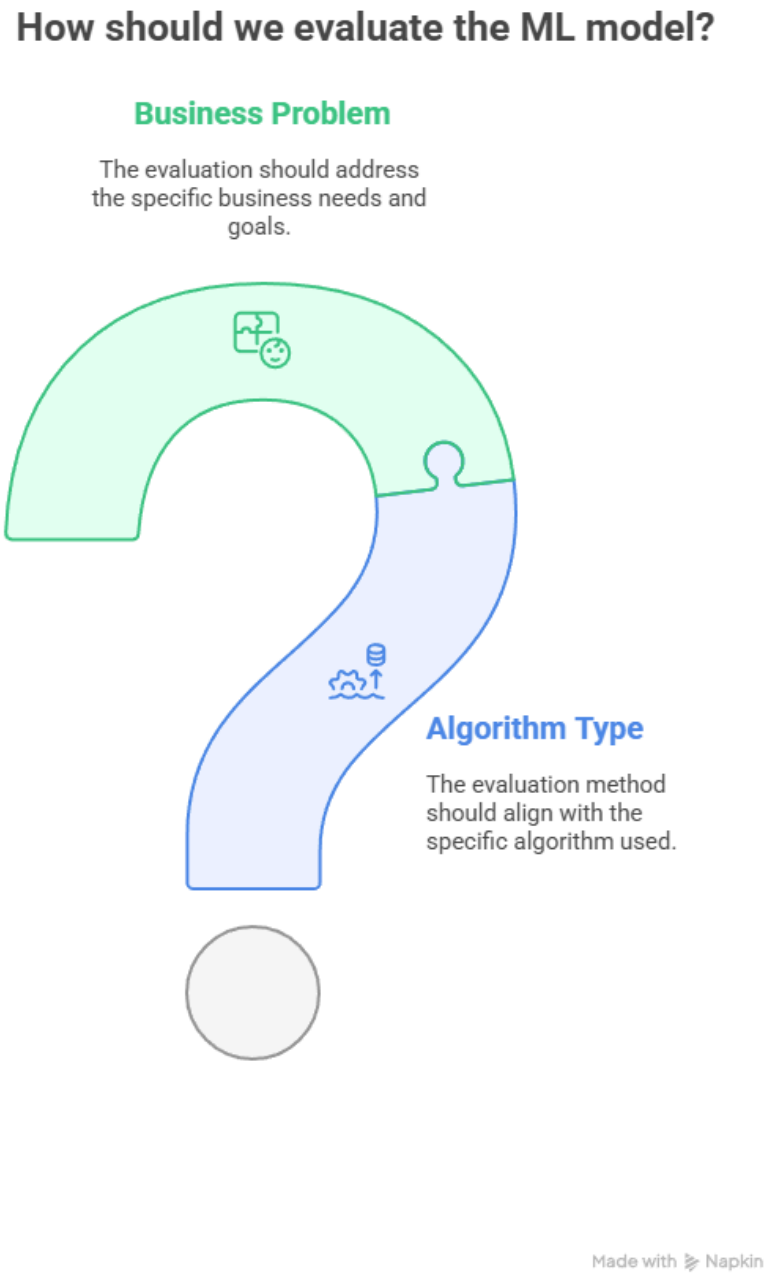
Evaluation Metrics in Machine Learning

When we build machine learning (ML) models, it's important to measure how well they're performing. However, **how we evaluate a model depends on two important factors:**

1. **The type of machine learning algorithm used**
2. **The kind of business problem we're solving**



1. Classification Models

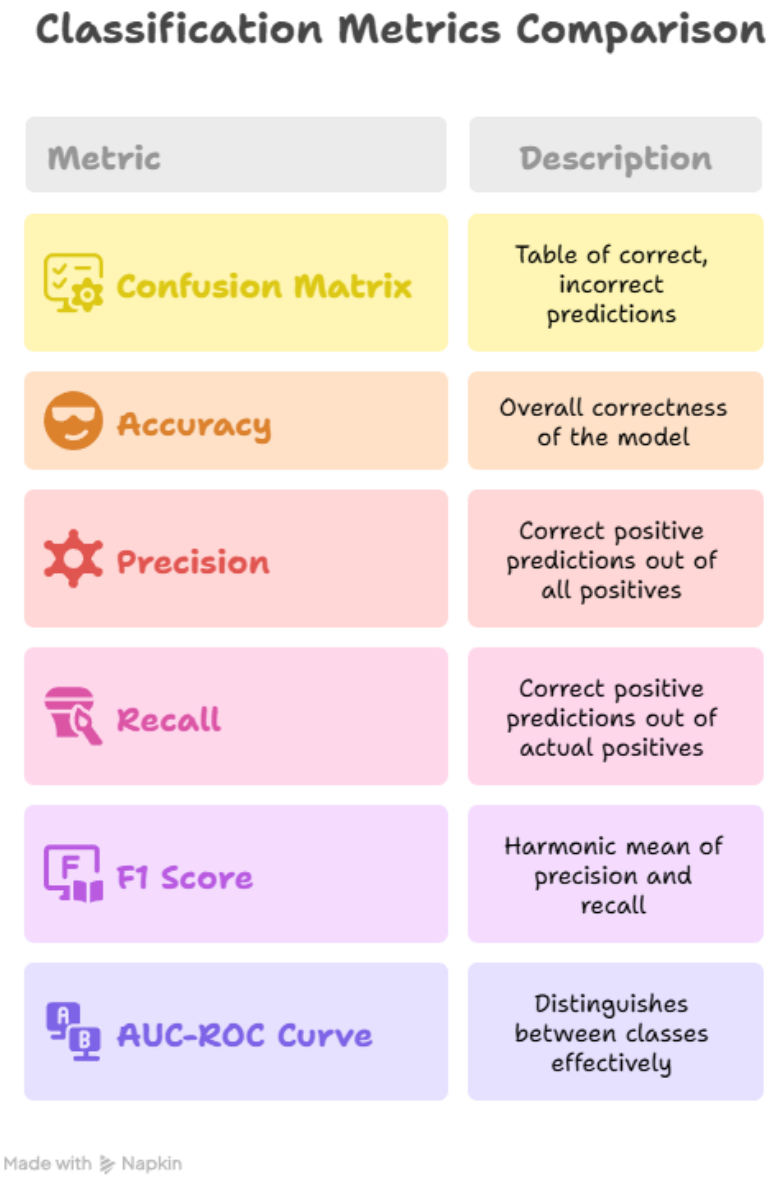
These models predict categories or labels. For example:

- Is this email spam or not?
- Will the customer buy or not?
- Is the transaction fraudulent or not?

Output Type: Discrete (e.g., 0 = No, 1 = Yes)

Common Metrics:

- **Confusion Matrix**: A table that shows correct and incorrect predictions.
- **Accuracy**: How often the model is right.
- **Precision**: Of all positive predictions, how many were correct?
- **Recall**: Of all actual positives, how many did the model catch?
- **F1 Score**: A balance between precision and recall.
- **AUC-ROC Curve**: Shows how well the model distinguishes between classes.



2. Regression Models

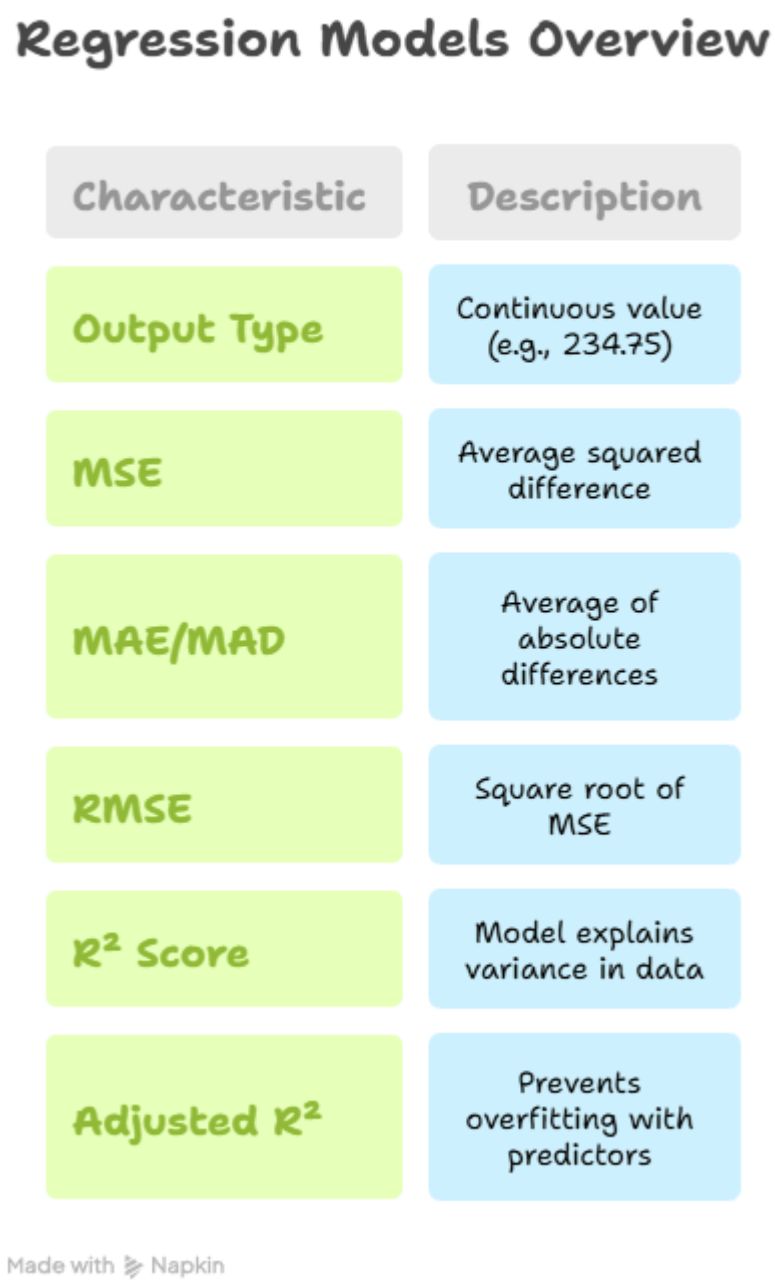
These models predict continuous numbers. For example:

- What will the house price be?
- How much will a customer spend next month?

Output Type: Continuous value (e.g., 234.75)

Common Metrics:

- **MSE (Mean Squared Error)**: Average squared difference between predicted and actual values.
- **MAE/MAD (Mean Absolute Error/Deviation)**: Average of absolute differences.
- **RMSE (Root Mean Squared Error)**: Square root of MSE for easier interpretation.
- **R^2 Score (Coefficient of Determination)**: How well the model explains the variance in the data.
- **Adjusted R^2**: Like R^2, but adjusts for the number of predictors to prevent overfitting.



3. Clustering Models

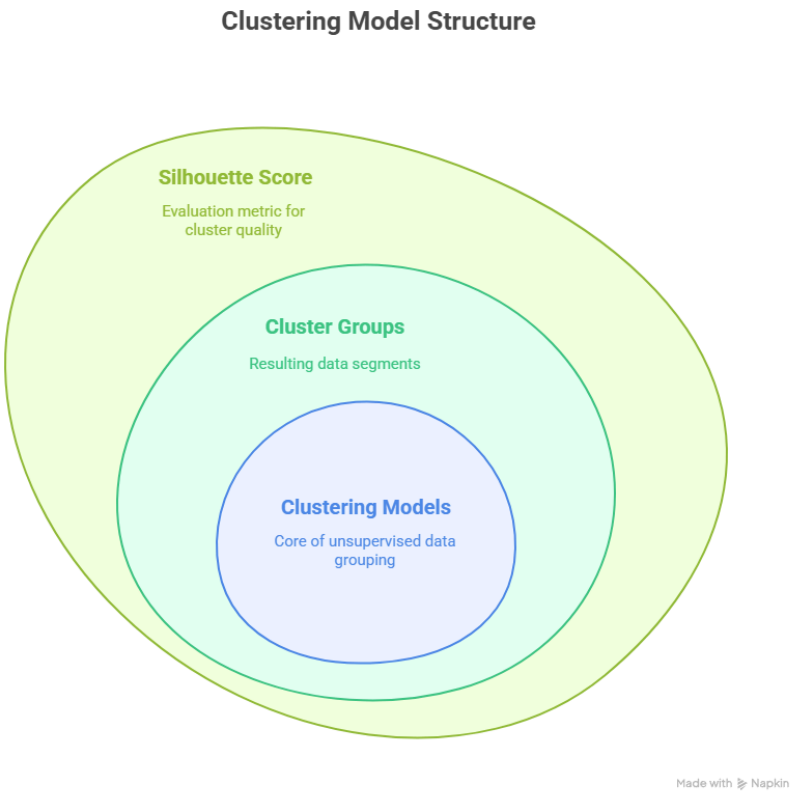
These models group data points without predefined labels. For example:

- Customer segmentation
- Grouping similar products

Output Type: Cluster groups (unsupervised)

Common Metric:

- **Silhouette Score:** Measures how well each data point fits into its cluster (higher is better).



Classification Models

These models predict categories or labels. For example:

- Is this email spam or not?
- Will the customer buy or not?
- Is the transaction fraudulent or not?

Output Type: Discrete (e.g., 0 = No, 1 = Yes)

Common Evaluation Metrics:

Confusion Matrix

A **confusion matrix** is a simple yet powerful tool that shows how well your model is predicting the actual classes.

It compares the predicted values with the actual values in a tabular format.

It's called an **"n × n matrix"**, where **n** is the number of classes in your target variable. For binary classification (0 or 1), it's a **2×2 matrix**:

	Predicted: No (0)	Predicted: Yes (1)
Actual: No (0)	True Negative (TN)	False Positive (FP)
Actual: Yes (1)	False Negative (FN)	True Positive (TP)

Let's Take a Simple Example:

Email Spam Detection

We built a binary classification model to predict whether an email is **spam (1)** or **not spam (0)**.

We tested the model on **10 emails** and got the following results:

Actual (True) Label	Predicted Label
1 (Spam)	1 (Spam)
0 (Not Spam)	0 (Not Spam)
1 (Spam)	0 (Not Spam)
1 (Spam)	1 (Spam)
0 (Not Spam)	0 (Not Spam)
0 (Not Spam)	1 (Spam)
1 (Spam)	1 (Spam)
0 (Not Spam)	0 (Not Spam)
1 (Spam)	0 (Not Spam)
0 (Not Spam)	0 (Not Spam)

Step 1: Build the Confusion Matrix

Let's count the outcomes:

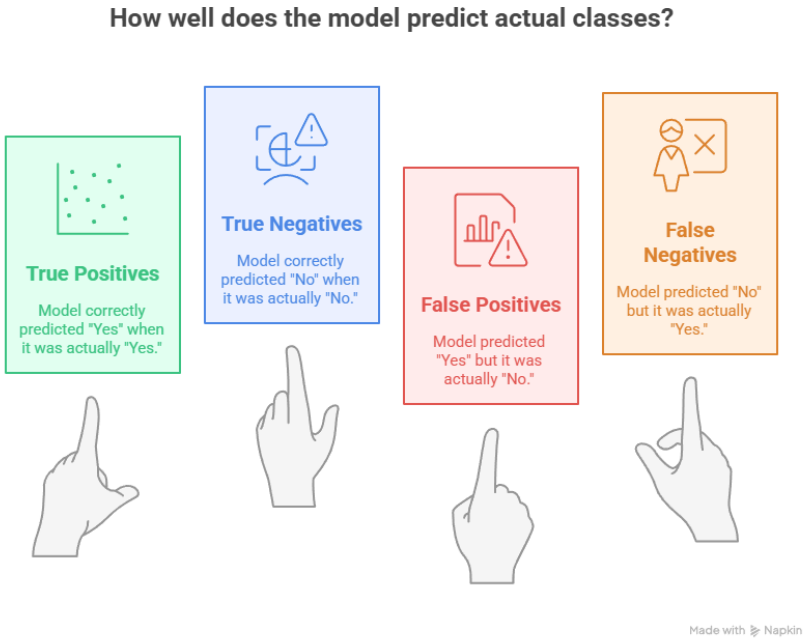
	Predicted: Spam (1)	Predicted: Not Spam (0)
Actual: Spam (1)	**3** (True Positive)	**2** (False Negative)
Actual: Not Spam (0)	**1** (False Positive)	**4** (True Negative)

Interpretation

Term	Meaning	Value	Explanation
TP (True Positive)	Model correctly predicted Spam as Spam	3	Good catch
TN (True Negative)	Model correctly predicted Not Spam as Not Spam	4	Accurate filtering
FP (False Positive)	Model predicted Spam, but it was Not Spam	1	Incorrectly marked a normal email as spam (type I error)
FN (False Negative)	Model predicted Not Spam, but it was actually Spam	2	Missed actual spam (type II error)

Quick Definitions:

- **True Positive (TP):** Model predicted “Yes” and it was correct.
- **True Negative (TN):** Model predicted “No” and it was correct.
- **False Positive (FP):** Model predicted “Yes” but it was actually “No.”
- **False Negative (FN):** Model predicted “No” but it was actually “Yes.”



Confusion Matrices for KNN, Naive Bayes & Decision Tree

Actual Labels (`y_test`):

Sample #	Actual (y_test)
1	0
2	1
3	1
4	0
5	0
6	0
7	0
8	1

Predictions by Models:

Sample #	KNN Prediction	Naive Bayes Prediction	Decision Tree Prediction
1	0	0	0
2	1	1	0
3	0	1	1
4	1	1	0
5	1	0	0
6	0	0	0
7	0	0	0
8	1	0	1

Confusion Matrices:

KNN

	Predicted: 0	Predicted: 1
Actual: 0	3 (TN)	2 (FP)
Actual: 1	1 (FN)	2 (TP)

Naive Bayes

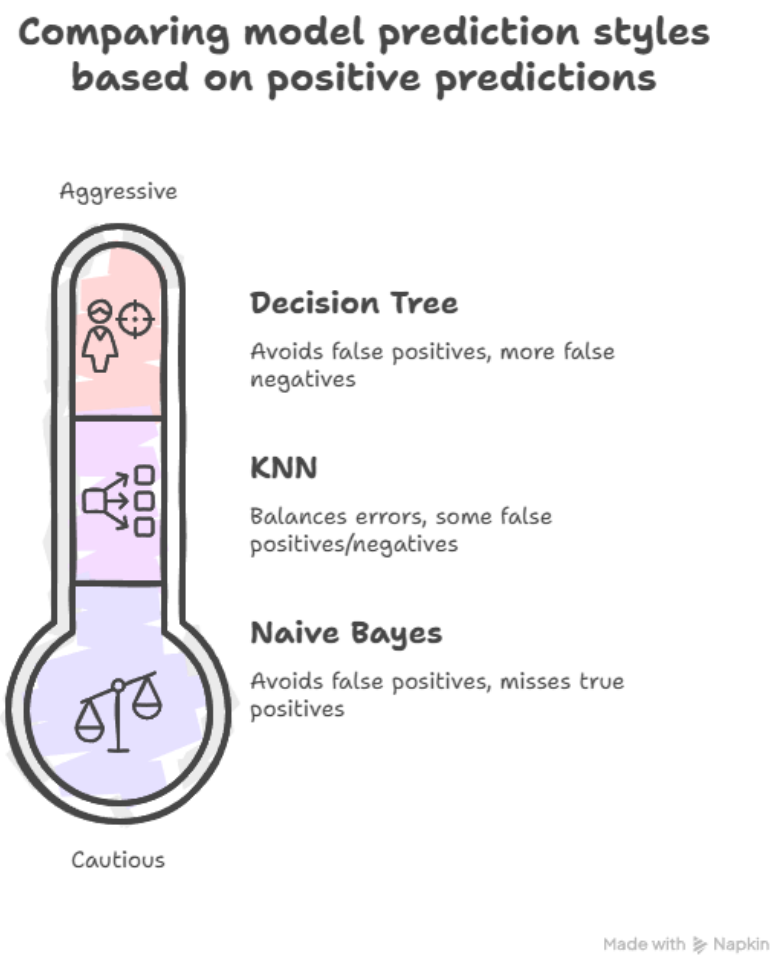
	Predicted: 0	Predicted: 1
Actual: 0	4 (TN)	1 (FP)
Actual: 1	3 (FN)	0 (TP)

Decision Tree

	Predicted: 0	Predicted: 1
Actual: 0	5 (TN)	0 (FP)
Actual: 1	2 (FN)	1 (TP)

Summary:

- **KNN** predicted a good balance but made some false positive and false negative errors.
- **Naive Bayes** was more cautious predicting positives, resulting in zero true positives but fewer false positives.
- **Decision Tree** perfectly avoided false positives but missed more actual positives (higher false negatives).



Accuracy

Formula:

```
Accuracy = (TP + TN) / (TP + TN + FP + FN)
```

(= Correct predictions / Total predictions)

Accuracy tells you **how often the model is right overall.**

Good For:

Balanced datasets (i.e., the number of 0s and 1s in the target are approximately equal)

Example—Balanced Dataset:

Predicting Whether a Student Passes or Fails an Exam

- **1 = Pass, 0 = Fail**

You test your model on **12 students**, and the actual vs. predicted results are:

Student	Actual (A)	Predicted (P)
1	1	1
2	0	0
3	1	1
4	0	1
5	1	0
6	1	1
7	0	0
8	0	0
9	1	1
10	0	0
11	1	1
12	0	0

Step 1: Confusion Matrix Counts

	Predicted: 1 (Pass)	Predicted: 0 (Fail)
Actual: 1 (Pass)	TP = 5	FN = 1
Actual: 0 (Fail)	FP = 1	TN = 5

Step 2: Apply Accuracy Formula

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

$$\text{Accuracy} = \frac{5 + 5}{5 + 5 + 1 + 1} = \frac{10}{12} = 0.8333$$

Final Accuracy: 83.33%

Interpretation

- The model **correctly predicted 10 out of 12 outcomes.**
- So, it is **83.33% accurate.**
- Because the number of students who passed and failed is balanced (6 each), this is a **reliable metric** to judge the model's overall performance.

Conclusion: The model is performing well with high accuracy, making correct predictions most of the time.

Why Accuracy Can Be Misleading (Imbalanced Dataset):

Detecting Fraudulent Transactions

- You have **1,000** transactions.
- Only **5** are actually **fraudulent (1)**—the rest **995 are normal (0).**
- Your model predicts **"Not Fraud"** (0) for **every** transaction.

Confusion Matrix

Let's build the confusion matrix based on what the model predicted:

	Predicted: Fraud (1)	Predicted: Not Fraud (0)
Actual: Fraud (1)	0 (TP)	5 (FN)
Actual: Not Fraud (0)	0 (FP)	995 (TN)

Step-by-Step Accuracy Calculation

Formula:

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

Substitute values:

$$\text{Accuracy} = \frac{0 + 995}{0 + 995 + 0 + 5} = \frac{995}{1000}$$

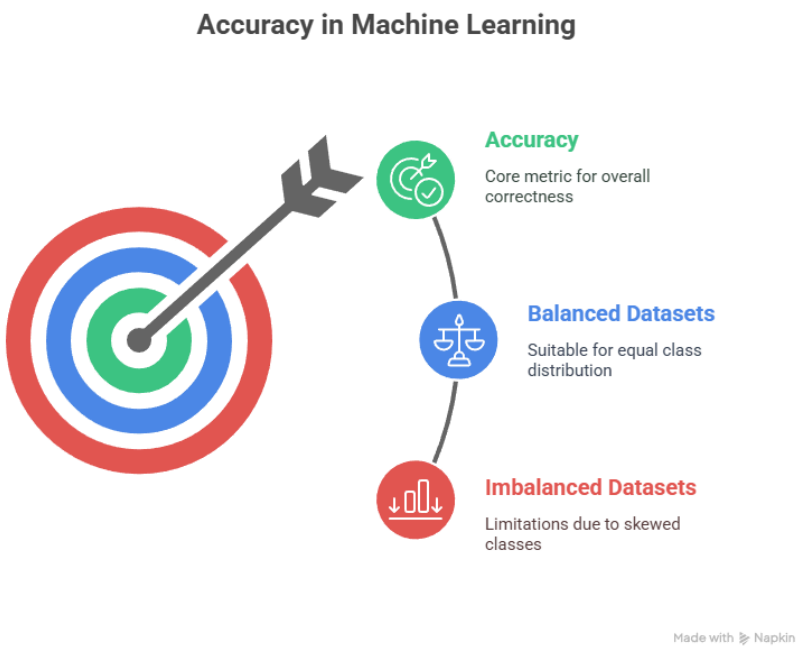
Final Accuracy: 99.5%

Interpretation

- **99.5% accuracy** seems **very high**—but it's misleading.
- The model **never predicted any fraud**—it **missed all 5 fraud cases** (False Negatives).
- It only did well because the dataset is **heavily imbalanced** (almost all transactions are non-fraud).
- In reality, this model is **useless for fraud detection**.

Why Accuracy Is Misleading Here

- The dataset is **imbalanced:** only 0.5% are fraud.
- The model can be lazy and predict "Not Fraud" always, and still get **very high accuracy**.
- But it's **failing at the task it's meant for**—identifying fraud.



Precision

Formula:

```
Precision = TP / (TP + FP)
```

(= Of all predicted "Yes", how many were actually "Yes"?)

Use When:

False Positives are costly

Example: Approving a loan to someone who can't repay

Example:

Loan Approval Prediction

You built a model to predict whether a person should be **approved for a loan.**

- **1 = Approved**
- **0 = Not Approved**

Your model makes predictions on 10 loan applicants.

Applicant	Actual (A)	Predicted (P)
1	1	1
2	0	1
3	1	1
4	0	0
5	1	1
6	1	0
7	0	0
8	0	0
9	1	1
10	0	1

Step 1: Confusion Matrix Counts

	Predicted: 1 (Approved)	Predicted: 0 (Not Approved)
Actual: 1 (Approved)	TP = 4	FN = 1
Actual: 0 (Not Approved)	FP = 2	TN = 3

Step 2: Precision Formula

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

Final Precision: 66.67%

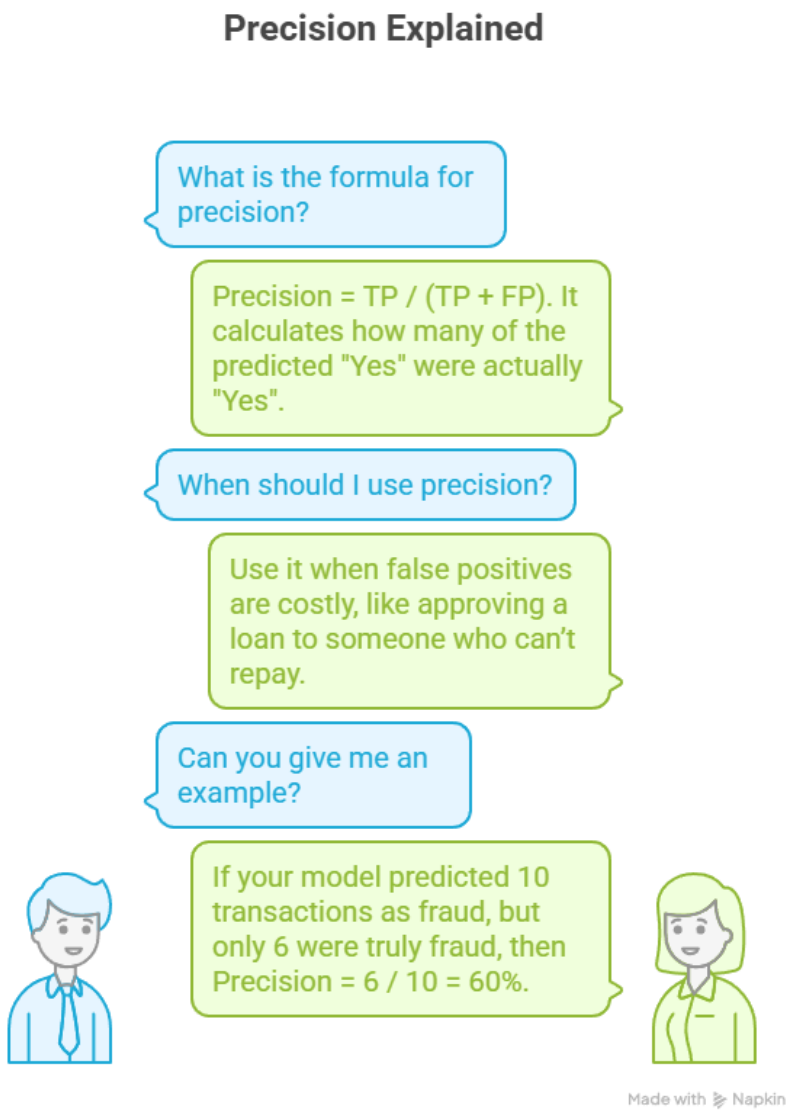
Interpretation

- Model **predicted 6 people should be approved** (Predicted = 1).
- Out of those 6:
- **4 were actually eligible** (True Positives)
- **2 were not eligible** (False Positives)
- So, only **66.67% of approved applicants were actually correct.**

Why Precision Matters Here

In **loan approval**, **False Positives are costly**:

- Approving someone who **can't repay** can lead to **financial losses.**
- Therefore, **precision** tells you **how trustworthy your approvals are.**
- A low precision means you're **approving too many unqualified applicants**, which is risky.



Recall (Sensitivity / True Positive Rate)

Formula:

```
Recall = TP / (TP + FN)
```

(= Of all actual “Yes”, how many did the model catch?)

Use When:

False Negatives are costly

Example: Detecting cancer—you don’t want to miss any positive cases

Example:

Cancer Detection

You built a model to detect whether a person has **cancer**:

- **1 = Has cancer**
- **0 = No cancer**

You run the model on **10 patients**. Here's the result:

Actual vs Predicted Table

Patient	Actual (A)	Predicted (P)
1	1	1
2	1	0
3	0	0
4	1	1
5	0	0
6	1	1
7	1	0
8	0	0
9	0	0
10	1	1

Step 1: Confusion Matrix Counts

	Predicted: 1 (Cancer)	Predicted: 0 (No Cancer)
Actual: 1 (Cancer)	TP = 4	FN = 2
Actual: 0 (No Cancer)	FP = 0	TN = 4

Step 2: Recall Formula

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

Final Recall: 66.67%

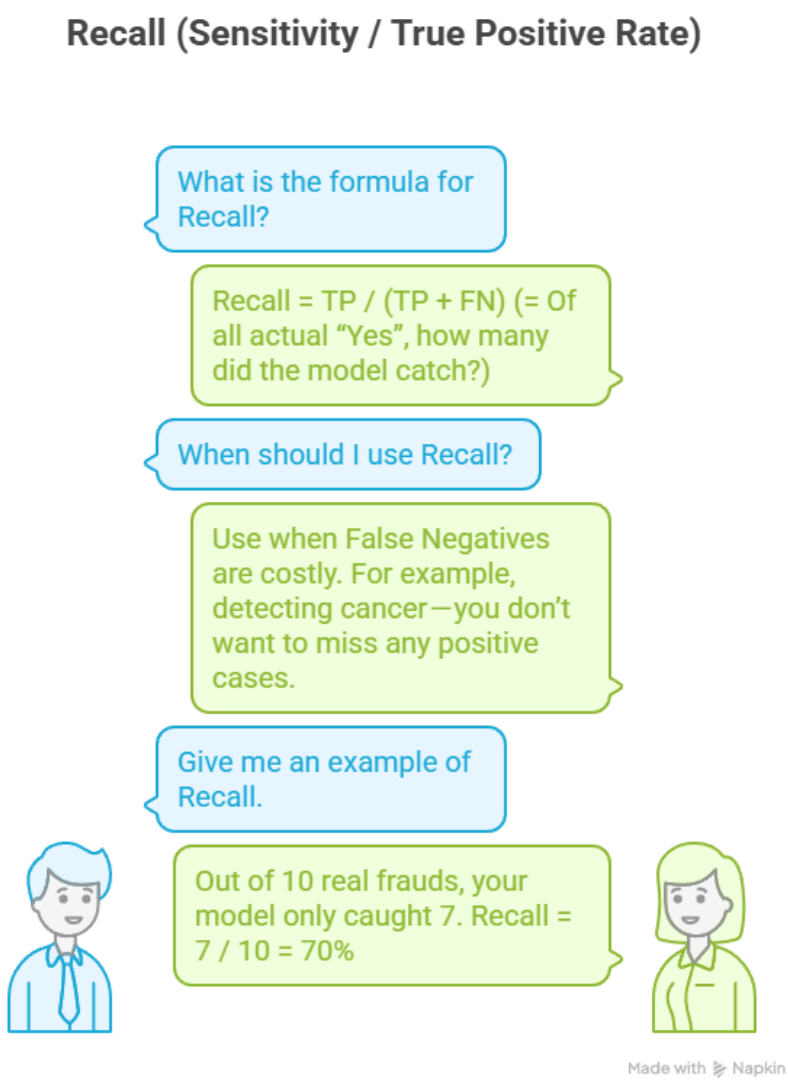
Interpretation

- There were **6 patients who actually had cancer**.
- The model **correctly caught 4 of them** (True Positives), but **missed 2** (False Negatives).
- So the **recall is 66.67%**, meaning the model **identified two-thirds of the actual cancer cases**.

Why Recall Matters Here

In **critical cases like cancer detection**, **missing a positive case (False Negative)** can be **life-threatening**.

- A person with cancer who gets a "No cancer" prediction might not get treated.
- So it's **more important to catch as many actual positive cases as possible**, even if that means occasionally having a false alarm (False Positive).



F1 Score

Formula:

```
F1 = 2 * (Precision * Recall) / (Precision + Recall)
```

F1 Score is the **harmonic mean** of precision and recall.

Best For:

Imbalanced datasets where both FP and FN matter

- Score ranges from 0 to 1
- Higher is better

Example:

Job Candidate Screening

A company uses a model to **predict whether a candidate is suitable for a job.**

- **1 = Suitable**
- **0 = Not Suitable**

The model is evaluated on **10 candidates.**

Actual vs Predicted Table

Candidate	Actual (A)	Predicted (P)
1	1	1
2	0	1
3	1	1
4	1	0
5	0	0
6	1	1
7	0	0
8	0	1
9	1	1
10	0	0

Step 1: Confusion Matrix Counts

	Predicted: 1 (Suitable)	Predicted: 0 (Not Suitable)
Actual: 1 (Suitable)	TP = 4	FN = 1
Actual: 0 (Not Suitable)	FP = 2	TN = 3

Step 2: Precision

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{4}{4+2} = \frac{4}{6} = 0.6667$$

Step 3: Recall

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{4}{4+1} = \frac{4}{5} = 0.8$$

Step 4: F1 Score Formula

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

$$= 2 \times \frac{0.6667 \times 0.8}{0.6667 + 0.8}$$

$$= 2 \times \frac{0.53336}{1.4667} \approx 2 \times 0.3636 = \mathbf{0.7272}$$

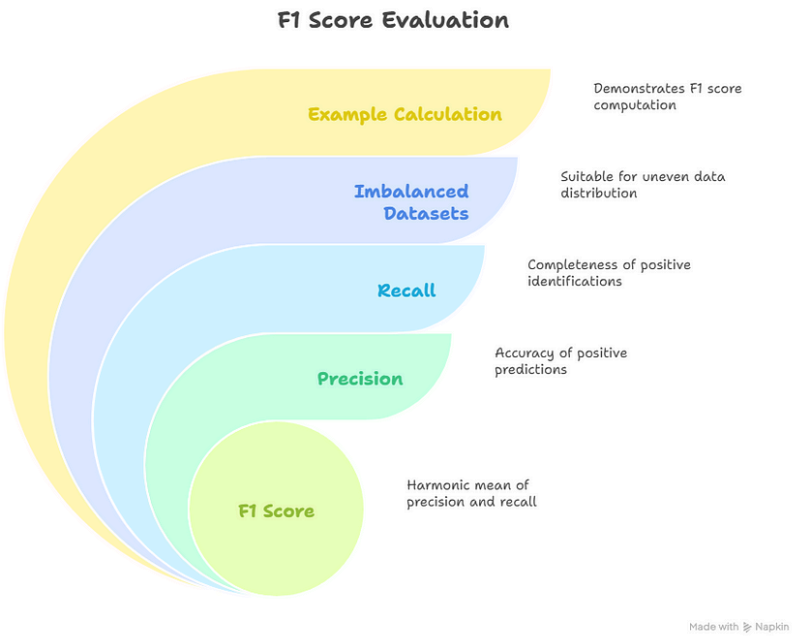
Final F1 Score: 0.727 (or 72.7%)

Interpretation

- **Precision** = 66.67% → Of all predicted suitable candidates, 66.67% were actually suitable.
- **Recall** = 80% → Of all truly suitable candidates, 80% were caught by the model.
- The **F1 Score** = 72.7% balances both.

Why F1 Score Matters

- F1 Score is helpful when **both false positives and false negatives matter**.
- In this case:
- A **false positive** means wasting time interviewing an unsuitable person.
- A **false negative** means **missing a great candidate**.
- F1 combines both into one metric—useful especially when the dataset is **imbalanced** (e.g., fewer qualified candidates).



AUC-ROC Curve

The ROC Curve (Receiver Operating Characteristic) plots **True Positive Rate vs False Positive Rate** at different thresholds.

Instead of using just `model.predict()`, we use `model.predict_proba()` to get probabilities like:

[0.2, 0.8, 0.3, 0.72, 0.42, 0.65]

If we set a threshold of 0.5, any value > 0.5 becomes class 1, else class 0.

By adjusting the threshold:

- You reduce false negatives (catch more positives)
- But you may increase false positives

Example:

Predicting whether an email is spam (1) or not (0)

Model outputs probabilities (using `predict_proba`) for 6 emails:

Email #	Actual Class	Predicted Probability of Spam (1)
1	0	0.20
2	1	0.80
3	0	0.30
4	1	0.72
5	0	0.42
6	1	0.65

Step 1: Choose Thresholds and Classify

Let's try thresholds: **0.5**, **0.4**, and **0.7**

Threshold	Predicted Classes (Prob > threshold = 1)
0.5	Emails 2,4,6 → Class 1; Others → 0
0.4	Emails 2,4,5,6 → Class 1; Others → 0
0.7	Emails 2,4 → Class 1; Others → 0

Step 2: Calculate TPR and FPR for each threshold

Recall:

- **TPR (Recall)** = **TP / (TP + FN)** = Proportion of actual positives correctly predicted
- **FPR = FP / (FP + TN)** = Proportion of actual negatives incorrectly predicted as positive

Threshold 0.5

Email #	Actual	Predicted (threshold 0.5)
1	0	0
2	1	1
3	0	0
4	1	1
5	0	0
6	1	1

- TP = 3 (emails 2,4,6 correctly predicted as spam)
- FN = 0 (no missed spam)
- FP = 0 (no non-spam predicted as spam)
- TN = 3 (emails 1,3,5 correctly predicted as not spam)

$$TPR = \frac{3}{3+0} = 1.0$$

$$FPR = \frac{0}{0+3} = 0.0$$

Threshold 0.4

Email #	Actual	Predicted (threshold 0.4)
1	0	0
2	1	1
3	0	0
4	1	1
5	0	1
6	1	1

- TP = 3 (emails 2,4,6)
- FN = 0
- FP = 1 (email 5 is incorrectly predicted as spam)
- TN = 2 (emails 1,3)

$$TPR = \frac{3}{3+0} = 1.0$$

$$FPR = \frac{1}{1+2} = \frac{1}{3} \approx 0.333$$

Threshold 0.7

Email #	Actual	Predicted (threshold 0.7)
1	0	0
2	1	1
3	0	0
4	1	1
5	0	0
6	1	0

- TP = 2 (emails 2,4)
- FN = 1 (email 6 missed)
- FP = 0
- TN = 3 (emails 1,3,5)

$$TPR = \frac{2}{2+1} = \frac{2}{3} \approx 0.667$$

$$FPR = \frac{0}{0+3} = 0.0$$

Step 3: Plot ROC Points

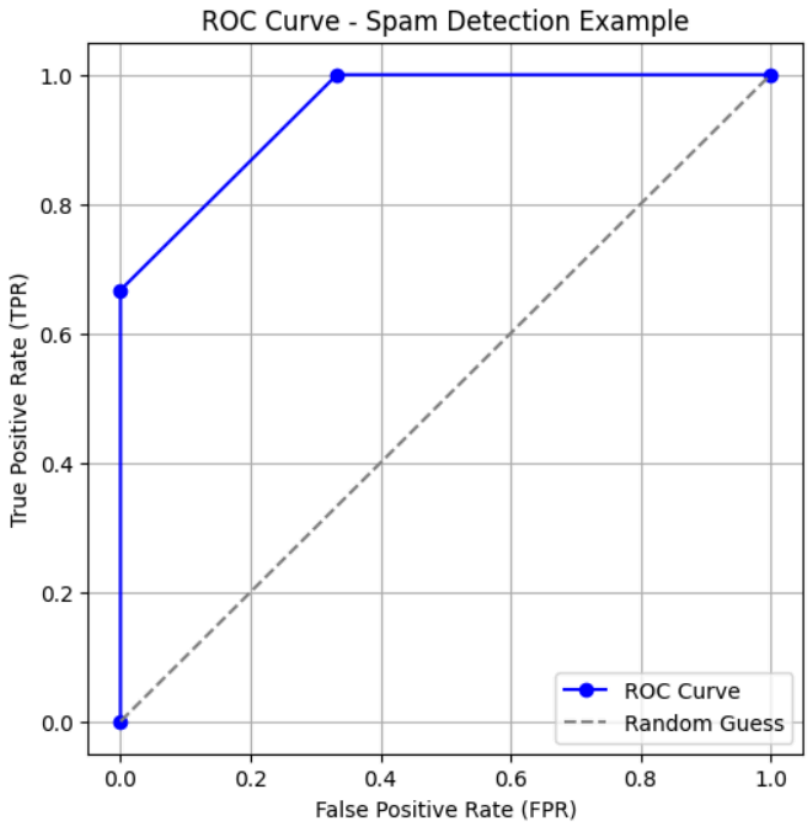
Threshold	FPR	TPR
0.7	0.0	0.667
0.5	0.0	1.0
0.4	0.333	1.0

Step 4: Plotting (simple graph)

- X-axis: FPR
- Y-axis: TPR

Points:

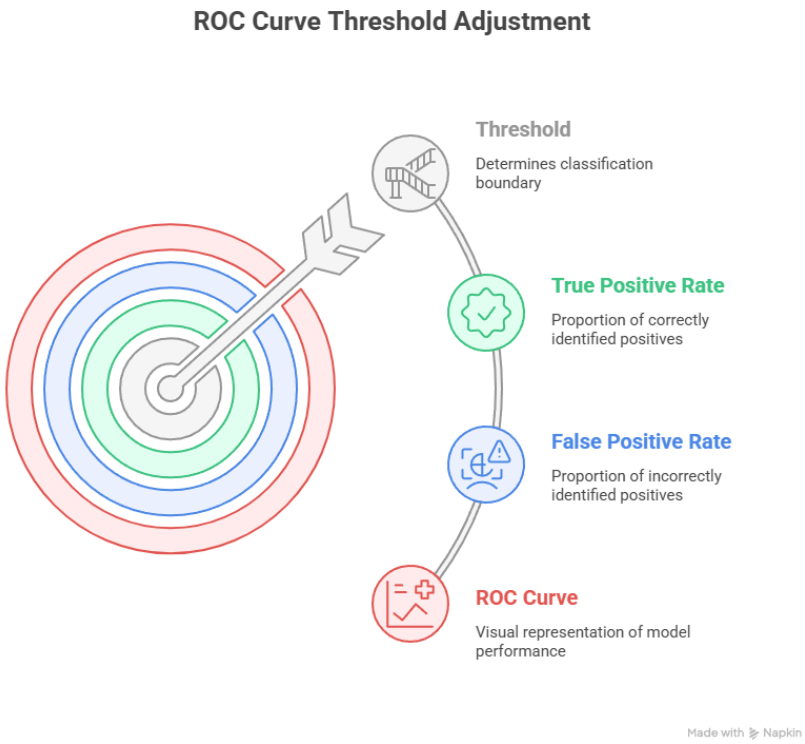
- (0.0, 0.667)
- (0.0, 1.0)
- (0.333, 1.0)



Interpretation:

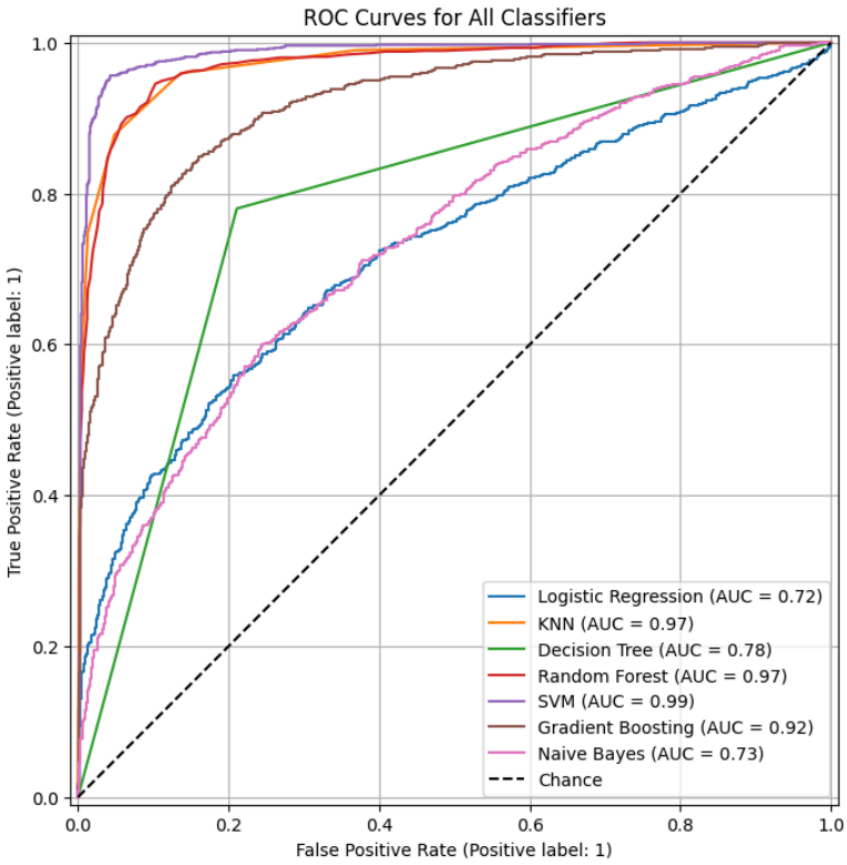
- When the threshold is **high (0.7)**, the model is **conservative** — fewer false positives (FPR=0), but misses some true positives (TPR=0.667).
- When the threshold is **medium (0.5)**, the model catches all true positives (TPR=1) with no false positives (FPR=0).
- When the threshold is **low (0.4)**, the model catches all positives but allows some false positives (FPR=0.333).

The ROC curve shows the **trade-off** between catching more positives vs. introducing false alarms.



Comparing Classifiers Using ROC Curves: Which Model Performs Best

```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import RocCurveDisplay

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

X, y = make_classification(n_samples=10000, n_features=20, n_classes=2,
                           n_informative=15, class_sep=0.5, random_state=42)


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


scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

models = {
    "Logistic Regression": LogisticRegression(),
    "KNN": KNeighborsClassifier(),
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "SVM": SVC(probability=True),
    "Gradient Boosting": GradientBoostingClassifier(),
    "Naive Bayes": GaussianNB()
}

plt.figure(figsize=(10, 8))

for name, model in models.items():
    if name in ["SVM", "KNN"]:
        model.fit(X_train_scaled, y_train)
        y_score = model.predict_proba(X_test_scaled)[:, 1]
    else:
        model.fit(X_train, y_train)
        y_score = model.predict_proba(X_test)[:, 1]

    RocCurveDisplay.from_predictions(y_test, y_score, name=name, ax=plt.gca())

plt.plot([0, 1], [0, 1], "k--", label="Chance")
plt.title("ROC Curves for All Classifiers")
plt.legend(loc="lower right")
plt.grid(True)
plt.show()
```



1. SVM performs almost perfectly with an AUC of 0.999.

2. KNN and Random Forest are also very strong performers (AUC ≈ 0.97).

3. Logistic Regression and Naive Bayes have the weakest performance among the listed models (AUCs ~0.72–0.73).

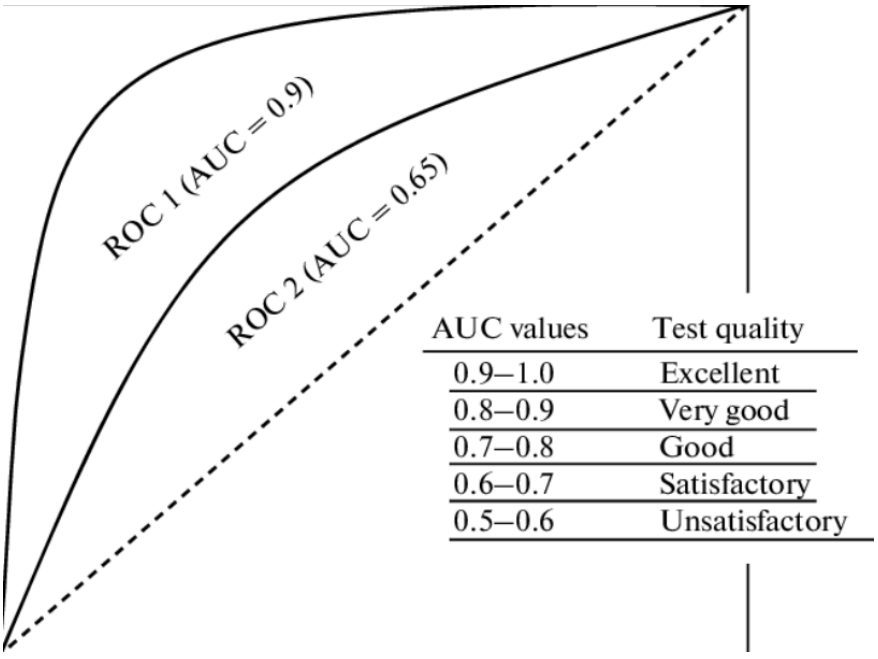
4. The Decision Tree shows moderate performance (AUC = 0.78) but is significantly outperformed by ensemble methods.

AUC (Area Under Curve) – What's a Good Score?

AUC Score	Interpretation
0.90 – 1.00	Excellent
0.80 – 0.90	Very Good
0.70 – 0.80	Good
0.60 – 0.70	Satisfactory
0.50 – 0.60	Unsatisfactory
< 0.50	Worse than random

ROC Curve Summary:

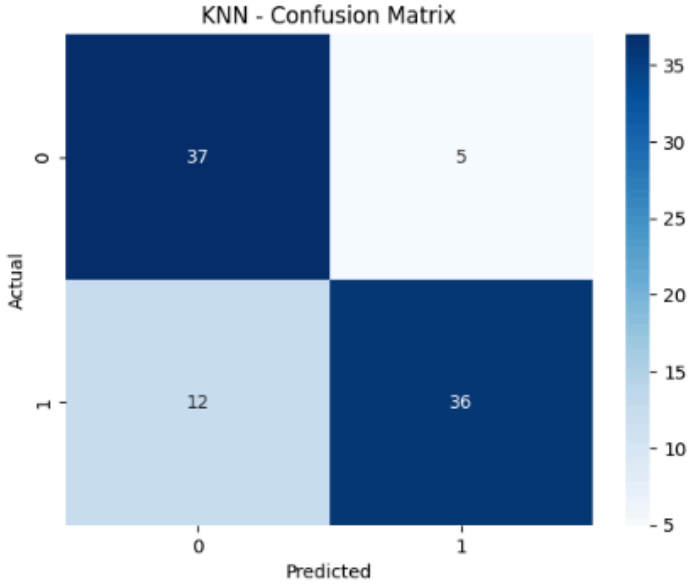
- ROC = 1.0 → Perfect model
- ROC = 0.5 → Random guess
- ROC < 0.5 → Worse than guessing



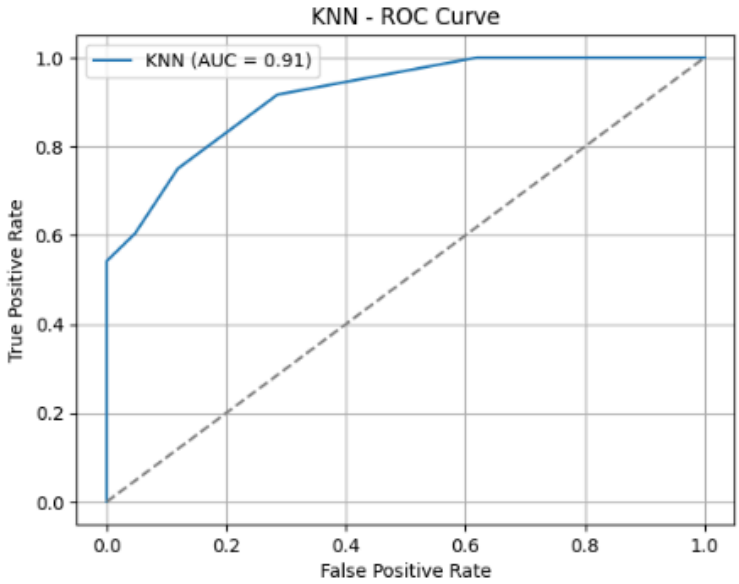
Here's a **Python example for each classification evaluation metric** using a **simple dataset** and three classifiers: **K-Nearest Neighbors (KNN)**, **Naive Bayes (NB)**, and **Decision Tree (DT)**.

```python
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import (confusion_matrix, accuracy_score, precision_score,
    recall_score, f1_score, roc_auc_score, roc_curve)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
import seaborn as sns

X, y = make_classification(
    n_samples=300, n_features=5, n_informative=3,
    n_redundant=0, n_classes=2, random_state=42
)

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

models = {
    'KNN': KNeighborsClassifier(),
    'Naive Bayes': GaussianNB(),
    'Decision Tree': DecisionTreeClassifier(random_state=42)
}

for name, model in models.items():
    print(f"\nModel: {name}")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:")
    print(cm)

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f"{name} - Confusion Matrix")
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred)
    rec = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_proba)

    print(f"Accuracy: {acc:.2f}")
    print(f"Precision: {prec:.2f}")
    print(f"Recall: {rec:.2f}")
    print(f"F1 Score: {f1:.2f}")
    print(f"AUC-ROC: {auc:.2f}")

    fpr, tpr, thresholds = roc_curve(y_test, y_proba)
    plt.plot(fpr, tpr, label=f'{name} (AUC = {auc:.2f})')
    plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
    plt.title(f"{name} - ROC Curve")
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend()
    plt.grid(True)
    plt.show()
```

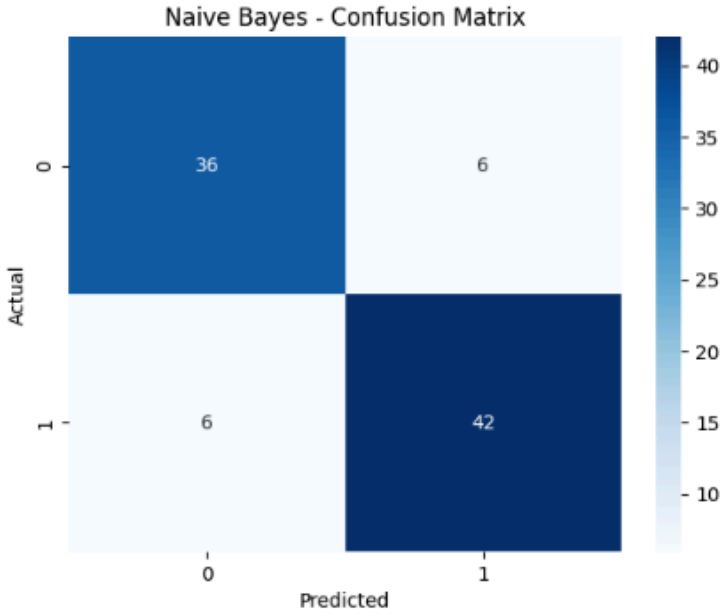
```
Model: KNN
Confusion Matrix:
[[37  5]
 [12 36]]
```



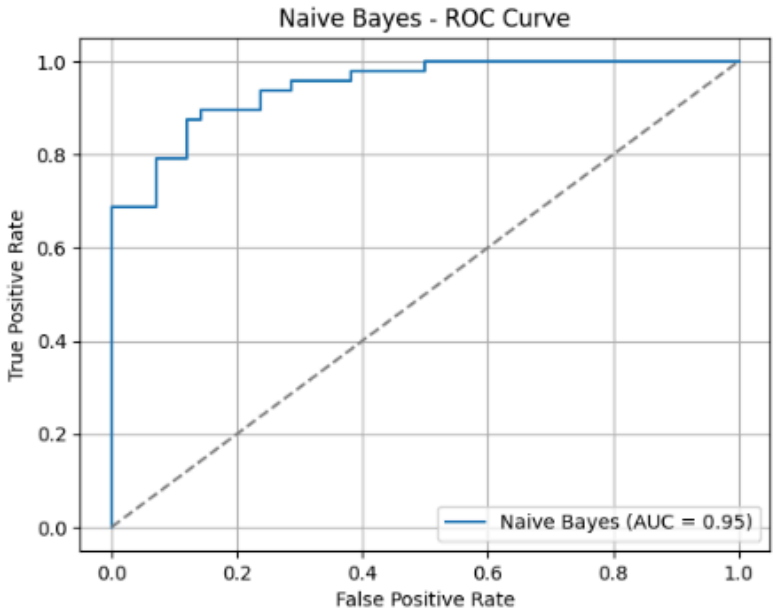

```
Accuracy: 0.81
Precision: 0.88
Recall: 0.75
F1 Score: 0.81
AUC-ROC: 0.91
```



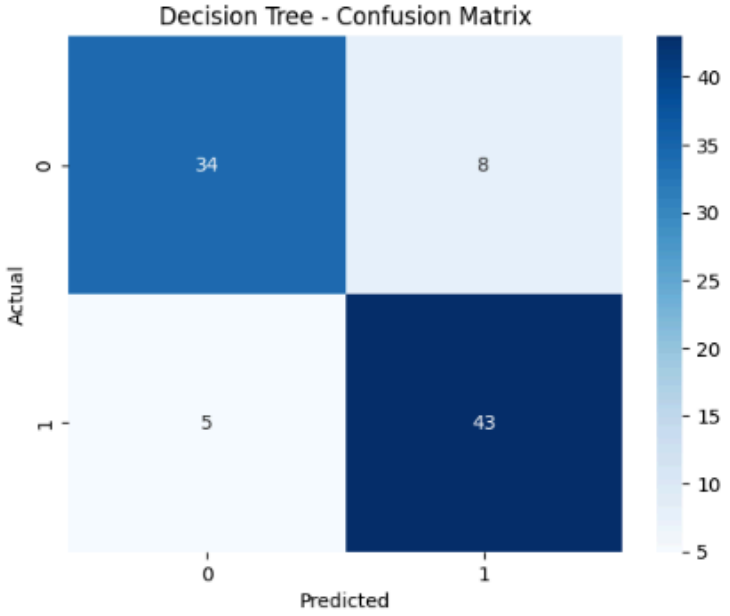
```
Model: Naive Bayes
Confusion Matrix:
[[36  6]
 [ 6 42]]
```



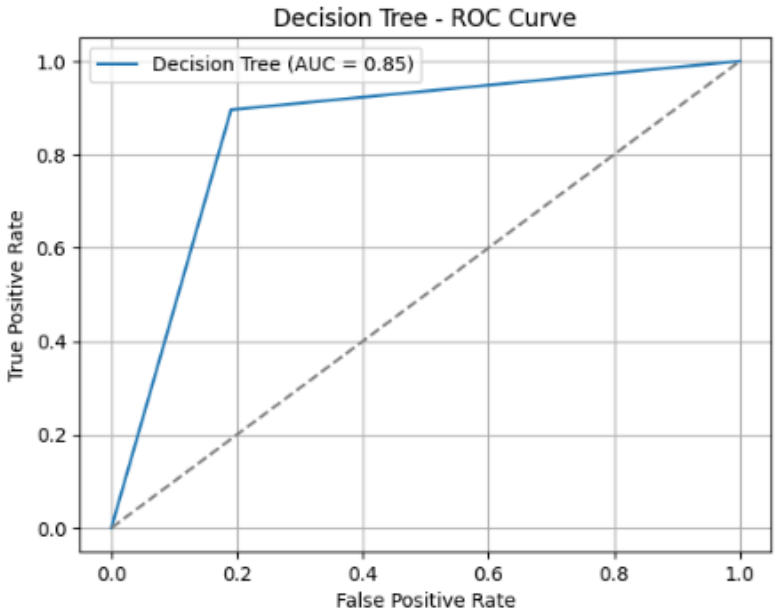

```
Accuracy: 0.87
Precision: 0.88
Recall: 0.88
F1 Score: 0.88
AUC-ROC: 0.95
```



```
Model: Decision Tree
Confusion Matrix:
[[34  8]
 [ 5 43]]
```




```
Accuracy: 0.86
Precision: 0.84
Recall: 0.90
F1 Score: 0.87
AUC-ROC: 0.85
```



Regression Models

When building a **regression model** in machine learning, the goal is to **predict a continuous value**—like house prices, sales, or temperatures.

But how do we know if our model is doing a good job?

That's where **regression evaluation metrics** come in. These metrics help us measure the difference between what the model predicted and what actually happened.

What Are Residuals?

Residual (or error) = Actual value—Predicted value
The smaller the residuals, the better the model.

Let's Look at an Example

Actual values (y):

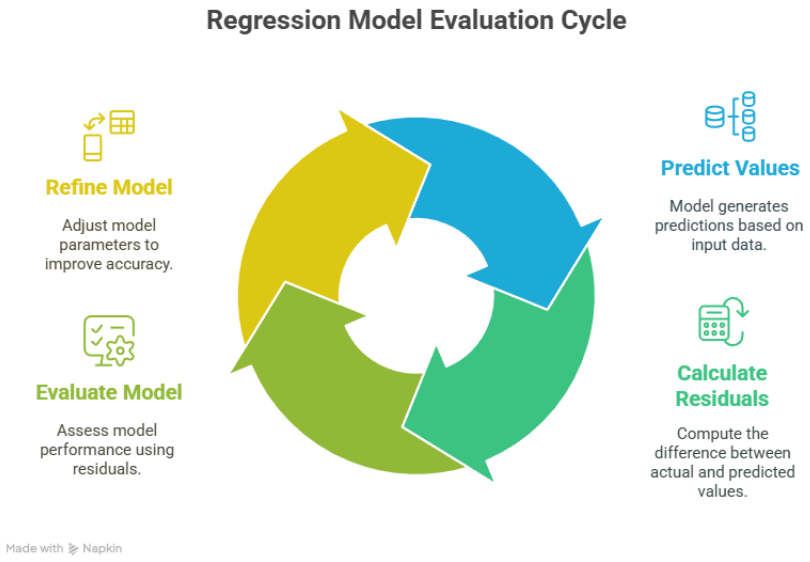
```
[23, 45, 55, 65, 70]
```

Predicted values (ŷ):

```
[20, 50, 50, 60, 85]
```

Residuals (y—ŷ):

```
[3, -5, 5, 5, -15]
```



Common Evaluation Metrics:

Mean Squared Error (MSE)

MSE measures the **average of the squared differences** between actual and predicted values.

Formula:

$$MSE = \Sigma (actual - predicted)^2 / n$$

Manual Example:

Case	Actual (y)	Predicted (ŷ)	Error (y - ŷ)	Squared Error (Error²)	Interpretation
A	2	1	1	1	Slight underestimation
	3	6	-3	9	Overestimated by 3
	7	3	4	16	Underestimated by 4
	9	2	7	49	Large underestimation
	10	7	3	9	Underestimated by 3
				Total: 84	
				MSE = 84 / 5 = 16.8	**Poor overall performance**

Case	Actual (y)	Predicted (ŷ)	Error (y - ŷ)	Squared Error (Error²)	Interpretation
B	2	1	1	1	Slight underestimation
	4	4	0	0	Perfect prediction
	7	5	2	4	Small underestimation
	9	8	1	1	Slight underestimation
	10	10	0	0	Perfect prediction
				Total: 6	
				MSE = 6 / 5 = 1.2	**Much better model performance**

Interpretation Summary:

- **Case A:** High errors and large variance in predictions → **poor model fit.**
- **Case B:** Errors are small or zero → **closer to actual values** → **better model fit.**

When is MSE useful?

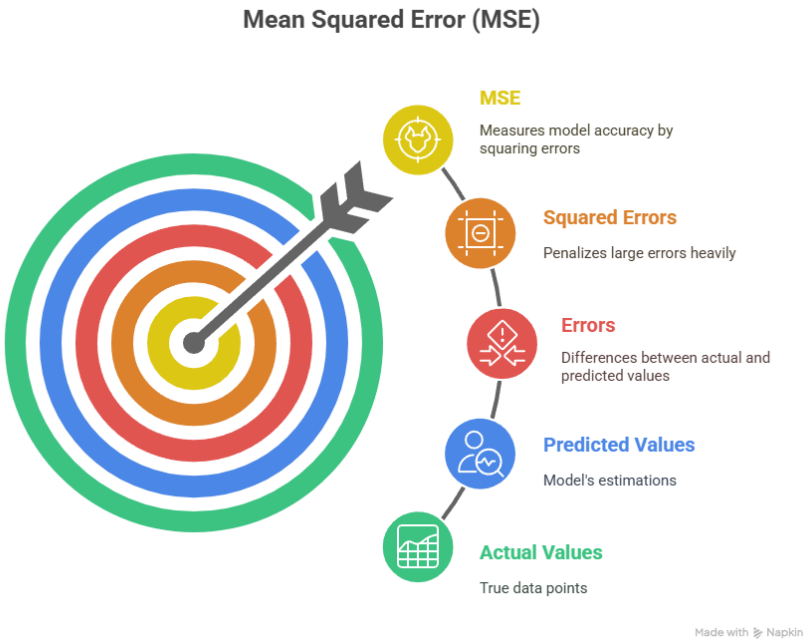
- **Lower MSE means a better model.**

Disadvantages:

- **Units are squared**, making it harder to interpret.
- **Useful when we want to heavily penalize large errors.**

- **Very sensitive to outliers.**

Why?
A single large error (like predicting 100 instead of 10) increases MSE dramatically.



Root Mean Squared Error (RMSE)

RMSE is just the **square root of MSE**, which brings the error back to the same units as the target variable.

Formula:

$$RMSE = \sqrt{(\Sigma\ (actual - predicted)^2 / n)}$$

Manual Example:

Index	Actual (y)	Predicted (ŷ)	Error (y - ŷ)	Squared Error	Interpretation
1	2	1	1	1	Slight underestimation
2	4	4	0	0	Perfect prediction
3	7	5	2	4	Underestimated by 2
4	9	8	1	1	Slight underestimation
5	10	10	0	0	Perfect prediction
				Total = 6	

Calculating Mean Squared Error (MSE)

MSE formula:

$$\text{MSE} = \frac{\sum(\text{Error})^2}{n}$$

Where:

- Squared Errors = [1, 0, 4, 1, 0]
- Total Squared Error = 6
- Number of observations (n) = 5

$$\text{MSE} = \frac{6}{5} = 1.2$$

Interpretation:

- An **MSE of 1.2** indicates **low average squared error**.
- Most predictions are **close to the actual values** (0–2 units off).
- Two of five predictions are **perfect**, and none have large errors.
- This suggests the model is **quite accurate and consistent**.

Benefits of RMSE:

- Same units as the target (e.g., dollars, degrees).
- Still penalizes large errors.
- Differentiable (useful for optimization in ML).

Disadvantage:

- Like MSE, it still **penalizes large errors heavily**.

Mean Absolute Error (MAE) or MAD (Mean Absolute Deviation)

This metric takes the **average of absolute errors**, treating **all errors equally**.

Formula:

$$MAE = \Sigma \, |actual - predicted| \, / \, n$$

Manual Example:

Index	Actual (y)	Predicted (ŷ)	Absolute Error = |y − ŷ|	Interpretation
1	2	1	|2 − 1| = **1**	Slight underestimation
2	4	4	|4 − 4| = **0**	Perfect prediction
3	7	5	|7 − 5| = **2**	Underestimated by 2
4	9	8	|9 − 8| = **1**	Slight underestimation
5	10	10	|10 − 10| = **0**	Perfect prediction
			Total = 4	

Calculating Mean Absolute Error (MAE)

Formula:

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

- Total Absolute Error = 1 + 0 + 2 + 1 + 0 = **4**
- Number of data points (n) = 5

$$\text{MAE} = \frac{4}{5} = 0.8$$

Interpretation:

- **MAE of 0.8** means that, on average, the model's predictions are **0.8 units away** from the actual values.

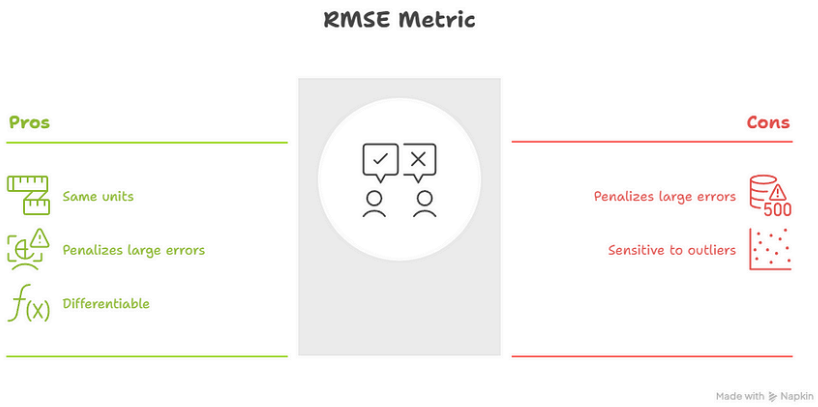
- This is a **low MAE**, showing that the model makes small, consistent errors.
- It's easier to interpret than MSE because it uses **absolute values**, not squares (so errors aren't exaggerated).
- MAE is also **robust to outliers** compared to MSE.

Benefits of MAE:

- **Easy to understand**
- **Same units as target**
- **Not sensitive to outliers** (treats all errors equally)

Disadvantage:

- **Not differentiable at zero**
  (This is a problem during optimization in some algorithms like gradient descent.)



Metric	Formula	Penalizes Outliers?	Units	Differentiable?
MSE	Average of squared errors	Yes	No (squared)	Yes
RMSE	Square root of MSE	Yes	Yes	Yes
MAE	Average of absolute errors	No	Yes	No

R^2 Score:

When we build **regression models**, one key question we ask is:

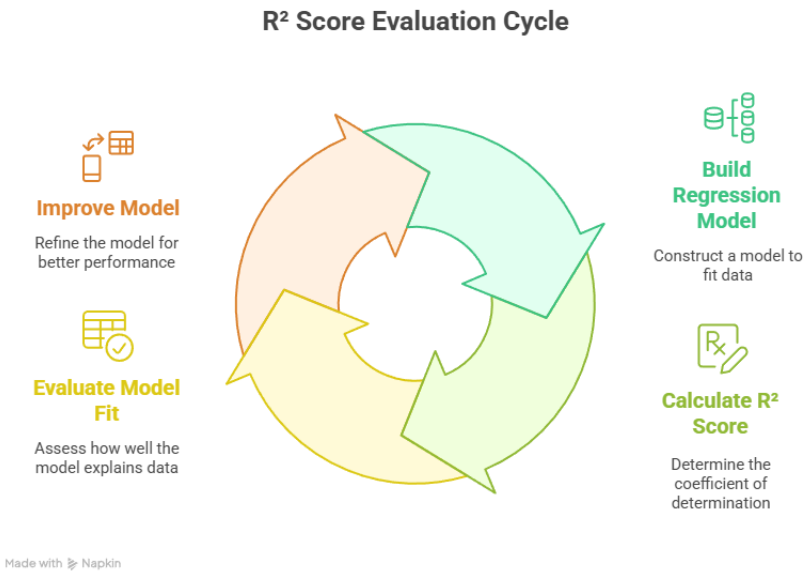
"How well is my model fitting the data?"

That's where **R^2 Score** (R-squared), also called the **coefficient of determination**, comes in.

What Is R^2 Score?

R^2 tells us **how much of the variation in the target variable (y) is explained by the model.**

- **Higher R^2** → better fit
- **$R^2 = 1$** → perfect prediction
- **$R^2 = 0$** → model is no better than just predicting the average
- **$R^2 < 0$** → model is worse than a horizontal line (very poor fit)



R^2 Score Formula

$$R^2 = 1 - \frac{SSR}{TSS}$$

Where:

- **SSR** = Sum of Squared Residuals = $\sum(y_i - \hat{y}_i)^2$
- **TSS** = Total Sum of Squares = $\sum(y_i - \bar{y})^2$
- y_i: actual values
- $\hat{y}_i$: predicted values
- $\bar{y}$: mean of actual values

Manual Example

Index	Actual (y)	Predicted (ŷ)	Error (y - ŷ)	(y - ŷ)² (Squared Error)
1	10	12	-2	4
2	20	18	2	4
3	30	33	-3	9
4	40	37	3	9
5	50	49	1	1
				Total SSR = 27

Calculate Mean of Actuals

$$\bar{y} = \frac{10 + 20 + 30 + 40 + 50}{5} = \frac{150}{5} = 30$$

Calculate TSS (Total Sum of Squares)

$$\text{TSS} = \sum (y_i - \bar{y})^2$$

Index	Actual (y)	$y_i - \bar{y}$	$(y_i - \bar{y})^2$
1	10	-20	400
2	20	-10	100
3	30	0	0
4	40	10	100
5	50	20	400
			TSS = 1000

Calculate SSR (Sum of Squared Residuals)

$$\text{SSR} = \sum (y_i - \hat{y}_i)^2$$

Index	Actual (y)	Predicted (ŷ)	Residual $y - \hat{y}$	$(y - \hat{y})^2$
1	10	12	-2	4
2	20	18	2	4
3	30	33	-3	9
4	40	37	3	9
5	50	49	1	1
				SSR = 27

Compute R^2 Score

$$R^2 = 1 - \frac{SSR}{TSS} = 1 - \frac{27}{1000} = 1 - 0.027 = \boxed{0.973}$$

Interpretation:

- **R^2 = 0.973** means the model explains **97.3% of the total variation** in the actual values.
- Only **2.7%** of the variation is due to error (residuals).
- This indicates a **very strong model fit.**

Interpretation Cases

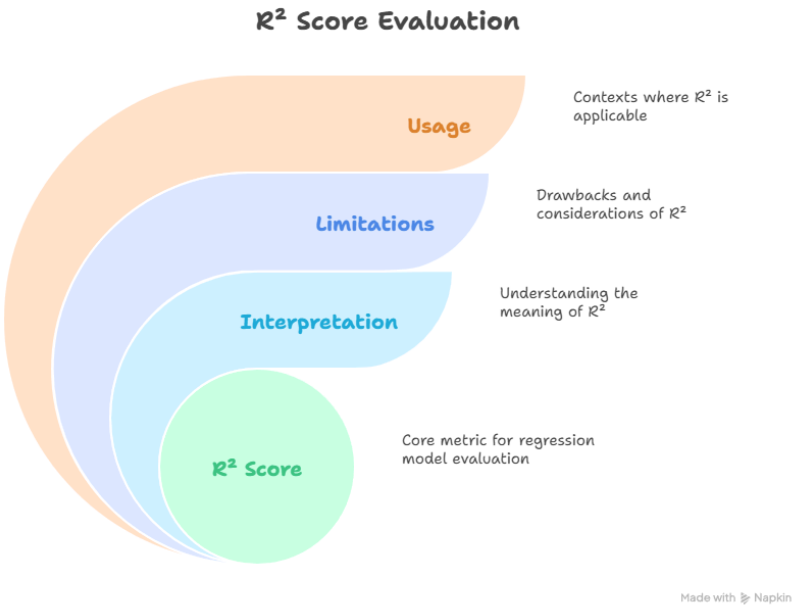
Case	What It Means	R^2 Value
Model always predicts mean	SSR = TSS → no learning	$R^2 = 0$
Model is perfect	SSR = 0 → no error	$R^2 = 1$
Model performs well	0 < SSR < TSS → some error	$0 < R^2 < 1$
Model is worse than average	SSR > TSS	$R^2 < 0$

Limitations of R^2 Score

1. **R^2 increases when you add more features**, even if they're irrelevant
   → This can give a **false impression** of a better model
2. **R^2 doesn't tell you** whether your predictions are biased
3. **Only works for regression problems**—not for classification

When to Use R^2

Use R^2 **to evaluate model performance** when you're doing regression **and your features are meaningful.** But always pair it with **other metrics** like RMSE or MAE for a fuller picture.



Adjusted R^2 Score:

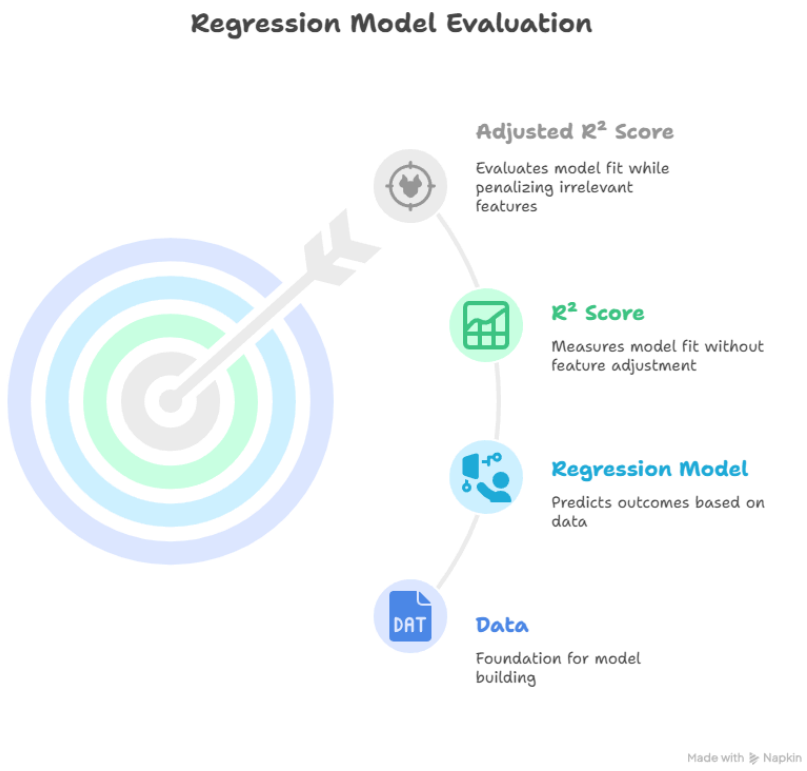
When building regression models, we often use **R^2 Score** to evaluate how well the model fits the data. But there's a catch:

R^2 always increases when you add more features, even if they're useless.

That's where **Adjusted R^2 Score** comes in—it adjusts for the number of predictors in your model.

What is Adjusted R^2 Score?

Adjusted R^2 tells you how well your model explains the variance in the target variable **after penalizing for irrelevant features.**



Formula:

$$\text{Adjusted } R^2 = 1 - \left(\frac{(1 - R^2)(n - 1)}{n - k - 1} \right)$$

Where:

- **R²** = regular R^2 score
- **n** = number of observations (rows)
- **k** = number of independent features (columns)

Manual Example

Let's say:

- $R^2 = 0.85$
- n = 100 (data points)
- k = 10 (features used in the model)

Plug into the formula:

$$\text{Adjusted } R^2 = 1 - \left(\frac{(1 - 0.85)(100 - 1)}{100 - 10 - 1} \right) = 1 - \left(\frac{(0.15)(99)}{89} \right) = 1 - (14.85/89) = 1 - 0.167 = 0.833$$

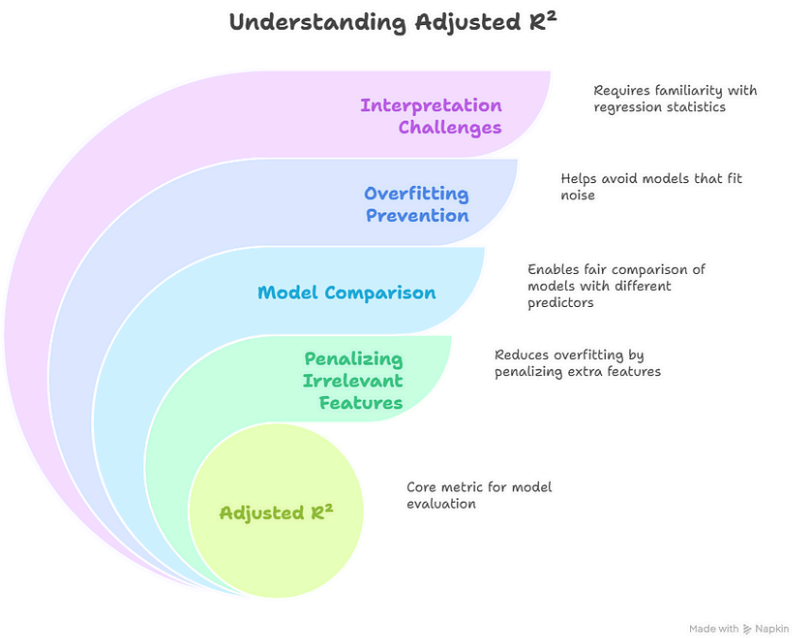
Adjusted R^2 = 0.833, which is slightly lower than $R^2 = 0.85$—because the formula penalized the number of features used.

Advantages of Adjusted R²

- Penalizes **irrelevant features**
- More reliable when comparing **models with different numbers of predictors**
- Helps prevent **overfitting**

Disadvantages

- Can still increase with noisy features if they add slight predictive power
- Not intuitive to interpret unless you're familiar with regression statistics



Python Example for Each Regression Evaluation Metric Using a Simple Dataset and Linear Regression Model

```python
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=100, n_features=5, noise=10, random_state=42)

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

model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error (MSE): {mse:.2f}")

rmse = np.sqrt(mse)
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")

mae = mean_absolute_error(y_test, y_pred)
print(f"Mean Absolute Error (MAE): {mae:.2f}")

r2 = r2_score(y_test, y_pred)
print(f"R² Score: {r2:.2f}")

n = X_test.shape[0]
k = X_test.shape[1]
adj_r2 = 1 - (1 - r2) * (n - 1) / (n - k - 1)
print(f"Adjusted R² Score: {adj_r2:.2f}")
```

```
Mean Squared Error (MSE): 113.45
Root Mean Squared Error (RMSE): 10.65
Mean Absolute Error (MAE): 8.44
R² Score: 0.99
Adjusted R² Score: 0.99
```

Quick Reference Guide: Classification & Regression Evaluation Metrics

Classification Evaluation Metrics:

Metric	Formula / Concept	Best For	Range	Key Notes
Accuracy	(TP + TN) / (TP + TN + FP + FN)	Balanced datasets	0 to 1	Misleading on imbalanced data
Precision	TP / (TP + FP)	When **False Positives** are costly	0 to 1	High Precision → fewer false positives
Recall	TP / (TP + FN)	When **False Negatives** are costly	0 to 1	High Recall → fewer false negatives
F1 Score	2 × (Precision × Recall) / (Precision + Recall)	Imbalanced datasets	0 to 1	Harmonic mean of precision & recall
AUC-ROC	Area under ROC curve	Comparing classifiers	0 to 1	Higher AUC = better model
Confusion Matrix	Table of TP, TN, FP, FN	Model interpretation	N/A	Visual metric showing prediction errors

Regression Evaluation Metrics:

Metric	Formula / Concept	Best For	Units	Key Notes
MSE	Mean of squared errors	Penalizing large errors	Squared units	Sensitive to outliers, differentiable
RMSE	√MSE	Interpretability	Same as target	Still penalizes large errors, differentiable
MAE (MAD)	Mean of absolute errors	Outlier-resistant	Same as target	Treats all errors equally, **not** differentiable at 0
R^2 Score	1 - SSR / TSS	Variance explanation	Unitless (0 to 1)	Higher = better fit; may be misleading with irrelevant features
Adjusted R^2	$1 - (1 - R^2) \times (n - 1)/(n - k - 1)$	Model comparison with extra features	Unitless	Penalizes unnecessary features