

SD LAB3: DATA PRE-PROCESSING

NAME: EBA BOKALLI ANNA

DEPARTMENT: SOFTWARE ENGINEERING

Campus B

Introduction

Data pre-processing is a crucial step in the data analysis pipeline. It involves collecting data, cleaning it, and transforming it into a format suitable for analysis. Key aspects of this process include:

1. **Data Collection:** Gathering data from various sources, such as files or databases.
2. **Data Cleaning:** Identifying and rectifying errors or inconsistencies in the data, including handling missing values and outliers.
3. **Data Transformation:** Normalizing and scaling data to ensure it is in a suitable format for analysis and modelling.
4. **Feature Engineering:** Creating new features from existing data to improve model performance.
5. **Feature Selection:** Identifying and selecting the most relevant features for the analysis.

1. Data Collection

This is the first step in order to build databases through the gathering of data from various sources.

```
import pandas as pd
# Prompt for the author's name
author_name = input("Enter your name: ")

# Load the penguins dataset
data = pd.read_csv('penguins.csv') # Replace with your actual file path
print(f"\nAuthor: {author_name}")
print("Initial Data:")
print(data.head())
```

This code will load the penguins dataset found into the current directory and display the first few rows, allowing you to see the structure and contents of the data, including columns like species, island, culmen length, culmen depth, flipper length, body mass, and sex.

OUTPUT

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

Figure 1: loaded penguins dataset

2. Data Cleaning

a) Inspecting the missing values

```
# Step 2.1: Inspect for missing values
missing_values = penguins.isnull().sum()

# Display the count of missing values for each column
missing_values[missing_values > 0]
```

It gives the states for each column in the dataset, helping us to identify which columns have missing values.

```
culmen_length_mm    2
culmen_depth_mm     2
flipper_length_mm   2
body_mass_g         2
sex                10
dtype: int64
```

Figure 2: inspecting missing values

b) Handling missing values

Dropping rows or columns with missing data

```
# Step 2.2: Handle missing values by dropping rows
penguins_cleaned_dropped = penguins.dropna()

# Display the shape of the original and cleaned dataset
print(f"Original dataset shape: {penguins.shape}")
print(f"Cleaned dataset shape (dropped): {penguins_cleaned_dropped.shape}")
```

It prints the shapes of the original and cleaned datasets, allowing you to see how many rows were removed.

```
Original dataset shape: (344, 7)
Cleaned dataset shape (dropped): (334, 7)
```

Figure 3: dropping count for missing rows or columns

Imputing missing values with the median, mode

```
# Step 2.2: Handle missing values by imputing with the mean
penguins_imputed = penguins.fillna(penguins.mean(numeric_only=True))
# Display the shape of the original and cleaned dataset
print(f"Original dataset shape: {penguins.shape}")
print(f"Cleaned dataset shape (imputed): {penguins_imputed.shape}")
# with the mode
penguins_imputed_mode = penguins.fillna(penguins.mode().iloc[0])

# Display the shape of the original and cleaned dataset
print(f"Original dataset shape: {penguins.shape}")
print(f"Cleaned dataset shape (imputed with mode): {penguins_imputed_mode.shape}")
```

It replace missing values in numerical columns with the mean as well as mode of each column, and print the shapes of the original and imputed datasets to show that the number of rows remains the same.

```
Original dataset shape: (344, 7)
Cleaned dataset shape (imputed): (344, 7)
Original dataset shape: (344, 7)
Cleaned dataset shape (imputed with mode): (344, 7)
```

Figure 4: median and mode count the number rows or columns with missing values

3. Handling Outliers

a) Identifying outliers in the 'fare' and 'age' columns using box plots.

Outliers are data points that significantly differ from other observations in a dataset. According to our penguins dataset, it was assumed that the 'fare' was body_mass_g and 'age' was culmen_length_mm.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Step 3.1: Create box plots to identify outliers
plt.figure(figsize=(15, 5))

# Box plot for body mass
plt.subplot(1, 3, 1)
sns.boxplot(y=penguins['body_mass_g'])
plt.title('Box Plot of Body Mass (g)')

# Box plot for culmen Length
plt.subplot(1, 3, 2)
sns.boxplot(y=penguins['culmen_length_mm'])
plt.title('Box Plot of Culmen Length (mm)')

# Box plot for culmen depth
plt.subplot(1, 3, 3)
sns.boxplot(y=penguins['culmen_depth_mm'])
plt.title('Box Plot of Culmen Depth (mm)')

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

The box plots will display the distributions of body_mass_g, culmen_length_mm, culmen_depth_mm. Any outliers will appear as points outside the whiskers of the box.

OUTPUT

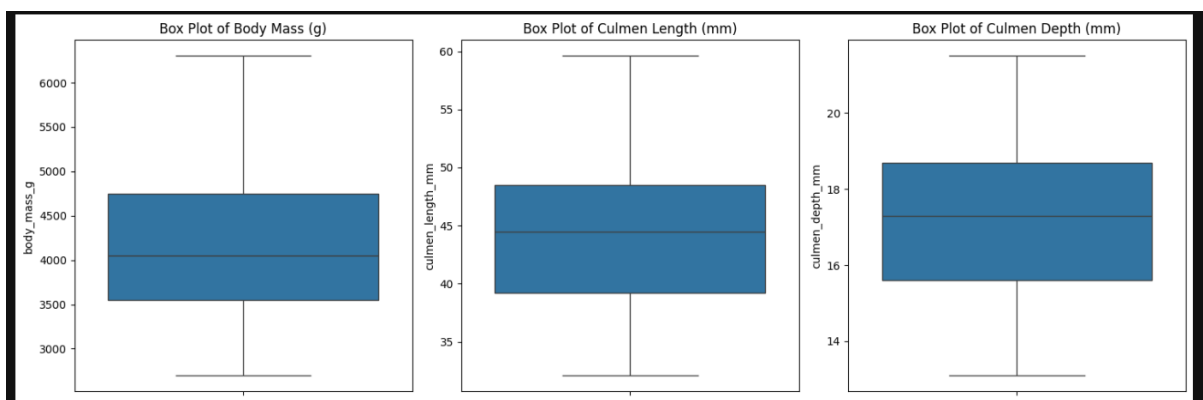


Figure 5: configured box plots with various data distributions

b) Removing the outliers

```
# Step 3.2: Remove outliers based on IQR for body_mass_g, culmen_length_mm, and culmen_depth_mm

def remove_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return df[(df[column] >= lower_bound) & (df[column] <= upper_bound)]

# Start with the original dataset
penguins_no_outliers = penguins.copy()

# Remove outliers for all three columns
for col in ['body_mass_g', 'culmen_length_mm', 'culmen_depth_mm']:
    penguins_no_outliers = remove_outliers(penguins_no_outliers, col)

# Display the shape of the original and cleaned dataset
print(f"Original dataset shape: {penguins.shape}")
print(f"Cleaned dataset shape (no outliers): {penguins_no_outliers.shape}")
```

- **Function Definition:** The `remove_outliers` function calculates the (Interquartile Range) IQR for a given column and determines the lower and upper bounds for removing outliers.
- **Loop through Columns:** We iterate through the specified columns and apply the `remove_outliers` function to each one.
- **Output:** We print the shapes of the original and cleaned datasets to observe how many rows were removed.

```
Original dataset shape: (344, 7)
Cleaned dataset shape (no outliers): (342, 7)
```

Figure 6: no outliers observed, removal of redundant data

4. Data Normalization

```
from sklearn.preprocessing import MinMaxScaler

# Step 4.1: Normalize the numerical features using Min-Max scaling
scaler = MinMaxScaler()

# Apply Min-Max scaling to selected columns
penguins_normalized = penguins_no_outliers.copy() # Using the dataset without outliers
penguins_normalized[['body_mass_g', 'culmen_length_mm', 'culmen_depth_mm']] = scaler.fit_transform(
    penguins_normalized[['body_mass_g', 'culmen_length_mm', 'culmen_depth_mm']]
)

# Display the first few rows of the normalized dataset
print(penguins_normalized[['body_mass_g', 'culmen_length_mm', 'culmen_depth_mm']].head())
```

We import `MinMaxScaler` from `sklearn.preprocessing`. We create a scaler object and apply it to the selected columns, transforming their values to a range between zero and one. Finally, we display the normalized values for verification.

OUTPUT

	body_mass_g	culmen_length_mm	culmen_depth_mm
0	0.291667	0.254545	0.666667
1	0.305556	0.269091	0.511905
2	0.152778	0.298182	0.583333
4	0.208333	0.167273	0.738095
5	0.263889	0.261818	0.892857

Figure 7: rows normalised and scaled between zero and one.

4. Feature Engineering

a) Create family size Column

The penguins dataset does not contain family-related columns; we will create a new feature based on the average body mass of the penguins, which might be a proxy for size.

```
import pandas as pd

# Load the penguins dataset
penguins_df = pd.read_csv('penguins.csv') # Adjust the path if necessary

# Display the first few rows to verify loading
print(penguins_df.head())

# Step 5.1: Create a new feature based on body mass
penguins_df['average_body_mass'] = penguins_df.groupby('species')['body_mass_g'].transform('mean')

# Display the first few rows to verify the new column
print(penguins_df[['species', 'body_mass_g', 'average_body_mass']].head())
```

We create a new column `average_body_mass` that calculates the mean body mass for each species.

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	\
0	Adelie	Torgersen	39.1	18.7	181.0	
1	Adelie	Torgersen	39.5	17.4	186.0	
2	Adelie	Torgersen	40.3	18.0	195.0	
3	Adelie	Torgersen	NaN	NaN	NaN	
4	Adelie	Torgersen	36.7	19.3	193.0	
	body_mass_g	sex				
0	3750.0	MALE				
1	3800.0	FEMALE				
2	3250.0	FEMALE				
3	NaN	NaN				
4	3450.0	FEMALE				
	species	body_mass_g	average_body_mass			
0	Adelie	3750.0	3700.662252			
1	Adelie	3800.0	3700.662252			
2	Adelie	3250.0	3700.662252			
3	Adelie	NaN	3700.662252			
4	Adelie	3450.0	3700.662252			

Figure 8: showing species with their various average mass

b) Create size category column

The first row is used to verify the new column

```
# Step 5.2: Create a size_category column based on body mass
def categorize_size(mass):
    if pd.isna(mass): # Check for NaN
        return 'Unknown' # Assign a category for NaN values
    elif mass < 3500:
        return 'Small'
    elif 3500 <= mass < 5000:
        return 'Medium'
    else:
        return 'Large'

penguins_df['size_category'] = penguins_df['body_mass_g'].apply(categorize_size)

# Display the first few rows to verify the new column
print(penguins_df[['body_mass_g', 'size_category']].head())
```

The categorize size function, we check for “NaN” values using “pd.isna (mass)”. It is derived from the “body_mass_g” values, which assigns categories based on specified thresholds:

- **Small:** body mass < 3500
- **Medium:** 3500 <= body mass < 5000
- **Large:** body mass > 5000
- If a “NaN” is encountered, we return “Unknown”. This ensures that missing values are explicitly categorized instead of defaulting to an unintended value.

OUTPUT

	body_mass_g	size_category
0	3750.0	Medium
1	3800.0	Medium
2	3250.0	Small
3	NaN	Unknown
4	3450.0	Small

Figure 9: showing the penguin body mass with their various size categories as the new column

If a “NaN” is encountered, we return “Unknown”. This ensures that missing values are explicitly categorized instead of defaulting to an unintended value.

i. Features Selection

Correlation Analysis

First we need to display the data frame to observe its structure using “print (penguins_df.info ())”.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 344 entries, 0 to 343
Data columns (total 9 columns):
#   Column                Non-Null Count  Dtype
---  -
0   species                344 non-null    object
1   island                 344 non-null    object
2   culmen_length_mm       342 non-null    float64
3   culmen_depth_mm        342 non-null    float64
4   flipper_length_mm      342 non-null    float64
5   body_mass_g            342 non-null    float64
6   sex                    334 non-null    object
7   average_body_mass      344 non-null    float64
8   size_category          344 non-null    object
dtypes: float64(5), object(4)
memory usage: 24.3+ KB
```

Figure 10: data frame which consists of 9 columns listing all the species, dtype, non-null count

```
# Step 6.1: Correlation analysis
numeric_columns = penguins_df.select_dtypes(include=['float64', 'int64']) # Select only numeric columns
correlation_matrix = numeric_columns.corr()

# Display the correlation matrix
plt.figure(figsize=(8, 5))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')
plt.show()
```

- **select_dtypes (include= ['float64', 'int64']):** This method filters the Data Frame to include only columns of type "float64" and "int64", which are numeric types suitable for correlation analysis.
- **corr ():** This function calculates the correlation matrix for the numeric columns.

OUTPUT

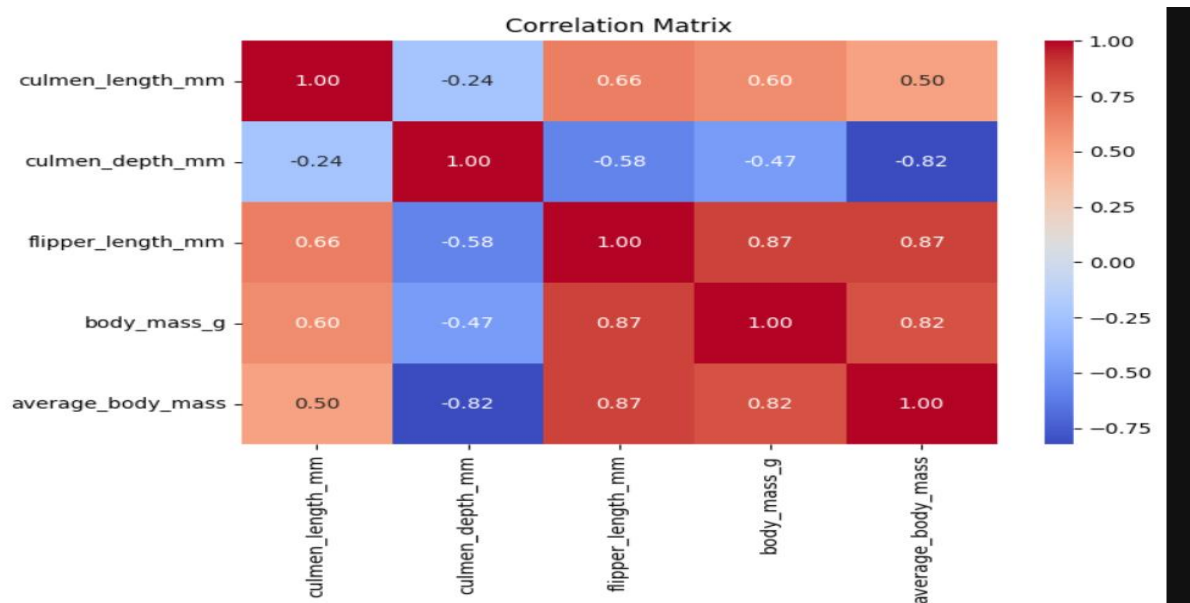


Figure 11:

5) Model Building

We will use a Random Forest model to assess feature importance.

- We prepare the features and target variable for modelling.
- We fit a Random Forest classifier and extract feature importance.
- the resulting Data Frame shows which features contribute the most to the model's predictions.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Prepare the data for modeling (drop non-numeric and target columns)
# Assuming we want to predict species based on body_mass_g, culmen_length_mm, and culmen_depth_mm
features = penguins_df[['body_mass_g', 'culmen_length_mm', 'culmen_depth_mm']]
target = penguins_df['species']

# Train a Random Forest model to assess feature importance
model = RandomForestClassifier(random_state=42)
model.fit(features, target)

# Get feature importance
importance = model.feature_importances_

# Create a DataFrame for visualization
importance_df = pd.DataFrame({'Feature': features.columns, 'Importance': importance})
importance_df = importance_df.sort_values(by='Importance', ascending=False)

# Display feature importance
print(importance_df)
```

OUTPUT

	Feature	Importance
1	culmen_length_mm	0.453113
0	body_mass_g	0.286489
2	culmen_depth_mm	0.260397

Figure12: showing the most important feature of our penguin dataset

Now, we will split the data into training and testing sets and build a logistic regression model.

```
from sklearn.model_selection import train_test_split

# Step 7.1: Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)

# Display the shape of the resulting sets
print(f"Training feature set shape: {X_train.shape}")
print(f"Testing feature set shape: {X_test.shape}")
```

OUTPUT

```
Training feature set shape: (275, 3)
Testing feature set shape: (69, 3)
```

Figure13: Showing the count for training and testing shape

Predictions using the Model

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

# Example dataset creation (replace this with your actual dataset)
X = np.array([[1, 2], [np.nan, 3], [7, 6], [np.nan, np.nan], [4, 5]])
y = np.array([0, 1, 0, 1, 0]) # Corresponding labels

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an imputer to fill NaN values with the mean
imputer = SimpleImputer(strategy='mean')


# Fit the imputer on the training data and transform both training and test data
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)

# Fit the logistic regression model
logistic_model = LogisticRegression(max_iter=200)
logistic_model.fit(X_train_imputed, y_train)

# Make predictions on the test set
y_pred = logistic_model.predict(X_test_imputed)

# Print predictions
print("Predictions:", y_pred)
```

OUTPUT



```
Predictions: [0]
```

Figure13: outcome showing the predicted value to be zero

- **Imputation:** The “SimpleImputer” is used to fill in NaN values with the mean of each column. This allows you to keep all your data without dropping any rows.
- **Data Splitting:** The dataset is split into training and test sets using “train_test_split”.
- **Model Fitting:** The logistic regression model is fitted using the imputed training data.
- **Predictions:** Predictions are made on the imputed test set.