Pipeline

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),          # Step 1: Standardize data
    ('pca', PCA(n_components=2)),          # Step 2: Reduce dimensions
    ('knn', KNeighborsClassifier(n_neighbors=5))  # Step 3: KNN model
])
```

4.3 Split Data into Train/Test

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

4.4 Fit and Evaluate Model

```
pipeline.fit(X_train, y_train)

test_score = pipeline.score(X_test, y_test)

print(f"Accuracy: {test_score:.3f}")
```

📈 **Output:** `Accuracy: 0.900`

4.5 Predictions & Confusion Matrix

```
y_pred = pipeline.predict(X_test)

conf_matrix = confusion_matrix(y_test, y_pred)


plt.figure()

sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='d',

            xticklabels=labels, yticklabels=labels)

plt.title('Classification Pipeline Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.tight_layout()

plt.show()
```

🧩 **Error Observation:**

- 2 *Virginica* misclassified as *Versicolor*
- 1 *Versicolor* misclassified as *Virginica*

🔍 5. Hyperparameter Tuning with GridSearchCV

To avoid overfitting during tuning, use **cross-validation** instead of testing directly on the test set.

5.1 Define a New Pipeline

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA()),
    ('knn', KNeighborsClassifier())
])
```

5.2 Define Parameter Grid

```
param_grid = {
    'pca__n_components': [2, 3],
    'knn__n_neighbors': [3, 5, 7]
}
```

5.3 Cross-Validation Strategy

```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

5.4 Run GridSearchCV

```
best_model = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=cv,
    scoring='accuracy',
    verbose=2
)
```

5.5 Fit the Model

```
best_model.fit(X_train, y_train)
```

✅ **Cross-validation** runs for all parameter combinations.

5.6 Evaluate Best Model

```
test_score = best_model.score(X_test, y_test)
print(f"Best Model Accuracy: {test_score:.3f}")
```

📈 **Output:** `Best Model Accuracy: 0.933`

5.7 Display Best Parameters

```
print(best_model.best_params_)
```

🧩 **Output:**

```
{'knn__n_neighbors': 3, 'pca__n_components': 3}
```

5.8 Plot Confusion Matrix for Best Model

```
y_pred = best_model.predict(X_test)

conf_matrix = confusion_matrix(y_test, y_pred)


plt.figure()

sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='d',

            xticklabels=labels, yticklabels=labels)

plt.title('Optimized KNN Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.tight_layout()

plt.show()
```

📊 **Observation:** Only 2 errors — both *Virginica* misclassified as *Versicolor*.

🧠 6. Summary

Step	Purpose	Tool/Method
Data Scaling	Standardize features	`StandardScaler()`

Dimensionality Reduction	Reduce to fewer components	`PCA()`
Model	Classify using distance	`KNeighborsClassifier()`
Pipeline	Combine steps	`Pipeline()`
Model Tuning	Optimize hyperparameters	`GridSearchCV()`
Evaluation	Accuracy, Confusion Matrix	`score()`, `confusion_matrix()`

🏁 7. Key Takeaways

- **Pipeline** simplifies ML workflows
- **Cross-validation** avoids overfitting
- **GridSearchCV** can tune preprocessing + model together
- **Best practices:** always scale, tune, and validate

Cheat Sheet: Evaluating and Validating Machine Learning Models

Module 5 Summary and Highlights

Congratulations! You have completed this lesson. At this point in the course, you know:

- Supervised learning evaluation assesses a model's ability to predict outcomes for unseen data, often using a train/test split to estimate performance.
- Key metrics for classification evaluation include accuracy, confusion matrix, precision, recall, and the F1 score, which balances precision and recall.

- Regression model evaluation metrics include MAE, MSE, RMSE, R-squared, and explained variance to measure prediction accuracy.
- Unsupervised learning models are evaluated for pattern quality and consistency using metrics like Silhouette Score, Davies-Bouldin Index, and Adjusted Rand Index.
- Dimensionality reduction evaluation involves Explained Variance Ratio, Reconstruction Error, and Neighborhood Preservation to assess data structure retention.
- Model validation, including dividing data into training, validation, and test sets, helps prevent overfitting by tuning hyperparameters carefully.
- Cross-validation methods, especially K-fold and stratified cross-validation, support robust model validation without overfitting to test data.
- Regularization techniques, such as ridge (L2) and lasso (L1) regression, help prevent overfitting by adding penalty terms to linear regression models.
- Data leakage occurs when training data includes information unavailable in real-world data, which is preventable by separating data properly and mindful feature selection.
- Common modelling pitfalls include misinterpreting feature importance, ignoring class imbalance, and making causal inferences without sufficient evidence.”
- Feature importance assessments should consider redundancy, scale sensitivity, and avoid misinterpretation, as well as inappropriate assumptions about causation.