Outline

The problem with NaN

Real datasets have holes. A single NaN breaks math operations downstream.

Running example: Yerevan apartment rent. Some rows have unknown balcony, others have unknown year_built.

```
>>> df.head()
   area  rooms  district     year_built  balcony  rent
0  45.0     2    Kentron        1985        Yes     280
1  62.0     3    Arabkir          NaN        No     340
2   NaN     2    Komitas        1992        NaN     310
3  38.0     1    Kentron        1978        No      245

>>> (df["area"] * 1000).mean()
nan                          # one NaN poisons the whole column
```

Before any model: detect, decide, document.

Types of missingness

Why data is missing matters more than how much. Plain-English first, statistics names in parentheses:

- **Innocent missing (MCAR).** The reason a value is missing has nothing to do with any other variable. Example: random data-entry hiccup. *Safe to drop or impute.*
- **Structured missing (MAR).** Missingness depends on *other observed* variables. Example: older buildings less likely to have `year_built` recorded - missingness depends on the (observable) age. *A model can recover it from the other columns.*
- **Biased missing (MNAR).** Missingness depends on the *value itself*. Example: tenants don't report rent when it's unusually high - the value affects whether it's reported. *Dangerous - imputation will be biased; need domain reasoning.*

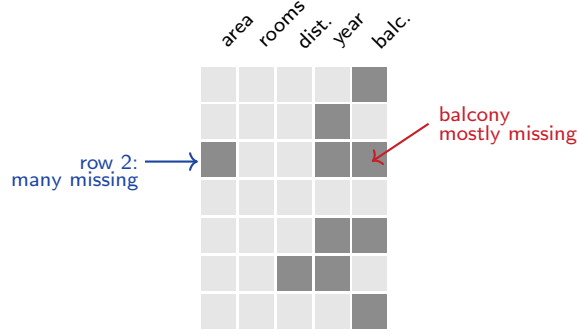
You can't fully distinguish these from data alone. You can usually rule things out by talking to whoever collected the data.

Detect: visualize the pattern

```
df.isna().sum()                    # per column
df.isna().mean() * 100             # percent
df.isna().any(axis=1).mean()       # row-wise fraction
import missingno as msno
msno.matrix(df)                    # heatmap
```

Look for two patterns:

- A **column** mostly missing ⇒ consider dropping it.
- A **row** with many missing fields ⇒ MAR pattern likely.



Strategy 1 & 2: drop vs. simple impute

Drop - cheap, sometimes wrong:

```
df.dropna()                  # any
    NaN
df.dropna(thresh=4)          # keep
    rows w/ >= 4 non-NaN
df.dropna(axis=1)            # drop
    NaN columns
```

- Use when missing rate is small ($< 5\%$) AND MCAR-ish.
- Risk: throwing away signal. If `balcony` is missing for 30% of rows, dropping them loses a third of the data.

Simple impute - fast baseline:

```
from sklearn.impute import
    SimpleImputer

num = SimpleImputer(strategy="
    median")
cat = SimpleImputer(strategy="
    most_frequent")
df["area"] = num.fit_transform(df
    [["area"]])
```

- Numeric: median (robust to outliers), mean (only if symmetric).
- Categorical: mode.
- Constants: `fill_value=0` or `"Unknown"`.

Predict-first: what does mean imputation hide?

You have area data for 100 apartments. 20 values are missing. You fill the 20 holes with the mean of the other 80.

Predict-first: what does mean imputation hide?

You have area data for 100 apartments. 20 values are missing. You fill the 20 holes with the mean of the other 80.

Predict before reading on: after imputation,

- What happens to the *variance* of the column?
- What happens to its *correlation* with rent?
- What happens to a model's reported *confidence*?

Predict-first: what does mean imputation hide?

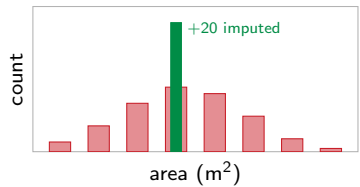
You have area data for 100 apartments. 20 values are missing. You fill the 20 holes with the mean of the other 80.

Predict before reading on: after imputation,

- What happens to the *variance* of the column?
- What happens to its *correlation* with rent?
- What happens to a model's reported *confidence*?

- **Variance shrinks** - the 20 imputed values are identical, so they contribute 0 to spread.
- **Correlation attenuates** - 20 rows say "I am the mean" regardless of rent.
- **Confidence is overstated** - the model treats imputed values as real measurements.

Mean imputation is a fine baseline, but it lies about uncertainty.



Green spike at the mean = lost variance.

Smarter imputation + the indicator trick

Model-based imputation. Predict each missing value from the other features.

```
from sklearn.impute import KNNImputer, IterativeImputer
KNNImputer(n_neighbors=5).fit_transform(X)
IterativeImputer(random_state=509).fit_transform(X)    # ~MICE
```

- KNN: find the 5 most similar rows, average their value.
- Iterative (MICE-style): regress each column on the others, loop until stable.
- Cost: slower, can leak if not fit on train only.

Missingness as a feature. Add a binary indicator column:

```
df["balcony_missing"] = df["balcony"].isna().astype(int)
df["balcony"] = df["balcony"].fillna("Unknown")
```

Why: *the fact that something is missing* can itself be predictive. A landlord who didn't report `year_built` may be hiding a very old building.

Two more cleanups: duplicates and outliers

Duplicates.

```
df.duplicated().sum()              # count
df.drop_duplicates()               # exact
df.drop_duplicates(                # subset
    subset=["area","rooms","district"])
```

- Exact duplicates: usually safe to drop.
- *Near*-duplicates (same apartment listed twice with tiny price diff): need a domain rule.
- Sometimes duplicates are real (two units of one model car) - check before dropping.

Outliers.

```
# IQR rule (Tukey)
q1, q3 = df["rent"].quantile([.25, .75])
iqr = q3 - q1
mask = df["rent"].between(q1 - 1.5*iqr,
                          q3 + 1.5*iqr)
# z-score rule
from scipy.stats import zscore
mask = abs(zscore(df["rent"])) < 3
# model-based: IsolationForest (sklearn)
```

- Don't drop blindly. Investigate first.
- Outliers are often data-entry errors OR the most interesting rows.

Why encode at all?

Most ML models (linear regression, k-NN, gradient boosting trees, neural networks) operate on **numbers**. They can't multiply, take dot products with, or compute distances on the string `"Kentron"`.

Encoding turns categorical strings (or category codes) into numerical features that preserve the right structure.

Pick the encoding by two questions:

1. **Are the categories ordered?** (e.g., *small* $<$ *medium* $<$ *large*.)
2. **How many unique values?** (Few $\rightarrow$ one-hot is fine. Many $\rightarrow$ one-hot explodes.)

One-hot encoding: the default for unordered

Each category becomes its own 0/1 column.

Before:

district
Kentron
Arabkir
Komitas
Kentron

After one-hot:

d=Kentron	d=Arabkir	d=Komitas
1	0	0
0	1	0
0	0	1
1	0	0

```
pd.get_dummies(df, columns=["district"], drop_first=False)
from sklearn.preprocessing import OneHotEncoder
OneHotEncoder(handle_unknown="ignore", sparse_output=False)
```

Production essential: `handle_unknown="ignore"`. Otherwise an unseen test-set district crashes the model. `drop="first"` also matters - next slide.

The dummy trap (link back to L01b)

Including *all* one-hot columns AND an intercept makes the design matrix singular:

$$\underbrace{1}_{\text{intercept}} = \underbrace{\mathbb{1}[d{=}\text{Kentron}] + \mathbb{1}[d{=}\text{Arabkir}] + \mathbb{1}[d{=}\text{Komitas}]}_{\text{sum of all dummies}}$$

The all-ones column equals the sum of the dummies. Columns are linearly dependent $\Rightarrow \mathbf{X}^\top\mathbf{X}$ not invertible $\Rightarrow$ OLS fails (L01b Section 4).

Two fixes:

- `pd.get_dummies(..., drop_first=True)` - drops one category, which becomes the reference baseline absorbed into θ_0.
- `OneHotEncoder(drop="first")` (sklearn).

Exceptions where you keep ALL dummies:

- Ridge / Lasso (regularization makes the system invertible anyway).
- Trees (no intercept, no matrix inversion).
- When you want all coefficients interpretable on the same footing (no reference category).

Label / ordinal encoding: ONLY for ordered

Maps each category to an integer. **Imposes an order.**

```
from sklearn.preprocessing import OrdinalEncoder
# CORRECT use: t-shirt sizes
OrdinalEncoder(categories=[["S","M","L","XL"]]).fit_transform(...)
# WRONG use: city names
OrdinalEncoder().fit_transform(df[["district"]])  # treats Kentron=0
    < Arabkir=1 < Komitas=2
```

The common mistake: feeding label-encoded city names to a linear regression. The model sees Komitas = 2× Arabkir, which is nonsense.

Rule: use ordinal encoding only when you can write down the order on paper and defend it.

High-cardinality: target and frequency encoding

Some categoricals have hundreds or thousands of levels (zip codes, product IDs). One-hot explodes; ordinal lies. Two production tricks:

Target encoding. Replace each category by the mean target value among rows of that category.

```
# example: district -> mean rent in that district (computed on TRAIN
    only!)
mean_rent = df_train.groupby("district")["rent"].mean()
df["district_te"] = df["district"].map(mean_rent)
```

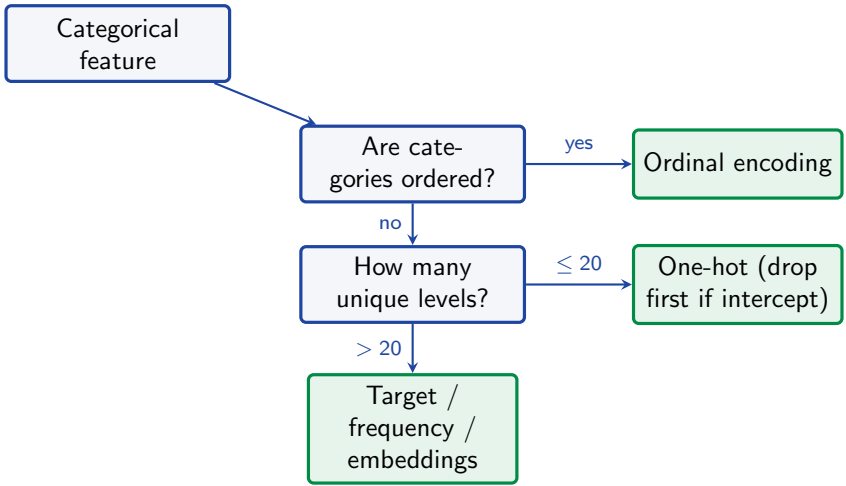
Why this is leakage if computed on full data: the mean depends on `rent`, which is what you're predicting. If you compute the mean using test rows, you've used the answer as an input.

Frequency / count encoding. Replace each category by how often it appears.

```
freq = df_train["district"].value_counts(normalize=True)
df["district_freq"] = df["district"].map(freq)
```

Both must be *fit on the training fold only*. Use `category_encoders` library + CV-aware variants in practice.

Encoding decision flowchart



Embeddings (a learned dense vector per category) are the modern high-cardinality default for neural networks - covered in the NN chapter.

Datetime preprocessing

Raw dates aren't directly usable. Extract components, and use *cyclic* encoding for periodic features.

```
df["dt"] = pd.to_datetime(df["listed_at"])
df["year"]    = df["dt"].dt.year
df["month"]   = df["dt"].dt.month
df["weekday"] = df["dt"].dt.dayofweek          # 0=Mon
df["hour"]    = df["dt"].dt.hour
```

Cyclic features need sin/cos encoding. `hour`=23 and `hour`=0 are 1 hour apart, not 23. Label encoding lies; one-hot wastes columns. Trick:

```
import numpy as np
df["hour_sin"] = np.sin(2 * np.pi * df["hour"] / 24)
df["hour_cos"] = np.cos(2 * np.pi * df["hour"] / 24)
```

Also useful: time since a reference event (`days_since_listed`), holiday indicators, weekend flag.

Predict-first: why scale?

You have three features:

- `area` in m^2, ranging 20-200
- `rooms`, ranging 1-5
- `year_built`, ranging 1950-2024

Predict-first: why scale?

You have three features:

- `area` in m^2, ranging 20-200
- `rooms`, ranging 1-5
- `year_built`, ranging 1950-2024

Predict before reading on:

- In k-NN with Euclidean distance, which feature dominates the neighbor calculation?
- In gradient descent on linear regression, does the step size that works for θ_{area} also work for θ_{year}?
- Do decision trees care about feature scale at all?

Predict-first: why scale?

You have three features:

- `area` in m^2, ranging 20-200
- `rooms`, ranging 1-5
- `year_built`, ranging 1950-2024

Predict before reading on:

- In k-NN with Euclidean distance, which feature dominates the neighbor calculation?
- In gradient descent on linear regression, does the step size that works for θ_{area} also work for θ_{year}?
- Do decision trees care about feature scale at all?

Answers: kNN distance is dominated by `year_built` (largest range). GD oscillates: one good learning rate per feature is impossible. Trees: **no** - they only use threshold comparisons.

Three scalers: formulas + when to use

StandardScaler (z-score):

$$x' = \frac{x - \mu}{\sigma}$$

Output: mean 0, std 1.

Best for *roughly Gaussian* features. Default choice for linear models.

MinMaxScaler:

$$x' = \frac{x - \min}{\max - \min}$$

Output: range $[0, 1]$.

Best for *bounded inputs* (image pixels, percentages, anything with natural floor/ceiling).

RobustScaler:

$$x' = \frac{x - \text{med}}{\text{IQR}}$$

Output: median 0, IQR 1.

Best when *outliers* would distort mean/std. Uses median + IQR instead.

- All three are linear ($x' = a\,x + b$). They don't change the *shape* of the distribution.
- For a skewed feature: apply log / power transform *first*, then scale.

Three scalers: code comparison

Same interface for all three - drop-in replaceable:

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
    RobustScaler

# 1. Standardize (z-score)
sc = StandardScaler().fit(X_train)
X_train_z = sc.transform(X_train)
X_test_z  = sc.transform(X_test)          # NEVER refit on test

# 2. Min-max to [0, 1]
mm = MinMaxScaler().fit(X_train)
X_train_mm = mm.transform(X_train)

# 3. Robust (median / IQR)
rs = RobustScaler().fit(X_train)
X_train_rs = rs.transform(X_train)
```

The `fit` call learns μ, σ (or min/max, or median/IQR) from *training data only*. The `transform` call applies the same statistics to both train and test. Pipelines (later) automate this.

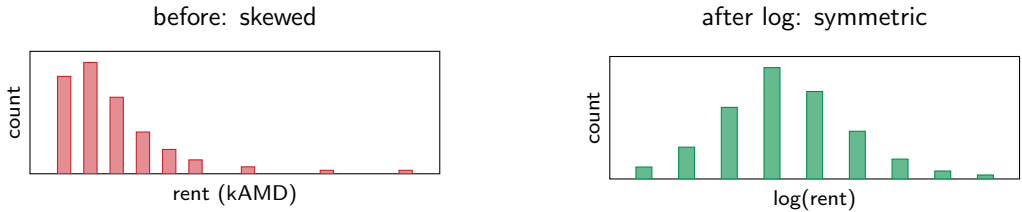
When does scaling matter?

Model family	**Scale?**	**Why**
Linear regression with GD (L02)	yes	step size needs it
Linear regression with normal eq. (L01b)	weakly	numerical conditioning
Ridge / Lasso	yes	penalty depends on coeff. magnitudes
k-NN, k-means, SVM-RBF	yes	distances dominated by big-range features
Logistic regression with GD	yes	same as linear+GD
PCA	yes	variance dominated by big-range features
Decision trees, random forests, GBMs	**no**	only use thresholds
Gaussian Naive Bayes	no	per-feature mean/var anyway

Trees only look at thresholds (`if area > 60 ...`) - rescaling doesn't change which split is best. Distance-based and gradient-based methods care a lot.

Log and power transforms for skew

`rent` is right-skewed: most apartments 200-400 kAMD, a long tail of luxury at 1500+. Models with symmetric-noise assumptions both *underfit the tail* AND *overpredict near the mean*. A monotonic transform compresses the tail and restores symmetry.




```
df["log_rent"] = np.log1p(df["rent"])                # log(1 + x), handles 0
from sklearn.preprocessing import PowerTransformer
PowerTransformer(method="yeo-johnson").fit(Xtr)      # any real input
PowerTransformer(method="box-cox").fit(Xtr)          # requires x > 0
```

After training in log-space, invert with `np.expm1` to report MAE in real units.

The leakage rule (the single most important slide)

Every preprocessor has a *state* learned from data: a mean, a list of categories, a min/max. The rule:

Fit on train. Transform on train and test.
Never fit on test. Never fit on train+test combined.

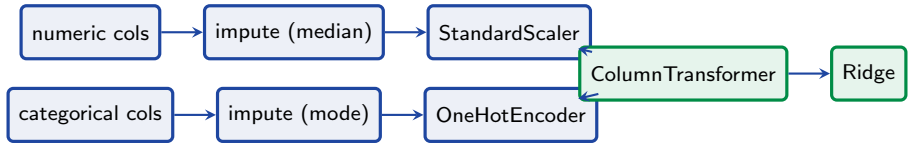
Examples of how leakage hides:

- Scaling on full data before train/test split $\Rightarrow$ test mean leaks into train.
- Mean-imputing on full data $\Rightarrow$ same.
- Target encoding on full data $\Rightarrow$ test target leaks into a feature.
- One-hot on full data $\Rightarrow$ usually OK, but unseen test categories will crash unless `handle_unknown="ignore"`.

Effect: inflated CV scores during dev, ugly surprises in production.

sklearn Pipeline + ColumnTransformer (the right way)

Wrap the chain so leakage is structurally impossible. Conceptually:




```
numeric = Pipeline([("imp", SimpleImputer(strategy="median")),
                    ("sc",  StandardScaler())])
categorical = Pipeline([("imp", SimpleImputer(strategy="most_frequent")),
                        ("ohe", OneHotEncoder(handle_unknown="ignore", drop=
                            "first"))])
pre = ColumnTransformer([("num", numeric,      ["area", "rooms", "year_built"
    ]),
                         ("cat", categorical, ["district", "balcony"])])
model = Pipeline([("pre", pre), ("ridge", Ridge(alpha=1.0))])
model.fit(X_train, y_train)          # ONE call -- no leakage possible
```

Recap

- **Missing data:** detect the pattern, prefer median/mode for a baseline, use KNN / iterative when you need accuracy, add an indicator column when missingness itself is informative. Drop duplicates after a quick sanity check. Treat outliers as signals to investigate, not noise to delete.
- **Categorical encoding:** one-hot for low cardinality (drop one column if you have an intercept - the dummy trap from L01b). Ordinal only for genuinely ordered categories. Target / frequency / embeddings for high cardinality. Cyclic sin/cos for periodic features like hour-of-day.
- **Scaling:** StandardScaler for Gaussian-ish, MinMax for bounded, Robust for outlier-heavy. Required for GD / kNN / Ridge / Lasso / PCA. Not needed for trees. Log / power transforms come first if the feature is skewed.
- **The one rule:** fit any preprocessor on train data only. Wrap everything in a `Pipeline` and you can't accidentally violate it.

HW01c - practice

1. On the Yerevan rent dataset: detect the missingness pattern, choose a strategy, justify your choice in 2-3 sentences.
2. Build a `ColumnTransformer` for the numeric + categorical features. Fit Ridge regression in a single Pipeline. Report 5-fold CV MAE.
3. Deliberately leak: scale on full data, then on train-only. Compare CV scores and discuss why the leaky version looks better. *This is the most important question.*
4. Bonus 1: log-transform rent and re-fit. Report MAE in AMD (not log AMD) by inverting the transform with `np.expm1`.
5. Bonus 2: add a cyclic feature for `listed_at` hour-of-day. Does Ridge improve?
6. Bonus 3: replace Ridge with a tree-based model (e.g., `HistGradientBoostingRegressor`). Drop the scaler. Compare to the Ridge pipeline.