Data Minnig

CBS 3007

L51 + L52

NAME	REG NO
Asmith	21BBS0066
Saksham Agnihotri	21BBS0080
Dhoop Patel	21BBS0140

Experiment - 1

GitHub Link - https://github.com/Abstract-state/Data-Minning

About the Data:

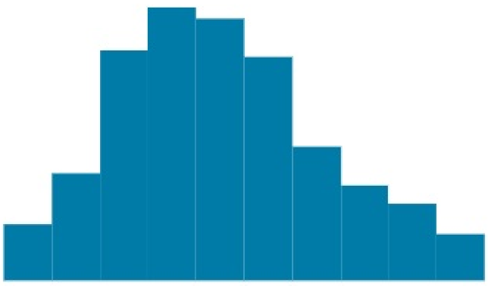
This HR Analytics dataset contains comprehensive information on employee demographics, job roles, satisfaction levels, compensation, and work experience, providing valuable insights for understanding and improving various aspects of human resources management within an organization.

EmpID

1470
unique values

Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Unique	1470	
Most Common	RM1465	0%

Age



18 60

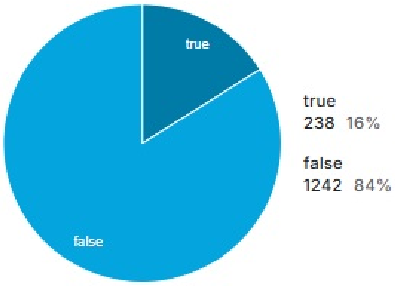
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	36.9	
Std. Deviation	9.13	
Quantiles	18	Min
	30	25%
	36	50%
	43	75%
	60	Max

AgeGroup

26-35	41%
36-45	32%
Other (398)	27%

Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Unique	5	
Most Common	26-35	41%

✓ Attrition



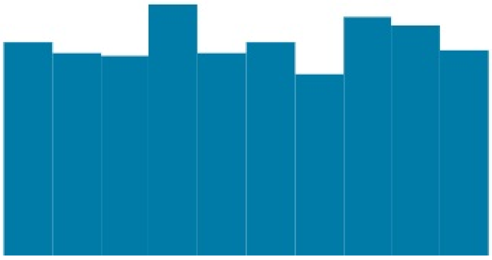
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
True	238	16%
False	1242	84%

BusinessTravel

Travel_Rarely	70%
Travel_Frequently	19%
Other (159)	11%

Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Unique	4	
Most Common	Travel_Rarely	70%

DailyRate



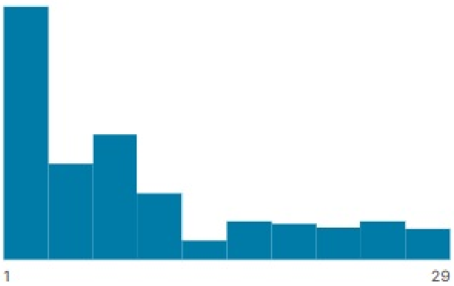
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	801	
Std. Deviation	403	
Quantiles	102	Min
	465	25%
	801	50%

Department

Research & Development	65%
Sales	30%
Other (63)	4%

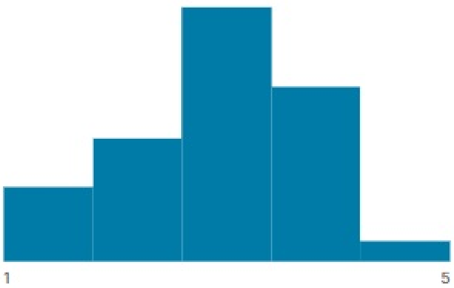
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Unique	3	
Most Common	Research &...	65%

DistanceFromHome



Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	9.22	
Std. Deviation	8.13	
Quantiles	1	Min
	2	25%
	7	50%
	14	75%
	29	Max

Education



Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	2.91	
Std. Deviation	1.02	
Quantiles	1	Min
	2	25%
	3	50%
	4	75%
	5	Max

EducationField

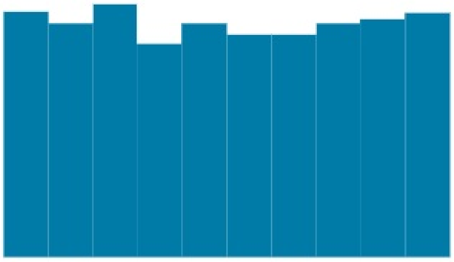
Life Sciences	41%
Medical	32%
Other (403)	27%

Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Unique	6	
Most Common	Life Sciences	41%

EmployeeCount

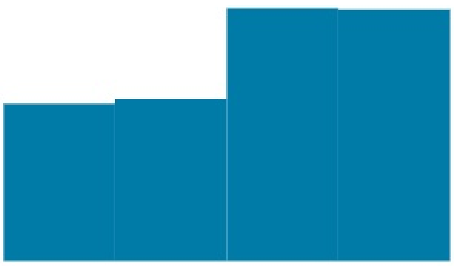
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	1	
Std. Deviation	0	
Quantiles	1	Min
	1	25%
	1	50%
	1	75%
	1	Max

EmployeeNumber



Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	1.03k	
Std. Deviation	606	
Quantiles	1	Min
	494	25%
	1028	50%
	1569	75%
	2068	Max

EnvironmentSatisfaction



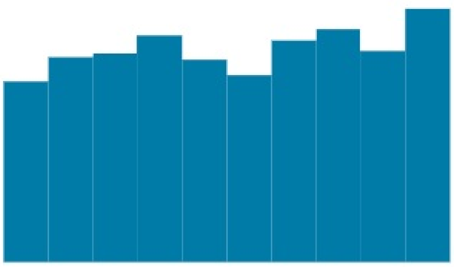
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	2.72	
Std. Deviation	1.09	
Quantiles	1	Min
	2	25%
	3	50%
	4	75%
	4	Max

Gender

Male	60%
Female	40%

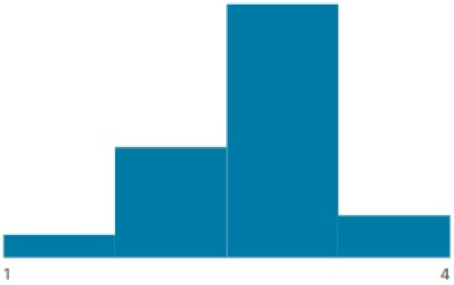
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Unique	2	
Most Common	Male	60%

HourlyRate



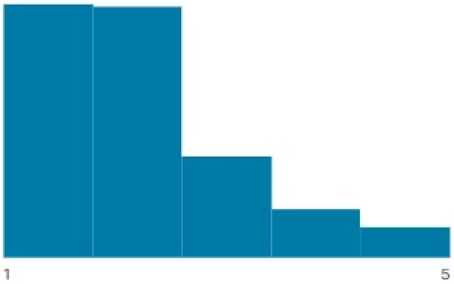
Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	65.8	
Std. Deviation	20.3	
Quantiles	30	Min
	48	25%
	66	50%
	83	75%
	100	Max

JobInvolvement



Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	2.73	
Std. Deviation	0.71	
Quantiles	1	Min
	2	25%
	3	50%
	3	75%
	4	Max

JobLevel



Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Mean	2.06	
Std. Deviation	1.11	
Quantiles	1	Min
	1	25%
	2	50%
	3	75%
	5	Max

JobRole

Sales Executive	22%
Research Scientist	20%
Other (858)	58%

Valid	1480	100%
Mismatched	0	0%
Missing	0	0%
Unique	9	
Most Common	Sales Exec...	22%

Dataset Columns:

The dataset used in this project contains the following columns:

- **EmpID:** Employee ID
- **Age:** Age of the employee
- **AgeGroup:** Age group to which the employee belongs
- **Attrition:** Employee attrition status (whether the employee has left the organization or is still active)
- **BusinessTravel:** Frequency of business travel for the employee
- **DailyRate:** Daily rate of pay for the employee
- **Department:** Department in which the employee works
- **DistanceFromHome:** Distance in miles from the employee's home to the workplace
- **Education:** Level of education attained by the employee
- **EducationField:** Field of education of the employee
- **EmployeeCount:** Number of employees
- **EmployeeNumber:** Unique identifier for each employee
- **EnvironmentSatisfaction:** Employee's satisfaction level with the work environment
- **Gender:** Gender of the employee
- **HourlyRate:** Hourly rate of pay for the employee
- **JobInvolvement:** Employee's level of job involvement
- **JobLevel:** Level of the employee's job position
- **JobRole:** Role of the employee within the organization
- **JobSatisfaction:** Employee's satisfaction level with their job
- **MaritalStatus:** Marital status of the employee
- **MonthlyIncome:** Monthly income of the employee
- **SalarySlab:** Categorization of monthly income into salary slabs
- **MonthlyRate:** Monthly rate of pay for the employee
- **NumCompaniesWorked:** Number of companies the employee has worked for in the past
- **Over18:** Whether the employee is over 18 years old

- **OverTime:** Whether the employee works overtime or not
- **PercentSalaryHike:** Percentage increase in salary for the employee
- **PerformanceRating:** Performance rating of the employee
- **RelationshipSatisfaction:** Employee's satisfaction level with work relationships
- **StandardHours:** Standard working hours for the employee
- **StockOptionLevel:** Level of stock options granted to the employee
- **TotalWorkingYears:** Total number of years the employee has worked
- **TrainingTimesLastYear:** Number of training sessions attended by the employee in the last year
- **WorkLifeBalance:** Employee's work-life balance satisfaction level
- **YearsAtCompany:** Number of years the employee has worked at the current company
- **YearsInCurrentRole:** Number of years the employee has been in the current role
- **YearsSinceLastPromotion:** Number of years since the employee's last promotion
- **YearsWithCurrManager:** Number of years the employee has been working with the current manager

This code snippet imports essential libraries for data analysis and machine learning: NumPy for numerical operations, Pandas for data manipulation, Matplotlib and Seaborn for data visualization, and scikit-learn's LabelEncoder for encoding categorical variables. Additionally, it imports the re module for working with regular expressions.

In [ ]:
```
#Importing Necessary Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
import re
```

This line of code reads a CSV file named "HR_Analytics.csv" into a Pandas DataFrame named `Hr_data`, allowing for easy data manipulation and analysis using the powerful tools provided by the Pandas library.

In [ ]:
```
#Reading the dataset
Hr_data = pd.read_csv("HR_Analytics.csv")
```

This line of code displays the first five rows of the `Hr_data` DataFrame, providing a quick look at the structure and contents of the dataset.

In [ ]:
```
Hr_data.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...
0	RM297	18	18-25	Yes	Travel_Rarely	230	Research & Development	3	3	Life Sciences	...
1	RM302	18	18-25	No	Travel_Rarely	812	Sales	10	3	Medical	...
2	RM458	18	18-25	Yes	Travel_Frequently	1306	Sales	5	3	Marketing	...
3	RM728	18	18-25	No	Non-Travel	287	Research & Development	5	2	Life Sciences	...
4	RM829	18	18-25	Yes	Non-Travel	247	Research & Development	8	1	Medical	...

5 rows × 38 columns

This line of code provides a summary of the `Hr_data` DataFrame, including the number of non-null entries, data types of each column, and memory usage, which helps in understanding the dataset's completeness and structure.

In [ ]:
```
Hr_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1480 entries, 0 to 1479
Data columns (total 38 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   EmpID                     1480 non-null   object 
 1   Age                       1480 non-null   int64  
 2   AgeGroup                  1480 non-null   object 
 3   Attrition                 1480 non-null   object 
 4   BusinessTravel            1480 non-null   object 
 5   DailyRate                 1480 non-null   int64  
 6   Department                1480 non-null   object 
 7   DistanceFromHome          1480 non-null   int64  
 8   Education                 1480 non-null   int64  
 9   EducationField            1480 non-null   object 
 10  EmployeeCount             1480 non-null   int64  
 11  EmployeeNumber            1480 non-null   int64  
 12  EnvironmentSatisfaction   1480 non-null   int64  
 13  Gender                    1480 non-null   object 
 14  HourlyRate                1480 non-null   int64  
 15  JobInvolvement            1480 non-null   int64  
 16  JobLevel                  1480 non-null   int64  
 17  JobRole                   1480 non-null   object 
 18  JobSatisfaction           1480 non-null   int64  
 19  MaritalStatus             1480 non-null   object 
 20  MonthlyIncome             1480 non-null   int64  
 21  SalarySlab                1480 non-null   object 
 22  MonthlyRate               1480 non-null   int64  
 23  NumCompaniesWorked        1480 non-null   int64  
 24  Over18                    1480 non-null   object 
 25  OverTime                  1480 non-null   object 
 26  PercentSalaryHike         1480 non-null   int64  
 27  PerformanceRating         1480 non-null   int64  
 28  RelationshipSatisfaction  1480 non-null   int64  
 29  StandardHours             1480 non-null   int64  
 30  StockOptionLevel          1480 non-null   int64  
 31  TotalWorkingYears         1480 non-null   int64  
 32  TrainingTimesLastYear     1480 non-null   int64  
 33  WorkLifeBalance           1480 non-null   int64  
 34  YearsAtCompany            1480 non-null   int64  
 35  YearsInCurrentRole        1480 non-null   int64  
 36  YearsSinceLastPromotion   1480 non-null   int64  
 37  YearsWithCurrManager      1423 non-null   float64
dtypes: float64(1), int64(25), object(12)
memory usage: 439.5+ KB
```

This line of code generates descriptive statistics for the `Hr_data` DataFrame, including measures such as count, mean, standard deviation, minimum, maximum, and quartiles for numeric columns, which helps in understanding the distribution and central tendencies of the data.

In [ ]:
```python
Hr_data.describe()
```

Out[ ]:

	Age	DailyRate	DistanceFromHome	Education	EmployeeCount	EmployeeNumber	EnvironmentSatisfaction
count	1480.000000	1480.000000	1480.000000	1480.000000	1480.0	1480.000000	1480.000000
mean	36.917568	801.384459	9.220270	2.910811	1.0	1031.860811	2.724324
std	9.128559	403.126988	8.131201	1.023796	0.0	605.955046	1.092579
min	18.000000	102.000000	1.000000	1.000000	1.0	1.000000	1.000000
25%	30.000000	465.000000	2.000000	2.000000	1.0	493.750000	2.000000
50%	36.000000	800.000000	7.000000	3.000000	1.0	1027.500000	3.000000
75%	43.000000	1157.000000	14.000000	4.000000	1.0	1568.250000	4.000000
max	60.000000	1499.000000	29.000000	5.000000	1.0	2068.000000	4.000000

8 rows × 26 columns

LABEL ENCODING

This line of code generates descriptive statistics for the `Hr_data` DataFrame, including measures such as count, mean, standard deviation, minimum, maximum, and quartiles for numeric columns, which helps in understanding the distribution and central tendencies of the data.

In [ ]:
```python
#Encoding four coloumns
label_encoders = {}
columns_to_encode = ['Attrition', 'BusinessTravel', 'Gender', 'JobRole']
for column in columns_to_encode:
```

```
    label_encoders[column] = LabelEncoder()
    Hr_data[column] = label_encoders[column].fit_transform(Hr_data[column].astype(str))
```

This line of code displays the first five rows of the `Hr_data` DataFrame after the categorical columns specified in the previous code snippet have been encoded into numerical values. This allows you to verify the changes made by the `LabelEncoder`.

In [ ]:
```
Hr_data.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...
0	RM297	18	18-25	1	3	230	Research & Development	3	3	Life Sciences	...
1	RM302	18	18-25	0	3	812	Sales	10	3	Medical	...
2	RM458	18	18-25	1	2	1306	Sales	5	3	Marketing	...
3	RM728	18	18-25	0	0	287	Research & Development	5	2	Life Sciences	...
4	RM829	18	18-25	1	0	247	Research & Development	8	1	Medical	...

5 rows × 38 columns

EXPERIMENT QUESTIONS

1. DATA PRE - PROCESSING

A. Ignore the tuple

This line of code prints the number of rows in the `Hr_data` DataFrame, which helps you determine how many data entries are present in the dataset before any processing. Note that `Hr_data.shape[0]` returns the number of rows; if you want the number of columns, you would use `Hr_data.shape[1]`.

In [ ]:
```
# No of coloumns before
print(Hr_data.shape[0])
```

```
1480
```

This code snippet removes any rows from the `Hr_data` DataFrame that contain missing values using the `dropna` method with `how='any'`, which means rows with at least one `NaN` value are dropped. The `head()` method then displays the first five rows of the resulting `Hr_data_Ignore_Tuple` DataFrame to show the dataset after removing rows with missing values.

In [ ]:
```
Hr_data_Ignore_Tuple = Hr_data.dropna(how='any')

Hr_data_Ignore_Tuple.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...
0	RM297	18	18-25	1	3	230	Research & Development	3	3	Life Sciences	...
1	RM302	18	18-25	0	3	812	Sales	10	3	Medical	...
2	RM458	18	18-25	1	2	1306	Sales	5	3	Marketing	...
3	RM728	18	18-25	0	0	287	Research & Development	5	2	Life Sciences	...
4	RM829	18	18-25	1	0	247	Research & Development	8	1	Medical	...

5 rows × 38 columns

This line of code prints the number of rows remaining in the `Hr_data_Ignore_Tuple` DataFrame after removing rows with missing values. It provides a count of data entries that have complete information, indicating how many rows were retained after the deletion.

In [ ]:
```
print(f"Number of rows after deleting tuples: {Hr_data_Ignore_Tuple.shape[0]}")
```

```
Number of rows after deleting tuples: 1423
```

B. Fill in the missing values manually

This line of code prints the number of rows in the `Hr_data_Manually_Enter` DataFrame, which remains unchanged from the original `Hr_data` DataFrame, as no rows were deleted. It confirms that the number of rows is the same after filling missing values with either the mode or mean.

In [ ]:
```
Hr_data_Manually_Enter = Hr_data.copy()
for column in Hr_data.columns:
    if Hr_data[column].dtype == 'object':
        # For categorical columns, fill with the mode (most frequent value)
        Hr_data_Manually_Enter[column].fillna(Hr_data[column].mode()[0], inplace=True)
    else:
        # For numerical columns, fill with the mean
        Hr_data_Manually_Enter[column].fillna(Hr_data[column].mean(), inplace=True)

Hr_data_Manually_Enter.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...	F
0	RM297	18	18-25	1	3	230	Research & Development	3	3	Life Sciences	...	
1	RM302	18	18-25	0	3	812	Sales	10	3	Medical	...	
2	RM458	18	18-25	1	2	1306	Sales	5	3	Marketing	...	
3	RM728	18	18-25	0	0	287	Research & Development	5	2	Life Sciences	...	
4	RM829	18	18-25	1	0	247	Research & Development	8	1	Medical	...	

5 rows × 38 columns

In [ ]:
```
print(f"Number of rows after deleting tuples: {Hr_data_Manually_Enter.shape[0]}")
```

```
Number of rows after deleting tuples: 1480
```

C. Use a global constant to fill in the missing value

This code snippet creates a copy of the `Hr_data` DataFrame named `Hr_data_global_constant` and fills missing values using global constants: `-1` for numerical columns and `'NONE'` for categorical columns. It then displays the first five rows of the updated DataFrame, showing how missing values have been replaced by the specified constants.

In [ ]:
```
# Define global constants
constant_numeric = -1
constant_string = 'NONE'

# Fill missing values using the global constant
Hr_data_global_constant = Hr_data.copy()

for column in Hr_data_global_constant.columns:
    if Hr_data_global_constant[column].dtype == 'object':
        # For categorical columns, fill with the global constant
        Hr_data_global_constant[column].fillna(constant_string, inplace=True)
    else:
        # For numerical columns, fill with the global constant
        Hr_data_global_constant[column].fillna(constant_numeric, inplace=True)

# Display the dataset with global constant filled values
Hr_data_global_constant.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...	F
0	RM297	18	18-25	1	3	230	Research & Development	3	3	Life Sciences	...	
1	RM302	18	18-25	0	3	812	Sales	10	3	Medical	...	
2	RM458	18	18-25	1	2	1306	Sales	5	3	Marketing	...	
3	RM728	18	18-25	0	0	287	Research & Development	5	2	Life Sciences	...	
4	RM829	18	18-25	1	0	247	Research & Development	8	1	Medical	...	

5 rows × 38 columns

This line of code prints the number of rows in the `Hr_data_global_constant` DataFrame after filling missing values with global constants. Since no rows were deleted, this count will be the same as the original number of rows in the dataset.

In [ ]:
```
# Print the number of rows after filling missing values with global constants
print(f"Number of rows after filling missing values with global constants: {Hr_data_global_constant.shape[0]}")
```

```
Number of rows after filling missing values with global constants: 1480
```

D. Use a measure of central tendency for the attribute (e.g., the mean or median)

This code snippet creates a copy of the `Hr_data` DataFrame named `Hr_data_central_tendency`. It then fills missing values using central tendency measures: the mode (most frequent value) for categorical columns and the mean for numerical columns. The `head()` method displays the first five rows of this updated DataFrame, showing how missing values have been replaced by these central tendency measures.

In [ ]:
```
# Use mean for numerical columns and mode for categorical columns
Hr_data_central_tendency = Hr_data.copy()
for column in Hr_data_central_tendency.columns:
    if Hr_data_central_tendency[column].dtype == 'object':
        # For categorical columns, fill with the mode (most frequent value)
        Hr_data_central_tendency[column].fillna(Hr_data_central_tendency[column].mode()[0], inplace=True)
    else:
        # For numerical columns, fill with the mean
        Hr_data_central_tendency[column].fillna(Hr_data_central_tendency[column].mean(), inplace=True)

# Display the dataset with central tendency filled values
Hr_data_central_tendency.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...	F
0	RM297	18	18-25	1	3	230	Research & Development	3	3	Life Sciences	...	
1	RM302	18	18-25	0	3	812	Sales	10	3	Medical	...	
2	RM458	18	18-25	1	2	1306	Sales	5	3	Marketing	...	
3	RM728	18	18-25	0	0	287	Research & Development	5	2	Life Sciences	...	
4	RM829	18	18-25	1	0	247	Research & Development	8	1	Medical	...	

5 rows × 38 columns

This line of code prints the number of rows in the `Hr_data_central_tendency` DataFrame after filling missing values with central tendency measures (mean for numerical columns and mode for categorical columns). Since no rows were deleted, this count will be the same as the original number of rows in the dataset.

In [ ]:
```
# Print the number of rows after filling missing values with central tendency
print(f"Number of rows after filling missing values with central tendency: {Hr_data_central_tendency.shape[0]}"
```

```
Number of rows after filling missing values with central tendency: 1480
```

E. Use the most probable value to fill in the missing value

This code snippet creates a copy of the `Hr_data` DataFrame named `Hr_data_most_probable`. It fills missing values in all columns with the mode (most frequent value) of each respective column. The `head()` method then displays the first five rows of the updated DataFrame, showing how missing values have been replaced by the most frequent value in each column.

In [ ]:
```
# Use mode for all columns
Hr_data_most_probable = Hr_data.copy()
for column in Hr_data_most_probable.columns:
    # Fill with the mode (most frequent value)
    Hr_data_most_probable[column].fillna(Hr_data_most_probable[column].mode()[0], inplace=True)

# Display the dataset with most probable value filled values
Hr_data_most_probable.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...	F
0	RM297	18	18-25	1	3	230	Research & Development	3	3	Life Sciences	...	
1	RM302	18	18-25	0	3	812	Sales	10	3	Medical	...	
2	RM458	18	18-25	1	2	1306	Sales	5	3	Marketing	...	
3	RM728	18	18-25	0	0	287	Research & Development	5	2	Life Sciences	...	
4	RM829	18	18-25	1	0	247	Research & Development	8	1	Medical	...	

5 rows × 38 columns

This line of code prints the number of rows in the `Hr_data_most_probable` DataFrame after filling missing values with the most probable value (mode) for each column. Since no rows were deleted, this count will be the same as the original number of rows in the dataset.

In [ ]:
```python
# Print the number of rows after filling missing values with the most probable value
print(f"Number of rows after filling missing values with the most probable value: {Hr_data_most_probable.shape[(
```

```
Number of rows after filling missing values with the most probable value: 1480
```

2. Binnig

A. Equal Frequency Binning

This code snippet creates a copy of the `Hr_data_central_tendency` DataFrame named `Hr_data_binnig_equal_frequency`. It then applies equal-frequency binning to specified numerical columns (`'Age'`, `'MonthlyIncome'`, and `'YearsAtCompany'`), dividing each column into 4 bins and creating new columns to store the binned values. Finally, it displays the first five rows of the DataFrame, focusing on the new columns that contain the equal-frequency binned data.

In [ ]:
```python
Hr_data_binnig_equal_frequency = Hr_data_central_tendency.copy()
def equal_frequency_binning(column, num_bins):
    return pd.qcut(column, q=num_bins, labels=False, duplicates='drop')

columns_to_bin = ['Age', 'MonthlyIncome', 'YearsAtCompany']
num_bins = 4

for column in columns_to_bin:
    Hr_data_binnig_equal_frequency[f'{column}_EqualFreqBinned'] = equal_frequency_binning(Hr_data_binnig_equal_

# Display the dataset with equal frequency binned values
Hr_data_binnig_equal_frequency[[col for col in Hr_data_binnig_equal_frequency.columns if 'EqualFreqBinned' in co
```

Out[ ]:

	Age_EqualFreqBinned	MonthlyIncome_EqualFreqBinned	YearsAtCompany_EqualFreqBinned
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

B. Equal Width Binning

This code snippet creates a copy of the `Hr_data_central_tendency` DataFrame named `Hr_data_binning_equal_width`. It then applies equal-width binning to specified numerical columns (`'Age'`, `'MonthlyIncome'`, and `'YearsAtCompany'`), dividing each column into 4 bins of equal width and creating new columns to store these binned values. Finally, it displays the first five rows of the DataFrame, focusing on the columns that contain the equal-width binned data.

In [ ]:
```python
Hr_data_binning_equal_width = Hr_data_central_tendency.copy()

def equal_width_binning(column, num_bins):
    return pd.cut(column, bins=num_bins, labels=False, duplicates='drop')

columns_to_bin = ['Age', 'MonthlyIncome', 'YearsAtCompany']
num_bins = 4

for column in columns_to_bin:
    Hr_data_binning_equal_width[f'{column}_Equal_Width_Binned'] = equal_width_binning(Hr_data_binning_equal_wid

# Display the dataset with equal width binned values
Hr_data_binning_equal_width[[col for col in Hr_data_binning_equal_width.columns if 'Equal_Width_Binned' in col]
```

Out[ ]:

	Age_Equal_Width_Binned	MonthlyIncome_Equal_Width_Binned	YearsAtCompany_Equal_Width_Binned
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

3. Normalization

Min-Max Normalization

$$x_i = \frac{x_i - \min(X)}{\max(X) - \min(X)}$$

This code snippet creates a copy of the `Hr_data_central_tendency` DataFrame named `Hr_data_min_max`. It then applies Min-Max normalization to all numerical columns, scaling their values to a range between 0 and 1 by subtracting the minimum value and dividing by the range (max - min). Finally, it displays the first five rows of the normalized DataFrame.

In [ ]:
```
# Perform Min-Max Normalization
Hr_data_min_max = Hr_data_central_tendency.copy()
for column in Hr_data_min_max.select_dtypes(include=['float64', 'int64']):
    Hr_data_min_max[column] = (Hr_data_min_max[column] - Hr_data_min_max[column].min()) / (Hr_data_min_max[colu

# Display the dataset with min-max normalization
Hr_data_min_max.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...	F
0	RM297	0.0	18-25	1	3	0.091625	Research & Development	0.071429	0.50	Life Sciences	...	
1	RM302	0.0	18-25	0	3	0.508232	Sales	0.321429	0.50	Medical	...	
2	RM458	0.0	18-25	1	2	0.861847	Sales	0.142857	0.50	Marketing	...	
3	RM728	0.0	18-25	0	0	0.132427	Research & Development	0.142857	0.25	Life Sciences	...	
4	RM829	0.0	18-25	1	0	0.103794	Research & Development	0.250000	0.00	Medical	...	

5 rows × 38 columns

Z-Score Normalization

$$x_i = \frac{x_i - \mu}{\sigma}$$

This code snippet creates a copy of the `Hr_data_central_tendency` DataFrame named `Hr_data_z_score`. It then applies Z-Score normalization to all numerical columns, standardizing their values by subtracting the mean and dividing by the standard deviation. This transforms the data to have a mean of 0 and a standard deviation of 1. Finally, it displays the first five rows of the standardized DataFrame.

In [ ]:
```
# Perform Z-Score Normalization
Hr_data_z_score = Hr_data_central_tendency.copy()
for column in Hr_data_z_score.select_dtypes(include=['float64', 'int64']):
    Hr_data_z_score[column] = (Hr_data_z_score[column] - Hr_data_z_score[column].mean()) / Hr_data_z_score[colu

# Display the dataset with z-score normalization
Hr_data_z_score.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField
0	RM297	-2.07235	18-25	1	3	-1.417381	Research & Development	-0.764988	0.087116	Life Sciences
1	RM302	-2.07235	18-25	0	3	0.026333	Sales	0.095894	0.087116	Medical
2	RM458	-2.07235	18-25	1	2	1.251753	Sales	-0.519022	0.087116	Marketing
3	RM728	-2.07235	18-25	0	0	-1.275986	Research & Development	-0.519022	-0.889641	Life Sciences
4	RM829	-2.07235	18-25	1	0	-1.375210	Research & Development	-0.150073	-1.866397	Medical

5 rows × 38 columns

Decimal Scalling

$$x_i = \frac{x_i}{10^j}$$

This code snippet creates a copy of the `Hr_data_central_tendency` DataFrame named `Hr_data_decimal_scaling`. It then applies decimal scaling to all numerical columns by dividing each value by a power of 10 that scales the maximum absolute value to a range between 0 and 1. Finally, it displays the first five rows of the DataFrame with the decimal-scaled values.

In [ ]:
```
Hr_data_decimal_scaling = Hr_data_central_tendency.copy()
for column in Hr_data_decimal_scaling.select_dtypes(include=['float64', 'int64']):
    Hr_data_decimal_scaling[column] = Hr_data_decimal_scaling[column] / 10**np.ceil(np.log10(Hr_data_decimal_sca

# Display the dataset with decimal scaling
```

```
Hr_data_decimal_scaling.head()
```

Out[ ]:

	EmpID	Age	AgeGroup	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	...
0	RM297	0.18	18-25	1	3	0.0230	Research & Development	0.03	0.3	Life Sciences	...
1	RM302	0.18	18-25	0	3	0.0812	Sales	0.10	0.3	Medical	...
2	RM458	0.18	18-25	1	2	0.1306	Sales	0.05	0.3	Marketing	...
3	RM728	0.18	18-25	0	0	0.0287	Research & Development	0.05	0.2	Life Sciences	...
4	RM829	0.18	18-25	1	0	0.0247	Research & Development	0.08	0.1	Medical	...

5 rows × 38 columns

4. Visualization

The `%matplotlib inline` magic command is used in Jupyter notebooks to enable the inline display of Matplotlib plots. This means that plots will be displayed directly below the code cells that generate them, making it easier to view and analyze visualizations within the notebook.

In [ ]:

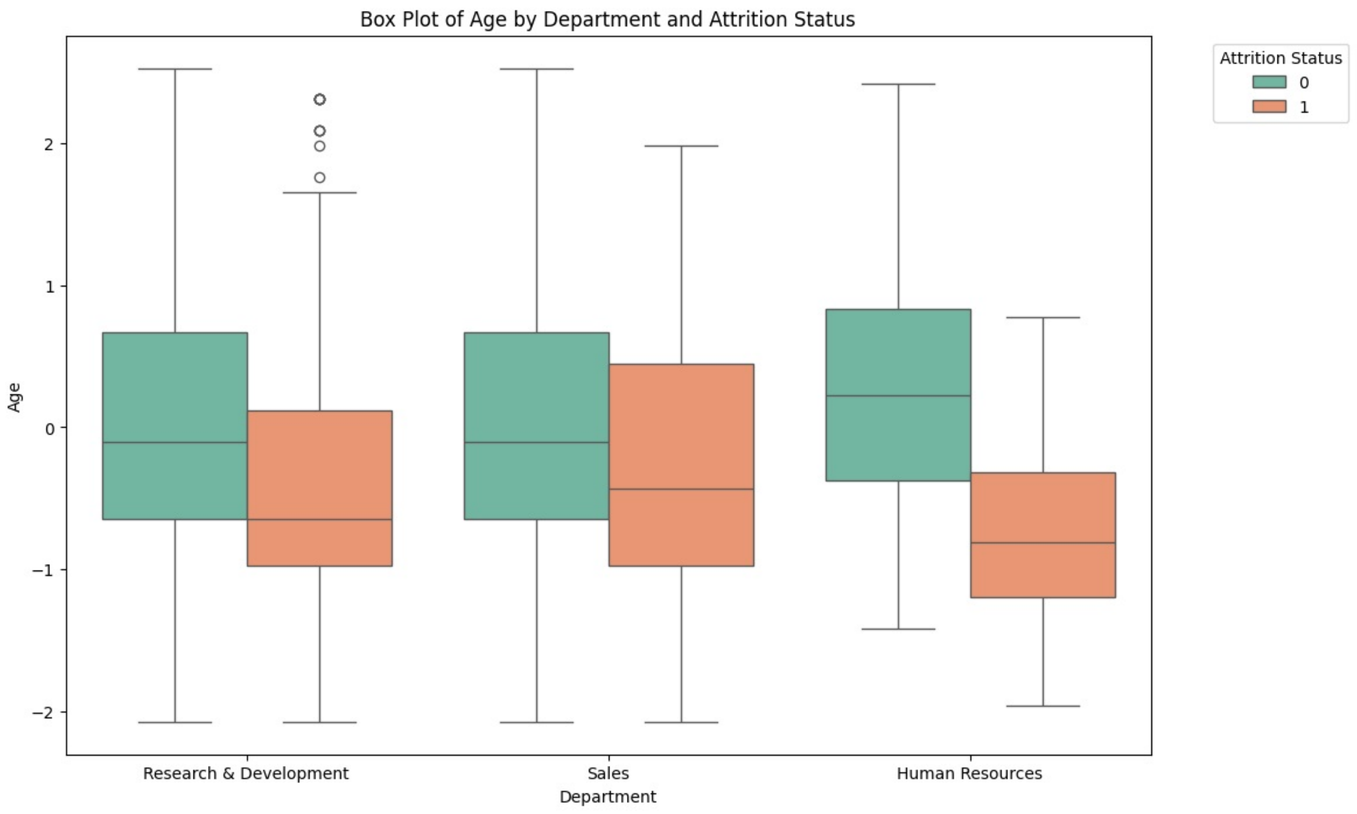
```
%matplotlib inline
```

A. a. Box Plot

This code creates a box plot using Seaborn to visualize the distribution of `Age` across different `Department` categories, with color-coding based on `Attrition` status. The plot is sized to 12x8 inches and includes a legend placed outside the plot area to the right. The title and axis labels provide context for the plot, showing how `Age` varies by `Department` and whether `Attrition` status affects this distribution.

In [ ]:

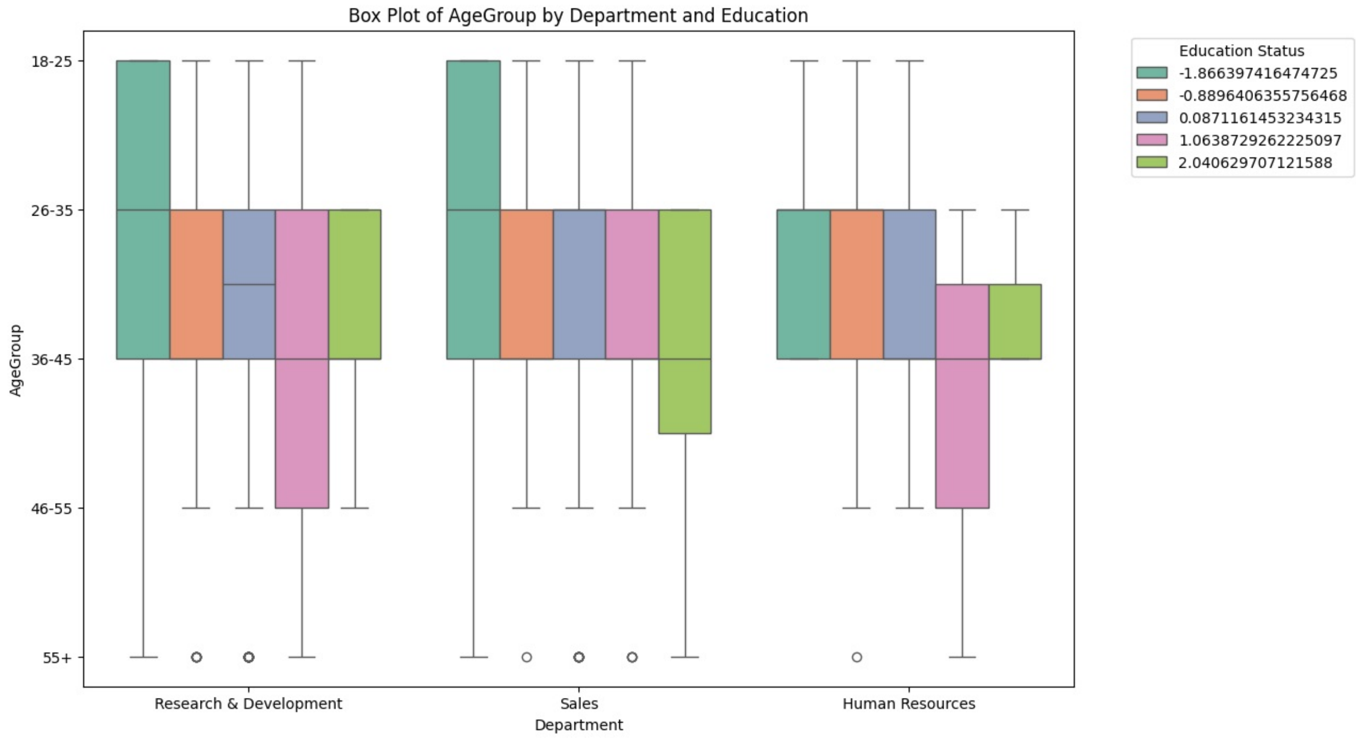
```
plt.figure(figsize=(12, 8))
sns.boxplot(data=Hr_data_z_score, x='Department', y='Age', hue='Attrition', palette='Set2')
plt.title('Box Plot of Age by Department and Attrition Status')
plt.xlabel('Department')
plt.ylabel('Age')
plt.legend(title='Attrition Status', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



A. b. Box Plot

This code creates a box plot to visualize the distribution of `AgeGroup` across different `Department` s, colored by `Education` levels. The plot is generated using Seaborn with a specified figure size and color palette.

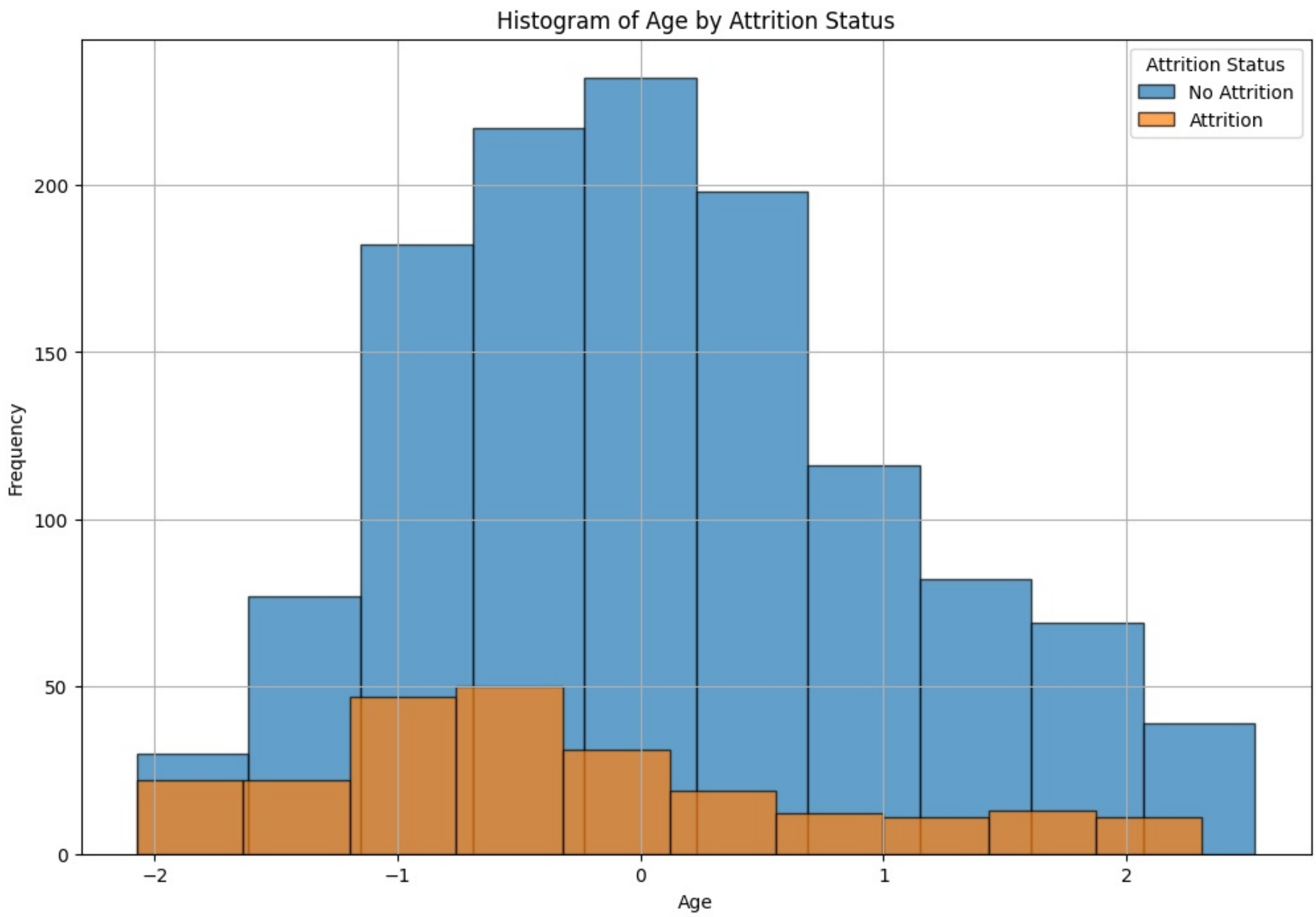
In [ ]:
```
plt.figure(figsize=(12, 8))
sns.boxplot(data=Hr_data_z_score, x='Department', y='AgeGroup', hue='Education', palette='Set2')
plt.title('Box Plot of AgeGroup by Department and Education')
plt.xlabel('Department')
plt.ylabel('AgeGroup')
plt.legend(title='Education Status', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.show()
```



B. a. Histogram

This code snippet creates a histogram to compare the distribution of `Age` between employees with `Attrition` (status 1) and without `Attrition` (status 0). The histogram is plotted with 10 bins, edge colors for clarity, and semi-transparent bars. The plot includes a title, axis labels, and a legend to differentiate between the two attrition statuses. The grid is enabled for better readability of the plot.

In [ ]:
```
plt.figure(figsize=(12, 8))
Hr_data_z_score[Hr_data_z_score['Attrition'] == 0]['Age'].plot(kind='hist', bins=10, edgecolor='black', alpha=0
Hr_data_z_score[Hr_data_z_score['Attrition'] == 1]['Age'].plot(kind='hist', bins=10, edgecolor='black', alpha=0
plt.title('Histogram of Age by Attrition Status')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.legend(title='Attrition Status')
plt.grid(True)
plt.show()
```

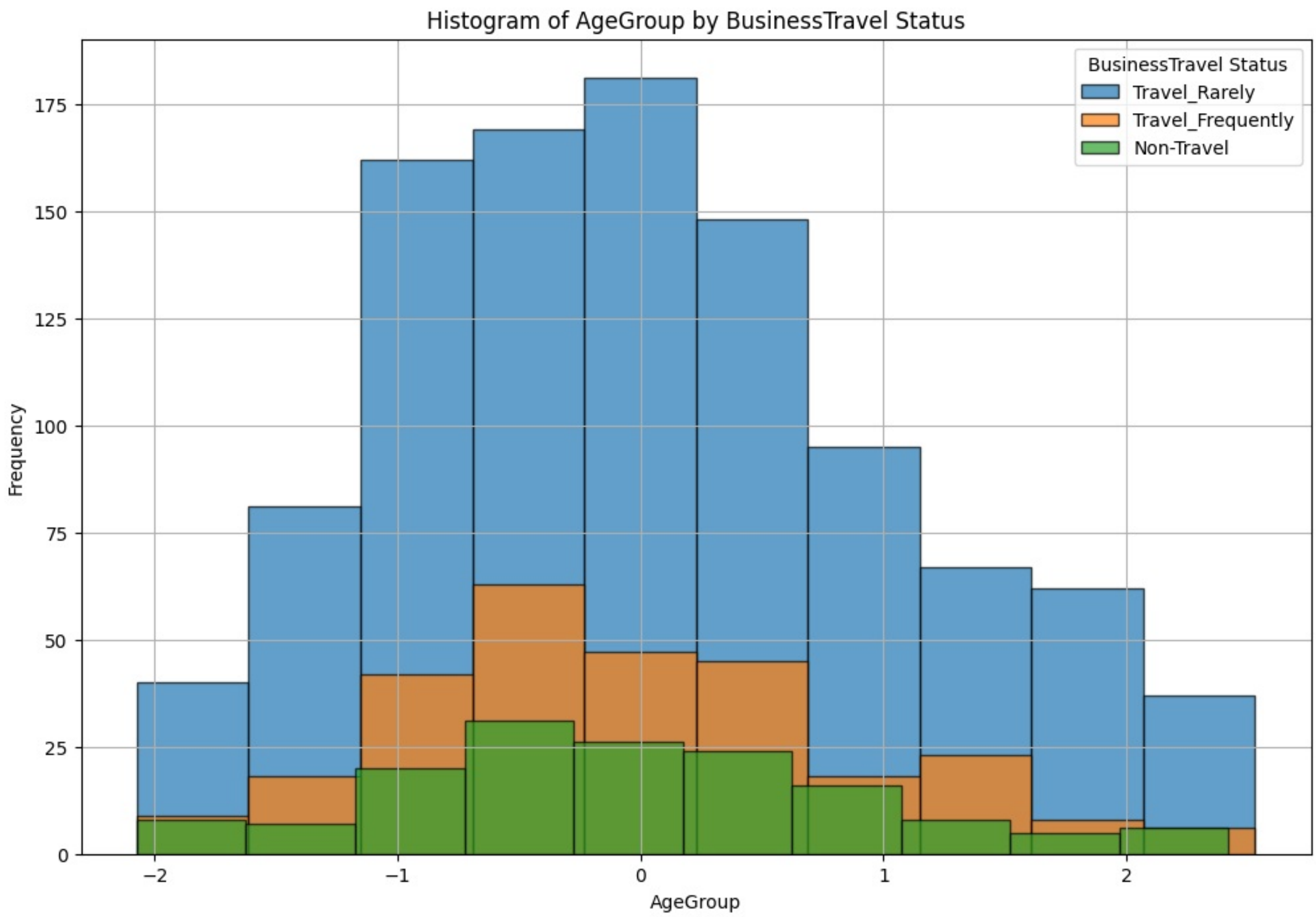


B. b. Histogram

This code creates histograms of `AgeGroup` for employees based on their `BusinessTravel` status: traveling rarely, frequently, or not at all. It plots these distributions separately to compare the age groups within different travel categories. The plot includes labels, a legend, and a grid for better visualization.

In [ ]:

```
plt.figure(figsize=(12, 8))
Hr_data_z_score[Hr_data_z_score['BusinessTravel'] == 3]['Age'].plot(kind='hist', bins=10, edgecolor='black', al
Hr_data_z_score[Hr_data_z_score['BusinessTravel'] == 2]['Age'].plot(kind='hist', bins=10, edgecolor='black', al
Hr_data_z_score[Hr_data_z_score['BusinessTravel'] == 0]['Age'].plot(kind='hist', bins=10, edgecolor='black', al
plt.title('Histogram of AgeGroup by BusinessTravel Status')
plt.xlabel('AgeGroup')
plt.ylabel('Frequency')
plt.legend(title='BusinessTravel Status')
plt.grid(True)
plt.show()
```

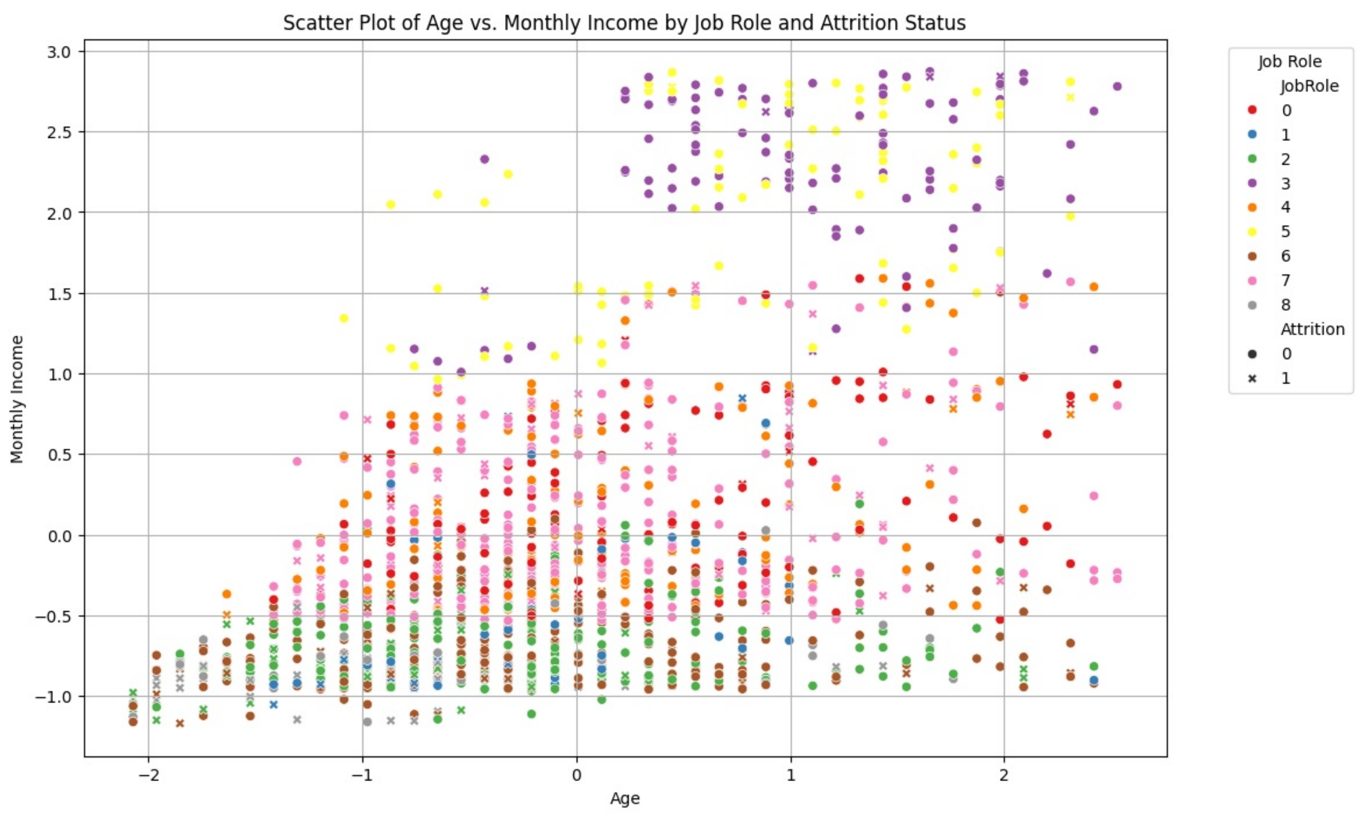


C. a. Scatter Plot

This code generates a scatter plot using Seaborn to visualize the relationship between `Age` and `MonthlyIncome`. Points are colored by `JobRole` and styled by `Attrition` status. The plot is sized to 12x8 inches and includes a title, axis labels, and a legend positioned outside the plot area to the right. Grid lines are enabled to improve readability.

In [ ]:

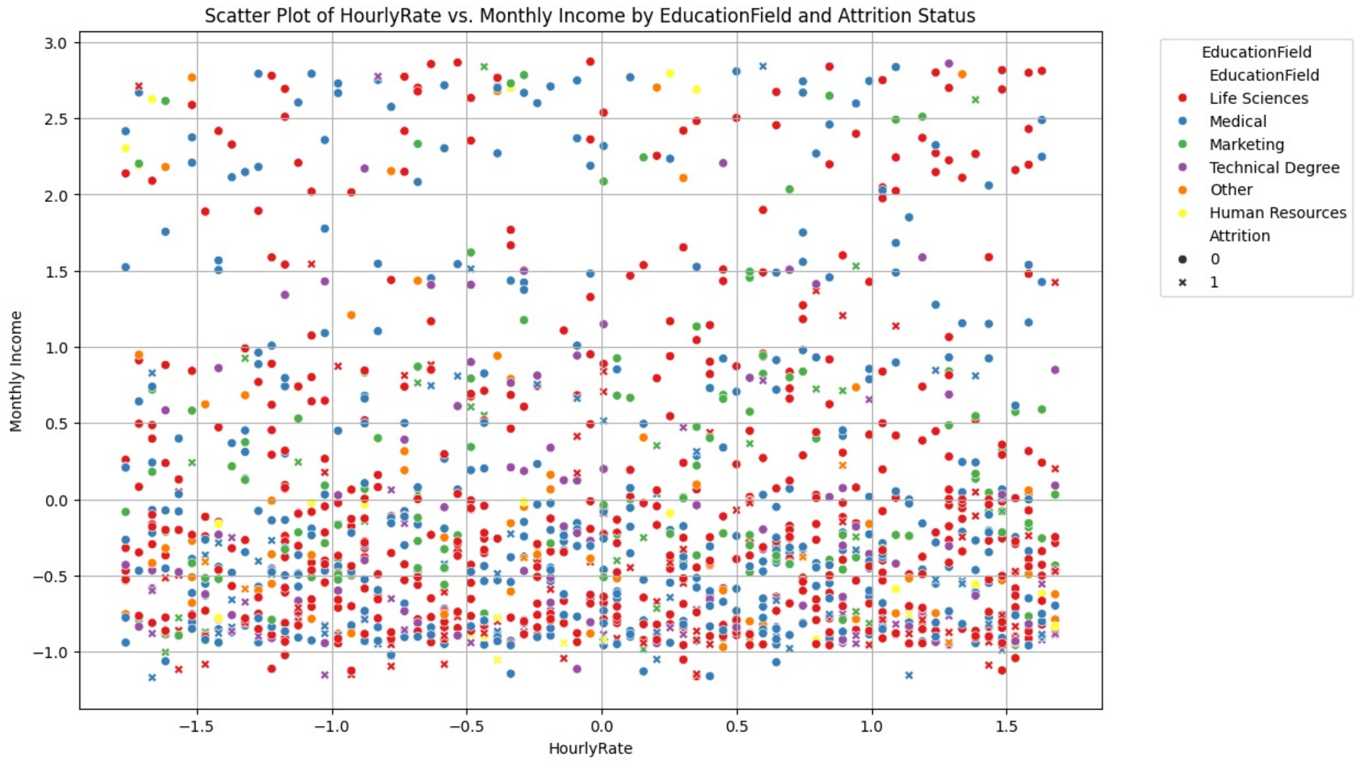
```
plt.figure(figsize=(12, 8))
sns.scatterplot(data=Hr_data_z_score, x='Age', y='MonthlyIncome', hue='JobRole', style='Attrition', palette='Se
plt.title('Scatter Plot of Age vs. Monthly Income by Job Role and Attrition Status')
plt.xlabel('Age')
plt.ylabel('Monthly Income')
plt.legend(title='Job Role', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.show()
```



C. b. Scatter Plot

This code creates a scatter plot to visualize the relationship between `HourlyRate` and `MonthlyIncome`, with points colored by `EducationField` and styled by `Attrition` status. The plot includes a legend, a title, and a grid for better clarity.

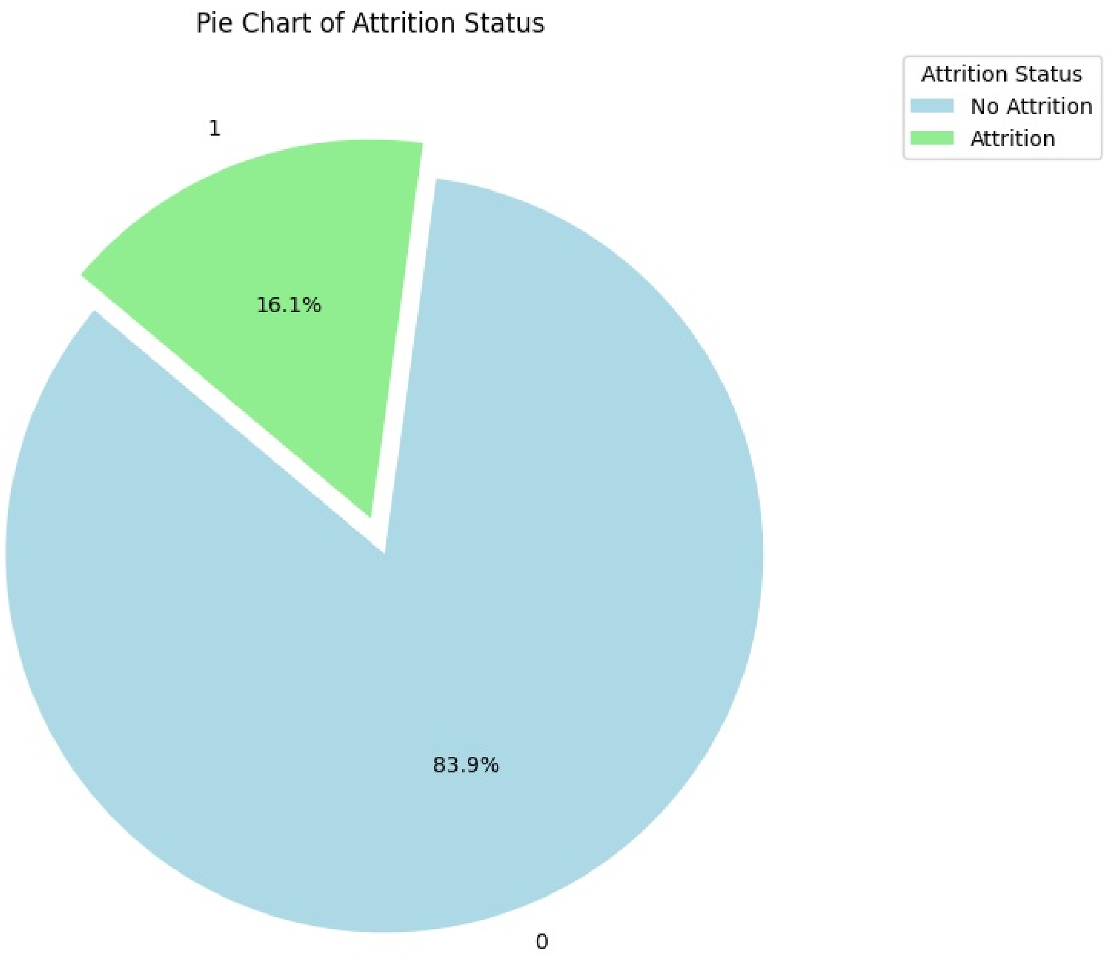
In [ ]:
```
plt.figure(figsize=(12, 8))
sns.scatterplot(data=Hr_data_z_score, x='HourlyRate', y='MonthlyIncome', hue='EducationField', style='Attrition
plt.title('Scatter Plot of HourlyRate vs. Monthly Income by EducationField and Attrition Status')
plt.xlabel('HourlyRate')
plt.ylabel('Monthly Income')
plt.legend(title='EducationField', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.show()
```



D. a. Pie Chart

This code creates a pie chart to display the distribution of `Attrition` status in the `Hr_data` DataFrame. It uses different colors to represent each status and includes percentage labels on the slices. The pie chart is set with a start angle of 140 degrees and an exploded slice for the 'No Attrition' category for emphasis. The plot is sized to 8x8 inches, and a legend is placed outside the plot area to the right, labeling the statuses as 'No Attrition' and 'Attrition.'

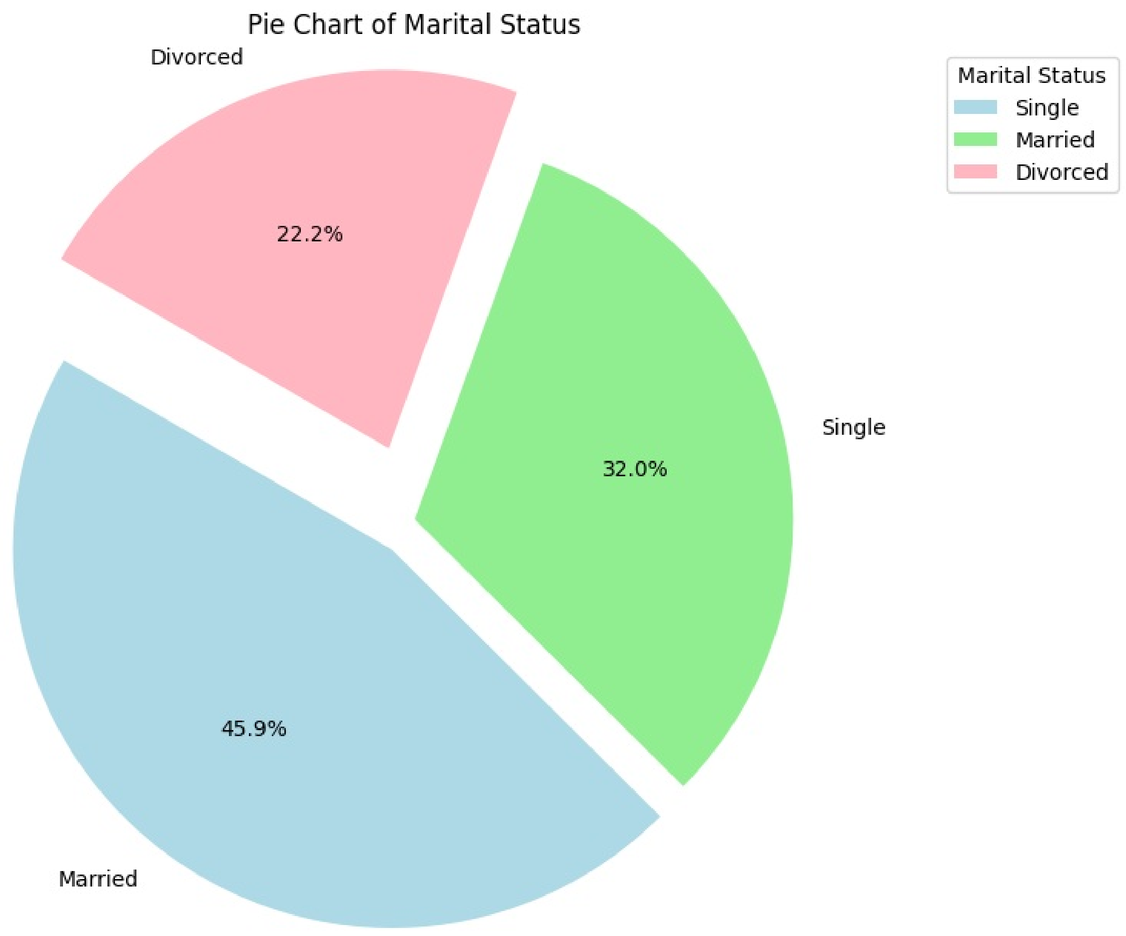
In [ ]:
```
plt.figure(figsize=(8, 8))
attrition_counts = Hr_data['Attrition'].value_counts()
plt.pie(attrition_counts, labels=attrition_counts.index, autopct='%1.1f%%', startangle=140, colors=['lightblue'
plt.title('Pie Chart of Attrition Status')
plt.legend(title='Attrition Status', labels=['No Attrition', 'Attrition'], bbox_to_anchor=(1.05, 1), loc='upper
plt.show()
```



D. b. Pie Chart

This code creates a pie chart to display the distribution of `MaritalStatus` among employees. It uses different colors and an explode effect to highlight the 'Single' and 'Divorced' categories. The plot includes a legend, labels, and a title for better interpretation.

In [ ]:
```
plt.figure(figsize=(8, 8))
MaritalStatus_counts = Hr_data['MaritalStatus'].value_counts()
plt.pie(MaritalStatus_counts, labels=MaritalStatus_counts.index, autopct='%1.1f%%', startangle=150, colors=['li
plt.title('Pie Chart of Marital Status')
plt.legend(title='Marital Status', labels=['Single', 'Married', 'Divorced'], bbox_to_anchor=(1.05, 1), loc='upp
plt.show()
```


Pie Chart of Marital Status
Divorced
22.2%
Single
32.0%
Married
45.9%
Marital Status
Single
Married
Divorced