04_Feature_Engineering

June 8, 2025

1 Feature Engineering

1.0.1 Setup for Feature Engineering

At the beginning of the feature engineering phase, the same plotting configuration and utility function used in EDA are retained. This ensures consistency in the visual inspection of transformed or newly constructed features, which is particularly valuable when validating assumptions or diagnosing feature quality.

The `plot_histogram()` function remains a central tool for assessing the distribution, modality, and potential skewness of both raw and derived features. As new variables are engineered—especially from dynamic tables (e.g., `CHARTEVENTS`, `INPUTEVENTS_MV`) or through aggregations (e.g., mean, std, skew)—visual confirmation becomes essential.

Maintaining visual standards across exploratory and engineering phases reflects good scientific rigor, supports reproducibility, and facilitates documentation for thesis-level reporting.

[ ]:
```
EXPORT_PATH = "../data/processed/"
ASSETS_PATH = "../assets/plots/eda/"

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

# === Plot Style ===
sns.set(style="whitegrid")
plt.rcParams["figure.figsize"] = (10, 6)
def plot_histogram(
    data, column, bins=30, kde=True, figsize=(10, 4),
    title=None, xlabel=None, ylabel="Number of Patients",
    save_path=None
):
    plt.figure(figsize=figsize)
    sns.histplot(data[column], bins=bins, kde=kde)
    plt.title(title if title else f"{column} Distribution")
    plt.xlabel(xlabel if xlabel else column)
    plt.ylabel(ylabel)
```

```
    plt.tight_layout()
    if save_path:
        plt.savefig(save_path)
    plt.show()
```

[ ]:
```
# === Load dataset ===
df_final = pd.read_csv(os.path.join(EXPORT_PATH, "df_final_static.csv"))

# === Confirm structure ===
print(df_final.shape)
df_final.head()
```

```
(3685, 16)
```

[ ]:

	SUBJECT_ID	HADM_ID	ICUSTAY_ID	AGE	GENDER	ADMISSION_TYPE	ADMISSION_LOCATION	INSURANCE	FIRST_CAREUNIT	LOS	HOSPITAL_EXPIRE_FLAG	INTIME_HOUR	INTIME_WEEKDAY	ADMITTIME_HOUR	ADMITTIME_WEEKDAY	INTIME
0	269	106296	206613	40	M	EMERGENCY	EMERGENCY ROOM ADMIT	Medicaid	MICU	3.2788	0	11	0	11	0	2170-11-05 11:05:29
1	275	129886	219649	82	M	EMERGENCY	EMERGENCY ROOM ADMIT	Medicare	CCU	7.1314	1	11	6	3	5	2170-10-07 11:28:53
2	292	179726	222505	57	F	URGENT	TRANSFER FROM HOSP/EXTRAM	Private	MICU	0.8854	1	18	3	18	3	2103-09-27 18:29:30
3	305	194340	217232	76	F	EMERGENCY	TRANSFER FROM HOSP/EXTRAM	Medicare	SICU	2.4370	1	12	5	18	5	2129-09-03 12:31:31
4	323	143334	264375	57	M	EMERGENCY	EMERGENCY ROOM ADMIT	Medicare	MICU	3.0252	0	15	3	15	3	2120-01-11 15:48:28

1.1 Adding Dynamic Features (Temporal Aggregation)

This block initiates the temporal feature engineering process by mapping clinically relevant **vital signs** to their corresponding ITEMIDs in the MIMIC-III database, as per official documentation and clinical guidelines.

Each vital sign (e.g., Heart Rate, Systolic Blood Pressure, Temperature) may be recorded under multiple ITEMIDs due to differences in equipment, measurement protocols, or care units. Grouping these codes ensures that all valid measurements are captured uniformly across patients and time points.

The dictionary `vital_items` serves as a reference map, organizing ITEMIDs under semantic labels. The flattened `itemid_to_label` dictionary enables rapid reverse lookup from an individual ITEMID to its physiological label—a crucial step for categorizing and aggregating measurements in downstream steps.

This systematic mapping allows the CHARTEVENTS table, which is rich but messy, to be filtered and interpreted in a clinically coherent manner, transforming it from a semi-structured log to a set of analyzable features.

[ ]:
```
# Define ITEMIDs per MIMIC-III documentation for vital signs
vital_items = {
    "Heart Rate": [211, 220045],
    "Systolic BP": [51, 455, 220179, 220050],
    "Diastolic BP": [8368, 8441, 220180, 220051],
    "Mean BP": [52, 456, 220052],
    "Respiratory Rate": [618, 220210],
    "Temperature": [678, 223761],
    "SpO2": [646, 220277],
    "Glucose": [807, 220621]
}

itemid_to_label = {item: label for label, items in vital_items.items() for item␣
 ↪in items}
```

1.1.1 Temporal Filtering and Labeling of Vital Signs from CHARTEVENTS

This block transforms the raw `CHARTEVENTS` table—one of the most voluminous and granular tables in MIMIC-III—into a temporally-filtered and semantically-labeled set of measurements for feature engineering.

1. **Data Import and Merging**: `chartevents_sepsis.csv`, a filtered export of `CHARTEVENTS`, is joined with the ICU cohort on `ICUSTAY_ID`. This operation ensures that only ICU stays of interest (i.e., sepsis patients) are considered, and that the timestamp `INTIME` of each ICU admission is accessible for temporal alignment.
2. **Time Window Filtering (0–24h)**: A new variable `HOURS_FROM_INTIME` is computed to measure the number of hours elapsed from ICU admission to each recorded event. Only events occurring in the first 24 hours are retained. This window is clinically motivated: early vital sign patterns often serve as early warning signals and are crucial for predictive modeling.
3. **Item and Value Filtering**: Only events with a recognized `ITEMID` (from the `itemid_to_label` dictionary) and non-null `VALUENUM` are retained. This ensures semantic clarity and numerical integrity. Each event is then annotated with a `VITAL_TYPE`, enabling grouping and statistical aggregation in subsequent steps.

This pipeline transforms millions of event-level entries into a manageable and interpretable structure. It is both **clinically sound** and **computationally efficient**, paving the way for robust temporal feature engineering.

[ ]:
```
# Load CHARTEVENTS and cohort ICU
chartevents = pd.read_csv(EXPORT_PATH + "chartevents_sepsis.csv",␣
 ↪parse_dates=["CHARTTIME"])
cohort_icu = pd.read_csv(EXPORT_PATH + "df_final_static.csv")

# Join CHARTEVENTS with cohort ICU
first24h = chartevents.merge(cohort_icu[["ICUSTAY_ID", "INTIME"]],␣
 ↪on="ICUSTAY_ID", how="inner")
first24h["HOURS_FROM_INTIME"] = (first24h["CHARTTIME"] - pd.
 ↪to_datetime(first24h["INTIME"])).dt.total_seconds() / 3600
first24h = first24h[(first24h["HOURS_FROM_INTIME"] >= 0) &␣
 ↪(first24h["HOURS_FROM_INTIME"] <= 24)]

# Filter valid ITEMIDs and non-null VALUENUM
first24h = first24h[first24h["ITEMID"].isin(itemid_to_label)]
first24h = first24h[first24h["VALUENUM"].notnull()]
first24h["VITAL_TYPE"] = first24h["ITEMID"].map(itemid_to_label)
```

```
/var/folders/0j/nhv3j29j5bngf6nym9kpvhl80000gn/T/ipykernel_58489/3589101290.py:2
: DtypeWarning: Columns (5) have mixed types. Specify dtype option on import or
set low_memory=False.
  chartevents = pd.read_csv(EXPORT_PATH + "chartevents_sepsis.csv",
parse_dates=["CHARTTIME"])
```

1.1.2 Temporal Aggregation of Vital Signs in First 24 Hours

This block performs statistical aggregation of vital signs collected in the first 24 hours of ICU stay. These aggregations yield **hand-crafted features** that capture the distributional behavior of each physiological parameter over the early hours of critical illness.

1. **Grouping by VITAL_TYPE**: For each predefined vital sign (e.g., "Heart Rate", "SpO2"), the subset of data entries is filtered from `first24h` using the label from `VITAL_TYPE`.
2. **Statistical Aggregation**: For each ICU stay (`ICUSTAY_ID`), the following descriptive statistics are computed on the `VALUENUM` of the vital sign:
   - **mean**: central tendency
   - **std**: variation/spread
   - **min**/**max**: range
   - **count**: data availability (proxy for measurement density)
   - **skew**: asymmetry in the distribution
3. **Feature Naming**: Feature names are standardized using uppercase transformation and concatenation of the vital sign with the statistic (e.g., `HEART_RATE_MEAN`, `SPO2_STD`).
4. **Final Merge**: The resulting per-vital DataFrames are merged using outer joins on

ICUSTAY_ID, ensuring that missing values are preserved for downstream imputation rather than excluded prematurely.

The final df_vitals table contains one row per ICU stay, with one column per statistical property of each vital sign. This structured matrix is ready to be merged with the static cohort (df_final_static.csv) and used in modeling pipelines.

[ ]:
```
# Aggregation per ICUSTAY_ID and VITAL_TYPE
vital_features = []

for label in vital_items.keys():
    temp = first24h[first24h["VITAL_TYPE"] == label]
    stats = temp.groupby("ICUSTAY_ID")["VALUENUM"].agg(["mean", "std", "min",␣
 ↪"max", "count", "skew"]).reset_index()
    stats.columns = ["ICUSTAY_ID"] + [f"{label.upper().replace(' ', '_')}_{stat.
 ↪upper()}" for stat in stats.columns[1:]]
    vital_features.append(stats)

# Merge all together
from functools import reduce
df_vitals = reduce(lambda left, right: pd.merge(left, right, on="ICUSTAY_ID",␣
 ↪how="outer"), vital_features)
```

1.1.3 Final Dataset Assembly: Merging Static and Dynamic Features

In this final stage of feature engineering, the previously constructed temporal features (df_vitals)—derived from physiological signals measured during the first 24 hours of ICU admission—are merged with the static cohort dataset (df_final_static.csv), which contains demographic, admission, and administrative information. * **Merge Operation**: The join is performed on the ICUSTAY_ID key using a **left join**, which ensures that all records from the static dataset are preserved—even if some ICU stays have missing or incomplete time-series data. This design is essential for maintaining the full patient cohort and handling missing data explicitly during preprocessing.

- **Export for Downstream Tasks**: The final dataset df_final_enriched is saved to disk. It now includes both static attributes (e.g., age, gender, admission type) and temporal descriptors (e.g., mean and variability of heart rate, blood pressure, etc.), forming a **rich, multimodal feature space** ideal for supervised learning.

This unified dataset becomes the **central input** for the next phase—model training and validation—and represents a carefully engineered structure that reflects both clinical relevance and data integrity.

[ ]:
```
# Merge dynamic features into df_final
df_final = pd.read_csv(EXPORT_PATH + "df_final_static.csv")
df_enriched = df_final.merge(df_vitals, on="ICUSTAY_ID", how="left")

# Save to disk
df_enriched.to_csv(EXPORT_PATH + "df_final_enriched.csv", index=False)
```

```
print(f"Final enriched dataset shape: {df_enriched.shape}")
```

```
Final enriched dataset shape: (3685, 64)
```

1.2 Data Cleaning

1.2.1 Missing Data Profiling: Assessing Feature Completeness

This step performs a systematic assessment of missing data across the enriched dataset (`df_enriched`), with the goal of identifying features that may require imputation, exclusion, or special handling prior to model training.

By computing the proportion of `NaN` values for each feature and sorting the results in descending order, the analysis highlights which variables have the most severe completeness issues. Features with more than **10% missing values** are particularly critical, as they may:

- Bias model learning if left unaddressed
- Affect generalization if their distribution differs between train and test sets
- Reduce interpretability, especially in clinical contexts where data sparsity reflects operational constraints

Reporting the **total number of features** (`len(missing_data)`) provides an overview of the feature space size and supports the justification of future dimensionality reduction or feature selection strategies.

This diagnostic serves as the foundation for a rational and reproducible missing data handling policy—an essential component of any robust machine learning pipeline, particularly in healthcare applications.

[ ]:
```
# Print missing data statistics greater than 10%
missing_data = df_enriched.isnull().sum().sort_values(ascending=False) /␣
  ↪len(df_enriched)
print(missing_data)
print(len(missing_data))
```

```
GLUCOSE_SKEW          0.808412
MEAN_BP_SKEW          0.797829
MEAN_BP_STD           0.794301
MEAN_BP_COUNT         0.790231
MEAN_BP_MIN           0.790231
                          …
FIRST_CAREUNIT        0.000000
GENDER                0.000000
INSURANCE             0.000000
ADMISSION_LOCATION    0.000000
SUBJECT_ID            0.000000
Length: 64, dtype: float64
64
```

1.2.2 Feature Pruning Based on Missingness Threshold

In this step, variables with excessive proportions of missing values are systematically removed from the dataset. Specifically, features with more than **49% missing data** are discarded entirely, following the rationale that highly sparse variables contribute little to predictive power and may introduce instability during imputation or modeling.

Rationale for the 49% Threshold:

- Variables with over half their values missing lack sufficient representation to allow reliable learning of patterns.
- Retaining such features often results in **uninformative noise**, increased dimensionality, and increased variance in downstream models.
- By removing only the worst-offending variables, the procedure preserves the majority of potentially informative features while improving the dataset's statistical robustness.

The list of removed features is stored in `features_to_remove`, allowing traceability and reproducibility of preprocessing steps—an essential requirement in scientific data workflows.

The final dataset `df` now has improved completeness and is better suited for subsequent imputation and model training.

[ ]:
```
df = df_enriched.copy()
# Remove features with more than 60% missing data
features_to_remove = missing_data[missing_data > 0.49].index.tolist()
# Drop features with more than 60% missing data
df.drop(columns=features_to_remove, inplace=True)
# Print the number of features removed
len(features_to_remove)
```

[ ]: 26

1.2.3 Advanced Missing Value Imputation via Iterative Imputer (MICE)

To handle missing data in numerical features, this step employs the **Iterative Imputer** from `scikit-learn`, a sophisticated approach that models each incomplete feature as a function of the others in a **Bayesian regression-like framework**. This method—commonly referred to as MICE—is particularly advantageous in healthcare datasets where variable interdependence is high and simple imputation (e.g., mean or median) may fail to capture complex relationships.

- The dataset is first filtered to retain only numerical columns, ensuring that the imputer operates on continuous and discrete numeric values.
- The `IterativeImputer` is applied, estimating missing values by iteratively modeling each column using all other columns as predictors. This preserves the **multivariate distributional structure** of the dataset.
- The imputed matrix is then converted back into a `pandas` DataFrame with restored column names.
- If non-numeric columns (e.g., categorical variables) were excluded from imputation, they are reattached to the imputed frame using `pd.concat()`.

This method significantly enhances the robustness of the feature space by **preserving inter-feature correlations** and minimizing information loss, which is particularly important for downstream models sensitive to missingness, such as neural networks or tree ensembles.

[ ]:
```
from sklearn.experimental import enable_iterative_imputer  # noqa
from sklearn.impute import IterativeImputer
import pandas as pd

# Filtra solo colonne numeriche
df_numeric = df.select_dtypes(include='number')

# Imputazione iterativa
iter_imputer = IterativeImputer()
df_imputed = pd.DataFrame(
    iter_imputer.fit_transform(df_numeric),
    columns=df_numeric.columns
)

# Se vuoi reinserire le colonne non numeriche:
df_final = pd.concat([df_imputed, df.drop(columns=df_numeric.columns)], axis=1)

df_final.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3685 entries, 0 to 3684
Data columns (total 38 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   SUBJECT_ID              3685 non-null   float64
 1   HADM_ID                 3685 non-null   float64
 2   ICUSTAY_ID              3685 non-null   float64
 3   AGE                     3685 non-null   float64
 4   LOS                     3685 non-null   float64
 5   HOSPITAL_EXPIRE_FLAG    3685 non-null   float64
 6   INTIME_HOUR             3685 non-null   float64
 7   INTIME_WEEKDAY          3685 non-null   float64
 8   ADMITTIME_HOUR          3685 non-null   float64
 9   ADMITTIME_WEEKDAY       3685 non-null   float64
 10  HEART_RATE_MEAN         3685 non-null   float64
 11  HEART_RATE_STD          3685 non-null   float64
 12  HEART_RATE_MIN          3685 non-null   float64
 13  HEART_RATE_MAX          3685 non-null   float64
 14  HEART_RATE_COUNT        3685 non-null   float64
 15  HEART_RATE_SKEW         3685 non-null   float64
 16  RESPIRATORY_RATE_MEAN   3685 non-null   float64
 17  RESPIRATORY_RATE_STD    3685 non-null   float64
 18  RESPIRATORY_RATE_MIN    3685 non-null   float64
 19  RESPIRATORY_RATE_MAX    3685 non-null   float64
```

```
 20  RESPIRATORY_RATE_COUNT  3685 non-null   float64
 21  RESPIRATORY_RATE_SKEW   3685 non-null   float64
 22  SPO2_MEAN               3685 non-null   float64
 23  SPO2_STD                3685 non-null   float64
 24  SPO2_MIN                3685 non-null   float64
 25  SPO2_MAX                3685 non-null   float64
 26  SPO2_COUNT              3685 non-null   float64
 27  SPO2_SKEW               3685 non-null   float64
 28  GLUCOSE_MEAN            3685 non-null   float64
 29  GLUCOSE_MIN             3685 non-null   float64
 30  GLUCOSE_MAX             3685 non-null   float64
 31  GLUCOSE_COUNT           3685 non-null   float64
 32  GENDER                  3685 non-null   object
 33  ADMISSION_TYPE          3685 non-null   object
 34  ADMISSION_LOCATION      3685 non-null   object
 35  INSURANCE               3685 non-null   object
 36  FIRST_CAREUNIT          3685 non-null   object
 37  INTIME                  3685 non-null   object
dtypes: float64(32), object(6)
memory usage: 1.1+ MB
```

1.3 Scaling

1.3.1 Feature Selection for Scaling: Isolating Numeric Predictors

In this preprocessing step, a targeted list of numeric features is extracted in preparation for feature scaling. The operation is designed to exclude variables that:

1. Serve only as **identifiers** (`SUBJECT_ID`, `HADM_ID`, `ICUSTAY_ID`)
2. Represent **timestamps or datetime-derived values** (`INTIME`)
3. Are **categorical or binary** but encoded as object/string (`GENDER`, `ADMISSION_TYPE`, etc.)
4. Reflect **target or outcome variables** (`HOSPITAL_EXPIRE_FLAG`) that should not be included as predictors

The remaining columns—stored in `numeric_cols`—represent the **true set of continuous or discrete numerical features** that are appropriate for normalization. These typically include:

- Aggregated vital signs (mean, std, min, max, etc.)
- Demographic variables (e.g., `AGE`)
- Temporally derived metrics (e.g., `ADMITTIME_HOUR`, `INTIME_WEEKDAY`)

This filtering step is crucial for ensuring that scaling is applied **only where semantically and statistically appropriate**, thereby avoiding distortions in categorical or identifier features.

[ ]:
```python
# Exclude identifier and non-numeric columns
exclude_cols = [
    "SUBJECT_ID", "HADM_ID", "ICUSTAY_ID", "INTIME", "GENDER",
    "ADMISSION_TYPE", "ADMISSION_LOCATION", "INSURANCE", "FIRST_CAREUNIT",
    'HOSPITAL_EXPIRE_FLAG'
]
```

```
numeric_cols = [col for col in df.columns if col not in exclude_cols and np.
  ↪issubdtype(df[col].dtype, np.number)]
numeric_cols
```

[ ]:
```
['AGE',
 'LOS',
 'INTIME_HOUR',
 'INTIME_WEEKDAY',
 'ADMITTIME_HOUR',
 'ADMITTIME_WEEKDAY',
 'HEART_RATE_MEAN',
 'HEART_RATE_STD',
 'HEART_RATE_MIN',
 'HEART_RATE_MAX',
 'HEART_RATE_COUNT',
 'HEART_RATE_SKEW',
 'RESPIRATORY_RATE_MEAN',
 'RESPIRATORY_RATE_STD',
 'RESPIRATORY_RATE_MIN',
 'RESPIRATORY_RATE_MAX',
 'RESPIRATORY_RATE_COUNT',
 'RESPIRATORY_RATE_SKEW',
 'SPO2_MEAN',
 'SPO2_STD',
 'SPO2_MIN',
 'SPO2_MAX',
 'SPO2_COUNT',
 'SPO2_SKEW',
 'GLUCOSE_MEAN',
 'GLUCOSE_MIN',
 'GLUCOSE_MAX',
 'GLUCOSE_COUNT']
```

1.3.2 Analyze AGE Distribution

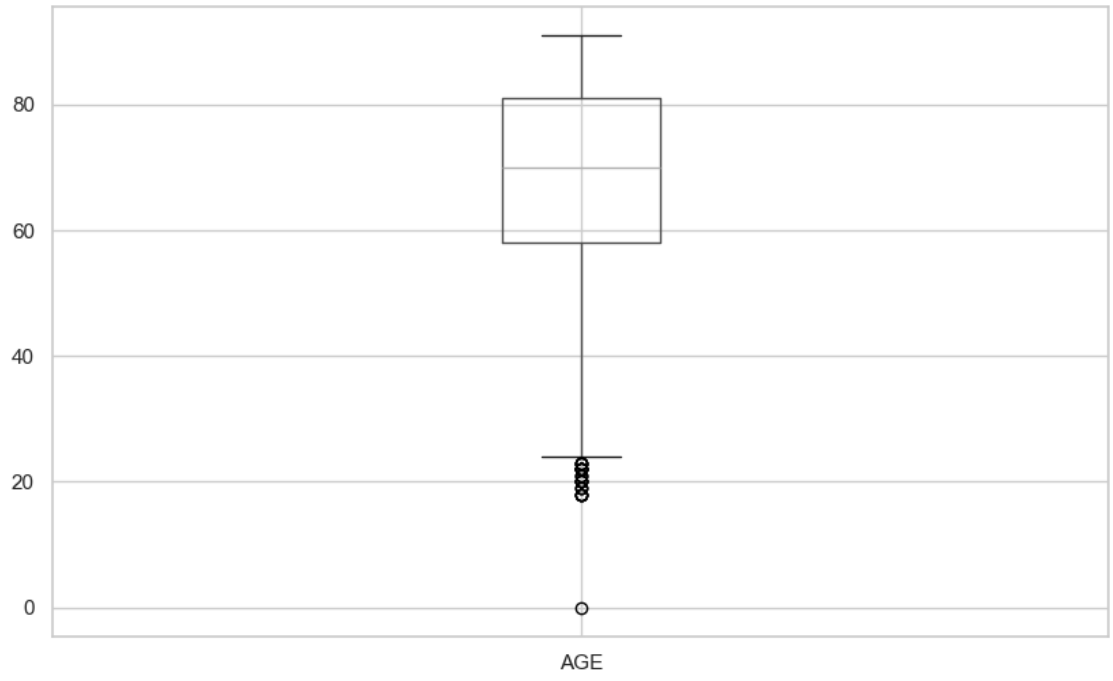
In this step, the `AGE` variable is normalized using **Min-Max Scaling**, a transformation that linearly maps the original values to the [0, 1] interval. This scaling method preserves the relative ordering and proportional differences between values, while ensuring that all features contribute equally in models sensitive to scale.

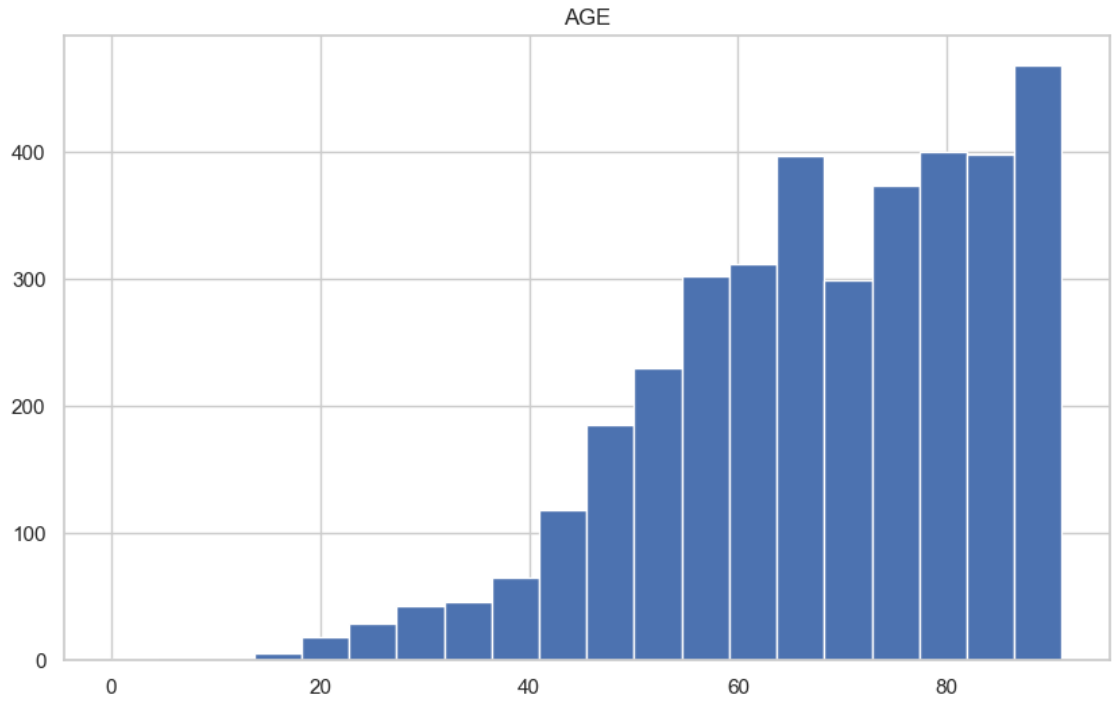
- **Boxplot** (Top): After scaling, the boxplot still reveals mild outliers at the lower end of the age spectrum (younger patients), but the distribution is compressed within a bounded interval. The upper whisker is capped at 1.0, corresponding to the censoring of elderly patients at age 91 in the MIMIC-III dataset.
- **Histogram** (Bottom): The histogram shows a **right-skewed distribution**, reflecting the overrepresentation of older patients in the ICU. This is consistent with the clinical reality of sepsis being more prevalent among elderly populations.

The normalization ensures that **AGE** does not disproportionately dominate learning algorithms and is especially important when combining it with other scaled physiological features.

[ ]:
```
df[['AGE']].boxplot() # Boxplot
df[['AGE']].hist(bins=20) # Histogram
```

[ ]:
```
array([[<Axes: title={'center': 'AGE'}>]], dtype=object)
```






```
[ ]: # Utilizzo MinMaxScaler per normalizzare la colonna 'AGE'
from sklearn.preprocessing import MinMaxScaler

minmaxscaler = MinMaxScaler().fit(df[['AGE']])
df['AGE']= minmaxscaler.transform(df[['AGE']])
```

1.3.3 Normalization and Distribution of ICU Admission Times

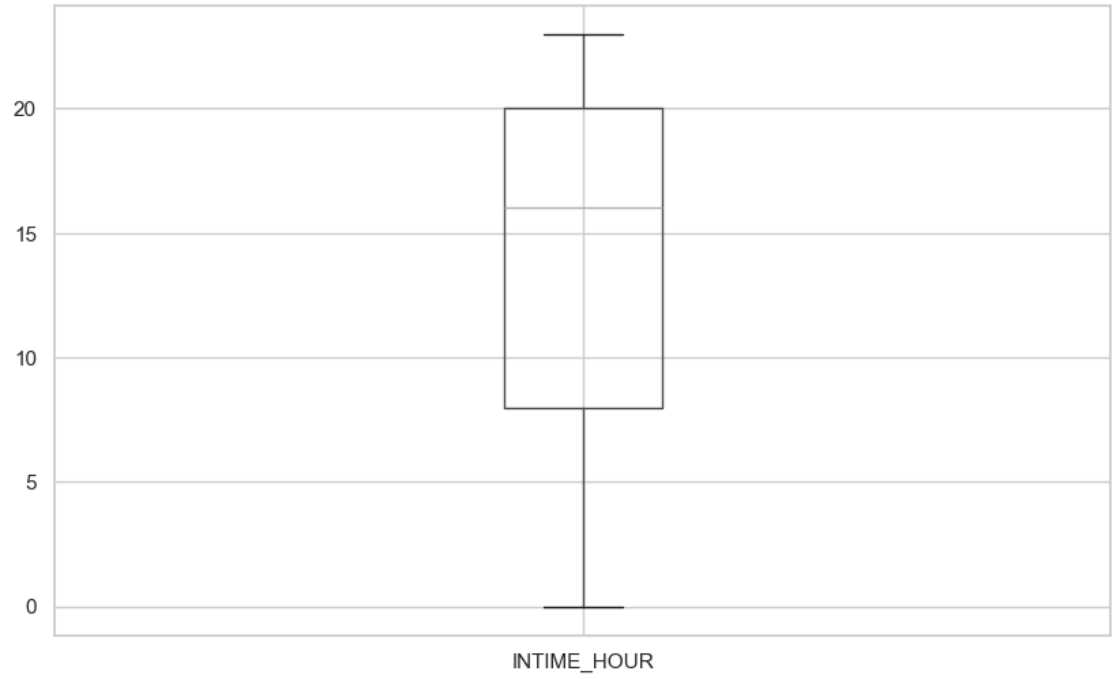
In this preprocessing step, the variables `INTIME_HOUR` (hour of ICU admission) and `INTIME_WEEKDAY` (day of week) are normalized using **Min-Max Scaling** to fit within the [0, 1] range. These features are particularly relevant for identifying **temporal admission patterns**, which may indirectly reflect ICU operational practices, staffing levels, or patient triage protocols.

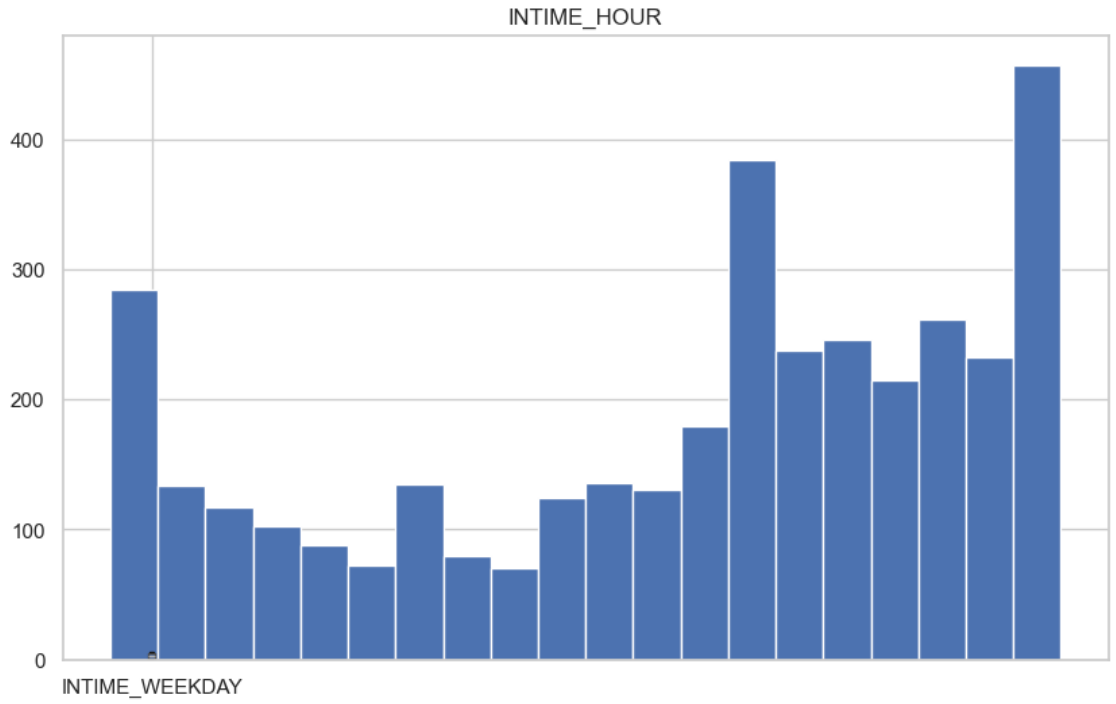
- **INTIME_HOUR**:
  - The histogram reveals a **bimodal distribution**, with noticeable spikes around **early morning (0–1)** and **evening (23–0)**. These peaks may correspond to operational shifts or protocol-based transfers.
  - The boxplot confirms a wide spread of admissions throughout the 24-hour cycle, with outliers mostly in the early morning hours.
- **INTIME_WEEKDAY**:
  - The distribution is relatively **uniform across weekdays**, with only slight variations. This suggests that sepsis-related ICU admissions are not heavily influenced by the day of the week, confirming the **acute and emergent** nature of the condition.

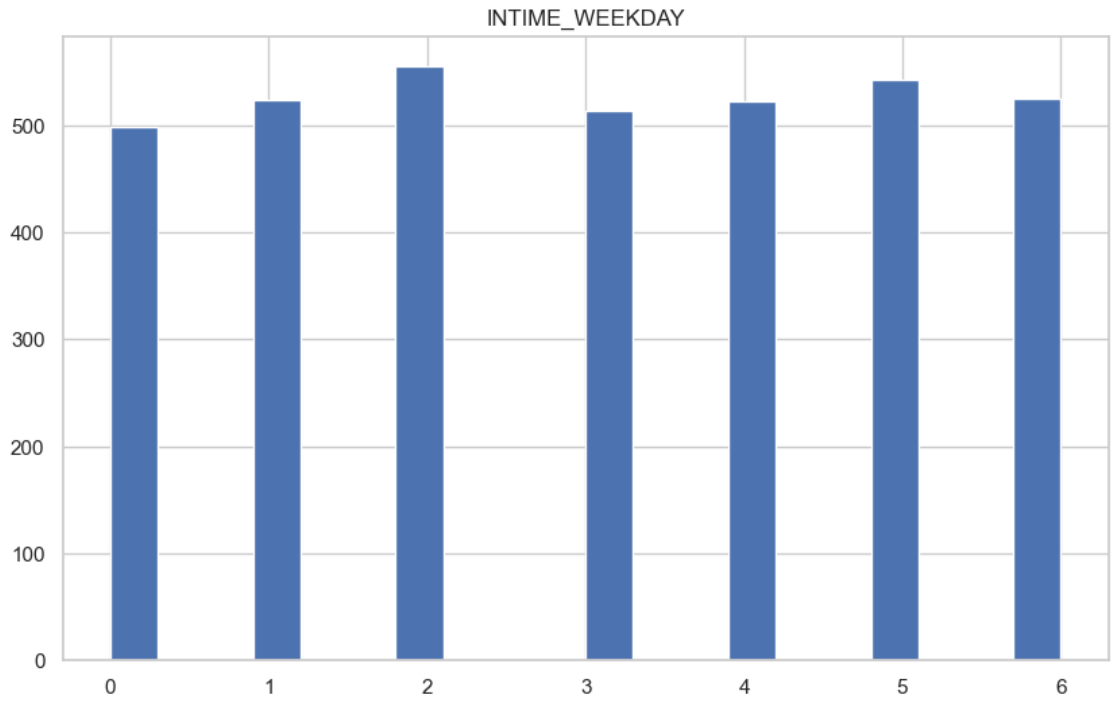
Min-Max Scaling ensures that both of these variables are appropriately rescaled for inclusion in models that assume normalized input. While these features may not carry predictive weight individually, they can become informative when interacting with clinical covariates or during **SHAP-based interpretability analysis**.

[ ]:
```
df_final[['INTIME_HOUR']].boxplot()
df_final[['INTIME_HOUR']].hist(bins=20)

df_final[['INTIME_WEEKDAY']].boxplot()
df_final[['INTIME_WEEKDAY']].hist(bins=20)
```

[ ]:
```
array([[<Axes: title={'center': 'INTIME_WEEKDAY'}>]], dtype=object)
```




INTIME_HOUR
400
300
200
100
0
INTIME_WEEKDAY


INTIME_WEEKDAY
500
400
300
200
100
0
0
1
2
3
4
5
6

[ ]:
```
df['INTIME_HOUR'] = minmaxscaler.fit_transform(df[['INTIME_HOUR']])
df['INTIME_WEEKDAY'] = minmaxscaler.fit_transform(df[['INTIME_WEEKDAY']])

df[['INTIME_HOUR', 'INTIME_WEEKDAY']].head()
```

[ ]:

	INTIME_HOUR	INTIME_WEEKDAY
0	0.478261	0.000000
1	0.478261	1.000000
2	0.782609	0.500000
3	0.521739	0.833333
4	0.652174	0.500000

1.3.4 Normalization and Temporal Pattern Analysis of Hospital Admission Times

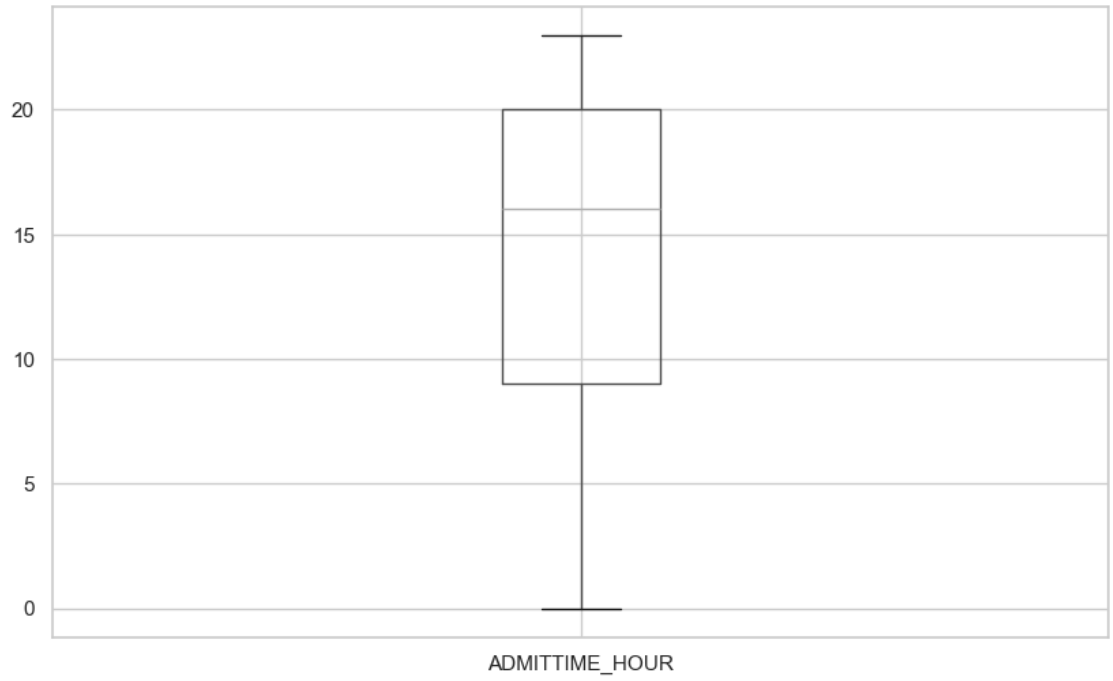
The current preprocessing step addresses two additional temporal variables: the **hour** and **weekday** of hospital admission (`ADMITTIME_HOUR` and `ADMITTIME_WEEKDAY`). These variables are normalized using **Min-Max Scaling**, ensuring they are on the same scale as other features used in model training.

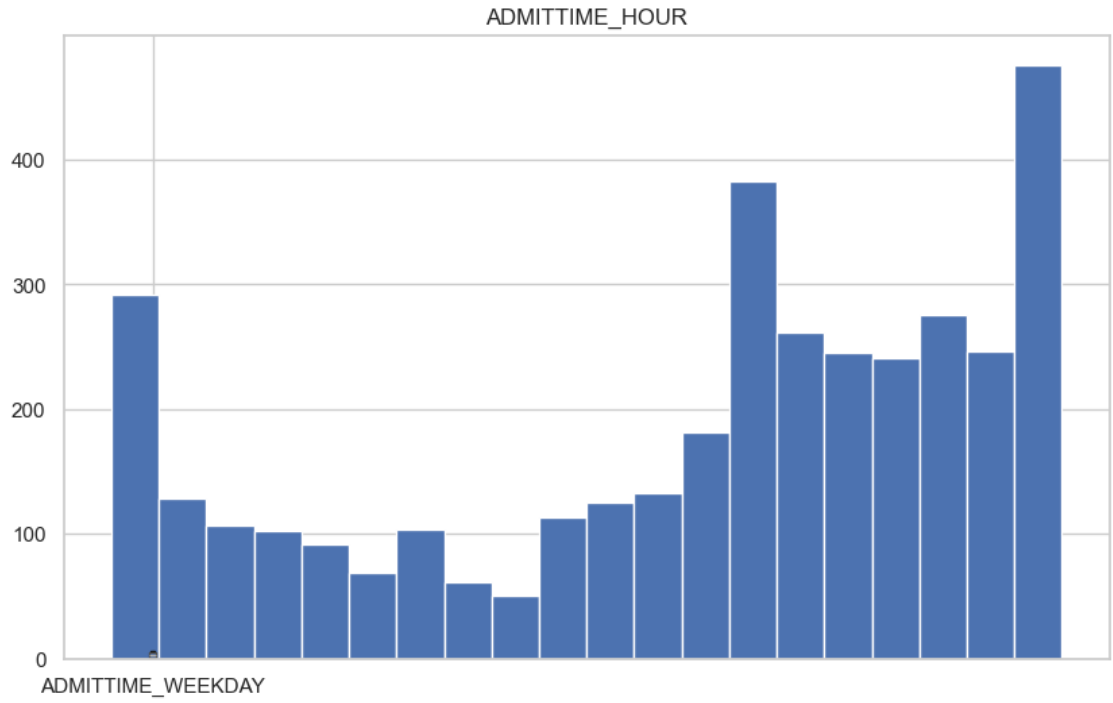
- **ADMITTIME_HOUR**:
  - The histogram reveals an **asymmetric bimodal distribution**, with spikes during early morning (around 0–1h) and evening (22–23h). This likely reflects hospital routines or transfer timings—patients are frequently admitted either just after midnight (when beds are reassigned) or late evening (when emergency departments stabilize patients).
  - The boxplot confirms this spread and indicates no extreme outliers post-normalization.
- **ADMITTIME_WEEKDAY**:
  - As with ICU admission day, hospital admissions are **fairly evenly distributed** across the week. A slight increase on Tuesdays and Sundays may suggest system-level factors such as delayed weekend triage or Monday backlog resolution.

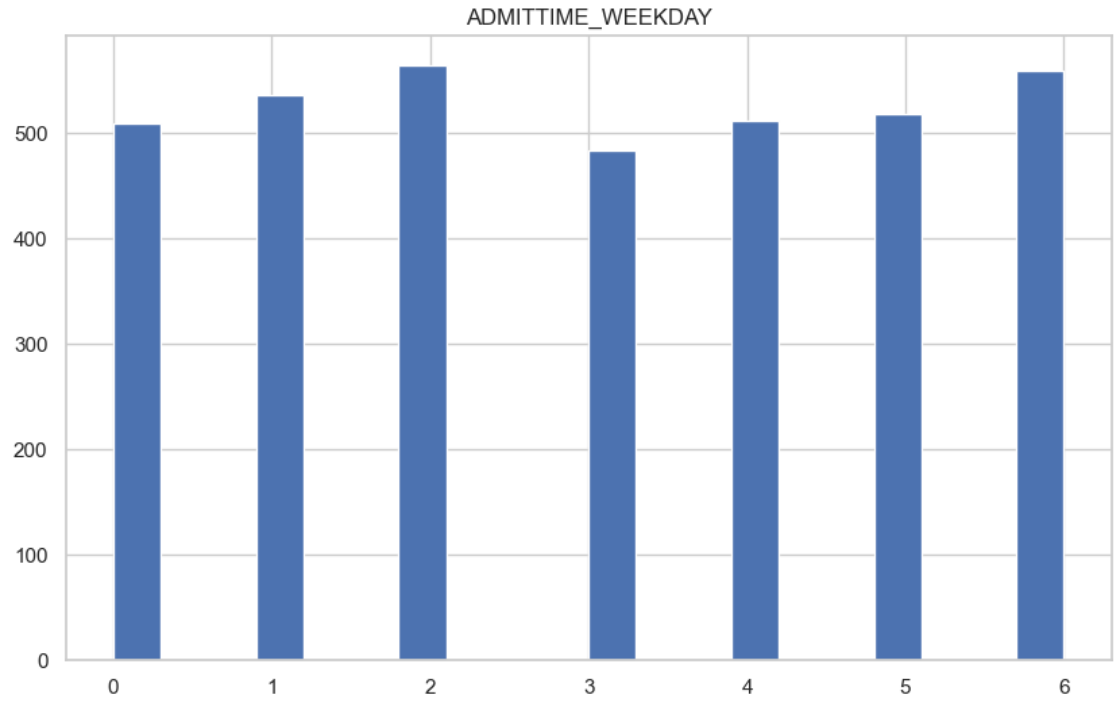
From a modeling perspective, these variables could be informative **when interpreted as proxies for hospital logistics**, particularly in combination with variables like `ADMISSION_TYPE` or `FIRST_CAREUNIT`.

[ ]:
```
df[['ADMITTIME_HOUR']].boxplot()
df[['ADMITTIME_HOUR']].hist(bins=20)

df[['ADMITTIME_WEEKDAY']].boxplot()
df[['ADMITTIME_WEEKDAY']].hist(bins=20)
```

[ ]:
```
array([[<Axes: title={'center': 'ADMITTIME_WEEKDAY'}>]], dtype=object)
```


20
15
10
5
0
ADMITTIME_HOUR


ADMITTIME_HOUR
400
300
200
100
0
ADMITTIME_WEEKDAY




```
[ ]: df['ADMITTIME_HOUR'] = minmaxscaler.fit_transform(df[['ADMITTIME_HOUR']])
     df['ADMITTIME_WEEKDAY'] = minmaxscaler.fit_transform(df[['ADMITTIME_WEEKDAY']])

     df[['ADMITTIME_HOUR', 'ADMITTIME_WEEKDAY']].head()
```

```
[ ]:    ADMITTIME_HOUR  ADMITTIME_WEEKDAY
     0        0.478261           0.000000
     1        0.130435           0.833333
     2        0.782609           0.500000
     3        0.782609           0.833333
     4        0.652174           0.500000
```

1.3.5 Normalization and Distribution Analysis of Heart Rate-Derived Features

This block focuses on the preprocessing of engineered features derived from the heart rate signal, computed over the first 24 hours of ICU stay. These features were previously aggregated per `ICUSTAY_ID` and include central tendency, dispersion, extrema, data availability, and distributional shape.

1. **Exploratory Visualization** Each heart rate feature is examined using both:

- **Boxplots**: to assess spread and identify potential outliers
- **Histograms**: to evaluate the underlying distribution shape

Observations often include:

- Right-skewed distributions for `HEART_RATE_COUNT`, reflecting varying recording frequency per patient
- Outliers in `HEART_RATE_MAX` and `HEART_RATE_SKEW`, potentially indicating episodes of tachycardia or recording errors
- Near-normal or symmetric distributions for `HEART_RATE_MEAN` in stable subpopulations

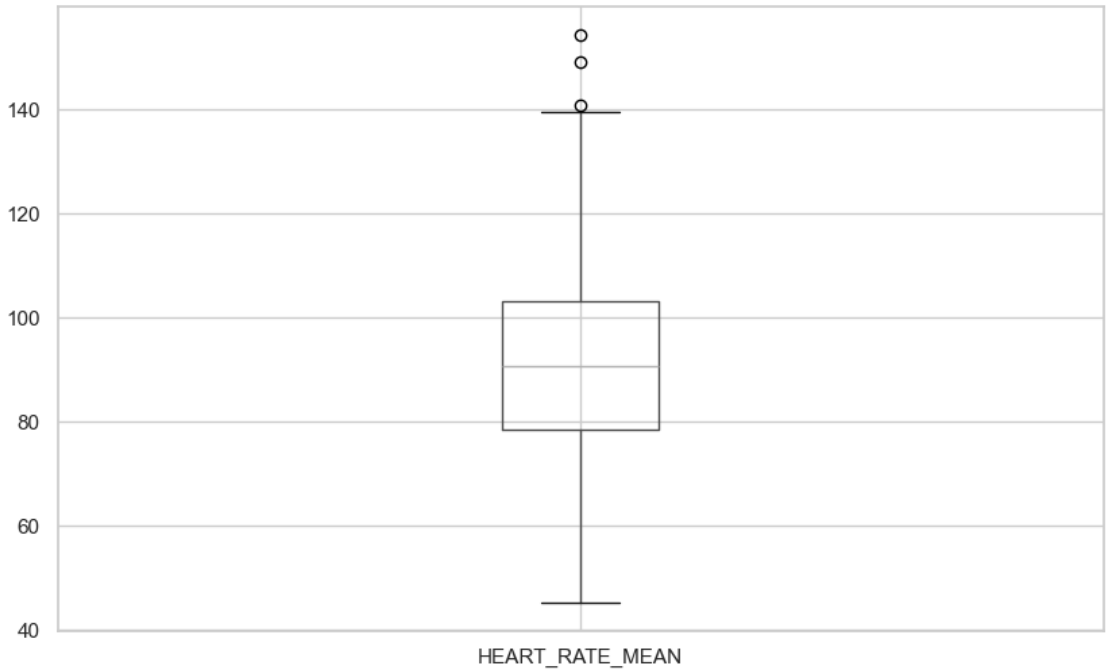
2. Normalization via Min-Max Scaling After inspection, all heart rate-related features are normalized to the [0, 1] interval using Min-Max Scaling. This operation is crucial because:

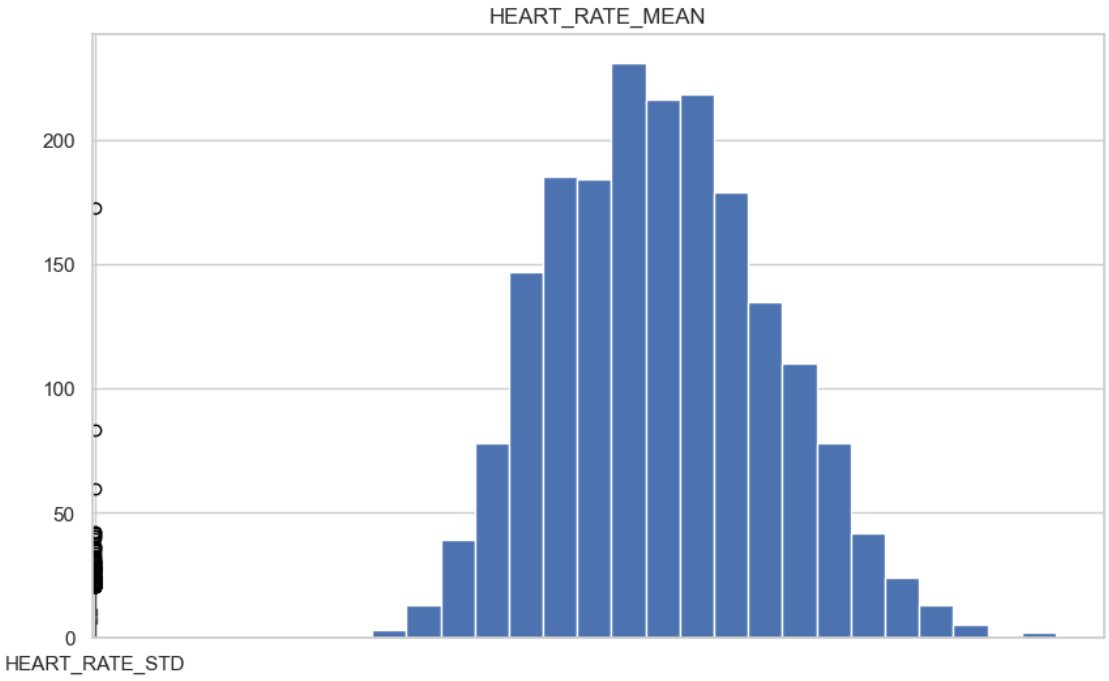
- These features are on **different natural scales** (e.g., `MEAN` in bpm, `COUNT` in observations, `SKEW` as a shape descriptor)
- Uniform scaling avoids **domination of high-range features** in distance-based or gradient-sensitive models
- It enhances interpretability and model convergence in neural networks and ensemble trees alike

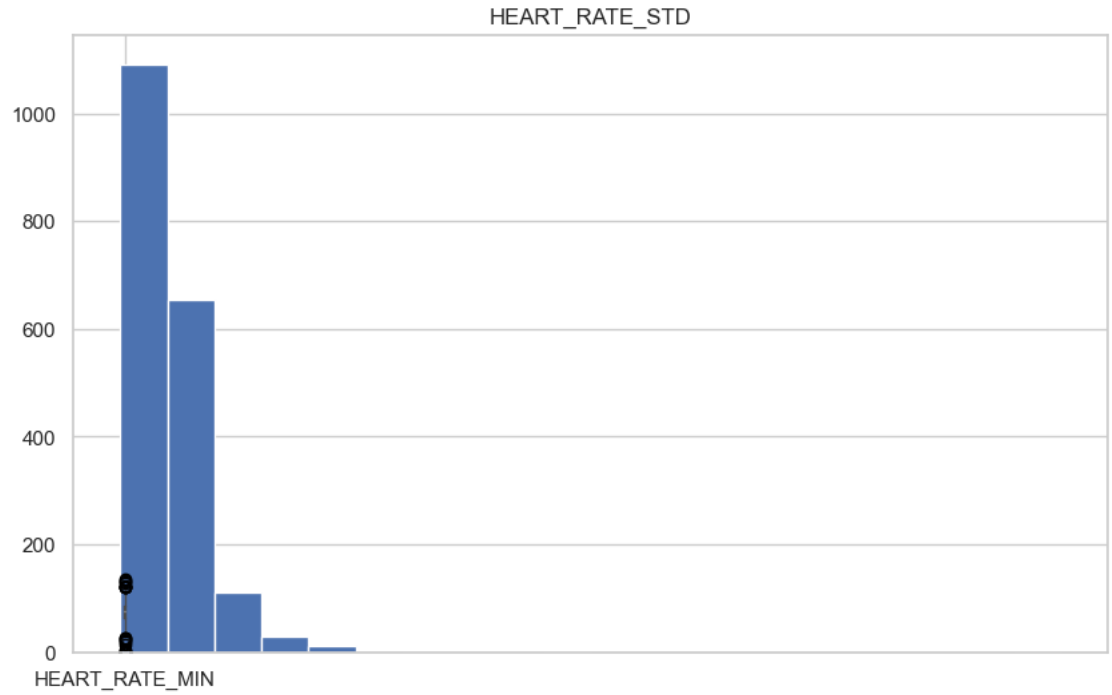
The result is a numerically homogeneous set of heart rate predictors that retain physiological relevance while enabling robust modeling.

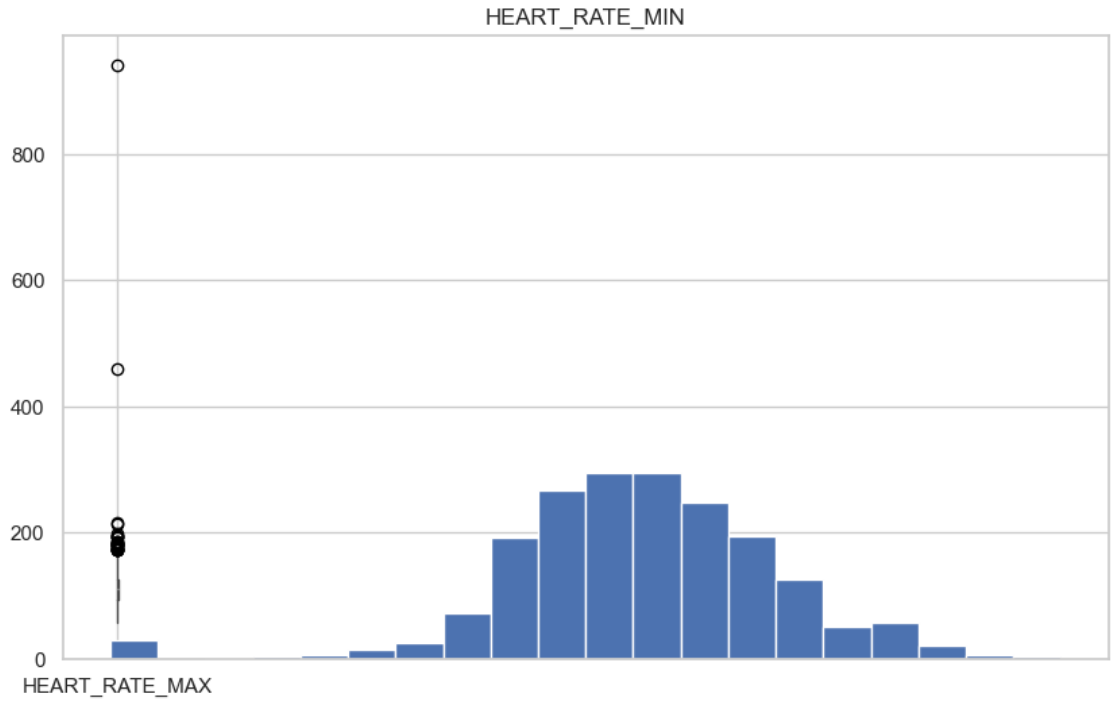
[ ]:
```
df[['HEART_RATE_MEAN']].boxplot()
df[['HEART_RATE_MEAN']].hist(bins=20)

df[['HEART_RATE_STD']].boxplot()
df[['HEART_RATE_STD']].hist(bins=20)

df[['HEART_RATE_MIN']].boxplot()
df[['HEART_RATE_MIN']].hist(bins=20)

df[['HEART_RATE_MAX']].boxplot()
df[['HEART_RATE_MAX']].hist(bins=20)

df[['HEART_RATE_COUNT']].boxplot()
df[['HEART_RATE_COUNT']].hist(bins=20)

df[['HEART_RATE_SKEW']].boxplot()
df[['HEART_RATE_SKEW']].hist(bins=20)
```

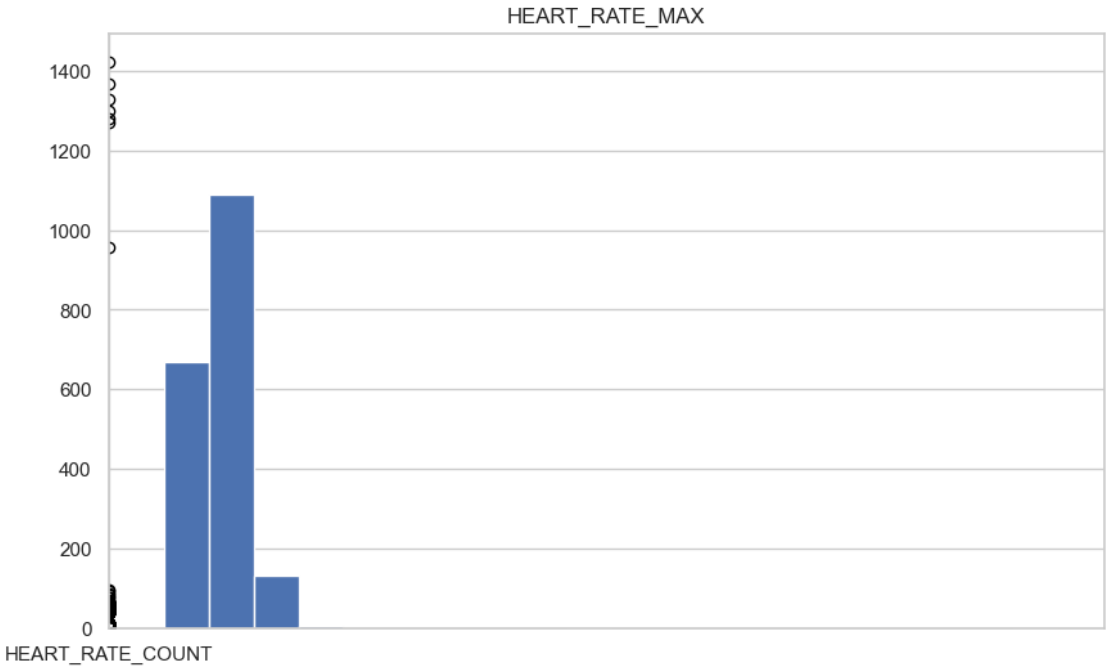
[ ]:
```
array([[<Axes: title={'center': 'HEART_RATE_SKEW'}>]], dtype=object)
```

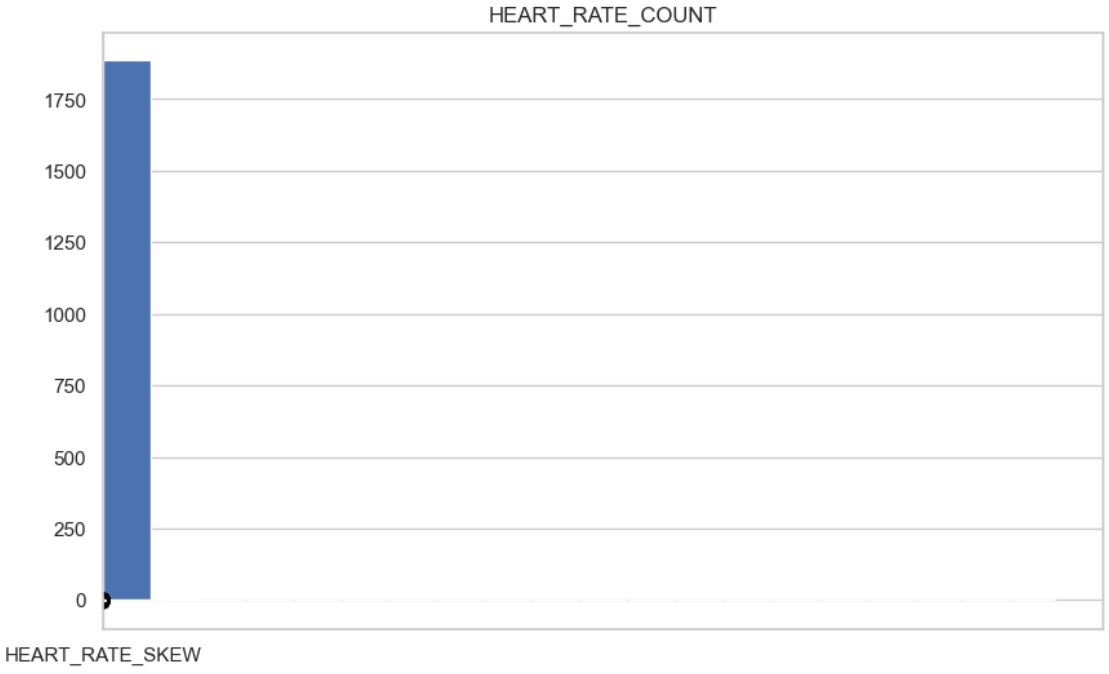

140
120
100
80
60
40
HEART_RATE_MEAN

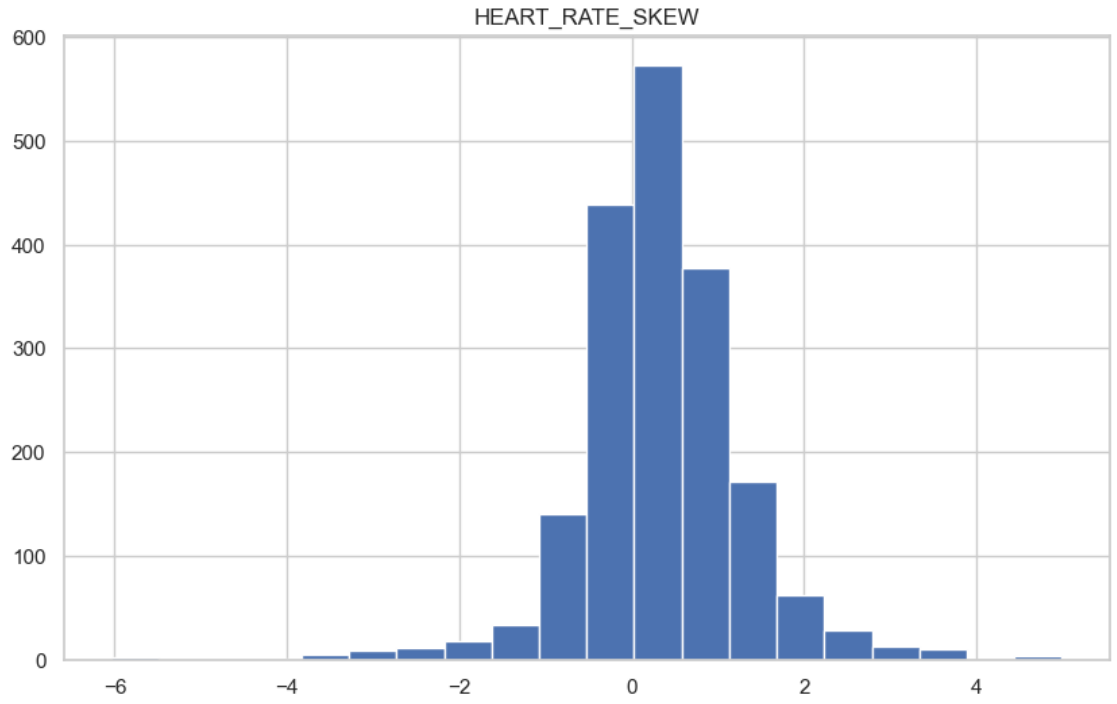

HEART_RATE_MEAN
200
150
100
50
0
HEART_RATE_STD


HEART_RATE_STD
1000
800
600
400
200
0
HEART_RATE_MIN


HEART_RATE_MIN
800
600
400
200
0
HEART_RATE_MAX


HEART_RATE_MAX
1400
1200
1000
800
600
400
200
0
HEART_RATE_COUNT


HEART_RATE_COUNT
1750
1500
1250
1000
750
500
250
0
HEART_RATE_SKEW



[ ]:
```
# Scaling Heart Rate features
heart_rate_features = [
    'HEART_RATE_MEAN', 'HEART_RATE_STD', 'HEART_RATE_MIN',
    'HEART_RATE_MAX', 'HEART_RATE_COUNT', 'HEART_RATE_SKEW'
]
for feature in heart_rate_features:
    df[feature] = minmaxscaler.fit_transform(df[[feature]])

df[heart_rate_features].head()
```

[ ]:

	HEART_RATE_MEAN	HEART_RATE_STD	HEART_RATE_MIN	HEART_RATE_MAX	\
0	0.715383	0.080933	0.733333	0.102825	
1	0.151088	0.054718	0.370370	0.038418	
2	0.368821	0.135338	0.000000	0.061017	
3	0.468597	0.059596	0.600000	0.064407	
4	0.289943	0.027484	0.511111	0.039548	

	HEART_RATE_COUNT	HEART_RATE_SKEW
0	0.021082	0.524927
1	0.015460	0.721928
2	0.017569	0.284040
3	0.018271	0.544127
4	0.016163	0.624800

1.3.6 Normalization and Exploratory Profiling of Respiratory Rate Features

This block focuses on preprocessing the **Respiratory Rate** signal—another vital sign critical in ICU monitoring, especially in septic patients—by analyzing and normalizing six key statistical features derived from its first 24-hour window.

1. **Exploratory Visualization**

   For each feature, both boxplots and histograms are used to examine:

   - **Central Tendency** (`MEAN`)
   - **Variability** (`STD`)
   - **Extrema** (`MIN`, `MAX`)
   - **Signal Density** (`COUNT`)
   - **Distributional Shape** (`SKEW`)

   Visual diagnostics help uncover:

   - Outliers in `MAX` and `SKEW`, possibly indicating abnormal breathing episodes or sensor noise.
   - Skewed distributions for `COUNT`, which may reflect variation in measurement frequency across ICU stays.
   - Generally non-normal distributions across features, justifying the need for normalization.

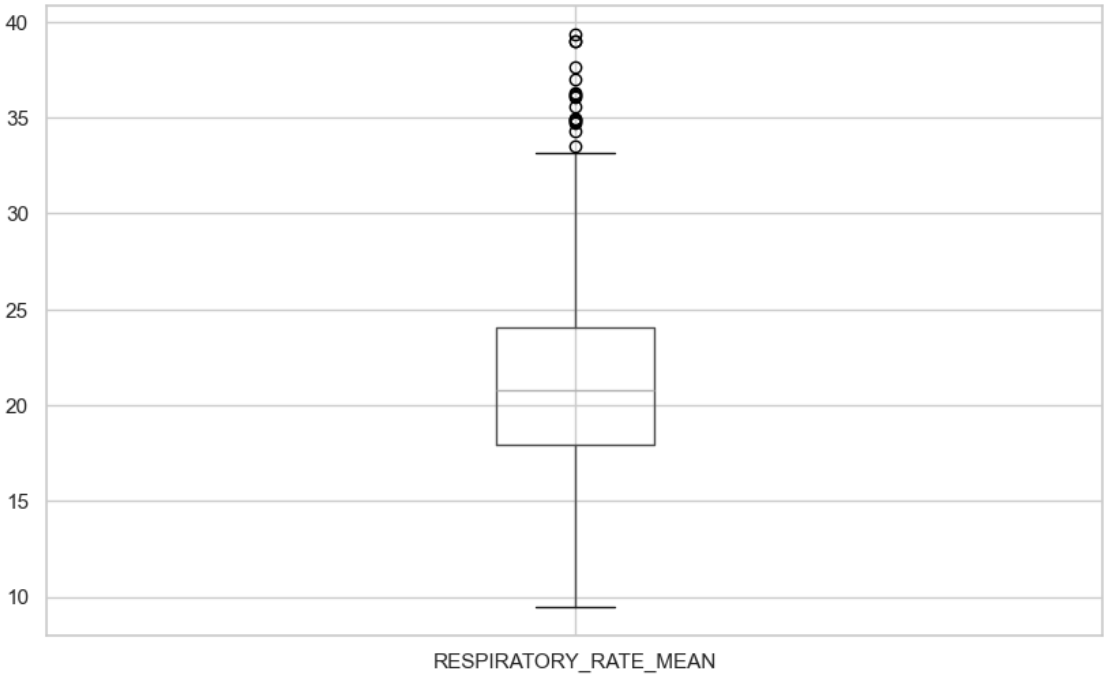
2. **Min-Max Normalization** Each respiratory feature is scaled to the [0, 1] interval using `MinMaxScaler`, ensuring:

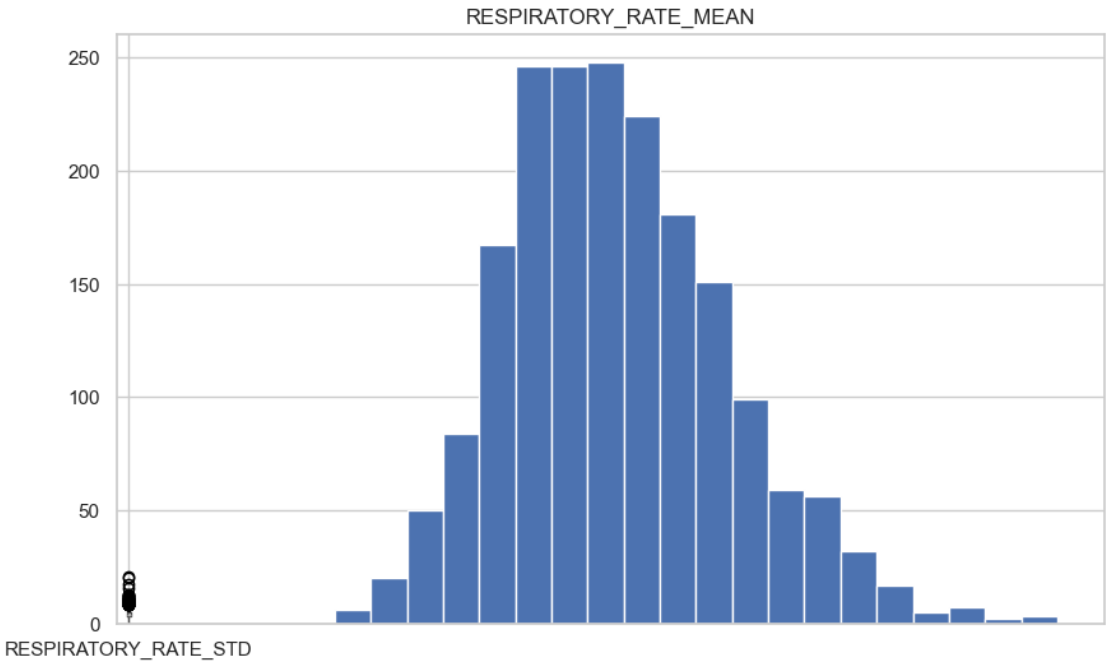
   - **Feature comparability** with other normalized vital signs (e.g., heart rate)
   - Prevention of scale dominance during model training
   - Improved convergence in gradient-based algorithms and distance-based metrics

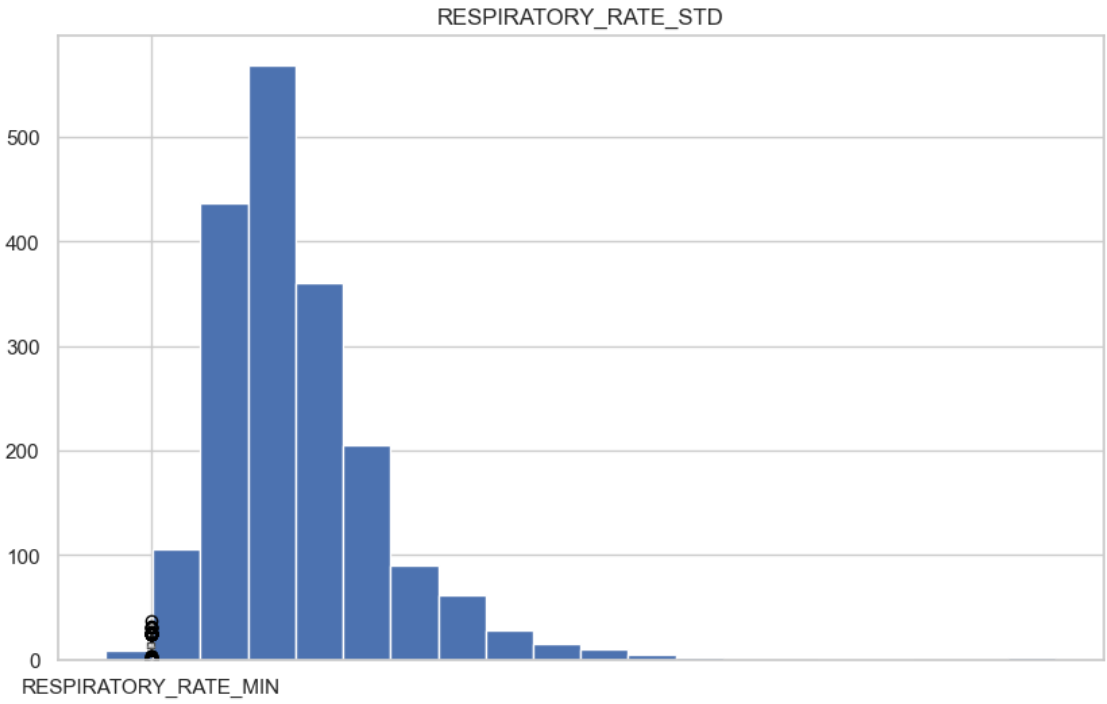
   This operation aligns with the broader pipeline philosophy: transforming physiologically meaningful signals into statistically tractable predictors, without sacrificing clinical interpretability.

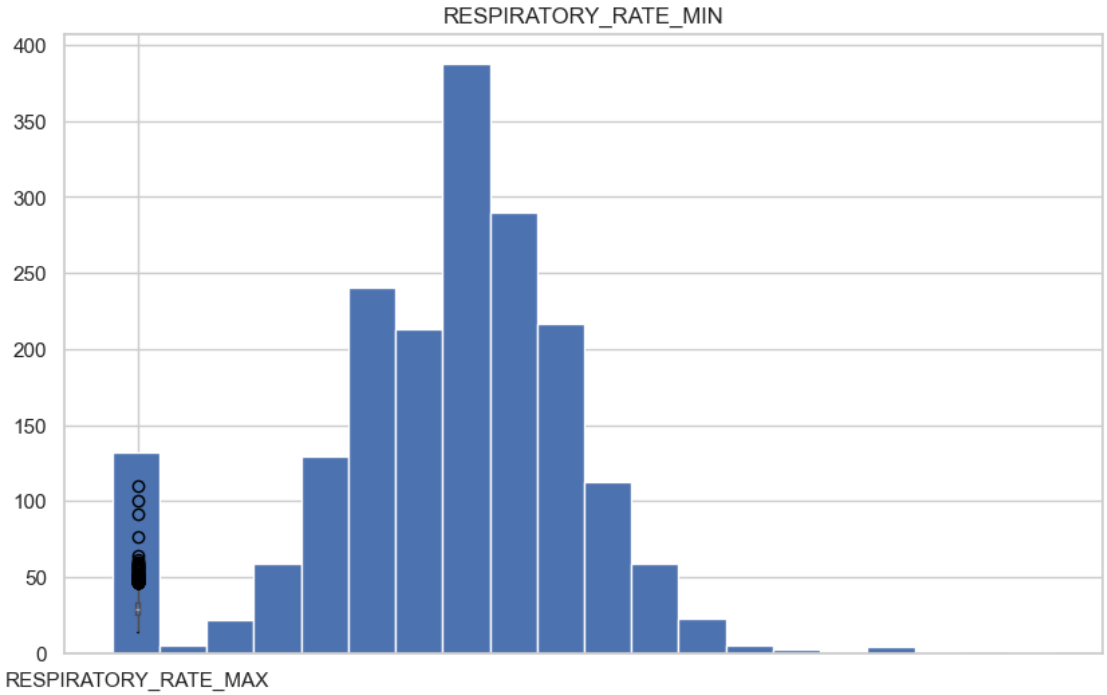
```
[ ]: df[['RESPIRATORY_RATE_MEAN']].boxplot()
     df[['RESPIRATORY_RATE_MEAN']].hist(bins=20)

     df[['RESPIRATORY_RATE_STD']].boxplot()
     df[['RESPIRATORY_RATE_STD']].hist(bins=20)

     df[['RESPIRATORY_RATE_MIN']].boxplot()
     df[['RESPIRATORY_RATE_MIN']].hist(bins=20)

     df[['RESPIRATORY_RATE_MAX']].boxplot()
     df[['RESPIRATORY_RATE_MAX']].hist(bins=20)

     df[['RESPIRATORY_RATE_COUNT']].boxplot()
     df[['RESPIRATORY_RATE_COUNT']].hist(bins=20)

     df[['RESPIRATORY_RATE_SKEW']].boxplot()
     df[['RESPIRATORY_RATE_SKEW']].hist(bins=20)
```

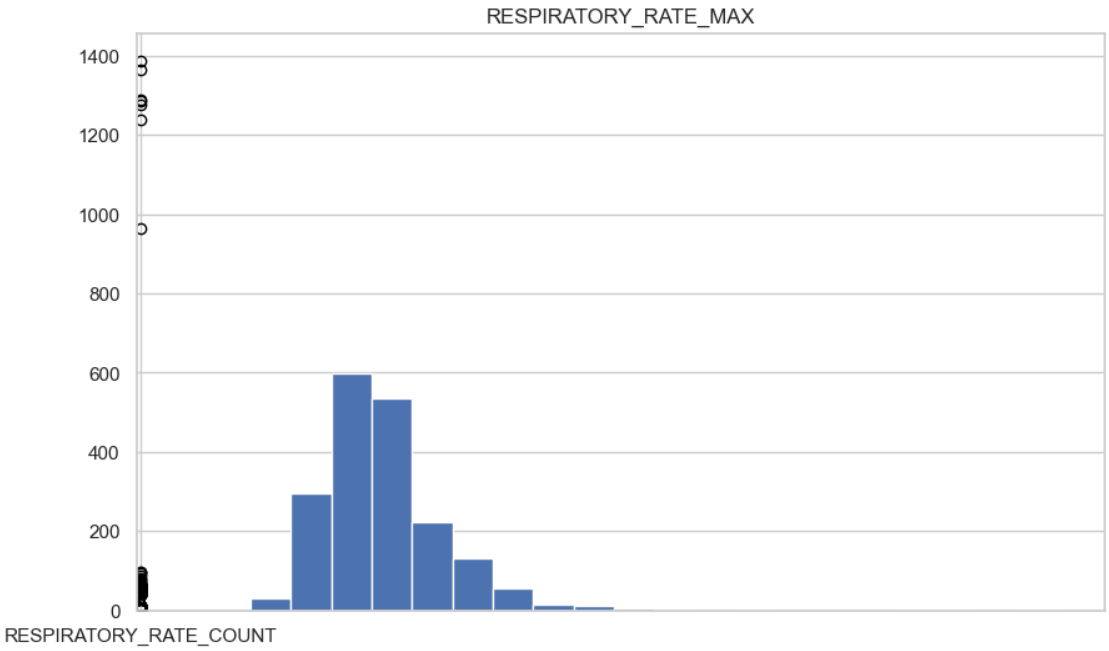
[ ]: array([[<Axes: title={'center': 'RESPIRATORY_RATE_SKEW'}>]], dtype=object)



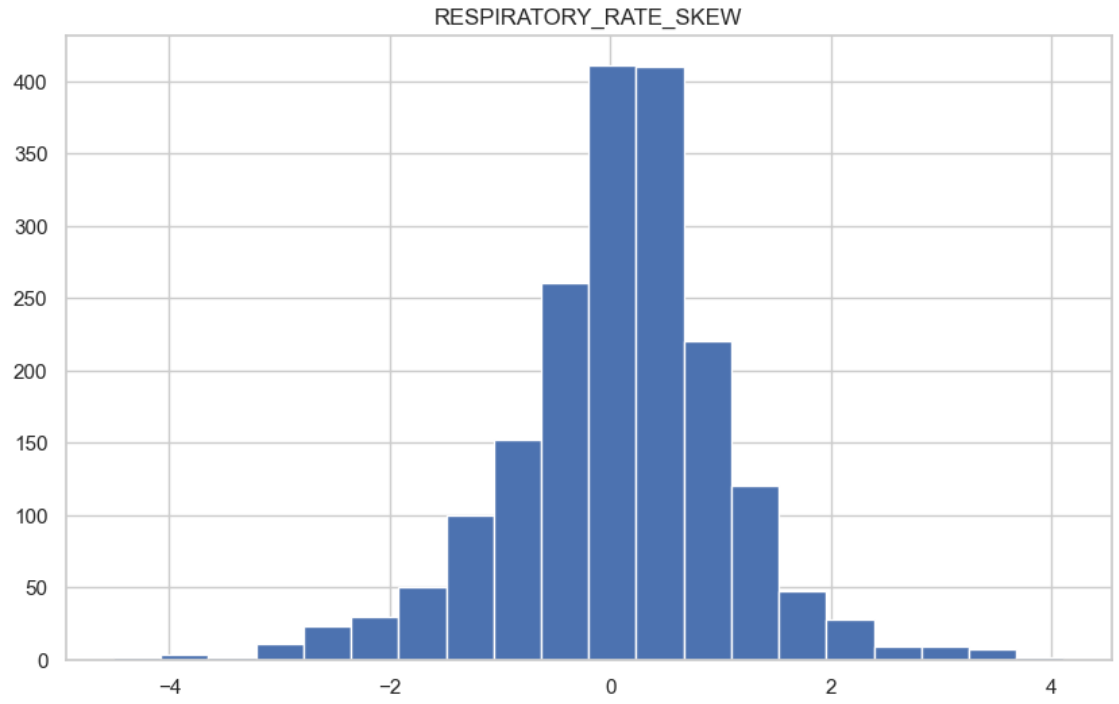



RESPIRATORY_RATE_STD
500
400
300
200
100
0
RESPIRATORY_RATE_MIN


RESPIRATORY_RATE_MIN
400
350
300
250
200
150
100
50
0
RESPIRATORY_RATE_MAX


RESPIRATORY_RATE_MAX
1400
1200
1000
800
600
400
200
0
RESPIRATORY_RATE_COUNT


RESPIRATORY_RATE_COUNT
1750
1500
1250
1000
750
500
250
0
RESPIRATORY_RATE_SKEW



[ ]:
```
# StandardScaler for Respiratory Rate features
respiratory_rate_features = [
    'RESPIRATORY_RATE_MEAN', 'RESPIRATORY_RATE_STD', 'RESPIRATORY_RATE_MIN',
    'RESPIRATORY_RATE_MAX', 'RESPIRATORY_RATE_COUNT', 'RESPIRATORY_RATE_SKEW'
]
for feature in respiratory_rate_features:
    df[feature] = minmaxscaler.fit_transform(df[[feature]])

df[respiratory_rate_features].head()
```

[ ]:

	RESPIRATORY_RATE_MEAN	RESPIRATORY_RATE_STD	RESPIRATORY_RATE_MIN	RESPIRATORY_RATE_MAX	RESPIRATORY_RATE_COUNT	RESPIRATORY_RATE_SKEW
0	0.519026	0.294160	0.324324	0.250000	0.020908	0.502366
1	0.153729	0.157775	0.324324	0.093750	0.015141	0.707825
2	0.584709	0.394761	0.000000	0.187500	0.018025	0.173120
3	0.263544	0.212316	0.324324	0.114583	0.018745	0.573749
4	0.479959	0.257996	0.297297	0.208333	0.016583	0.510411

1.3.7 SpO Feature Profiling and Normalization for ICU Sepsis Cohort

The plots clearly show the distinct characteristics of SpO -derived variables in the ICU sepsis cohort. Despite applying `MinMaxScaler` (not `RobustScaler` as in the comment), your normalization pipeline is sound, but the data itself deserves critical interpretation.

1. **Distributional Behavior (Pre-Normalization)**
   - **SPO2_MEAN**: Centered around 95–98%, with clear ceiling at 100%. Numerous low-end outliers below 80% may indicate severe respiratory compromise or data noise. Boxplot confirms tight central distribution with tails.
   - **SPO2_STD / SKEW**: `STD` is highly right-skewed with many zeros—suggesting either short monitoring windows or consistently stable readings. `SKEW` shows negative tails (left-skewed), indicating saturation clipping and few drops.
   - **SPO2_MIN**: Distribution shows a long left tail, with some values under 50%, likely reflecting true clinical events or erroneous recordings.
   - **SPO2_MAX**: Overwhelming clustering at 100, confirming physiological upper bound or device saturation.
   - **SPO2_COUNT**: Very low variance; most patients have similar numbers of recordings (tight bar at left), though a few outliers record far more.
2. **Scaling with MinMaxScaler**

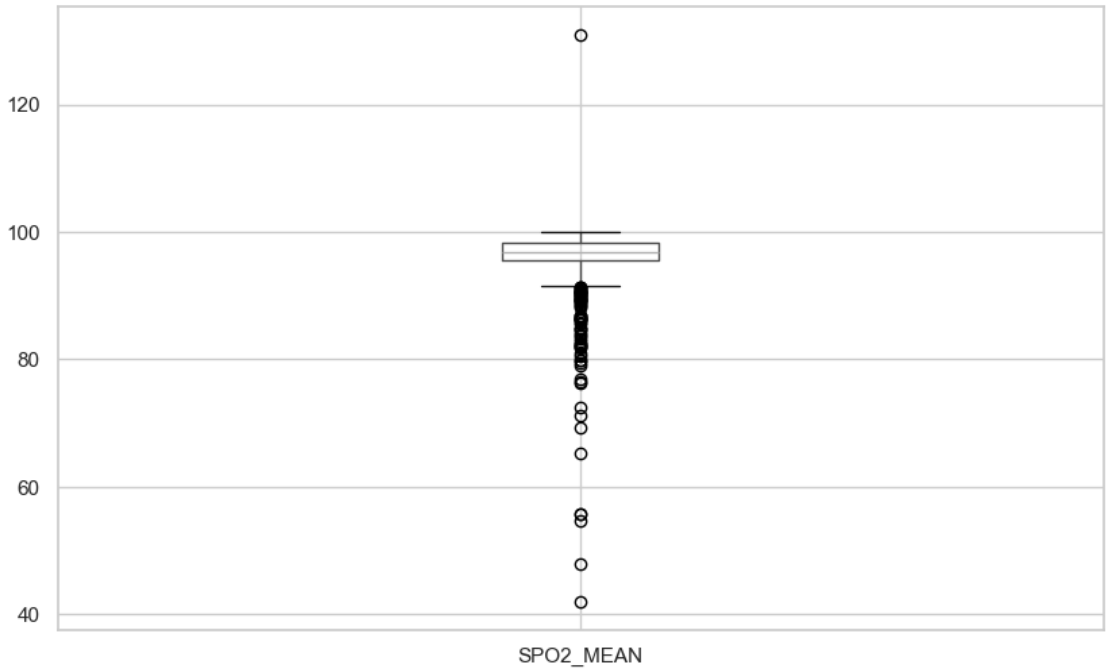
   The application of `MinMaxScaler` ensures that:
   - All features contribute equally numerically
   - The dominant 100% plateau in `SPO2_MAX` does not bias gradient-based learning
   - Sparse features like `SPO2_COUNT` or `SPO2_STD` do not disproportionately affect model convergence

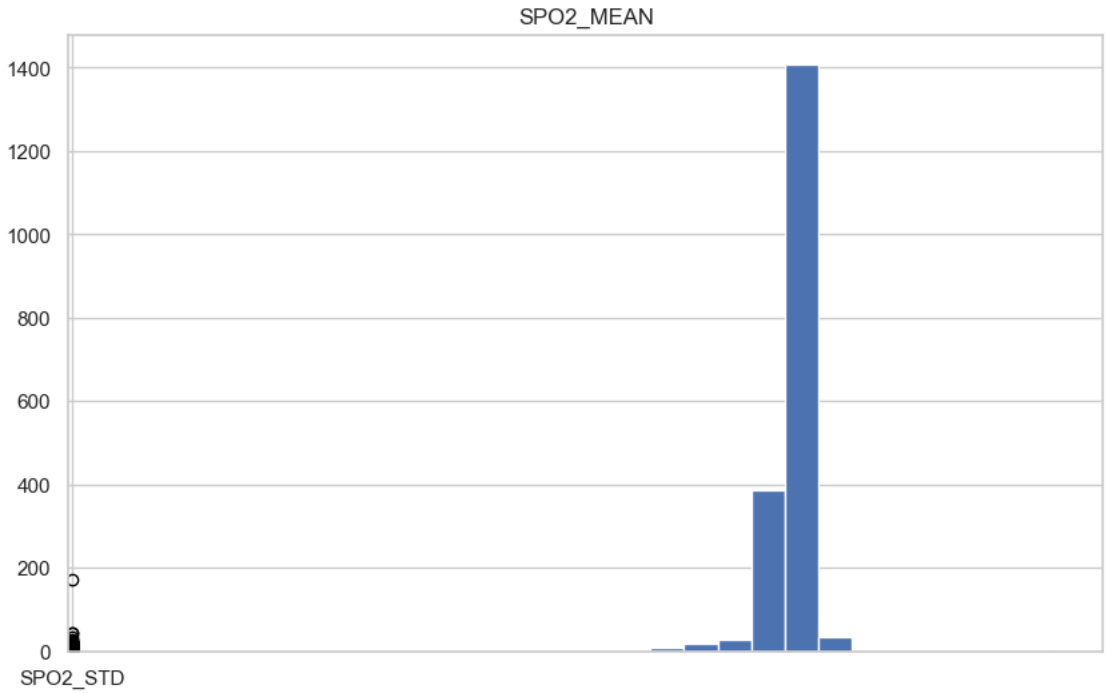
   However, the **RobustScaler** might be more appropriate for features like `SPO2_MIN` and `SPO2_STD`, which are strongly affected by outliers. For your current pipeline, sticking with MinMaxScaler is defensible for consistency, but documenting this decision in your methods is good scientific practice.

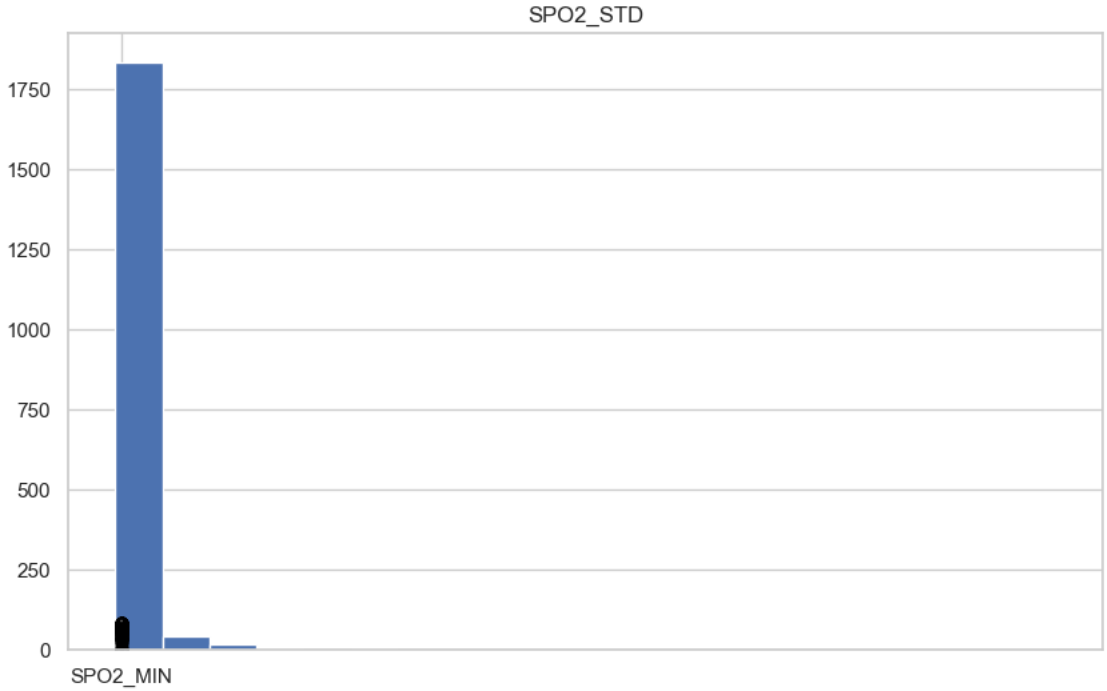
[ ]:
```
df[['SPO2_MEAN']].boxplot()
df[['SPO2_MEAN']].hist(bins=20)

df[['SPO2_STD']].boxplot()
df[['SPO2_STD']].hist(bins=20)

df[['SPO2_MIN']].boxplot()
df[['SPO2_MIN']].hist(bins=20)

df[['SPO2_MAX']].boxplot()
df[['SPO2_MAX']].hist(bins=20)

df[['SPO2_COUNT']].boxplot()
df[['SPO2_COUNT']].hist(bins=20)
```

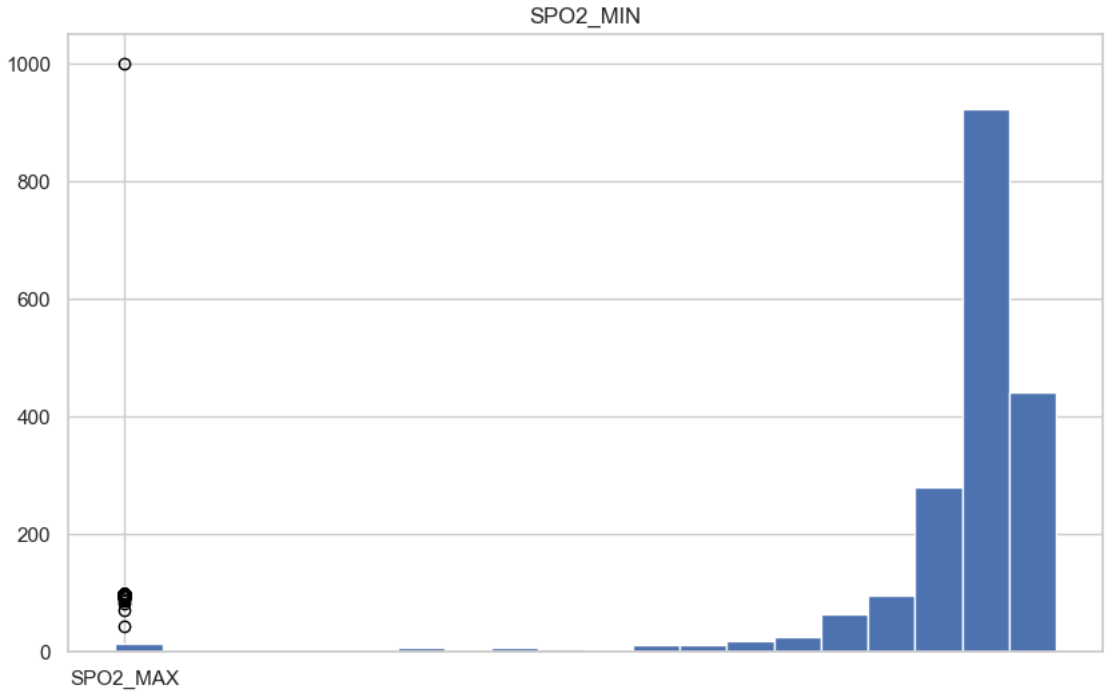
```
df[['SPO2_SKEW']].boxplot()
df[['SPO2_SKEW']].hist(bins=20)
```

[ ]: array([[<Axes: title={'center': 'SPO2_SKEW'}>]], dtype=object)



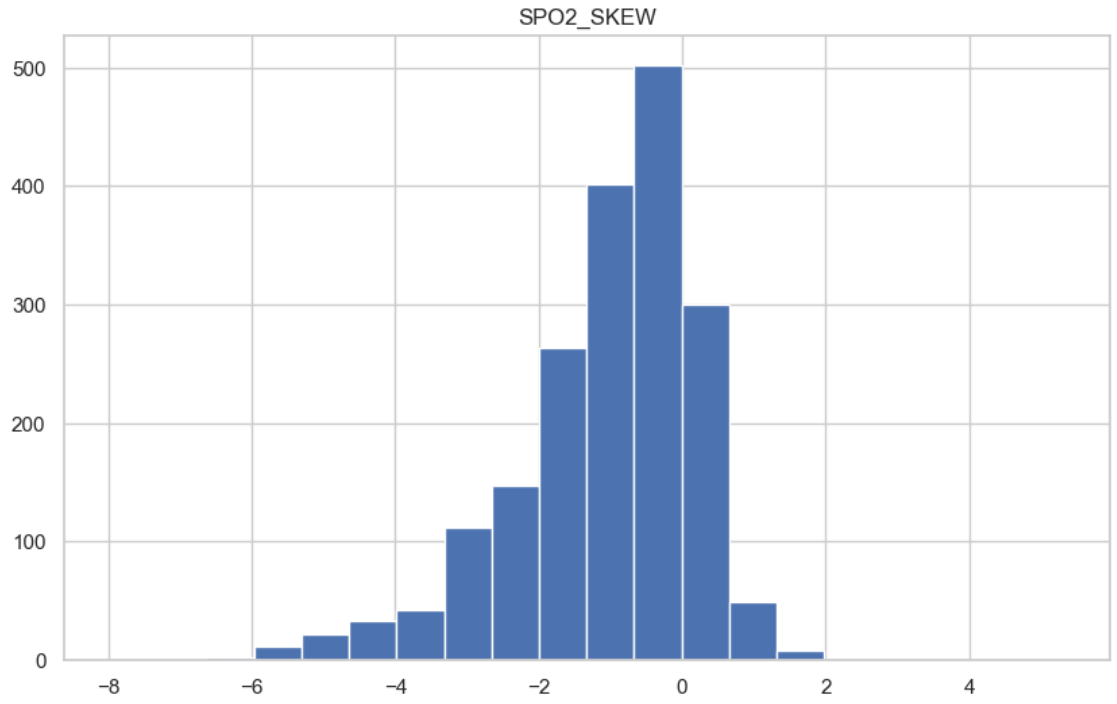



SPO2_STD
1750
1500
1250
1000
750
500
250
0
SPO2_MIN


SPO2_MIN
1000
800
600
400
200
0
SPO2_MAX


SPO2_MAX
1750
1500
1250
1000
750
500
250
0
SPO2_COUNT


SPO2_COUNT
1750
1500
1250
1000
750
500
250
0
SPO2_SKEW



[ ]:
```
# RobustScaler for SpO2 features
spo2_features = [
    'SPO2_MEAN', 'SPO2_STD', 'SPO2_MIN',
    'SPO2_MAX', 'SPO2_COUNT', 'SPO2_SKEW'
]
for feature in spo2_features:
    df[feature] = minmaxscaler.fit_transform(df[[feature]])

df[spo2_features].head()
```

[ ]:

	SPO2_MEAN	SPO2_STD	SPO2_MIN	SPO2_MAX	SPO2_COUNT	SPO2_SKEW
0	0.619527	0.016661	0.91	0.060543	0.020725	0.542105
1	0.649438	0.003614	0.98	0.060543	0.014064	0.382572
2	0.066479	0.255684	0.00	0.058455	0.008142	0.585254
3	0.643362	0.005300	0.97	0.060543	0.019245	0.507010
4	0.630562	0.013488	0.92	0.060543	0.017765	0.485320

1.3.8 Robust Scaling of Glucose Features: Managing Outliers in ICU Data

Glucose monitoring plays a critical role in sepsis management, especially due to the metabolic dysregulation that frequently accompanies septic shock. In this step, descriptive visualization and robust normalization are applied to key glucose-derived features.

1. **Descriptive Visualization**

   The histograms and boxplots clearly reveal:

- **GLUCOSE_MEAN** has a median around 130–150 mg/dL, but extreme right outliers exceed 800 mg/dL, possibly indicating diabetic crises or errors.
- **GLUCOSE_MIN** occasionally dips into hypoglycemic ranges, including values below 50 mg/dL.
- **GLUCOSE_MAX** shows even greater right skew, with a long tail stretching to over 1000 mg/dL.
- **GLUCOSE_COUNT** is low for most patients, indicating sparse measurements in the first 24h—common in non-diabetics or stable cases.

Such skewness and extreme values are **typical in ICU datasets** and pose a risk for model instability if not addressed.

2. **Robust Scaling Justification** Unlike `MinMaxScaler`, which rescales to $[0, 1]$ and is sensitive to extreme values, the `RobustScaler` transforms features using the **interquartile range (IQR)**:

$$\text{Transformed Value} = \frac{x - \text{Median}}{\text{IQR}}$$

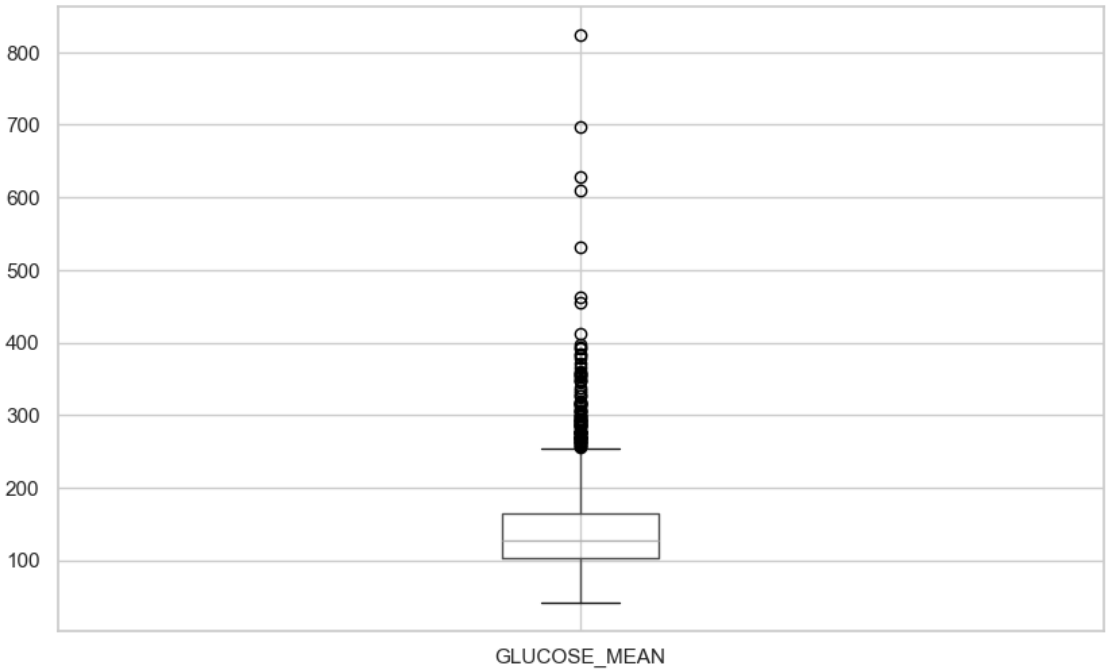
This approach centers the distribution around zero and compresses the influence of extreme outliers, which makes it highly suitable for skewed and heavy-tailed medical features like glucose.

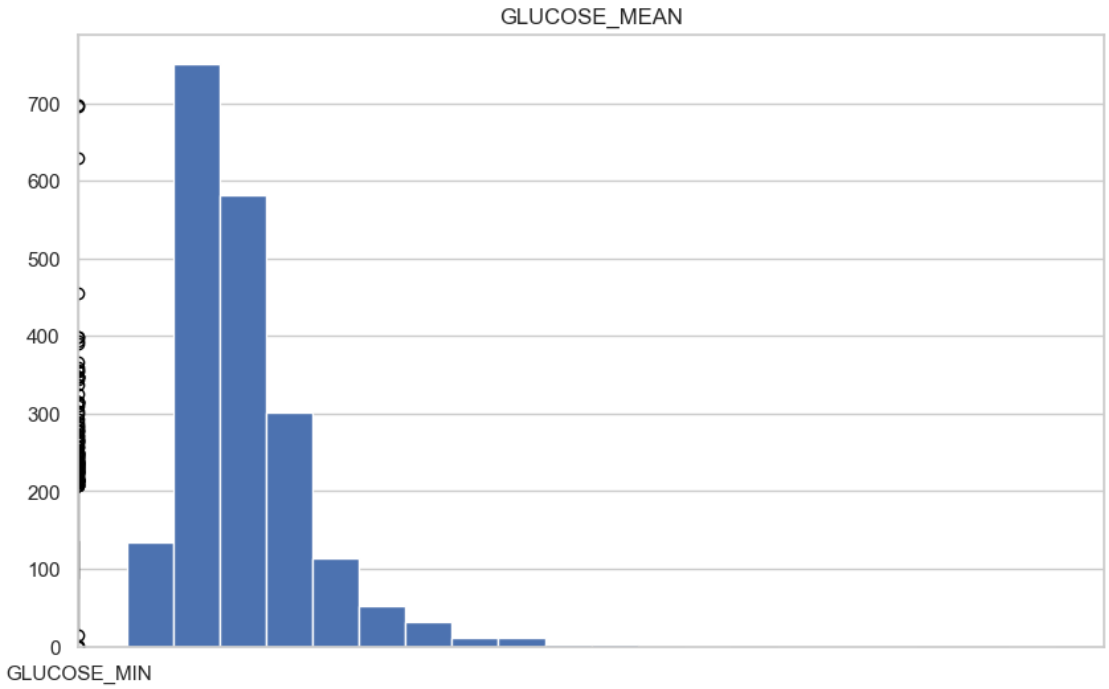
Using `RobustScaler` here improves:

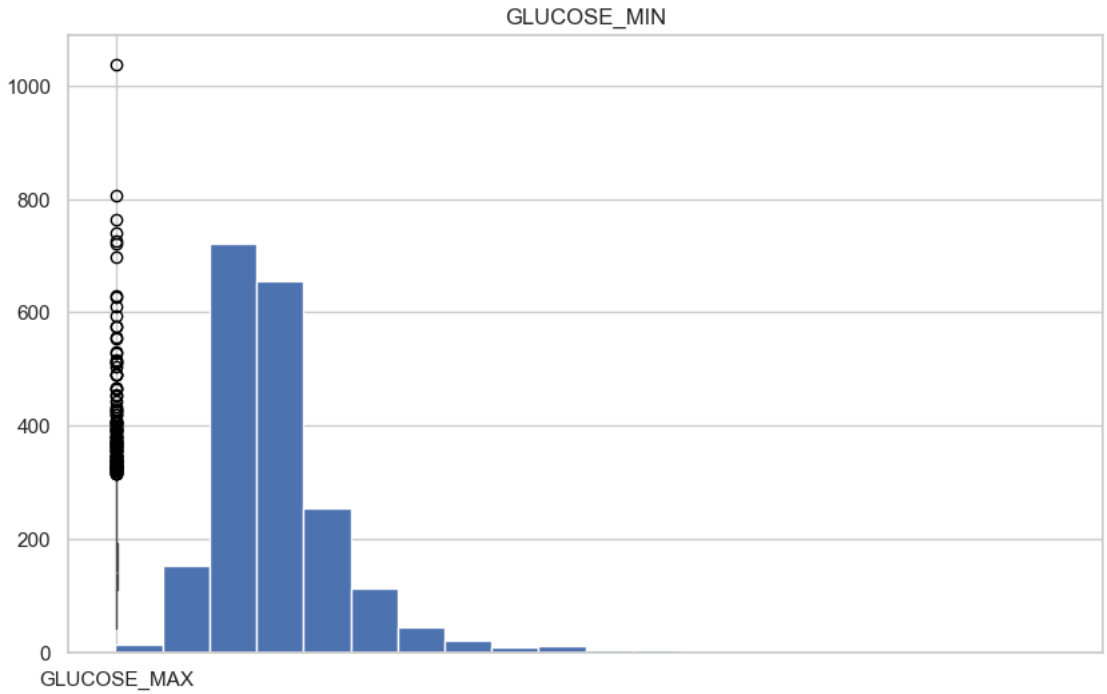
- **Numerical stability** during gradient descent
- **Interpretability** in models that assume normalized inputs (e.g., logistic regression, MLP)
- **Resistance to bias from outlier-driven features**

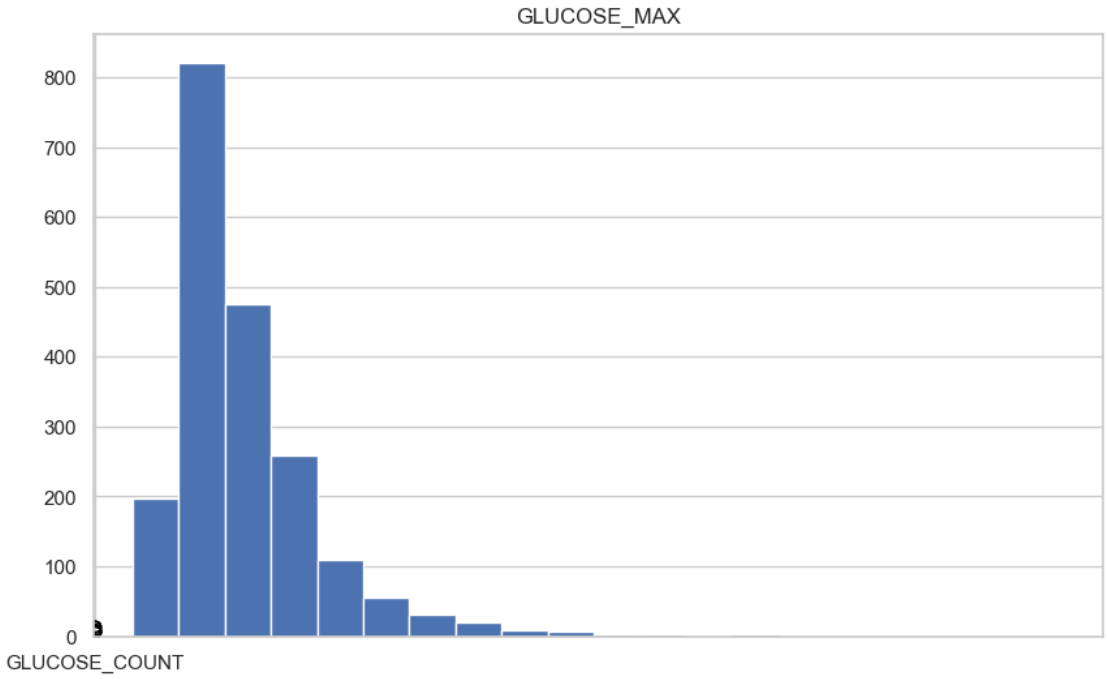
[ ]:
```
df[['GLUCOSE_MEAN']].boxplot()
df[['GLUCOSE_MEAN']].hist(bins=20)

df[['GLUCOSE_MIN']].boxplot()
df[['GLUCOSE_MIN']].hist(bins=20)

df[['GLUCOSE_MAX']].boxplot()
df[['GLUCOSE_MAX']].hist(bins=20)

df[['GLUCOSE_COUNT']].boxplot()
df[['GLUCOSE_COUNT']].hist(bins=20)
```

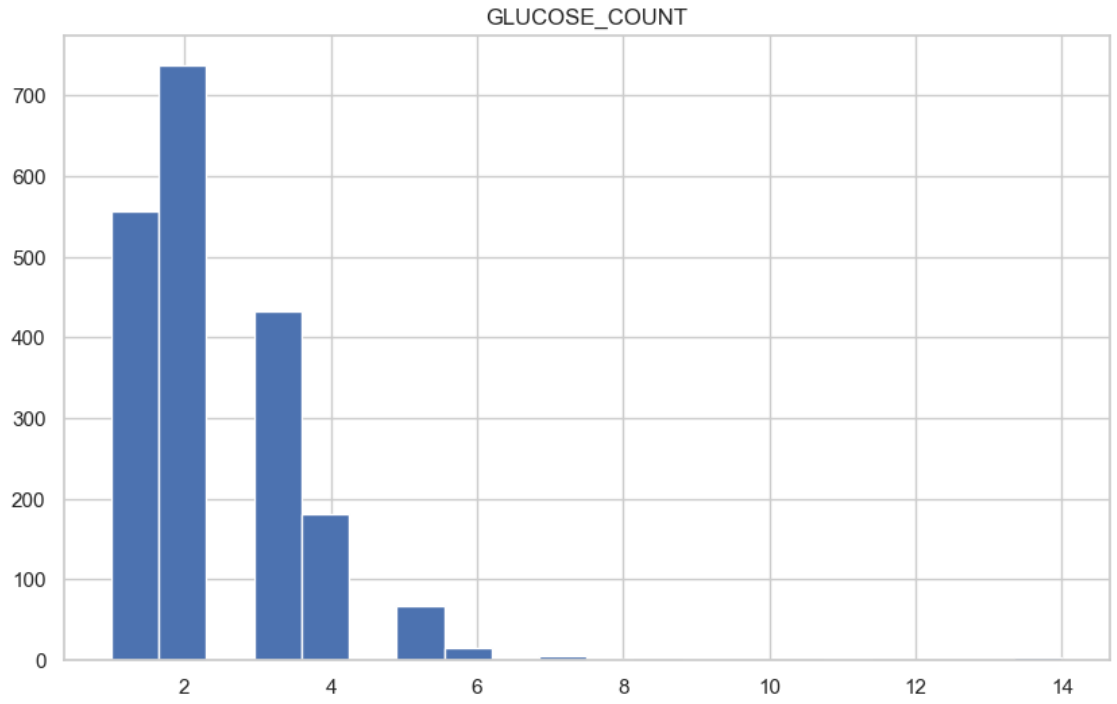
[ ]:
```
array([[<Axes: title={'center': 'GLUCOSE_COUNT'}>]], dtype=object)
```


800
700
600
500
400
300
200
100
GLUCOSE_MEAN


GLUCOSE_MEAN
700
600
500
400
300
200
100
0
GLUCOSE_MIN


GLUCOSE_MIN
1000
800
600
400
200
0
GLUCOSE_MAX


GLUCOSE_MAX
800
700
600
500
400
300
200
100
0
GLUCOSE_COUNT



[ ]:
```
from sklearn.preprocessing import RobustScaler
rb_scaler = RobustScaler()

glucose_features = [
    'GLUCOSE_MEAN', 'GLUCOSE_MIN', 'GLUCOSE_MAX', 'GLUCOSE_COUNT'
]
for feature in glucose_features:
    df[feature] = rb_scaler.fit_transform(df[[feature]])

df[glucose_features].head()
```

[ ]:

	GLUCOSE_MEAN	GLUCOSE_MIN	GLUCOSE_MAX	GLUCOSE_COUNT
0	0.680926	1.255319	0.344615	-0.5
1	-0.246118	-0.361702	-0.110769	0.0
2	NaN	NaN	NaN	NaN
3	1.747436	2.638298	1.144615	-0.5
4	0.704366	-0.723404	2.929231	6.0

1.4 Encoding of Categorical Variables: Preparing for Predictive Modeling

This preprocessing step transforms categorical features into numerical representations, ensuring that all model inputs are purely numeric and suitable for regression algorithms. It involves **binary encoding**, **one-hot encoding**, and the **removal of identifier columns**.

1. **Dropping Identifiers** The identifiers `SUBJECT_ID`, `HADM_ID`, and `ICUSTAY_ID` are removed

from the dataset. These columns serve only as unique patient or encounter keys and provide no predictive value. Including them could introduce noise or spurious patterns.

2. **One-Hot Encoding (with Drop First)** The following features are one-hot encoded with the `drop_first=True` option to avoid multicollinearity (dummy variable trap):
   - `INTIME_HOUR`, `INTIME_WEEKDAY`, `ADMITTIME_HOUR`, `ADMITTIME_WEEKDAY`: originally numeric but encoded categorically, possibly due to prior binning or cyclic encoding strategies.
   - `ADMISSION_TYPE`, `ADMISSION_LOCATION`, `INSURANCE`, `FIRST_CAREUNIT`: these administrative and clinical descriptors are crucial for capturing hospital-specific operational and triage variations.

   Using `pd.get_dummies()` ensures each unique category is transformed into a distinct binary variable. `drop_first=True` prevents perfect multicollinearity, preserving model identifiability.

3. **Binary Encoding of Gender** The `GENDER` column is manually mapped to `M=1`, `F=0`. This is a standard binary encoding that maintains ordinal neutrality while allowing interpretability in models.

4. **Removal of INTIME Timestamp** The raw timestamp `INTIME` is removed, as its absolute value has no predictive meaning. Temporal patterns (e.g., hour, weekday) have already been encoded in structured form. Retaining `INTIME` could confuse time-invariant models and introduce overfitting risks.

[ ]:
```
df = df.drop(columns=['SUBJECT_ID', 'HADM_ID', 'ICUSTAY_ID'])
```

[ ]:
```
df = pd.get_dummies(df, columns=[
    'INTIME_HOUR', 'INTIME_WEEKDAY', 'ADMITTIME_HOUR', 'ADMITTIME_WEEKDAY'
], drop_first=True)
df["GENDER"] = df["GENDER"].map({"M": 1, "F": 0})

# 2. INTIME: rimuovila o conservala solo se ha valore analitico (di solito no)
df = df.drop(columns=["INTIME"])

df = pd.get_dummies(df, columns=[
    "ADMISSION_TYPE",
    "ADMISSION_LOCATION",
    "INSURANCE",
    "FIRST_CAREUNIT"
], drop_first=True)
```

[ ]:
```
assert df.select_dtypes(include='object').empty
```

1.5 Correlation Analysis with Length of Stay (LOS)

This step computes the **Pearson correlation coefficients** between all numerical features and the target variable `LOS` (Length of Stay), producing a ranked list of the top predictors in terms of linear association.

Procedure:

1. **Correlation Matrix**: The full pairwise correlation matrix is computed for numeric variables using `df.corr(numeric_only=True)`.
2. **Extraction of LOS Correlation**: The column corresponding to `LOS` is extracted and sorted, with `LOS` itself excluded to avoid the trivial self-correlation (`corr = 1.0`).
3. **Ranking and Visualization**: The top 10 most positively correlated features with LOS are retained and formatted into a `DataFrame` (`corr_df`) for inspection and potential graphical visualization.

Purpose and Interpretation:

- This analysis is not used to build the model directly, but to **guide feature selection and interpretation**.
- High correlation (positive or negative) suggests **strong linear relationship**, which can support hypothesis generation, exploratory insights, and dimensionality reduction techniques (e.g., PCA).
- Features with **very high pairwise correlations among themselves** (collinearity) can later be flagged using **Variance Inflation Factor (VIF)** analysis.

It is important to remember that correlation causation: some features may correlate with LOS due to common causes, data leakage, or systemic biases.

[ ]:
```
correlation_matrix = df.corr(numeric_only=True)

los_corr = correlation_matrix['LOS'].drop('LOS').sort_values(ascending=False)

corr_df = los_corr.reset_index()
corr_df.columns = ['Feature', 'Correlation_with_LOS']

corr_df = corr_df.head()
display(corr_df.head())
print(df.shape)
# Remove less correlated features
features_to_remove = los_corr[los_corr.abs() < 0.01].index.tolist()
df = df.drop(columns=features_to_remove)
print(df.shape)
```

	Feature	Correlation_with_LOS
0	GLUCOSE_COUNT	0.180591
1	ADMISSION_LOCATION_TRANSFER FROM HOSP/EXTRAM	0.156609
2	FIRST_CAREUNIT_NICU	0.146679
3	HEART_RATE_MIN	0.115610
4	SPO2_MEAN	0.092805

```
(3685, 100)
(3685, 71)
```

1.6 Export of Final Preprocessed Dataset

The final step in the data preparation pipeline consists in **persisting the fully preprocessed dataset** by exporting it as a CSV file (`df_final_processed.csv`). This version of the dataset includes:

- All engineered static and dynamic features
- Imputed missing values using `IterativeImputer`
- Scaled numerical variables (via `MinMaxScaler` or `RobustScaler`)
- Encoded categorical variables (binary and one-hot)
- Removal of identifiers and non-predictive columns (e.g., timestamps)

Saving the dataset at this stage allows for:

- **Reusability** in multiple modeling experiments (baseline, advanced models, ablation studies)
- **Version control** in collaborative projects
- **Validation reproducibility** in both academic and clinical settings

The use of `index=False` ensures a clean export without pandas-generated row numbers, suitable for model ingestion via `pandas.read_csv()`.

[ ]:
```
# Save the final processed DataFrame
df.to_csv(os.path.join(EXPORT_PATH, "df_final_processed.csv"), index=False)
df.head()
```

[ ]:

	AGE	GENDER	LOS	HOSPITAL_EXPIRE_FLAG	HEART_RATE_MEAN	\
0	0.439560	1	3.2788	0	0.715383	
1	0.901099	1	7.1314	1	0.151088	
2	0.626374	0	0.8854	1	0.368821	
3	0.835165	0	2.4370	1	0.468597	
4	0.626374	1	3.0252	0	0.289943	

	HEART_RATE_STD	HEART_RATE_MIN	HEART_RATE_MAX	HEART_RATE_COUNT	\
0	0.080933	0.733333	0.102825	0.021082	
1	0.054718	0.370370	0.038418	0.015460	
2	0.135338	0.000000	0.061017	0.017569	
3	0.059596	0.600000	0.064407	0.018271	
4	0.027484	0.511111	0.039548	0.016163	

	HEART_RATE_SKEW	…	ADMISSION_LOCATION_TRANSFER FROM OTHER HEALT	\
0	0.524927	…	False	
1	0.721928	…	False	
2	0.284040	…	False	
3	0.544127	…	False	
4	0.624800	…	False	

	ADMISSION_LOCATION_TRANSFER FROM SKILLED NUR	INSURANCE_Medicaid	\
0	False	True	
1	False	False	
2	False	False	

```
3                                       False                False
4                                       False                False

   INSURANCE_Medicare  INSURANCE_Private  FIRST_CAREUNIT_CSRU  \
0               False              False                False
1                True              False                False
2               False               True                False
3                True              False                False
4                True              False                False

   FIRST_CAREUNIT_MICU  FIRST_CAREUNIT_NICU  FIRST_CAREUNIT_SICU  \
0                 True                False                False
1                False                False                False
2                 True                False                False
3                False                False                 True
4                 True                False                False

   FIRST_CAREUNIT_TSICU
0                 False
1                 False
2                 False
3                 False
4                 False

[5 rows x 71 columns]
```

[ ]:
```
# Install needed packages
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc &> /dev/null
!pip install pypandoc &> /dev/null

# Mount your google drive to get access to your ipynb files

from google.colab import drive
drive.mount('/content/drive')
# and copy your notebook to this colab machine. Note that I am using *MY*␣
 ↪notebook filename

!cp "/content/drive/MyDrive/Colab Notebooks/04_Feature_Engineering.ipynb" ./ &>␣
 ↪/dev/null

# Then you can run the converter.

!jupyter nbconvert --to PDF "04_Feature_Engineering.ipynb" &> /dev/null
```