DATA CLEANING

The objective of this task was to prepare the dataset for accurate analysis by ensuring that the data is free from inconsistencies, errors, and unnecessary noise. This document provides a detailed summary of the methodologies, assumptions, and techniques used to clean the data. By performing these steps, the final dataset was prepared for analysis and met the requirements of consistency, accuracy, and completeness as outlined in the assignment.

INSPECT THE DATA:

Methodology: The dataset was examined to understand its structure, data types, and potential inconsistencies. Initial inspection was done using methods such as .head(), .info(), and .describe(). These methods provided insight into the first few rows, column data types, summary statistics, and the presence of missing values.

Findings:

- The dataset contained **11,000 rows and 8 columns**.
- There are Null values in all columns except Id.
- Existence of duplicate Id's.
- Inconsistencies in Name and Department Columns.
- Incorrect email formats in Email column and Invalid date formats in Join Date column.
- Noise in Salary column.

RECORD QA ISSUES:

- **ID**: Duplicate entries for ID, with slight variations in Name, Department (e.g., "HR" vs. "HRs"), and Join Date. In some cases, one row had the correct email while another had the correct salary, leading to inconsistencies across fields.
- **Name**: The Name column contained extraneous words and inconsistencies, requiring cleaning to ensure uniform formatting.
- **Emai**l: Unformatted and non-professional emails.
- **JoinDate:** Inconsistent date formats.
- **Department:** Inconsistent and non-standardized department names.
- **Salary**:Noise in the Salary column.
- **Age**: Null values in Age column.

REMOVING DUPLICATES:

Removing duplicates is essential to ensure data integrity, improve analysis accuracy, enhance efficiency, and support informed decision-making.

Duplicates were identified using:

```
df.duplicated().sum()
```

Removing duplicates:

```
df.drop_duplicates(inplace=True)
```

REMOVING DUPLICATE ID'S

Problem with duplicate id's:

- **Inconsistent Data**: Duplicate entries may contain conflicting information across various columns (e.g., different names, emails, and departments).
- **Data Redundancy**: Having multiple records for the same ID leads to inflated counts and can skew analysis results.

Methodology:

- **Email Validation**: The code includes a function (is_valid_email) to ensure that any email address included in the merged record adheres to a valid format.
- **Sorting**: The duplicates are sorted by the Join_Date, allowing the code to prioritize the most recent entry when merging.
- **Iterative Merging**: The merge_duplicates function iterates through the grouped entries, filling in missing values based on predefined rules:
  - Names: Keep the name with the least number of characters.
  - Emails: Retain the first valid email found.
  - Join Dates: Always keep the latest joining date.
  - Departments: Keep the department with the least number of characters.
  - Salaries: Update the salary if it's non-null.
- **Group Application**: The merging process is applied to each group of duplicates using groupby(), consolidating the information into a single row for each unique ID.

Code:

```
# Define  functions for email validation and merging duplicates

def is_valid_email(email):
    """Check if the given email address has a valid format."""
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))

def merge_duplicates(group):
    """Merge duplicate rows in a DataFrame group."""
    # Sort group by 'Join Date' to prioritize latest data
    group_sorted = group.sort_values(by='Join_Date', ascending=False)

    # Start with the latest sorted row
    merged = group_sorted.iloc[0].copy()

    # Initialize a flag to check if a valid email has been found
    valid_email_found = False

    # Iterate through the sorted group to fill missing values and correct information
    for _, row in group_sorted.iterrows():
        # Name: Keep the name with the least number of characters and non-null values
        if pd.notnull(row['Name']):
            if pd.isnull(merged['Name']) or len(row['Name']) < len(merged['Name']):
                merged['Name'] = row['Name']

        # Email: Take the first valid email found
        if pd.notnull(row['Email']) and is_valid_email(row['Email']):
            merged['Email'] = row['Email']
            valid_email_found = True
```

```
        # Join Date: Always keep the latest join date and non-null values
        if pd.notnull(row['Join_Date']):
            merged['Join_Date'] = row['Join_Date']

        # Department: Keep the department with the least number of characters and non-null values
        if pd.notnull(row['Department']):
            if pd.isnull(merged['Department']) or len(row['Department']) < len(merged['Department']):
                merged['Department'] = row['Department']

        # Salary: Update the salary if it's non-null
        if pd.notnull(row['Salary']):
            merged['Salary'] = row['Salary']

    # If no valid email was found, set it to NaN
    if not valid_email_found:
        merged['Email'] = None

    return merged

# Group by 'Id' and apply the merge function to duplicates
cleaned_df = df.groupby('Id').apply(merge_duplicates).reset_index(drop=True)


# Display the final cleaned dataframe
cleaned_df.head()
```

HANDLING MISSING VALUES:

1) **Removing Rows with Null values in all the 6 columns.**

Code:

```
 #Removing rows that contains Nan values in 6 columns


columns_to_check = ['Name', 'Age', 'Email', 'Join_Date', 'Salary',
       'Department']

# Remove rows where all specified columns have null values

df= df.dropna(subset=columns_to_check, how='all')
```

2) **Filling Missing Values in Each column:**

- Name: Nan values are filled with Unknown.
- Department: Nan values are filled with Unknown.
- Age : Nan values are filled with mean Age.
- Salary: Nan values are filled with department wise median salary.
- Join_Date:
    - Forward fill to fill missing dates.
    - Backward fill to fill remaining null values.

Code:

```
  cleaned_df['Name']=cleaned_df['Name'].fillna('Unknown')

  cleaned_df['Department'].fillna('Unknown',inplace=True)

  #Imputing Age column with mean value of Age column
  mean_age=cleaned_df['Age'].mean()

  cleaned_df['Age']=cleaned_df['Age'].fillna(mean_age)

  # Calculate the median salary for each department

Median_depart_salary=cleaned_df.groupby('Department')['Salary'].median(
)

# Fill missing salaries with the department's median salary

cleaned_df['Salary']=cleaned_df['Salary'].fillna(cleaned_df['Department
'].map(Median_depart_salary))

# Dealing with Nan values using forward fill

cleaned_df['Join_Date'].fillna(method='ffill', inplace=True)

# To handle any remaining Nan values

cleaned_df['Join_Date'].fillna(method='bfill',inplace=True)
```

CORRECTING EMAIL FORMATS:

Objective : To ensure that all email addresses follow a standard format (username@domain.com).

Methodology:

1) **Defined a Email Validation Function:** A function (validate_email) is defined to check whether an email address is valid. It uses a regular expression pattern (^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$) to validate that the email has a proper format (e.g., username@domain.com).

   Steps:
   - If the email is missing (NaN), it returns False.
   - If the email format is invalid, it returns False; otherwise, it returns True.

2) **Applying Email Validation**: The validation function is applied to each row in the Email column to check if the email is valid.

   Steps:

   - The apply() method is used to apply the validate_email function.

- Emails that do not pass the validation are replaced with NaN using the lambda function: lambda x: x if validate_email(x) else np.nan.

3) **Removing Leading and Trailing Whitespace**: Ensures that any extra spaces at the beginning or end of the email addresses are removed.

Steps:

- The str.strip() function is applied to the Email column to remove whitespace from the start and end of each email address.

4) **Converting Emails to Lowercase**: Standardizes all email addresses by converting them to lowercase, as emails are typically case-insensitive.

Steps:

- The str.lower() function is applied to the Email column, ensuring that all email addresses are lowercase.

5) **Dropping Rows with Invalid Emails**: After identifying invalid email addresses (now represented as NaN), these rows are removed from the dataset.

Steps:

- The `dropna()` method is used to remove any rows where the `Email` column contains `NaN` values.

Code:

```
import numpy as np
# Function to validate email addresses
def validate_email(email):
    if pd.isnull(email):
        return False
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return re.match(pattern, email) is not None

# Apply email validation and replace unprofessional emails with NaN
cleaned_df['Email'] = cleaned_df['Email'].apply(lambda x: x if validate_email(x) else np.nan)

# Remove leading and trailing whitespace
cleaned_df['Email'] = cleaned_df['Email'].str.strip()

# Convert to lowercase
cleaned_df['Email'] = cleaned_df['Email'].str.lower()


# Drop rows with NaN values in 'Email'
cleaned_df = cleaned_df.dropna(subset=['Email'])
```

CLEANING NAME FIELDS:

Objective: To ensure names are consistently formatted and free from extraneous words.

Methodology:

1) **Define the clean_name Function**: The function clean_name is designed to clean the Name column in a dataset by removing titles and keeping only the first word of the name. This helps standardize the names for further analysis.

Steps:

- **Removing titles** (such as "Mr.", "Mrs.", "Dr.", etc.) to simplify the names.
- **Removing extra spaces** from the beginning and end of each name.
- **Keeping only the first word** of the name for consistency or simplicity.

Code:

```
# Simple function to clean names by keeping only the first word
def clean_name(name):
    # List of titles to remove
    titles = ["Mr.", "Mrs.", "Ms.", "Dr.", "Prof.", "Sir", "Jr.", "Sr."]

    # Remove titles
    for title in titles:
        name = name.replace(title, "")

    # Remove extra spaces
    name = name.strip()

    # Split the cleaned name into words
    words = name.split()

    # Keep only the first word
    if words:
        return words[0]
    else:
        return name  # Return the original name if it's empty or NaN

# Apply the clean_name function to the 'Name' column
cleaned_df['Name'] = cleaned_df['Name'].apply(clean_name)

# Display the cleaned DataFrame
cleaned_df.head()
```

STANDARDISE DATE FORMATS:

Objective: To ensure all dates in the 'Join Date' column follow YYYY-MM-DD format .

Methodology:

1) **Define the convert_to_yyyy_mm_dd Function**: This function is designed to standardize date strings in the Join_Date column by converting them to the YYYY-MM-DD format.

2) **Convert Date Format Using pd.to_datetime()**: The function attempts to convert the date string into the YYYY-MM-DD format using Panda's pd.to_datetime() function.

3) **Handle Invalid Dates with try-except**: The function uses a try-except block to handle invalid date formats. If the date string cannot be converted (for example, if the string is not a valid date), the function catches the error and returns NaN (missing value).

4) **Apply the Function to the Join_Date Column**: The convert_to_yyyy_mm_dd function is applied to every row in the Join_Date column using the apply() method.

Code:

```
# Function to convert date strings to YYYY-MM-DD format
def convert_to_yyyy_mm_dd(date_str):
    try:
        return pd.to_datetime(date_str, dayfirst=True).strftime('%Y-%m-%d')
    except:
        return np.nan  # Return NaN for any invalid dates

# Apply conversion to 'Join Date' column
cleaned_df['Join_Date'] = cleaned_df['Join_Date'].apply(convert_to_yyyy_mm_dd)

cleaned_df
```

CORRECT DEPARTMENT NAMES:

Objective: To standardise department names to ensure consistency.

Methodology:

1) **Define a List of Valid Department Names**: A list called department names is created, containing the valid and standardized department names: "Marketing", "Support", "Sales", "Engineering", and "HR".

2) Define the **clean_dept Function**: The clean_dept function is created to check and clean department names in the dataset. It compares each department name in the data with the valid names in department names.

3) **Check for Missing Values**: The function first checks if the department name is missing (NaN). If so, it returns the original value (NaN) without any modifications.

4) **Loop Through Valid Department Names**: The function then iterates through the list of valid department names (department_names) to check if the current department name contains any of the valid options.

5) **Check for Partial Matches**: During each iteration, the function checks if the valid department name (e.g., "HR") is part of the current department name in the dataset (e.g., "HRs"). If a match is found (e.g., if the department name contains "HRs" or "HR Department"), the function replaces the department name with the valid department name from the list (in this case, "HR").

6) **Apply the clean_dept Function to the Department Column**: The clean_dept function is applied to each row in the Department column of the Data Frame using the apply() method.

Code:

```
# List of valid department values
department_names = ['Marketing', 'Support', 'Sales', 'Engineering', 'HR']

# Function to clean department names
def clean_dept(dept):

 # Keep missing values (NaN) as they are

    if pd.isna(dept):
        return dept

    # Loop through the valid department names and find a match

    for valid_dept in department_names:

        if valid_dept in dept:  # Check if the valid department is part of the name

            return valid_dept  # Replace with the valid department name

    return dept  # If no match is found, keep the original name

# Apply the function to the 'Department' column

cleaned_df['Department'] = cleaned_df['Department'].apply(clean_dept)

# Display the cleaned DataFrame

cleaned_df.head()
```

CLEANING AGE COLUMN:

Objective: To ensure clean values in Age column.

Methodology:

1) **Impute Missing Age Values with the Mean**:

```
# Filling Missing values with Age mean
 mean_age = cleaned_df['Age']. mean()
 cleaned_df['Age'] = cleaned_df['Age'].fillna(mean_age)
```

2) **Converting Age datatype from float to int:**

```
cleaned_df['Age'] = cleaned_df['Age'].astype(int)
```

HANDLING SALARY NOISE:

Objective: To remove noises added to Salary column.

Methodology:

1) **Calculate the Median Salary for Each Department**

2) **Fill Missing Salaries with the Department's Median Salary**
3) **Round the 'Salary' Column to Two Decimal Places to remove noise in the Salary column.**

Code:

```
# finding department wise median Salary

Median_depart_salary = cleaned_df.groupby('Department')['Salary'].median()

# Mapping median Salary to Nan values

cleaned_df['Salary']=cleaned_df['Salary'].fillna(cleaned_df['Department'].map(Median_depart_salary)
)


#Removing noise in Salary column using round function

cleaned_df['Salary'] = cleaned_df['Salary'].round(2)
```

CONVERTING DATAFRAME TO CSV FILE

```
cleaned_df.to_csv(“cleaned_dataset.csv”,index=False)
```

OUTCOME:

After the data cleaning process, the dataset was reduced to 6,838 rows and 7 columns. All duplicate records were successfully merged, ensuring the most accurate and up-to-date information for each entry, including valid email addresses, standardized names, and departments. The cleaned dataset is now ready for further analysis.