6.04: Handling Missing (Null) Values

What are Missing Values?

Missing values are a common problem in data analysis and machine learning.
They happen when some information is not recorded for certain columns in a dataset.
These missing pieces can look like empty cells, `NaN`, `NA`, or words like "unknown".

Many machine learning algorithms fail if the dataset contains missing values. However, algorithms like K-nearest and Naive Bayes support data with missing values.

If we don't fix them, missing values can cause:

- Wrong or less accurate results
- Smaller dataset size
- Bias in the model
- Problems with models that need complete data

So, handling missing values in the right way is important to make sure our models give correct and fair results.

1. Creating Synthetic Dataset with Missing Values

In [4]:
```
import pandas as pd
import numpy as np

np.random.seed(0)
df = pd.DataFrame({
    "age": np.random.randint(18, 70, size=10).astype(float),
    "salary": np.random.normal(50000, 15000, size=10),
    "city": ["A","B","A","C","B","A","C","B","A","C"],
    "join_date": pd.date_range("2020-01-01", periods=10)
})

# inject some NaNs
df.loc[[1, 2, 5], "age"] = np.nan
df.loc[[2, 7], "salary"] = np.nan
df.loc[[3, 5], "city"] = np.nan

df_backup = df.copy()
df
```

Out[4]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	NaN	47729.641876	B	2020-01-02
2	NaN	NaN	A	2020-01-03
3	21.0	56158.977529	NaN	2020-01-04
4	21.0	52160.653567	B	2020-01-05
5	NaN	71814.102604	NaN	2020-01-06
6	27.0	61415.565877	C	2020-01-07
7	37.0	NaN	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

2. Inspect & quantify missingness (EDA)

In [5]:
```
# per-column counts and percent
df.isnull().sum()
```

Out[5]:
```
age          3
salary       2
city         2
join_date    0
dtype: int64
```

In [6]:
```
df.count()
```

Out[6]:
```
age           7
salary        8
city          8
join_date    10
dtype: int64
```

In [7]:
```
df.isnull().mean() *100
```

Out[7]:
```
age          30.0
salary       20.0
city         20.0
join_date     0.0
dtype: float64
```

In [8]:
```
# using df.info()
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        7 non-null      float64
 1   salary     8 non-null      float64
 2   city       8 non-null      object
 3   join_date  10 non-null     datetime64[ns]
dtypes: datetime64[ns](1), float64(2), object(1)
memory usage: 452.0+ bytes
```

What is `np.nan`?

- `np.nan` comes from the NumPy library in Python.
- `nan` means **"Not a Number"**.
- It is used to represent **missing or undefined values** in numerical data.
- In datasets, `np.nan` often works as a placeholder for missing values.

3. Strategy - Drop columns / rows by threshold

In [9]:
```
df
```

Out[9]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	NaN	47729.641876	B	2020-01-02
2	NaN	NaN	A	2020-01-03
3	21.0	56158.977529	NaN	2020-01-04
4	21.0	52160.653567	B	2020-01-05
5	NaN	71814.102604	NaN	2020-01-06
6	27.0	61415.565877	C	2020-01-07
7	37.0	NaN	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

In [10]:
```
df.isnull().mean()
```

Out[10]:
```
age          0.3
salary       0.2
city         0.2
join_date    0.0
dtype: float64
```

In [11]:
```
# drop columns with >40% missing
th = 0.40
drop_cols = df.columns[df.isnull().mean() > th]
df.drop(columns=drop_cols)
```

Out[11]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	NaN	47729.641876	B	2020-01-02
2	NaN	NaN	A	2020-01-03
3	21.0	56158.977529	NaN	2020-01-04
4	21.0	52160.653567	B	2020-01-05
5	NaN	71814.102604	NaN	2020-01-06
6	27.0	61415.565877	C	2020-01-07
7	37.0	NaN	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

In [12]:
```
# drop rows with >=2 missing values
df[df.isnull().sum(axis=1) < 2]
```

Out[12]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	NaN	47729.641876	B	2020-01-02
3	21.0	56158.977529	NaN	2020-01-04
4	21.0	52160.653567	B	2020-01-05
6	27.0	61415.565877	C	2020-01-07
7	37.0	NaN	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

4. Strategy - Simple imputation (pandas & sklearn)

In [13]:
```
# using pandas
# numeric: mean, median
df['age'] = df['age'].fillna(df['age'].median())

# categorical: mode or constant
# df['city'] = df['city'].fillna("Missing")
```

```
df['city'] = df['city'].fillna(df['city'].mode()[0])

df
```

Out[13]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	37.0	47729.641876	B	2020-01-02
2	37.0	NaN	A	2020-01-03
3	21.0	56158.977529	A	2020-01-04
4	21.0	52160.653567	B	2020-01-05
5	37.0	71814.102604	A	2020-01-06
6	27.0	61415.565877	C	2020-01-07
7	37.0	NaN	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

In [17]:
```
df = df_backup.copy()
df
```

Out[17]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	NaN	47729.641876	B	2020-01-02
2	NaN	NaN	A	2020-01-03
3	21.0	56158.977529	NaN	2020-01-04
4	21.0	52160.653567	B	2020-01-05
5	NaN	71814.102604	NaN	2020-01-06
6	27.0	61415.565877	C	2020-01-07
7	37.0	NaN	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

In [15]:
```
# Scikit-Learn
from sklearn.impute import SimpleImputer

num_imp = SimpleImputer(strategy="median")   # or "mean"
cat_imp_mfreq = SimpleImputer(strategy="most_frequent")
cat_imp_const = SimpleImputer(strategy="constant", fill_value="Missing") # strategy
```

```
# example: apply on single column
df['age'] = num_imp.fit_transform(df[['age']])[:, 0]
df['city'] = cat_imp_mfreq.fit_transform(df[['city']])[:, 0]
```

In [16]:
```
df
```

Out[16]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	37.0	47729.641876	B	2020-01-02
2	37.0	NaN	A	2020-01-03
3	21.0	56158.977529	A	2020-01-04
4	21.0	52160.653567	B	2020-01-05
5	37.0	71814.102604	A	2020-01-06
6	27.0	61415.565877	C	2020-01-07
7	37.0	NaN	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

5. Strategy - Time-series specifics

In [18]:
```
# forward-fill => imputing the values with the previous value instead
# backfill => In backward fill, the missing value is imputed using the next value.
df_ts = df.sort_values("join_date")
df_ts['salary'] = df_ts['salary'].ffill() #.bfill()
df_ts
```

Out[18]:

	age	salary	city	join_date
0	62.0	64251.326263	A	2020-01-01
1	NaN	47729.641876	B	2020-01-02
2	NaN	47729.641876	A	2020-01-03
3	21.0	56158.977529	NaN	2020-01-04
4	21.0	52160.653567	B	2020-01-05
5	NaN	71814.102604	NaN	2020-01-06
6	27.0	61415.565877	C	2020-01-07
7	37.0	61415.565877	B	2020-01-08
8	39.0	56657.948491	A	2020-01-09
9	68.0	55005.114911	C	2020-01-10

6. Strategy - Multivariate Approach

- In a multivariate approach, more than one feature is taken into consideration.
- There are two ways to impute missing values considering the multivariate approach.
- Using KNNImputer or IterativeImputer classes.

IterativeImputer

- IterativeImputer is used to fill missing values (NaNs) by modeling each feature as a function of the other features.
- Unlike SimpleImputer (which uses mean, median, or most frequent values), it can capture relationships between features.
- Very useful when missingness is not completely random and features are correlated.

In [20]:
```
# Create synthetic housing data
df = pd.DataFrame({
    'MedInc': [8.3, 8.0, np.nan, 5.5, 3.8, np.nan, 7.0], # Median Income
    'HouseAge': [41, 21, 52, np.nan, 52, 15, 30], # Age of the House
    'MedHouseVal': [4.5, 3.5, 3.5, 3.4, np.nan, 2.8, 3.0] #Median House Value
})

print("Original synthetic housing data with missing values:")
print(df)
```

```
Original synthetic housing data with missing values:
   MedInc  HouseAge  MedHouseVal
0     8.3      41.0          4.5
1     8.0      21.0          3.5
2     NaN      52.0          3.5
3     5.5       NaN          3.4
4     3.8      52.0          NaN
5     NaN      15.0          2.8
6     7.0      30.0          3.0
```

In [22]:
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

# Initialize IterativeImputer
imp = IterativeImputer()

# Fit and transform the dataset
df_imputed = pd.DataFrame(imp.fit_transform(df), columns=df.columns)

print("\nData after Iterative Imputer:")
print(df_imputed)
```

```
Data after Iterative Imputer:
     MedInc   HouseAge  MedHouseVal
0  8.300000  41.000000     4.500000
1  8.000000  21.000000     3.500000
2  5.459135  52.000000     3.500000
3  5.500000  48.897298     3.400000
4  3.800000  52.000000     2.592113
5  7.679367  15.000000     2.800000
6  7.000000  30.000000     3.000000
```

```
C:\Users\pc\.conda\envs\decodeaienv\Lib\site-packages\sklearn\impute\_iterative.py:8
95: ConvergenceWarning: [IterativeImputer] Early stopping criterion not reached.
  warnings.warn(
```

Steps

- Pick one column with missing values (target column).
- Use all other columns as predictors to fill the missing entries in this column.
- Move to the next column with missing values.
- Repeat for all columns with missing values.
- Repeat the entire process multiple times to refine imputations.

Nearest Neighbors Imputations (KNNImputer)

- Missing values are imputed using the k-Nearest Neighbors approach, where a Euclidean distance is used to find the nearest neighbors.
- Sensitive to feature scaling (must standardize/normalize first).

In [25]:
```
df
```

Out[25]:

	MedInc	HouseAge	MedHouseVal
0	8.3	41.0	4.5
1	8.0	21.0	3.5
2	NaN	52.0	3.5
3	5.5	NaN	3.4
4	3.8	52.0	NaN
5	NaN	15.0	2.8
6	7.0	30.0	3.0

In [26]:
```
from sklearn.impute import KNNImputer

# Initialize KNNImputer
knn_imp = KNNImputer(n_neighbors=2)

# Fit and transform
df_knn_imputed = pd.DataFrame(knn_imp.fit_transform(df), columns=df.columns)
```

```
print("\nDataFrame after KNN Imputer:")
print(df_knn_imputed)
```

```
DataFrame after KNN Imputer:
   MedInc  HouseAge  MedHouseVal
0    8.30      41.0         4.50
1    8.00      21.0         3.50
2    4.65      52.0         3.50
3    5.50      33.5         3.40
4    3.80      52.0         3.45
5    6.75      15.0         2.80
6    7.00      30.0         3.00
```

Iterative Vs KNNImputer

Aspect	IterativeImputer	KNNImputer
Method	Predicts missing values using regression models	Averages values from nearest neighbors
Captures Feature Relations?	Yes (global, model-based)	Partial (local similarity only)
Scalability	Moderate speed	Slower on large datasets
Data Assumptions	Works best if features correlate	Works best if rows are similar
Feature Scaling Needed?	Not mandatory (depends on model)	Yes (distance-based)
Best Use Case	Structured/tabular data with correlations	Datasets with natural clustering/neighbor similarity

7. Use “Missingness” as a Feature

In [28]:
```
X = pd.DataFrame({'Age':[20, 30, 10, np.nan, 10]})
X
```

Out[28]:

	Age
0	20.0
1	30.0
2	10.0
3	NaN
4	10.0

In [29]:
```
imputer = SimpleImputer(add_indicator=True)
imputer.fit_transform(X)
```

Out[29]:
```
array([[20. ,  0. ],
       [30. ,  0. ],
       [10. ,  0. ],
       [17.5,  1. ],
       [10. ,  0. ]])
```