Module-2.2 Data Preprocessing & Train–Test Split

Before we touch any ML algorithm—Linear Regression, Random Forest, Neural Networks—there is one thing that decides everything: **data**.

If you give **Garbage** to the **ML** model you will get **Garbage**.

1. Types of Data

1.1 High-level categories

We broadly deal with:

- **Numerical data**
- **Categorical data**
- **Text**
- **Images**
- **Time series**

Why do we care about types?

Different types need **different preprocessing**.

- Numerical → scaling, imputation with mean/median
- Categorical → encoding (one-hot, label encoding)
- Text → tokenization, embeddings
- Image → resizing, normalization
  If we treat everything the same, models will misinterpret the information.

1.1.1 Numerical Data

- **Continuous:** real-valued, can take many possible values.
  - Examples: height, weight, temperature, house price.
- **Discrete:** counts, integer-valued.
  - Examples: number of children, number of logins.

1.1.2 Categorical Data

- **Nominal:** categories without order
  - Example: city = {Tokyo, Delhi, Mumbai}, color = {red, blue, green}
- **Ordinal:** categories **with** order
  - Example: education level (10th < 12th < B.Tech < M.Tech < PhD), rating (bad < ok < good < excellent)

1.1.3 Text & Image

- Text: sequence of words/characters. Need NLP pipeline.
- Image: grid of pixels, usually represented as tensor (H × W × C).

NOTE: “Everything must eventually be converted into **numbers** for the model.”

1.1.4 Simple pandas view

```
import pandas as pd

df = pd.DataFrame({
    "age": [25, 32, 40],
    "salary": [40000, 55000, 70000],
    "gender": ["M", "F", "M"],
    "review": ["Good product", "Very bad experience", "Average"],
})

df
```

✓ 0.0s

	age	salary	gender	review
0	25	40000	M	Good product
1	32	55000	F	Very bad experience
2	40	70000	M	Average

2. Data Cleaning

We’ll look at:

1. Missing values

2. Outliers

3. Duplicates

Why clean data at all?

- ML algorithms are **mathematical functions**; they usually assume clean numeric input.

- Missing values, extreme errors, and duplicates can:

    - Distort statistics (mean, variance).

    - Mislead the model into wrong relationships.

    - Give **over-optimistic** evaluation if duplicates exist across train and test.

So cleaning is not “nice-to-have”; it’s **necessary for reliable models**.

2.1 Missing Values

Index	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	nan	6
3	4	Female	23	16	77
4	5	Female	nan	17	40
5	6	Female	22	17	76
6	7	Female	35	18	6
7	8	nan	23	18	94
8	9	Male	64	19	3
9	10	Female	30	19	72

2.1.1 Why values go missing?

Real-world reasons:

- Users skip form fields.
- Sensor downtime.
- Data entry mistakes.
- Some fields are not relevant for everyone.

2.1.2 Detecting missing in pandas

```python
import numpy as np
import pandas as pd

df = pd.DataFrame({
    "age": [25, 30, 35, 40, 28, 45, 50, 29, 29, 38],
    "salary": [15000, np.nan, 29000, 9000, 40000, 61000, np.nan, 38000, 38000, 38000],
    "gender": ["M", "F", "M", "F", "M", "F", "F", "M", "M", "M"],
})
print(df.isna().sum())
```

✓ 0.0s

```
age       0
salary    2
gender    0
dtype: int64
```

We first **quantify how bad the problem is**:

- If 1–2% values are missing → small problem.
- If 70% values missing in a column → maybe that column is useless.

2.1.4 Methods to handle missing values

(A) Drop rows / columns

```python
# Drop any row with at least one NaN
df_drop_rows = df.dropna()
print(df_drop_rows)
print("------------------------------")

# Drop columns with too many missing values (>60%)
missing_ratio = df.isna().mean()
cols_to_drop = missing_ratio[missing_ratio > 0.6].index
df_drop_cols = df.drop(columns=cols_to_drop)
print(df_drop_cols)
print("------------------------------")

print("missing ration:\n ",missing_ratio)
```

✓ 0.0s

```
   age   salary gender
0   25  15000.0      M
2   35  29000.0      M
3   40   9000.0      F
4   28  40000.0      M
5   45  61000.0      F
7   29  38000.0      M
8   29  38000.0      M
9   38  38000.0      M
------------------------------
   age   salary gender
0   25  15000.0      M
1   30      NaN      F
2   35  29000.0      M
3   40   9000.0      F
4   28  40000.0      M
5   45  61000.0      F
6   50      NaN      F
7   29  38000.0      M
8   29  38000.0      M
9   38  38000.0      M
------------------------------
missing ration:
  age       0.0
salary    0.2
gender    0.0
dtype: float64
```

Why do this?

- When **very few** rows have missing values, dropping them doesn't hurt much and avoids the complexity of imputation.
- When a column is **mostly missing**, imputing it is basically "guessing"; dropping it avoids noise.

When NOT to do this?

- When dataset is small; dropping reduces sample size and increases variance.

(B) Simple Imputation (mean / median / mode)

For numeric:

```python
from sklearn.impute import SimpleImputer

num_imputer = SimpleImputer(strategy="median")
df[["salary"]] = num_imputer.fit_transform(df[["salary"]])
print(df[["salary"]])
```

✓ 0.0s

```
    salary
0  15000.0
1  38000.0
2  29000.0
3   9000.0
4  40000.0
5  61000.0
6  38000.0
7  38000.0
8  38000.0
9  38000.0
```

For categorical:

```
cat_imputer = SimpleImputer(strategy="most_frequent")
df[["gender"]] = cat_imputer.fit_transform(df[["gender"]])
print(df[["gender"]])
```

✓ 0.0s

```
  gender
0      M
1      F
2      M
3      F
4      M
5      F
6      F
7      M
8      M
9      M
```

(C) Group-wise imputation

```
df["salary"] = df.groupby("gender")["salary"].transform(
    lambda s: s.fillna(s.median())
)
print(df["salary"])
```

✓ 0.0s

```
0    15000.0
1    38000.0
2    29000.0
3     9000.0
4    40000.0
5    61000.0
6    38000.0
7    38000.0
8    38000.0
9    38000.0
Name: salary, dtype: float64
```

- Salary distribution for “M” vs “F” might be different.
- Imputing global median ignores these differences.
- Group-wise imputation keeps **conditional structure**:
  P(salary | gender) better approximated than P(salary) alone.

(D) KNN

```
from sklearn.impute import KNNImputer


df_numeric = df.select_dtypes(include=['int64', 'float64'])
knn_imputer = KNNImputer(n_neighbors=3)
df_imputed = pd.DataFrame(
    knn_imputer.fit_transform(df_numeric),
    # knn_imputer.fit_transform(df[["age", "salary"]]),
    columns=df_numeric.columns
    # columns=df[["age", "salary"]]
)

df["salary"] = df_imputed["salary"]
print(df["salary"])
```

✓ 0.0s

```
0    15000.0
1    38000.0
2    29000.0
3     9000.0
4    40000.0
5    61000.0
6    38000.0
7    38000.0
8    38000.0
9    38000.0
Name: salary, dtype: float64
```

- We assume similar rows (neighbors in feature space) have similar values.
- KNN imputation captures **non-linear relationships** between variables.
- More powerful than simple mean/median, but more computationally expensive, and can overfit if dataset is small.

2.2 Outliers

2.2.1 What are outliers?

Values that are **very far** from the rest of the data.

Example:

- Salary: mostly 3–10 LPA, one entry 500 LPA (typo or CEO).
- Age: 0, 1, 2, ..., 120 are realistic; 1000 is not.

- [ ] Outliers severely affect statistics like mean and standard deviation.
- [ ] Algorithms like Linear Regression or K-Means can get heavily skewed by a few extreme points.
- [ ] But some outliers are **important events** (fraud transactions, anomalies), so we can't blindly remove them.

2.2.2 Detection methods

(A) Z-score method

For feature x:

$$z_i = \frac{x_i - \mu}{\sigma}$$

If |z_i| > 3, consider xi as an outlier

(rule of thumb for roughly normal data).

```
x = df["salary"]
mu = x.mean()
sigma = x.std()

z_scores = (x - mu) / sigma
print(z_scores)
outliers_z = np.abs(z_scores) > 3
df_outliers = df[outliers_z]
print(df_outliers)
df
```

✓ 0.0s

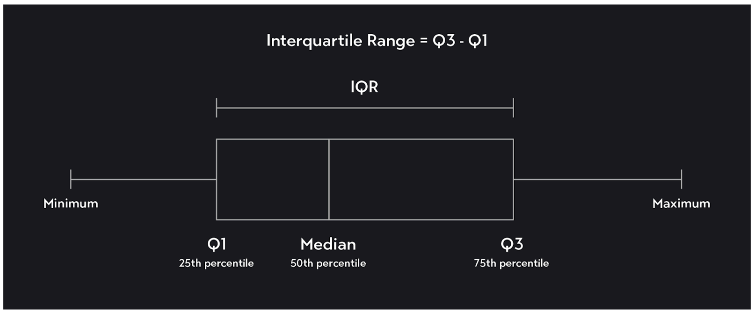
```
0   -0.317430
1   -0.315974
2   -0.316544
3   -0.317810
4   -0.315848
5   -0.314519
6   -0.315974
7   -0.315974
8   -0.315974
9    2.846049
Name: salary, dtype: float64
Empty DataFrame
Columns: [age, salary, gender]
Index: []
```

	age	salary	gender
0	25	15000.0	M
1	30	38000.0	F
2	35	29000.0	M
3	40	9000.0	F
4	28	40000.0	M
5	45	61000.0	F
6	50	38000.0	F
7	29	38000.0	M
8	29	38000.0	M
9	38	50000000.0	M

Z-score sometimes fails to detect extreme salary outliers, especially when the dataset is small and the distribution is skewed. Because z-score depends on mean and standard deviation, even one very large value can increase the standard deviation and reduce its own z-score.
So for small datasets and non-normal features like salary, IQR is usually a better choice.

- For a normal distribution, ~99.7% data lies within $\pm 3\sigma$.
- So anything beyond that is “very unlikely” under normal assumption.

(B) IQR method (more robust)



Compute:

- Q1=25th percentile
- Q3=75th percentile
- IQR = Q3−Q1

Then:

$\text{lower} = Q1 - 1.5 \cdot IQR, \quad \text{upper} = Q3 + 1.5 \cdot IQR$

Values outside [lower, upper] are flagged as outliers.

```python
x = df["salary"]
q1 = x.quantile(0.25)
q3 = x.quantile(0.75)
median = x.quantile(0.50)

iqr = q3 - q1

lower = q1 - 1.5 * iqr
upper = q3 + 1.5 * iqr

outliers_iqr = (x < lower) | (x > upper)
df_outliers_iqr = df[outliers_iqr]

print(median, q1, q3, lower, upper)
print("-----------------------------")
print(outliers_iqr)
print("-----------------------------")
print(df_outliers_iqr)
```

```
38000.0 31250.0 39500.0 18875.0 51875.0
-----------------------------
0     True
1    False
2    False
3     True
4    False
5     True
6    False
7    False
8    False
9     True
Name: salary, dtype: bool
-----------------------------
   age      salary gender
0   25     15000.0      M
3   40      9000.0      F
5   45     61000.0      F
9   38  50000000.0      M
```

- IQR uses **median-based** stats (Q1 and Q3), which are robust to extreme values.
- Works better than z-score when data is ***skewed or not normal***.

2.2.3 What to do with outliers?

Options:

1. **Correct obvious errors**
    - Age 300 → 30 (if you're sure it's a typo).
2. **Clip**

```python
df["salary_clipped"] = x.clip(lower, upper)
print(df["salary_clipped"])
```

✓ 0.0s

```
0    18875.0
1    38000.0
2    29000.0
3    18875.0
4    40000.0
5    51875.0
6    38000.0
7    38000.0
8    38000.0
9    51875.0
Name: salary_clipped, dtype: float64
```

3. **Remove rows**

```python
df_clean = df[~outliers_iqr]
df_clean
```

✓ 0.0s

	age	salary	gender	salary_clipped
1	30	38000.0	F	38000.0
2	35	29000.0	M	29000.0
4	28	40000.0	M	40000.0
6	50	38000.0	F	38000.0
7	29	38000.0	M	38000.0
8	29	38000.0	M	38000.0

4. **Keep them** if they are real, important signals.

2.3 Duplicates

2.3.1 Types

1. **Exact duplicate rows** (every column same).

```python
duplicate_rows = df[df.duplicated(keep=False)]
df_no_dup = df.drop_duplicates()
duplicate_rows
```

✓ 0.0s

	age	salary	gender	salary_clipped
7	29	38000.0	M	38000.0
8	29	38000.0	M	38000.0

2. **Entity duplicates** (same person twice with slight variation).

- [ ] Duplicates **inflate the importance** of those rows ⇒ model thinks they're more common.

- [ ] If duplicates appear in both train and test, the model has effectively **seen** test data during training, causing **over-optimistic accuracy**.

3. Train / Validation / Test Split

3.1 Why split the data?

We want model performance on **unseen data** (generalization).

But we only have a finite dataset. If we train and evaluate on the same data:

- Model could **memorize** the dataset and perform unrealistically well

3.2 Typical split ratios

- **Train:** 60–80%
- **Validation:** 10–20%
- **Test:** 10–20%

Example:

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_diabetes

data = load_diabetes()
X, y = data.data, data.target

# Train + temp
X_train, X_temp, y_train, y_temp = train_test_split( X, y, test_size=0.3, random_state=42)

# Temp -> val + test
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

print(X_train.shape, X_val.shape, X_test.shape)
```

✓ 0.0s

```
(309, 10) (66, 10) (67, 10)
```

- First split separates **learning** data from **evaluation** data.
- Second split separates evaluation into:
    - Validation: to choose best model.
    - Test: untouched, only for final evaluation.

4. Data Leakage

4.1 What is data leakage?

Data leakage happens when the model has access to information during training that it **wouldn't have in real-world prediction time**.

Examples:

- Using future info for prediction.
- Using target or post-target info to build features.
- Doing preprocessing on a full dataset (train+test) before splitting.

Real-life examples

1. Using future info for prediction

Scenario: Electricity load forecasting
You want to predict **tomorrow's load**.

Leakage mistake:
While creating features, you accidentally include:

- tomorrow's temperature
- or even tomorrow's load-related signals

Why it's leakage?
Because in real life **you don't have tomorrow's real data today**.
So your model is learning with future truth.

2. Leakage via preprocessing

If you scale the whole dataset before splitting:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i,\ \hat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2}$$

Then, transforming test data:

$$x_i' = \frac{x_i - \hat{\mu}}{\hat{\sigma}}$$

Here, μ^,σ^ depend on **test points themselves**, so test data is influencing its own transformation.

Correct way:
Compute μ^,σ^ using **only train data**, then **apply** to val/test.

4.3 Wrong vs Right code

❌ **Wrong:**

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

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

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # using full X (train + test) ❌

model = LinearRegression()
model.fit(X_scaled[:len(X_train)], y_train)
y_pred = model.predict(X_scaled[len(X_train):])
```

✅ **Right: use Pipeline**

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("model", LinearRegression())
])

pipe.fit(X_train, y_train)       # scaler.fit on X_train only ✅
y_pred = pipe.predict(X_test)    # scaler.transform on X_test
```

5. Temporal Splits (Time Series)

5.1 Why time series is special?

In time series, **order matters**.
You can't predict 2020 using information from 2021.

> Shuffling time and doing random train_test_split will mix future and past. That is unrealistic and creates temporal leakage.

```python
# Let's make 72 hourly samples (~3 days)
timestamps = pd.date_range("2024-01-01 00:00:00", periods=72, freq="h")

df = pd.DataFrame({"timestamp": timestamps})
df
```

✓ 0.0s

	timestamp
0	2024-01-01 00:00:00
1	2024-01-01 01:00:00
2	2024-01-01 02:00:00
3	2024-01-01 03:00:00
4	2024-01-01 04:00:00
...	...
67	2024-01-03 19:00:00
68	2024-01-03 20:00:00
69	2024-01-03 21:00:00
70	2024-01-03 22:00:00
71	2024-01-03 23:00:00

72 rows × 1 columns

```python
df["hour_of_day"] = df["timestamp"].dt.hour
df["day_of_week"] = df["timestamp"].dt.dayofweek
df.sample(3)
```

✓ 0.0s

	timestamp	hour_of_day	day_of_week
15	2024-01-01 15:00:00	15	0
41	2024-01-02 17:00:00	17	1
54	2024-01-03 06:00:00	6	2

```python
# Simple synthetic temperature pattern + noise
df["temp"] = 18 + 6*np.sin(2*np.pi*df["hour_of_day"]/24) + np.random.normal(0, 0.7, len(df))

# Synthetic load depends on temp + time-of-day + noise
df["load_kwh"] = (
    5000
    + 120*df["hour_of_day"]  # daily ramp effect
    + 80*df["temp"]          # temperature sensitivity
    + np.random.normal(0, 120, len(df))
)

df = df.sort_values("timestamp").reset_index(drop=True)

df.head(5)
```

✓ 0.0s

	timestamp	hour_of_day	day_of_week	temp	load_kwh
0	2024-01-01 00:00:00	0	0	17.706756	6255.178222
1	2024-01-01 01:00:00	1	0	19.215673	6612.833363
2	2024-01-01 02:00:00	2	0	22.138564	6970.690363
3	2024-01-01 03:00:00	3	0	20.505734	7042.669297
4	2024-01-01 04:00:00	4	0	23.555837	7353.455846

5.2 TimeSeriesSplit

```python
from sklearn.model_selection import TimeSeriesSplit

X = df[["temp", "hour_of_day", "day_of_week"]].values
y = df["load_kwh"].values

tscv = TimeSeriesSplit(n_splits=3)

for fold, (train_idx, val_idx) in enumerate(tscv.split(X)):
    print(f"Fold {fold}")
    print("Train:", train_idx[0], "->", train_idx[-1])
    print("Val:  ", val_idx[0], "->", val_idx[-1])
```

✓ 0.0s

```
Fold 0
Train: 0 -> 17
Val:   18 -> 35
Fold 1
Train: 0 -> 35
Val:   36 -> 53
Fold 2
Train: 0 -> 53
Val:   54 -> 71
```

- Each fold uses **earlier data to predict later data**.
- This respects temporal order and gives a more reliable estimate of future performance.
