

Week 4 Lab Session: Hands-On Data Preprocessing, Exploratory Data Analysis (EDA), and Feature Scaling

This **Hands-On Lab Prep Workbook** is designed for the practical laboratory session of **Machine Learning with Python for Managers** (Week 4). It provides step-by-step instructions for loading, cleaning, profiling, visualizing, and scaling real-world datasets.

By executing this lab, you will transition from theoretical data hygiene concepts to practical, production-ready Python pipelines, enabling you to audit and execute the data preparation phase of the machine learning lifecycle.

Lab Overview & Objectives

By completing this 60-minute hands-on lab, you will be able to:

1. **Build a Preprocessing Pipeline:** Load raw data, diagnose missing fields, and apply robust deletion and statistical imputation rules.
 2. **Execute Exploratory Data Analysis (EDA):** Formulate data profiles using univariate histograms, bivariate plots, and multivariate correlation heatmaps.
 3. **Detect Statistical Outliers:** Implement the **Interquartile Range (IQR) method** in Python to isolate and analyze anomalous transactions.
 4. **Implement Feature Scaling:** Apply and compare **Min-Max Normalization** (`MinMaxScaler`) and **Z-Score Standardization** (`StandardScaler`) on continuous features.
 5. **Develop a Pipeline Challenge:** Apply these combined techniques to clean and scale a retail credit risk portfolio.
-

PART I: Workspace Setup & Library Ingestion (5 Minutes)

Before writing data cleaning code, ensure that your Jupyter workspace is correctly configured and that the required scientific computing libraries are imported.

Step 1: Launch Your Jupyter Notebook

1. Open your **Anaconda Prompt** (Windows) or **Terminal** (macOS/Linux).
2. Activate your project virtual environment:
3. `conda activate ml_environment`
4. Launch Jupyter:
5. `jupyter notebook`
6. Navigate to your working folder `ML_For_Managers_Week4` and create a new Python 3 notebook named `Week4_Data_Prep_Lab.ipynb`.

Step 2: Ingest the Scientific Stack

In your first notebook cell, write and execute the following import block to load our data analysis, visualization, and scaling toolchain:

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.datasets import load_wine

# Set styling configurations for our plots
sns.set_theme(style="whitegrid")
plt.rcParams["figure.figsize"] = (8, 5)

print("Data Science stack successfully imported!")

```

PART II: Preprocessing & Cleaning the Titanic Dataset (15 Minutes)

We will use the famous **Titanic Passenger Dataset** to demonstrate standard database cleaning operations. This dataset is notoriously messy, containing missing records, unnecessary categories, and duplicated information.

Step 1: Load and Profile the Raw Data

We will load the dataset directly from the Seaborn library. Write and execute the following code to ingest and profile the data:

```

# Load the Titanic dataset
df_titanic = sns.load_dataset("titanic")

print("--- Initial Database Profile ---")
# Display dataset dimensions (rows, columns)
print(f"Dataset Shape: {df_titanic.shape}\n")

# Display a summary of data types, non-null counts, and memory usage
df_titanic.info()

```

Step 2: Diagnose Missing Data Gaps

Run the following script to calculate the exact count and percentage of missing (null) values in each column:

```

# Calculate absolute null values per column
null_counts = df_titanic.isnull().sum()

# Calculate the percentage of nulls
null_percentages = 100 * (df_titanic.isnull().sum() / len(df_titanic))

# Combine the results into a profiling DataFrame
df_missing_profile = pd.DataFrame({
    "Missing Values": null_counts,
    "Percentage (%)": null_percentages
})

```

```
})
```

```
# Display only the columns that have missing entries, sorted descending  
print(df_missing_profile[df_missing_profile["Missing Values"] >  
0].sort_values(by="Missing Values", ascending=False))
```

Step 3: Implement Cleaning and Imputation Rules

Based on our null-value diagnostics, we establish three systematic data hygiene rules:

1. **Drop Columns with Excessive Missingness:** The `deck` column is missing over 77% of its data. This column cannot be salvaged, so we drop it.
2. **Impute Missing Numbers with the Median:** The `age` column has 177 missing entries. Because age distributions can be skewed, we impute the missing values with the **column median** to prevent outliers from warping our statistics.
3. **Impute Missing Categories with the Mode:** The `embarked` column is missing 2 entries. Since this is categorical, we impute the missing records with the **most frequent value (mode)**.

Write and execute the following pipeline to apply these rules:

```
# Rule 1: Drop the heavily missing 'deck' column  
df_clean = df_titanic.drop(columns=["deck"])  
  
# Rule 2: Calculate the median age and fill missing values in-place  
age_median = df_clean["age"].median()  
df_clean["age"].fillna(age_median, inplace=True)  
print(f"Imputed missing Ages with Median: {age_median} years")  
  
# Rule 3: Find the most frequent port of embarkation and fill missing values  
embarked_mode = df_clean["embarked"].mode()[0]  
df_clean["embarked"].fillna(embarked_mode, inplace=True)  
print(f"Imputed missing Embarked ports with Mode: '{embarked_mode}'")  
  
# Verify that no missing values remain in our treated columns  
print(f"\nRemaining null values in treated columns: {df_clean[['age',  
'embarked']].isnull().sum().sum()}")
```

PART III: Exploratory Data Analysis (EDA) & Data Audits (15 Minutes)

Now that our dataset has been preprocessed and cleaned, we will execute **Exploratory Data Analysis (EDA)** to visually audit class behaviors, analyze demographic distributions, and interpret correlations.

Step 1: Univariate Analysis (Visualizing Distributions)

We will use a histogram to visualize the distribution of passenger ages, and a countplot to examine the survival rates of the cohorts.

```
# Create a 1-row, 2-column visual grid  
fig, axes = plt.subplots(1, 2, figsize=(16, 6))
```

```

# Subplot 1: Distribution of Passenger Age (Histogram with KDE)
sns.histplot(data=df_clean, x="age", bins=30, kde=True, color="skyblue",
ax=axes[0])
axes[0].set_title("Distribution of Passenger Age (Post-Imputation)",
fontsize=14, fontweight="bold")
axes[0].set_xlabel("Age (Years)")
axes[0].set_ylabel("Count")

# Subplot 2: Count Plot of Survival Rates
sns.countplot(data=df_clean, x="survived", palette="Set2", ax=axes[1])
axes[1].set_title("Absolute Count of Survivors (0 = Deceased, 1 = Survived)",
fontsize=14, fontweight="bold")
axes[1].set_xlabel("Survival Status")
axes[1].set_ylabel("Passenger Count")

plt.tight_layout()
plt.show()

```

Step 2: Bivariate Analysis (Analyzing Inter-Variable Interactions)

We will evaluate how passenger ticket class (`class`) and gender (`sex`) influenced survival rates using stacked and grouped bar charts.

```

fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Plot 1: Survival Rate broken down by Passenger Class
sns.barplot(data=df_clean, x="class", y="survived", ci=None, palette="muted",
ax=axes[0])
axes[0].set_title("Survival Probability by Ticket Class", fontsize=14,
fontweight="bold")
axes[0].set_xlabel("Passenger Class")
axes[0].set_ylabel("Survival Probability (0.0 - 1.0)")

# Plot 2: Count of Survivors grouped by Gender
sns.countplot(data=df_clean, x="class", hue="sex", palette="pastel",
ax=axes[1])
axes[1].set_title("Passenger Class Breakdown by Gender", fontsize=14,
fontweight="bold")
axes[1].set_xlabel("Passenger Class")
axes[1].set_ylabel("Passenger Count")

plt.tight_layout()
plt.show()

```

Step 3: Multivariate Analysis & Correlation Heatmaps

To understand the relationships between multiple continuous variables, we will calculate the correlation matrix and visualize it as a color-coded heatmap.

```

# Calculate the Pearson correlation matrix for our numerical columns

```

```

correlation_matrix = df_clean[["survived", "pclass", "age", "sibsp", "parch",
"fare"]].corr()

# Plot the heatmap using a cool/warm color map to highlight relationships
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f",
linewidths=0.5, vmin=-1, vmax=1)
plt.title("Correlation Matrix Heatmap of Numeric Titanic Features",
fontsize=14, fontweight="bold")
plt.show()

```

PART IV: Feature Scaling on the Wine Dataset (15 Minutes)

Distance-based models (like KNN) and gradient-descent algorithms are highly sensitive to varying scales in your input features. To demonstrate this, we will load the **Scikit-Learn Wine Dataset**, which contains features with vastly different scales, and apply **Z-Score Standardization** and **Min-Max Normalization**.

Step 1: Load the Wine Dataset and Inspect Feature Scales

Let us load the wine dataset and look at the scales of two specific features: **Alcohol** (percentage range -11% to 15%) and **Proline** (nitrogen compound range -270 to 1680).

```

# Load the raw dataset
wine = load_wine()

# Convert to a DataFrame
df_wine = pd.DataFrame(wine.data, columns=wine.feature_names)

# Inspect the summary statistics of our targeted columns
print("--- Raw Feature Scale Comparison ---")
print(df_wine[["alcohol", "proline"]].describe())

```

Step 2: Implement Z-Score Standardization (StandardScaler)

Z-Score Standardization centers your features around a mean of exactly 0 with a standard deviation of exactly 1. Execute the following code to standardize our features:

```

# Initialize the StandardScaler object
standard_scaler = StandardScaler()

# Fit and transform the raw DataFrame into standardized values
wine_standardized = standard_scaler.fit_transform(df_wine)

# Convert the resulting numpy array back into a Pandas DataFrame
df_wine_std = pd.DataFrame(wine_standardized, columns=df_wine.columns)

print("--- Standardized Feature Summary (Mean ≈ 0, Std = 1) ---")
print(df_wine_std[["alcohol", "proline"]].describe())

```

Step 3: Implement Min-Max Normalization (MinMaxScaler)

Min-Max Normalization compresses your numerical values into a strict, uniform range of exactly 0 to 1. Execute the following code to normalize our features:

```
# Initialize the MinMaxScaler object
minmax_scaler = MinMaxScaler()

# Fit and transform the raw DataFrame into normalized values
wine_normalized = minmax_scaler.fit_transform(df_wine)

# Convert back into a Pandas DataFrame
df_wine_norm = pd.DataFrame(wine_normalized, columns=df_wine.columns)

print("--- Normalized Feature Summary (Min = 0, Max = 1) ---")
print(df_wine_norm[["alcohol", "proline"]].describe())
```

Step 4: Visualizing the Impact of Scaling

Run the following code to generate histograms comparing the raw, standardized, and normalized distributions of the proline feature:

```
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Plot 1: Raw Proline Distribution
sns.histplot(df_wine["proline"], bins=15, color="salmon", kde=True,
ax=axes[0])
axes[0].set_title("Raw Proline Distribution\n(Scale: 270 to 1680)",
fontsize=12, fontweight="bold")
axes[0].set_xlabel("Raw Proline Value")

# Plot 2: Standardized Proline Distribution
sns.histplot(df_wine_std["proline"], bins=15, color="skyblue", kde=True,
ax=axes[1])
axes[1].set_title("Standardized Proline (Z-score)\n(Scale: Mean = 0, Std = 1)",
fontsize=12, fontweight="bold")
axes[1].set_xlabel("Z-Score")

# Plot 3: Normalized Proline Distribution
sns.histplot(df_wine_norm["proline"], bins=15, color="lightgreen", kde=True,
ax=axes[2])
axes[2].set_title("Normalized Proline (Min-Max)\n(Scale: 0.0 to 1.0)",
fontsize=12, fontweight="bold")
axes[2].set_xlabel("Normalized Value")

plt.tight_layout()
plt.show()
```

PART V: Capstone Challenge (10 Minutes)

The Business Case: "Automated Credit Portfolio Preprocessing"

You are the Lead Risk Analytics Manager at a retail credit institution. Your team has received a raw, messy batch of retail lending accounts containing missing fields, duplicates, and unscaled columns.

To prepare this portfolio for an automated default-prediction model, write a complete Python script that executes the following operational checklist:

1. **Duplicate Purging:** Identify and remove all duplicate passenger/customer records in-place.
2. **Missing Value Imputation:** Impute any missing numerical values in the **amount** column using the column **median**.
3. **Outlier Isolation:** Implement the **IQR method** to flag any transaction **amount** that falls outside the upper and lower outlier fences.
4. **Feature Normalization:** Scale the cleaned transaction **amount** column using **Min-Max Scaling** so that all inputs fit within a strict 0 to 1 range.

Python Capstone Starter Code & Solution

Write the following script in your notebook to run the complete automated credit cleaning and scaling pipeline:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# 1. Initialize our messy raw credit transaction database
raw_credit_data = {
    "Account_ID": ["C101", "C102", "C103", "C101", "C104", "C105", "C106"],
    "Age": [34, 45, 23, 34, 55, np.nan, 41], # Contains a missing value
    "Amount": [15.0, 22.5, 30.0, 15.0, 5000.0, 18.0, np.nan] # Duplicate row &
missing amount
}
df_credit = pd.DataFrame(raw_credit_data)
print("--- Raw Input Credit Portfolio ---")
print(df_credit, "\n")

# Step 1: Remove duplicate records based on identical rows
df_cleaned_portfolio = df_credit.drop_duplicates().reset_index(drop=True)
print("--- After Duplicate Elimination ---")
print(df_cleaned_portfolio, "\n")

# Step 2: Handle missing values in 'Amount' by imputing the column median
amount_median = df_cleaned_portfolio["Amount"].median()
df_cleaned_portfolio["Amount"].fillna(amount_median, inplace=True)
print(f"Imputed missing transaction Amounts with Median:
${amount_median:.2f}")

# Impute missing Ages with the column median as well
age_median = df_cleaned_portfolio["Age"].median()
df_cleaned_portfolio["Age"].fillna(age_median, inplace=True)
print(f"Imputed missing Ages with Median: {age_median:.0f} years\n")
```

```

# Step 3: Outlier Detection using the Interquartile Range (IQR) Method on the
'Amount' column
q1 = df_cleaned_portfolio["Amount"].quantile(0.25)
q3 = df_cleaned_portfolio["Amount"].quantile(0.75)
iqr = q3 - q1

lower_fence = q1 - (1.5 * iqr)
upper_fence = q3 + (1.5 * iqr)

print("--- Outlier Detection Bounds ---")
print(f"Lower Outlier Bound: ${lower_fence:.2f}")
print(f"Upper Outlier Bound: ${upper_fence:.2f}\n")

# Flag outliers in-place using a boolean mask
df_cleaned_portfolio["Is_Outlier"] = (df_cleaned_portfolio["Amount"] <
lower_fence) | \
                                (df_cleaned_portfolio["Amount"] >
upper_fence)
print("--- Flagged Outlier Records ---")
print(df_cleaned_portfolio, "\n")

# Step 4: Scale the 'Amount' feature using Min-Max Normalization to fit in a 0
to 1 range
# Note: For our model, we scale the outlier-treated column
scaler = MinMaxScaler()
df_cleaned_portfolio["Scaled_Amount"] =
scaler.fit_transform(df_cleaned_portfolio[["Amount"]])

print("--- Final Standardized, Scaled, and Cleaned Credit Portfolio ---")
print(df_cleaned_portfolio)

```

PART VI: Diagnostic Review Questions for Managers

Review these conceptual diagnostic questions to reinforce your strategic understanding of data preparation:

1. **The Danger of the "Mean Imputation" Shortcut:**
 - *Question:* Why is default-imputing missing values with the column *mean* a dangerous practice for highly skewed business variables like income, home values, or credit card transactions?
 - *Strategic Answer:* Highly skewed distributions (such as incomes) contain extreme outliers that pull the arithmetic mean significantly higher than the typical value. Imputing with the mean artificially inflates normal profiles, reduces the true statistical variance (compressing the standard deviation), and underestimates risk in predictive models. Using the **median** is a far more robust imputation strategy as it is unaffected by extreme outliers.
2. **Mitigating the "Accuracy Paradox" on Imbalanced Portfolios:**
 - *Question:* Your credit underwriting team reports a 99.2% Accuracy on a newly trained credit risk model. Why might you reject this model as strategically unsafe for production?

- *Strategic Answer:* In credit risk portfolios, actual defaults are rare events (often making up less than 1% of the dataset). A naive model can achieve 99% accuracy simply by labeling every single applicant as "Approved" (non-default), while failing to catch a single actual defaults. This is the **Accuracy Paradox**. To evaluate the model safely, you must reject simple accuracy and inspect metrics like **Recall** (the percentage of actual defaults successfully caught) and **Precision** (the reliability of your default flags).

3. The Technical Hazard of Data Leakage:

- *Question:* Why is it a critical error to perform feature scaling (such as `fit_transform`) on your entire raw dataset *before* executing your train-test split?
- *Strategic Answer:* If you scale your entire dataset before splitting it, the scaler will use information from your test set (the minimums, maximums, means, and standard deviations of the holdout data) to normalize the training set. This is **Data Leakage**. It leaks future, unseen test information into your active training pipeline, resulting in overly optimistic validation scores but poor performance when deployed to live customer transactions. You must always split your data first, execute `fit_transform` solely on the **training set**, and apply those exact pre-calculated scale parameters to the **testing set** using `.transform()`.