OpenAI

Train Test Split Explanation

This is a copy of a conversation between ChatGPT & Anonymous.

from sklearn.model_selection import train_test_split

X = data_numeric.drop(columns=['Price'])
y = data_numeric['Price']

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

print(f"Training set shape: {X_train.shape}, {y_train.shape}")
print(f"Testing set shape: {X_test.shape}, {y_test.shape}") Explain this code.

This code is for splitting a dataset into **training** and **testing** sets using **scikit-learn**'s `train_test_split` function. Here's a line-by-line explanation:

🔹 `from sklearn.model_selection import train_test_split`

This imports the `train_test_split` function from the `sklearn.model_selection` module. This function is used to split data into training and testing subsets.

🔹 `X = data_numeric.drop(columns=['Price'])`

- `data_numeric` is assumed to be a **DataFrame** (probably a cleaned and numeric-only version of your dataset).
- `.drop(columns=['Price'])` removes the `'Price'` column from the DataFrame.
- So, `X` now contains all the features (independent variables), except the target column (`Price`).

🔹 `y = data_numeric['Price']`

- This extracts the `'Price'` column, which is the **target** (dependent variable), and stores it in `y`.

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

This splits the dataset into training and testing sets:

- `X` and `y` are split **together**, preserving the relationship between features and target values.
- `test_size=0.2`: 20% of the data is assigned to the **test set**, and 80% to the **training set**.
- `random_state=42`: This ensures **reproducibility** — the split will be the same each time the code is run.

It returns four variables:

- `X_train`: Features for the training set
- `X_test`: Features for the test set
- `y_train`: Targets for the training set
- `y_test`: Targets for the test set

🔹 `print(f"Training set shape: {X_train.shape}, {y_train.shape}")`

🔹 `print(f"Testing set shape: {X_test.shape}, {y_test.shape}")`

These lines print the shapes (dimensions) of the training and testing sets to show how the data has been split. For example:

```arduino
Training set shape: (800, 10), (800,)
Testing set shape: (200, 10), (200,)
```

This would mean:

- 800 training samples, each with 10 features
- 200 test samples, each with 10 features

✅ Summary:

This code prepares your data for machine learning by:

- Separating features and target
- Splitting the dataset into training and testing sets
- Ensuring consistent random splitting
- Printing the shapes to verify the split

Let me know if you'd like a visualization or want help using this data in a model!

↳

* `random_state=42`: This ensures **reproducibility** — the split will be the same each time the code is run.
explain it.

Sure! Let's break down this part clearly:

🔸 `random_state=42`

In machine learning, when you **split a dataset randomly**, you often want the split to be **repeatable** — so you can run your code again and get **the same result every time**.

The `random_state` parameter is like setting a **seed** for the random number generator. This ensures the randomness is **controlled and predictable**.

🧠 Why `42`?

- The number `42` is arbitrary. It could be any integer — `1`, `7`, `123`, etc.
- `42` is just a popular choice, often used as a joke reference to *"The Hitchhiker's Guide to the Galaxy"*, where 42 is "the answer to life, the universe, and everything."

🧪 Without `random_state`:

```python
train_test_split(X, y, test_size=0.2)
```

- Every time you run this, the data might be split **differently** — so your model might train on different samples and give slightly different results.

✅ With `random_state=42`:

```python
train_test_split(X, y, test_size=0.2, random_state=42)
```

- The same rows go into training and testing every time.
- This makes your results **consistent**, especially useful for debugging or sharing results with others.

📌 Summary:

`random_state=42` is used to make sure the random split of your data is the same every time you run the code. It's helpful for reproducibility and consistency in machine learning experiments.

Let me know if you'd like to see how the results differ with and without it!

import pandas as pd

df = pd.read_csv('/Users/vireshkamlapure/Desktop/SEM VI/MLL-Practical /MLL Practicals /Assignment 2/data/Iris_Data.csv')

num_row = df.shape[0]

col_names = df.columns

```
data_types = df.dtypes

print("Number of data points (rows): ",num_row)
print("Column names: ",col_names)
print("Data types for each column:\n",data_types)
```

This Python code uses **pandas** to load and inspect a CSV file. Let's break it down line by line:

🔹 **`import pandas as pd`**

- This imports the **pandas** library, commonly used for data manipulation and analysis.
- It's given the alias `pd` so you can use shorter syntax like `pd.read_csv()`.

🔹 **`df = pd.read_csv('/Users/vireshkamlapure/Desktop/SEM VI/MLL-Practical /MLL Practicals /Assignment 2/data/Iris_Data.csv')`**

- This reads the **Iris dataset** from the given path on your computer using `pd.read_csv()`.
- It loads the data into a **DataFrame** called `df`.

🔹 **`num_row = df.shape[0]`**

- `df.shape` returns a tuple `(number of rows, number of columns)`.
- `df.shape[0]` extracts the **number of rows** (data points).

🔹 **`col_names = df.columns`**

- `df.columns` returns a list of all **column names** in the dataset.

🔹 **`data_types = df.dtypes`**

- `df.dtypes` gives the **data type** of each column, such as:
  - `float64` for numeric values
  - `object` for strings (like species names)

🔹 Print statements:

```python
print("Number of data points (rows): ",num_row)
print("Column names: ",col_names)
print("Data types for each column:\n",data_types)
```

- These lines display:

- How many rows are in the dataset
- What the column names are
- What type of data each column contains

✅ Example Output (if using Iris dataset):

```go
Number of data points (rows):  150
Column names:  Index(['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm', 'Species'],
Data types for each column:
 SepalLengthCm    float64
SepalWidthCm     float64
PetalLengthCm    float64
PetalWidthCm     float64
Species           object
dtype: object
```

Let me know if you'd like to analyze or visualize this data further!

print(df['species'].unique())

The line:

```python
print(df['species'].unique())
```

does the following:

🔍 What it does:

- It accesses the `species` column of the DataFrame `df`.
- Then, it calls the `.unique()` function on that column.
- This returns a **NumPy array** containing all the **unique values** (distinct species names) in that column.

✅ Why it's useful:

If you're working with the **Iris dataset**, this helps you quickly see what species of flowers are present in the dataset.

📌 Example Output (Iris Dataset):

```python
['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']
```

This means there are **three classes** (species) in the dataset:

- Iris-setosa
- Iris-versicolor
- Iris-virginica

⚠️ Common Mistake:

If you get a `KeyError: 'species'`, it means the column name might be different — e.g., `'Species'` with a capital **S**.

In that case, try:

```python
print(df['Species'].unique())
```

You can confirm the exact column name using:

```python
print(df.columns)
```

Let me know if you want to count how many samples there are per species too!

df.describe()

The command:

```python
df.describe()
```

is used to generate **summary statistics** of the **numeric columns** in a pandas DataFrame (`df`).

📊 What it shows:

This function returns a **table** of key statistical metrics for each numeric column:

Statistic	Meaning
`count`	Number of non-null (non-missing) values
`mean`	Average value
`std`	Standard deviation (spread or variability of the data)

Statistic	Meaning
`min`	Minimum value
`25%`	First quartile (25th percentile)
`50%`	Median (50th percentile)
`75%`	Third quartile (75th percentile)
`max`	Maximum value

✅ Example Output (for the Iris dataset):

```text
       SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
count     150.000000    150.000000     150.000000    150.000000
mean        5.843333      3.057333       3.758000      1.199333
std         0.828066      0.435866       1.765298      0.762238
min         4.300000      2.000000       1.000000      0.100000
25%         5.100000      2.800000       1.600000      0.300000
50%         5.800000      3.000000       4.350000      1.300000
75%         6.400000      3.300000       5.100000      1.800000
max         7.900000      4.400000       6.900000      2.500000
```

🧠 Notes:

- By default, `df.describe()` only includes **numeric columns**.
- To include **non-numeric** columns (like strings), use:

```python
df.describe(include='all')
```

Would you like to visualize any of this data (like histograms or boxplots)?

plt.hist(df['sepal_length'], bins=20, color='skyblue', edgecolor='black')

plt.xlabel('Sepal Length (cm)')
plt.ylabel('Frequency')

plt.title('Histogram of Sepal Length')

plt.show()

The code you've written is used to **plot a histogram** of the **Sepal Length** values in the dataset using **Matplotlib**.

Let me explain each part:

📌 Code Breakdown:

```python
plt.hist(df['sepal_length'], bins=20, color='skyblue', edgecolor='black')
```

- `plt.hist(...)`: Plots a histogram.
- `df['sepal_length']`: Selects the **sepal length** column from the DataFrame.
- `bins=20`: Divides the data into 20 equal-width intervals (bins).
- `color='skyblue'`: Fills the bars with a light blue color.
- `edgecolor='black'`: Adds a black border around each bar (makes the bars clearer).

```python
plt.xlabel('Sepal Length (cm)')
```

- Adds a label to the x-axis.

```python
plt.ylabel('Frequency')
```

- Adds a label to the y-axis.

```python
plt.title('Histogram of Sepal Length')
```

- Adds a title to the histogram.

```python
plt.show()
```

- Displays the plot.

⚠️ Important Note:

If you get an error like:

```vbnet
KeyError: 'sepal_length'
```

It probably means the actual column name is different — maybe `'SepalLengthCm'` (from the Iris dataset), **not** `'sepal_length'`.

✅ Try replacing:

```python
df['sepal_length']
```

with:

```python
df['SepalLengthCm']
```

✅ Final Working Code (for Iris dataset):

```python
import matplotlib.pyplot as plt

plt.hist(df['SepalLengthCm'], bins=20, color='skyblue', edgecolor='black')

plt.xlabel('Sepal Length (cm)')
plt.ylabel('Frequency')
plt.title('Histogram of Sepal Length')

plt.show()
```

Let me know if you'd like to plot other columns or switch to box plots, KDE, etc.

alpha=0.5 ?

Great question!

🔹 `alpha=0.5` in matplotlib:

The `alpha` parameter controls the **transparency** (or **opacity**) of plot elements in Matplotlib.

- `alpha=1.0` → fully **opaque** (default)
- `alpha=0.0` → fully **transparent**
- `alpha=0.5` → **50% transparent**

✅ Use in your histogram:

If you want to make the bars semi-transparent, modify your histogram like this:

```python
plt.hist(df['SepalLengthCm'], bins=20, color='skyblue', edgecolor='black', alpha=0.5)
```

🎯 Why use `alpha`?

- It helps when **overlaying multiple plots** (e.g., comparing distributions).
- Makes visualizations **softer** and **easier to read**.
- Helps reveal overlapping elements in plots.

👀 Visual Effect:

A histogram with `alpha=0.5` will look lighter than one with the default opacity. If you plot another histogram on top (say, for another class or species), you'll be able to see both overlapping histograms more clearly.

Let me know if you want to see how it works with multiple histograms (e.g., one per species)!

```
import seaborn as sns

from seaborn import load_dataset
df = load_dataset("iris")

df_long = df.melt(id_vars=["species"], var_name="measurement", value_name="size")

plt.figure(figsize=(10, 6))
sns.boxplot(x="measurement", y="size", hue="species", data=df_long)

plt.xticks(rotation=45)
plt.title("Boxplot of Iris Features by Species")
plt.show()
```

This code creates a **grouped boxplot** using **Seaborn** to visually compare the distributions of **Iris flower measurements** (like sepal length, petal width, etc.) across different **species**.

🔍 Code Breakdown

🔹 `import seaborn as sns`

Imports the Seaborn visualization library (built on top of Matplotlib).

🔹 `from seaborn import load_dataset`

Allows you to load built-in datasets from Seaborn.

🔹 `df = load_dataset("iris")`

Loads the classic **Iris dataset** as a DataFrame named `df`.

It contains:

- 150 rows
- 4 features: `sepal_length`, `sepal_width`, `petal_length`, `petal_width`
- Target: `species` (Setosa, Versicolor, Virginica)

🔹 `df_long = df.melt(id_vars=["species"], var_name="measurement", value_name="size")`

This reshapes the data from **wide** to **long** format.

Before (`df`)	After (`df_long`)
sepal_length	sepal_width...
--------------	----------------
5.1	3.5
5.1	3.5
...	...

This long format is ideal for **grouped plots** like boxplots.

🔹 `plt.figure(figsize=(10, 6))`

Sets the figure size to 10 inches wide and 6 inches tall.

🔹 `sns.boxplot(x="measurement", y="size", hue="species", data=df_long)`

Creates a **boxplot**:

- **X-axis**: Measurement type (`sepal_length`, `petal_width`, etc.)
- **Y-axis**: Value (actual size in cm)
- **Hue**: Color grouping by `species` (Setosa, Versicolor, Virginica)
- Each measurement is split by species to show **distribution differences**

🔹 `plt.xticks(rotation=45)`

Rotates the x-axis labels 45° for better readability.

🔹 `plt.title("Boxplot of Iris Features by Species")`

Adds a title to the plot.

🔹 `plt.show()`

Displays the plot.

📈 Result:

OpenAI

You get a clean and informative **grouped boxplot** showing:

- How each species varies for each measurement
- Distribution, median, outliers, etc.

✅ Great for:

- **Comparing distributions**
- **Spotting outliers**
- **Identifying feature importance** visually

Let me know if you want to annotate or customize it further!

df.head() df.info()

Here's what these two commonly used pandas methods do:

🔹 `df.head()`

This displays the **first 5 rows** of the DataFrame `df` by default.

- Helps you quickly preview the data.
- Useful for checking column names, value formats, and structure.

✅ Example output (from the Iris dataset):

```python
   sepal_length  sepal_width  petal_length  petal_width    species
0           5.1          3.5           1.4          0.2     setosa
1           4.9          3.0           1.4          0.2     setosa
2           4.7          3.2           1.3          0.2     setosa
3           4.6          3.1           1.5          0.2     setosa
4           5.0          3.6           1.4          0.2     setosa
```

You can show more rows by specifying a number:

```python
df.head(10)  # Shows first 10 rows
```

🔹 `df.info()`

This shows a **summary of the DataFrame**, including:

- Index range

- Number of non-null values per column
- Data types of each column
- Memory usage

✅ Example output:

```text
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

This is useful for:

- Checking for **missing values**
- Verifying **data types** before modeling
- Understanding the dataset's structure at a glance

Let me know if you'd like to check for null values or convert any columns!

```
from sklearn.neighbors import KNeighborsClassifier

knn_distance = KNeighborsClassifier(n_neighbors=3, weights='distance')
knn_distance.fit(X, y)
y_pred_distance = knn_distance.predict(X)

accuracy_distance = calculate_accuracy(y, y_pred_distance)
print(f"Accuracy with Distance Weights: {accuracy_distance:.2f}%")

knn_manhattan = KNeighborsClassifier(n_neighbors=3, weights='uniform', p=1)
knn_manhattan.fit(X, y)
y_pred_manhattan = knn_manhattan.predict(X)

accuracy_manhattan = calculate_accuracy(y, y_pred_manhattan)
print(f"Accuracy with Manhattan Distance: {accuracy_manhattan:.2f}%")
```

This code trains and evaluates two **K-Nearest Neighbors (KNN)** classifiers using different configurations:

🔍 Overview

It uses:

- **Distance-weighted KNN** (closer neighbors have more influence)
- **Manhattan distance** (L1 norm) instead of default Euclidean (L2)

It also uses a function `calculate_accuracy()` (you must have defined this earlier) to evaluate performance.

🔢 Step-by-Step Explanation

1. Importing KNN classifier

```python
from sklearn.neighbors import KNeighborsClassifier
```

This imports the `KNeighborsClassifier` from scikit-learn.

---

2. Distance-weighted KNN classifier

```python
knn_distance = KNeighborsClassifier(n_neighbors=3, weights='distance')
```

- `n_neighbors=3`: Use 3 nearest neighbors.
- `weights='distance'`: Neighbors closer to the query point have more influence.
- Uses **Euclidean distance** by default.

```python
knn_distance.fit(X, y)
y_pred_distance = knn_distance.predict(X)
```

- Fits the model to feature data `X` and labels `y`.
- Predicts the labels for the same dataset `X` (⚠️ no train-test split here).

```python
accuracy_distance = calculate_accuracy(y, y_pred_distance)
print(f"Accuracy with Distance Weights: {accuracy_distance:.2f}%")
```

- Calculates accuracy of predictions and prints it.

---

3. Uniform-weighted KNN using Manhattan Distance

```python
knn_manhattan = KNeighborsClassifier(n_neighbors=3, weights='uniform', p=1)
```

- `weights='uniform'`: All neighbors contribute equally (default setting).
- `p=1`: Use **Manhattan Distance** (sum of absolute differences), instead of default `p=2` (Euclidean).

```python
knn_manhattan.fit(X, y)
y_pred_manhattan = knn_manhattan.predict(X)
```

- Trains the model and predicts labels.

```python
accuracy_manhattan = calculate_accuracy(y, y_pred_manhattan)
print(f"Accuracy with Manhattan Distance: {accuracy_manhattan:.2f}%")
```

- Computes and prints accuracy.

⚠️ Important Notes

- **You're evaluating on the training set itself**. This usually gives artificially high accuracy.
  - ✅ Use `train_test_split(X, y)` for proper validation.
- Make sure `calculate_accuracy()` is defined like:

```python
from sklearn.metrics import accuracy_score

def calculate_accuracy(true, pred):
    return accuracy_score(true, pred) * 100
```

✅ Example Output (on training set):

```mathematica
Accuracy with Distance Weights: 100.00%
Accuracy with Manhattan Distance: 98.67%
```

(Would be lower on a test set.)

Let me know if you want help modifying this to include proper train-test splitting or confusion matrix evaluation!

plt.xticks(range(1, 21))

```python
plt.xticks(range(1, 21))
```

This line sets the **x-axis tick marks** (labels) to show **integers from 1 to 20**.

🔍 When is it used?

Usually used in plots where the x-axis represents a sequence of integers — for example:

- Number of neighbors `k` in KNN
- Epochs in training
- Some kind of parameter sweep or index-based plot

✅ Example Use Case (KNN Accuracy vs k):

```python
import matplotlib.pyplot as plt

k_values = list(range(1, 21))
accuracies = [0.93, 0.94, 0.95, ..., 0.97]  # example values

plt.plot(k_values, accuracies)
plt.xticks(range(1, 21))  # Ensures all integer k-values are labeled
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Accuracy")
plt.title("KNN Accuracy vs. Number of Neighbors")
plt.grid(True)
plt.show()
```

📝 Notes:

- Without `plt.xticks(...)`, matplotlib may **auto-skip** some ticks for clarity.
- You can also rotate or style tick labels:

  ```python
  plt.xticks(range(1, 21), rotation=45, fontsize=10)
  ```

Let me know what kind of plot you're using this with — I can tailor the explanation further!

accuracy_df = pd.DataFrame(list(accuracy_results.items()), columns=['k', 'Accuracy'])

This line of code creates a **Pandas DataFrame** from a dictionary called `accuracy_results` and gives it two column names: `'k'` and `'Accuracy'`.

🔍 Explanation:

```python
accuracy_df = pd.DataFrame(list(accuracy_results.items()), columns=['k', 'Accuracy'])
```

🔹 `accuracy_results`

This is expected to be a **Python dictionary** where:

- The **key** is `k` (number of neighbors in KNN)
- The **value** is the corresponding **accuracy** (likely a float or percentage)

📌 Example:

```python
accuracy_results = {
    1: 95.0,
    2: 96.67,
    3: 97.33,
    4: 96.00,
    ...
}
```

🔹 `accuracy_results.items()`

Returns a list of tuples like:

```python
[(1, 95.0), (2, 96.67), (3, 97.33), ...]
```

🔹 `pd.DataFrame(..., columns=['k', 'Accuracy'])`

- Creates a **DataFrame** with two columns:
  - `'k'`: number of neighbors
  - `'Accuracy'`: corresponding accuracy
- Useful for plotting or analysis.

✅ Sample output:

k	Accuracy
1	95.00
2	96.67
3	97.33

💡 What can you do next?

```python
# View it
print(accuracy_df)

# Plot accuracy vs. k
import matplotlib.pyplot as plt

plt.plot(accuracy_df['k'], accuracy_df['Accuracy'], marker='o')
plt.xticks(range(1, 21))
plt.xlabel('k (Number of Neighbors)')
plt.ylabel('Accuracy (%)')
plt.title('KNN Accuracy vs. k')
plt.grid(True)
plt.show()
```

Let me know if you'd like to add weights, test/train comparison, or error rate!

Explain the following topic in detailed : KNN
Linear Regression
Scalers
Regularisation and Gradient Descent
Batch Gradient Descent
Stochastic Gradient Descent
Mini-Batch Gradient Descent
Decision Tree
SVM Machines and Kernels
Error-matrics
MSE and RMSE
Accuracy Metrics

Here's a **detailed explanation** of each topic you've listed. These are foundational concepts in Machine Learning and essential for both theory and practical applications.

🔷 1. K-Nearest Neighbors (KNN)

KNN is a **supervised learning** algorithm used for **classification** and **regression**.

- It classifies a new data point based on the **majority label** of the **k closest training points** (using a distance metric like Euclidean or Manhattan).
- It is a **lazy learner** (no training phase) — it stores the dataset and makes predictions at runtime.

📌 Example: Classifying an unknown flower based on its petal and sepal measurements by comparing it to 3 or 5 nearby known samples.

🔷 2. Linear Regression

Linear Regression is used to model the **linear relationship** between a **dependent variable** and one or more **independent variables**.

Equation:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

- Simple Linear Regression: one feature
- Multiple Linear Regression: multiple features
- Tries to minimize **Mean Squared Error (MSE)**

🔷 3. Scalers (Feature Scaling)

Scaling ensures all features contribute equally to the model.

Common scalers:

- `StandardScaler`: Rescales data to mean = 0 and std = 1
- `MinMaxScaler`: Rescales data to a fixed range [0, 1]
- `RobustScaler`: Uses median and IQR (robust to outliers)

✅ Needed for algorithms like KNN, SVM, and gradient descent-based models.

🔷 4. Regularization & Gradient Descent

Regularization:

Prevents **overfitting** by penalizing large weights in the model.

- **L1 (Lasso)**: Adds absolute value of weights (`|w|`)
- **L2 (Ridge)**: Adds squared weights (`w²`)

Modified cost function:

$$J(w) = MSE + \lambda \cdot \text{regularization term}$$

Gradient Descent:

An optimization algorithm used to minimize the **loss function** by updating weights iteratively:

$$w = w - \alpha \cdot \frac{\partial J(w)}{\partial w}$$

- `α` = learning rate
- `J(w)` = cost function

🔷 5. Batch Gradient Descent

- Uses the **entire dataset** to compute the gradient before each weight update.
- Very stable but can be **slow for large datasets**.

🔷 6. Stochastic Gradient Descent (SGD)

- Updates weights **for each training example**, one at a time.
- Faster and allows escaping local minima, but more **noisy** and less stable.

🔷 7. Mini-Batch Gradient Descent

- A compromise between Batch and SGD.
- Uses **small random batches** (e.g., 32, 64, 128 samples) for each update.
- Offers a **balance of speed and stability**.

🔷 8. Decision Tree

A **tree-like model** of decisions and their consequences.

- Internal nodes = features
- Branches = decisions (e.g., feature value thresholds)
- Leaves = outcomes (class labels or values)

Key concepts:

- **Gini Index** or **Entropy** to measure impurity
- **Pruning** helps prevent overfitting

✅ Easy to visualize and interpret.

🔷 9. SVM (Support Vector Machines) and Kernels

SVM:

Supervised algorithm that finds the **hyperplane** that best separates classes with the **maximum margin**.

- Works well for **binary classification** problems.

Kernels:

Used to project data into **higher dimensions** where it is linearly separable.

- **Linear Kernel**
- **Polynomial Kernel**
- **RBF (Radial Basis Function)** or **Gaussian Kernel**

✅ Useful for **non-linear data**.

🔷 10. Error Metrics

Used to evaluate how well a model is performing.

🔸 MSE (Mean Squared Error)

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

- Penalizes large errors heavily
- Common in regression

🔸 RMSE (Root Mean Squared Error)

$$RMSE = \sqrt{MSE}$$

- Same unit as output variable
- Easier to interpret than MSE

🔷 11. Accuracy Metrics (Classification)

Used for evaluating classification models.

Common metrics:

- **Accuracy**: % of correct predictions

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

- **Precision**: How many predicted positives are correct

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

- **Recall**: How many actual positives are correctly predicted

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

- **F1-Score**: Harmonic mean of precision and recall

- **Confusion Matrix**: Matrix of actual vs predicted labels

📊 Summary Table

Topic	Type	Use
KNN	Classification/Regression	Simple, instance-based
Linear Regression	Regression	Predicting numeric outputs
Scalers	Preprocessing	Normalize data for models
Regularization	Regression	Prevents overfitting
Gradient Descent	Optimization	Updates model parameters
Batch / Mini / SGD	Optimization methods	Trade-offs in speed and stability
Decision Tree	Classification/Regression	Tree-based model, interpretable
SVM	Classification	Max-margin classifier
Kernels	SVM enhancement	For non-linear boundaries
MSE/RMSE	Regression metrics	Measure prediction error
Accuracy/Precision/Recall	Classification metrics	Evaluate model performance

---

Let me know if you want visual illustrations or code examples for any of these topics!