

## WEEK 2

# Data Cleaning I

Production-ready data cleaning skills for AI and ML pipelines

Day 6	<b>Missing Values</b>	Detection, quantification, basic handling strategies
Day 7	<b>Imputation Strategies</b>	Column-specific rules, indicators, train/test leakage
Day 8	<b>Duplicate Records</b>	Entity vs event, deduplication, aggregation
Day 9	<b>Data Types</b>	Numeric, datetime, mixed-type, schema stability
Day 10	<b>Outliers &amp; Ethics</b>	IQR, z-score, capping, transformation, fairness

### Tooling

pandas for table operations, numpy for numerical work  
isna, dropna, fillna, duplicated, drop\_duplicates, to\_numeric,  
to\_datetime, clip, np.log1p

```
import pandas as pd
import numpy as np
np.random.seed(42)
```

## DAY 6

# Missing Values

Detection, Quantification, and Basic Handling

### OBJECTIVES

- Interpret different causes of missingness
- Detect and summarize with isna/isnull
- Decide when to drop rows/columns vs impute
- Use mean/median/mode/constant imputation
- Create missingness indicators

### ACTIVITY

Analyze a partially observed behavioral dataset.  
Compare 'drop rows' vs 'impute + indicator' pipelines.

### ASSESSMENT

Justify strategy on a concrete feature.  
Produce missingness summary and imputations.

# What Missing Values Represent

Missing values encode multiple real-world situations

## Typical Meanings

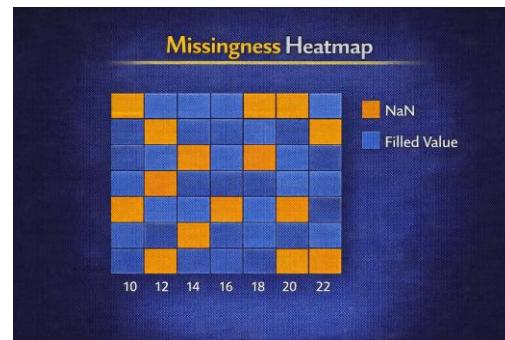
- Never collected (sensor offline, survey skipped)
  - Not applicable (field doesn't logically apply)
  - Withheld (privacy, 'prefer not to say')
  - Mis-encoded ('N/A', 'not reported', -999)

## Statistical Impact

- Non-random missingness biases means, variances, correlations
  - Sentinel values used as numbers can dominate averages and regression coefficients

## CONSTRUCTING DIFFERENT MISSING PATTERNS

```
df_missing = pd.DataFrame({  
    "user_id": [1, 2, 3, 4, 5, 6],  
    "income_reported": ["55000", "62000", "N/A", "not reported", None, "0"],  
    "pregnancy_weeks": [None, None, 20, 32, None, 0],  # Not applicable vs real  
    "age": [25, 30, np.nan, 40, 35, 28]  
})
```



# ML Impact of Missing Values

Treating all missing values the same is dangerous

## Linear & Neural Models

- Treat sentinel values (like -999) as extreme magnitudes
- Can dramatically skew predictions
- May learn spurious patterns from encoding

## Tree Models

- Can split strongly on sentinel values
- Risk overfitting to 'missing' as a category
- May seem robust but still affected

## Key Insight

Before cleaning, ask what each missing pattern means in the domain. Different causes require different handling strategies.

# Detecting Missing Values

Built-ins and custom tokens

## Core Functions

- df.isna() / df.isnull() for element-wise masks
- df.isna().sum() for per-column counts
- df.isna().mean() for per-column percentages

## Custom Tokens

- Pandas only flags NaN, None, NaT as missing
- Domain strings like 'N/A', 'unknown', '?' must be normalized to NaN early
- Risk: over-normalizing erases rare real categories

## NORMALIZE TOKENS THEN INSPECT

```
custom_missing = ["N/A", "NA", "not reported", "unknown", "?"]
df_norm = df_missing.copy()
df_norm["income_reported"] = df_norm["income_reported"].replace(custom_missing, np.nan)

print(df_norm)
print("\nMissing per column:\n", df_norm.isna().sum())
```

# Quantifying Missingness

Counts and percentages clarify which columns/rows are problematic

## Useful Summaries

- Per-column missing count and percentage
- Per-row missing count (record completeness)
- Group-level missingness (by segment/region)

## Decision Use

- Identify columns to drop or needs advanced imputation
- Identify subgroups with systematically poorer data collection

## COMPACT MISSINGNESS SUMMARY

```
def missing_summary(df: pd.DataFrame) -> pd.DataFrame:  
    total = df.isna().sum()  
    pct = (df.isna().mean() * 100).round(1)  
    return (pd.DataFrame({"missing_count": total, "missing_percent": pct})  
           .sort_values("missing_percent", ascending=False))  
  
summary = missing_summary(df_norm)  
df_norm["missing_per_row"] = df_norm.isna().sum(axis=1)
```

# Dropping Rows vs Dropping Columns

Easy but blunt - must be justified

## When to Drop Rows

- Missingness is rare and spread across rows
- Critical labels/predictors are missing and cannot be reliably imputed

Risk: Shrinks dataset, introduces selection bias

## When to Drop Columns

- Column is mostly missing and not business-critical
- Cannot construct trustworthy imputation

Risk: May discard predictive or fairness-relevant info

## BASIC DROPPING PATTERNS

```
critical_cols = ["income_reported", "age"]

df_drop_any = df_norm.dropna(how="any")          # Drop if ANY column missing
df_drop_crit = df_norm.dropna(subset=critical_cols)  # Drop only if critical cols missing

print("Original:", df_norm.shape, "Drop any:", df_drop_any.shape, "Drop critical:", df_drop_crit.shape)
```

# Basic Imputation — Mean, Median, Mode

Replace missing with estimates from observed data

## Mean

- Best for symmetric numeric features
- Reduces variance
- Sensitive to outliers

## Median

- Robust for skewed distributions
- Better for income, prices
- Preserves center better

## Mode

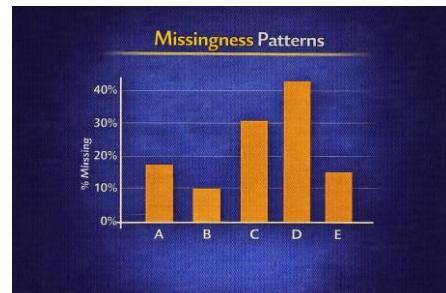
- For categorical features
- Inflates dominant category
- May hide missingness info

### NUMERIC MEAN/MEDIAN, CATEGORICAL MODE

```
df_imp = df_norm.copy()
df_imp["income_num"] = pd.to_numeric(df_imp["income_reported"], errors="coerce")

df_imp["age_mean_imp"] = df_imp["age"].fillna(df_imp["age"].mean())
df_imp["age_median_imp"] = df_imp["age"].fillna(df_imp["age"].median())

mode_income = df_imp["income_reported"].mode(dropna=True)[0]
df_imp["income_mode_imp"] = df_imp["income_reported"].fillna(mode_income)
```



# Constant Imputation + Missingness Indicator

Preserve information that value was originally missing

## When to Use Constant

- Features where constant reflects meaningful 'none' state
- When combining with binary indicator column
- Common constants: 0, -1, 'Unknown'

## Why Add Indicator

- Preserves info that value was missing
- Allows model to distinguish 'real 0' from 'imputed 0'
- Missingness itself may be predictive

### CONSTANT + INDICATOR FOR NUMERIC AND CATEGORICAL

```
df_const = df_imp.copy()

df_const["age_missing"] = df_const["age"].isna().astype(int)      # 1 if was missing
df_const["age_const"] = df_const["age"].fillna(-1)                 # Fill with sentinel

df_const["income_missing"] = df_const["income_reported"].isna().astype(int)
df_const["income_const"] = df_const["income_reported"].fillna("Unknown")
```

# ML Impact of Missingness and Imputation

Think of imputation as part of model design

## Effects on Models

- Changing means/variances → different linear coefficients
- Changing category frequencies → different priors/thresholds
- Indicators allow exploiting data-collection patterns

## Fairness Considerations

- If missingness concentrated in specific groups, naive imputation misrepresents them
- Dropping incomplete rows may systematically exclude certain users

### TOY COMPARISON: MEAN VS MEDIAN IN REGRESSION

```
from sklearn.linear_model import LinearRegression

X_mean = df_model["age"].fillna(age_mean).to_frame()
X_median = df_model["age"].fillna(age_median).to_frame()
y = df_model["clicked_ad"].values

coef_mean = LinearRegression().fit(X_mean, y).coef_[0]
coef_median = LinearRegression().fit(X_median, y).coef_[0]
```

# Missing Values Practice

Build and compare cleaning pipelines

## Activity Steps

1. Load partially observed user dataset
2. Normalize custom missing tokens to NaN
3. Produce missingness summary (per-column %, per-row)
4. Build Version A: drop rows with missing in key cols
5. Build Version B: impute + indicators
6. Train classifier on A and B; compare metrics

## Assessment

Written: Explain when you would prefer dropping rows over imputation on a churn dataset.

Coding: Implement a function that drops columns with `missing_percent > 0.7` and justify the threshold.

## SKELETON

```
df_raw = pd.read_csv("user_data.csv")
df_raw = df_raw.replace(["N/A", "NA", "not reported", "unknown", "?"], np.nan)
summary = missing_summary(df_raw)
```

## DAY 7

# Imputation Strategies

Column-Specific Rules and Avoiding Leakage

### OBJECTIVES

- Explain why imputation is never neutral
- Choose strategies based on feature type/distribution
- Implement column-wise rules with indicators
- Apply imputation in train/test-safe way
- Avoid data leakage from future information

### ACTIVITY

Build imputation function that learns on train, applies to test.

### ASSESSMENT

Conceptual: leakage scenarios and impact.

Coding: reusable imputation utilities.

# Why Imputation Is Not Neutral

Imputation assumes something about unobserved data

## Distribution Changes

- Mean/median imputation shrinks variance
- Mode/constant creates artificial spikes
- Correlations artificially strengthened/weakened

## Model Effects

- Linear models see reduced spread, altered coefficients
- Tree models see clusters at imputation constants
- Neural networks may learn imputation artifacts

### VARIANCE COMPARISON BEFORE/AFTER IMPUTATION

```
df_demo = df_imp.copy()
print("Age with NaN:\n", df_demo["age"].describe())

df_demo["age_mean_imp"] = df_demo["age"].fillna(df_demo["age"].mean())
print("\nAge after mean imputation:\n", df_demo["age_mean_imp"].describe()) # Note: std decreases
```

# Column-Specific Strategies

Different features warrant different imputation rules

## Guidelines

- Symmetric numeric → mean
- Skewed numeric → median
- Nominal categorical → mode or 'Unknown'
- Ordinal categorical → median or mode

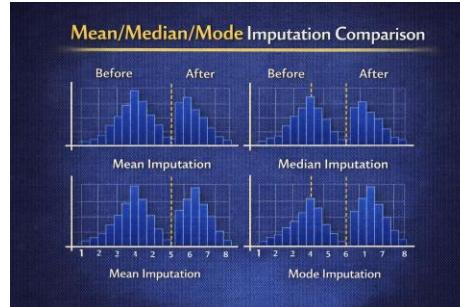
## Risks

- Using mean on heavy-tailed features biases estimates
- Inconsistent strategies across features hinder interpretation
- Document all choices!

## COLUMN-WISE RULE APPLICATION

```
strategies = {"age": ("median", None), "income_num": ("median", None),
              "income_reported": ("constant", "Unknown")}

for col, (kind, val) in strategies.items():
    if kind == "mean": df_cs[col + "_imp"] = df_cs[col].fillna(df_cs[col].mean())
    elif kind == "median": df_cs[col + "_imp"] = df_cs[col].fillna(df_cs[col].median())
    elif kind == "constant": df_cs[col + "_imp"] = df_cs[col].fillna(val)
```



# Distribution-Aware Imputation

Mean vs Median guided by empirical distribution

## Steps

1. Inspect summary stats (mean, median, quantiles)
2. Check for outliers and skewness
3. Prefer median when tails are long (income, prices)
4. Document the reasoning

## Example: Income

- Income is typically right-skewed
- Mean pulled up by high earners
- Median more representative of 'typical' user

### INCOME EXAMPLE

```
mean_inc = df_dist["income_num"].mean()  
median_inc = df_dist["income_num"].median()  
  
df_dist["income_mean_imp"] = df_dist["income_num"].fillna(mean_inc)  
df_dist["income_median_imp"] = df_dist["income_num"].fillna(median_inc)  
print("Mean:", mean_inc, "Median:", median_inc) # Median typically lower for skewed data
```

# Categorical Imputation — Mode vs Special Category

Balance simplicity and clarity about missingness

## Mode

- Simple, uses existing categories
- Increases dominance of majority category
- Hides that value was missing

## Special Category ('Unknown')

- Makes missingness explicit in feature space
- Can capture users who systematically avoid answering
- May be predictive signal itself

### MODE VS 'UNKNOWN'

```
mode_inc = df_cat["income_reported"].mode(dropna=True)[0]
df_cat["income_mode"] = df_cat["income_reported"].fillna(mode_inc)
df_cat["income_unknown"] = df_cat["income_reported"].fillna("Unknown")

print(df_cat[["income_reported", "income_mode", "income_unknown"]])
```

# Missingness Indicators and Model Behavior

Compact features marking originally missing values

## Benefits

- Preserve info that imputation would erase
- Useful when missingness related to outcome
- Models can learn from data-collection patterns

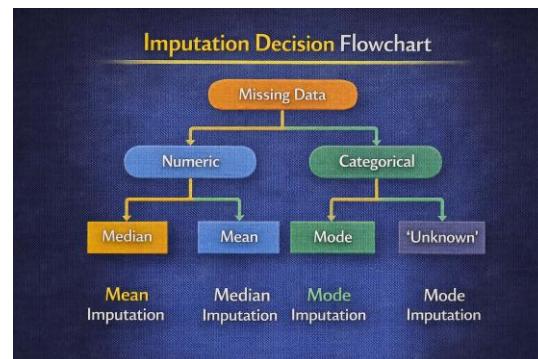
## Risks

- May overfit to data-collection quirks
- Indicators may pick up group-level differences
- Can increase model complexity

### ADD INDICATORS AND IMPUTE

```
df_ind = df_imp.copy()
for col in ["age", "income_num"]:
    df_ind[col + "_missing"] = df_ind[col].isna().astype(int) # Binary indicator
    med = df_ind[col].median()
    df_ind[col + "_imp"] = df_ind[col].fillna(med)

print(df_ind.head()) # Now has both imputed values and indicators
```



# Train/Test Separation and Data Leakage

Fit imputation parameters on train only

## Correct Pattern

1. Split into train and test
2. Fit imputation parameters on train only
3. Apply those parameters to both train and test
4. Never peek at test during fitting

## Leakage Effects

- Over-optimistic metrics (looks better than reality)
- Degraded production performance
- Future data is unknown at training time!

## SAFE MEDIAN IMPUTATION

```
train_df, test_df = train_test_split(df_leak, test_size=0.4, random_state=42)
medians = {c: train_df[c].median() for c in ["age", "income_num"]} # Fit on train

def apply_imp(df, med):
    out = df.copy()
    for c, v in med.items(): out[c] = out[c].fillna(v) # Apply train params
    return out
```

# Imputation Practice

Build leakage-free imputation utilities

## Activity

1. Implement `fit_imputer(train_df)` that computes medians for numeric, modes for categorical
2. Implement `transform_imputer(df, params)` to apply learned rules
3. Train classifier with and without indicators
4. Compare behavior and metrics

## Assessment

Conceptual: Explain a realistic imputation-leakage scenario and its impact on a deployed model.

Coding: Write justification for using median over mean on a skewed feature in a credit-risk model.

## SKELETON

```
def fit_imputer(train, num_cols, cat_cols):  
    return {"num_median": {c: train[c].median() for c in num_cols},  
            "cat_mode": {c: train[c].mode(dropna=True)[0] for c in cat_cols}}
```