

- ✓ Building a Healthcare Analytics Dataset from Raw Multi-Source Data
- ✓ Environment setup and synthetic data

Important — Do not modify these datasets

The synthetic datasets below are fixed and used by all learners. Do not modify this cell or change any data values, as **grading is based on the exact outputs** produced from these datasets.

You should only write code in the sections of the notebook that are clearly marked for you to complete.

```
# ⚠ DO NOT MODIFY THIS CELL
# This dataset is fixed and used for consistent grading across all learners.

import pandas as pd
import numpy as np

ehr_df = pd.DataFrame({
    "patient_id": ["P001", "P002", "P003", "P003", "P004"],
    "patient_name": ["John Doe", "Jane Smith", "Alice Brown", "Alice Brown", "Bob Lee"],
    "dob": pd.to_datetime(["1980-05-12", "1975-09-30", "1990-02-15", "1990-02-15", "1965-11-01"]),
    "encounter_id": ["E1", "E2", "E3", "E3", "E4"],
    "encounter_date": pd.to_datetime(["2023-01-10", "2023-01-12", "2023-01-15", "2023-01-15", "2023-01-18"]),
    "heart_rate": [72, -10, 85, 85, 220]
})

claims_df = pd.DataFrame({
    "member_id": ["001", "002", "003", "004"],
    "diagnosis_code": ["I10", "E11", "E11", "I10"],
    "claim_amount": [12000, 18000, 15000, 20000]
})

labs_df = pd.DataFrame({
    "lab_patient_id": ["P-001", "P-002", "P-003", "P-004"],
    "lab_date": pd.to_datetime(["2023-01-08", "2023-01-09", "2023-01-14", "2023-01-17"]),
    "lab_value": [100, 200, 150, 250]
})
```

```
        "glucose": [95, 5.5, 110, 6.2],  
        "unit": ["mg/dL", "mmol/L", "mg/dL", "mmol/L"]  
    })
```

▼ Step 1: Inspect dataset structure

What is expected

- Display the number of rows and columns of the EHR dataset
- Display all column names and their data types (For example, by showing the dataset shape and column data types.)

```
print(ehr_df.shape)  
ehr_df.info()  
  
(5, 6)  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5 entries, 0 to 4  
Data columns (total 6 columns):  
 #   Column           Non-Null Count  Dtype     
 ---  --     
 0   patient_id      5 non-null     object    
 1   patient_name    5 non-null     object    
 2   dob              5 non-null     datetime64[ns]   
 3   encounter_id    5 non-null     object    
 4   encounter_date  5 non-null     datetime64[ns]   
 5   heart_rate      5 non-null     int64     
 dtypes: datetime64[ns](2), int64(1), object(3)  
 memory usage: 372.0+ bytes
```

▼ Step 2: Identify duplicate records

What is expected

- Print the total number of duplicate rows
- Show which rows are duplicates using a clear row-level indicator (For example, by displaying a row-level indicator showing True/False for duplicates.)

```
print(ehr_df.duplicated().sum())
ehr_df.duplicated()
```

```
1
0
_____
0 False
1 False
2 False
3 True
4 False

dtype: bool
```

✓ Step 3: Remove duplicate records

What is expected

- Remove duplicate rows from the dataset
- Print the updated dataset shape (rows and columns)

```
ehr_df = ehr_df.drop_duplicates()
ehr_df.shape

(4, 6)
```

✓ Step 4: Identify invalid clinical values

What is expected

- Print any rows where heart rate values are invalid (less than zero)

```
ehr_df[ehr_df["heart_rate"] < 0]
```

	patient_id	patient_name	dob	encounter_id	encounter_date	heart_rate
1	P002	Jane Smith	1975-09-30	E2	2023-01-12	-10

▼ Step 5: Correct invalid clinical values

What is expected

- Replace invalid heart rate values with missing values (NaN)
- Print the updated heart rate column

```
ehr_df.loc[ehr_df["heart_rate"] < 0, "heart_rate"] = np.nan  
ehr_df["heart_rate"]
```

```
heart_rate  
0    72.0  
1    NaN  
2    85.0  
4   220.0
```

dtype: float64

▼ Step 6: Standardize patient identifiers across sources

What is expected

- Create a standardized patient identifier column
- Print the standardized identifiers from all three datasets

```
ehr_df["pid"] = ehr_df["patient_id"].str[1:]  
claims_df["pid"] = claims_df["member_id"]  
labs_df["pid"] = labs_df["lab_patient_id"].str.replace("P-", "")
```

```
print("EHR pid:")
print(ehr_df["pid"])

print("\nClaims pid:")
print(claims_df["pid"])

print("\nLabs pid:")
print(labs_df["pid"])

EHR pid:
0    001
1    002
2    003
4    004
Name: pid, dtype: object
```

```
Claims pid:
0    001
1    002
2    003
3    004
Name: pid, dtype: object
```

```
Labs pid:
0    001
1    002
2    003
3    004
Name: pid, dtype: object
```

▼ Step 7: Map diagnosis codes to clinical conditions

What is expected

- Map diagnosis codes to readable clinical condition names
- Print diagnosis codes alongside mapped conditions

```
claims_df["condition"] = claims_df["diagnosis_code"].map(
    {"I10": "Hypertension", "E11": "Diabetes"}
)
```

```
claims_df[["diagnosis_code","condition"]]
```

	diagnosis_code	condition
0	I10	Hypertension
1	E11	Diabetes
2	E11	Diabetes
3	I10	Hypertension

▼ Step 8: Resolve laboratory unit mismatches

What is expected

- Convert all glucose values to a consistent unit (mg/dL)
- Print original values, units, and converted values

```
labs_df["glucose_mgdl"] = np.where(  
    labs_df["unit"]=="mmol/L",  
    labs_df["glucose"]*18,  
    labs_df["glucose"]  
)  
  
labs_df[["glucose","unit","glucose_mgdl"]]
```

	glucose	unit	glucose_mgdl
0	95.0	mg/dL	95.0
1	5.5	mmol/L	99.0
2	110.0	mg/dL	110.0
3	6.2	mmol/L	111.6

▼ Step 9: Merge EHR, claims, and laboratory data

What is expected

- Merge all datasets using the standardized patient identifier
- Display the column names to confirm the merge
- Display key fields from each source system (for example, a patient identifier, an encounter field, a diagnosis field, and a laboratory value) to verify the merged result

```
merged_df = ehr_df.merge(claims_df, on="pid").merge(labs_df, on="pid")

# Display column names to confirm merge across datasets
print(merged_df.columns)

# Display key identifiers to confirm successful merge
merged_df[["pid", "encounter_id", "diagnosis_code", "glucose_mgdl"]]

Index(['patient_id', 'patient_name', 'dob', 'encounter_id', 'encounter_date',
       'heart_rate', 'pid', 'member_id', 'diagnosis_code', 'claim_amount',
       'condition', 'lab_patient_id', 'lab_date', 'glucose', 'unit',
       'glucose_mgdl'],
      dtype='object')

  pid  encounter_id  diagnosis_code  glucose_mgdl
0  001          E1           I10      95.0
1  002          E2           E11      99.0
2  003          E3           E11     110.0
3  004          E4           I10     111.6
```

▼ Step 10: Apply HIPAA safe harbor de-identification

What is expected

- Remove the patient name column, which represents a direct identifier in this dataset
- Print the column list to confirm that the patient_name column has been removed
- Create a generalized year-of-birth field from the date of birth (for example, extracting the year)
- To keep the output readable for grading, **print only the patient identifier and birth year columns**

Note: The full dataset should still be updated. You are only limiting what is printed.

```
merged_df = merged_df.drop(columns=["patient_name"])
merged_df["birth_year"] = merged_df["dob"].dt.year
# Display the column names to show that patient_name is no longer present
print(merged_df.columns)
# Display the patient identifier and generalized birth year columns
merged_df[["pid", "birth_year"]]

Index(['patient_id', 'dob', 'encounter_id', 'encounter_date', 'heart_rate',
       'pid', 'member_id', 'diagnosis_code', 'claim_amount', 'condition',
       'lab_patient_id', 'lab_date', 'glucose', 'unit', 'glucose_mgdl',
       'birth_year'],
      dtype='object')

  pid  birth_year
0  001        1980
1  002        1975
2  003        1990
3  004        1965
```

▼ Step 11: Create encounter utilization features

What is expected

- Count the number of encounters per patient
- Print the encounter count for each patient using the patient identifier

```
encounter_counts = merged_df.groupby("pid")["encounter_id"].nunique()
encounter_counts
```

```
encounter_id  
pid  
---  
001      1  
002      1  
003      1  
004      1  
  
dtype: int64
```

✓ Step 12: Engineer final model-ready features

What is expected

- Aggregate patient-level features
- Create a final feature table with one row per patient that includes a glucose-related feature and an encounter count
- Print the final feature table

```
merged_df.groupby("pid").agg(  
    avg_glucose=("glucose_mgdl","mean"),  
    encounters=("encounter_id","nunique")  
)
```

	avg_glucose	encounters
pid		
001	95.0	1
002	99.0	1
003	110.0	1
004	111.6	1

