EMPLOYEE RETENTION ANALYSIS USING MACHINE LEARNING

ABSTRACT

Employee attrition, a critical concern in the realm of Human Resources, directly impacts the stability and productivity of an organization. This project focuses on leveraging machine learning techniques, specifically Logistic Regression, to proactively predict potential employee attrition. The project aims to provide actionable insights for HR departments to implement effective retention strategies by analyzing key factors that influence employee satisfaction and engagement.

The project begins by exploring and understanding the dataset containing historical employee data, which includes attributes such as performance metrics, job satisfaction scores, years of experience, and more. Utilizing the Python programming language and popular libraries like Sklearn, Numpy, Pandas, and Matplotlib, the project establishes a robust data preprocessing pipeline to handle missing values, categorical variables, and data normalization. Exploratory Data Analysis (EDA) techniques are employed to uncover trends and patterns within the dataset, shedding light on potential indicators of attrition.

Logistic Regression, a well-suited classification algorithm for binary outcomes, is employed to build a predictive model. Through a systematic process of feature selection and model tuning, the algorithm learns to identify relevant features and relationships between the provided employee attributes and the likelihood of attrition. The trained model is then rigorously evaluated using accuracy, precision, recall, and F1-score metrics to ensure its reliability and generalizability.

Furthermore, the project incorporates a visual component, generating insightful graphs and visualizations using Matplotlib. These visuals aid in presenting the findings to stakeholders in a comprehensible manner, enabling HR teams to grasp the significance of different factors contributing to attrition.

The outcome of this project is a user-friendly predictive tool capable of identifying potential attrition cases within the organization. This tool empowers HR departments with a proactive approach to address employee dissatisfaction, enabling timely intervention and targeted retention strategies. Organizations can curate a more engaging work environment and foster long-term employee commitment by understanding the significant factors that contribute to attrition.

In summary, the "Employee Retention Analysis using Machine Learning" project merges data science and HR expertise to tackle the critical challenge of employee attrition. Through the utilization of Logistic Regression, exploratory data analysis, and a comprehensive dataset, the project delivers a valuable tool that assists organizations in maintaining a satisfied and motivated workforce, ultimately contributing to increased productivity and growth.

TABLE OF CONTENTS

1. INTRODUCTION

Employee attrition, the phenomenon of employees leaving a company, is a critical concern for organizations across industries. The ability to understand and predict employee attrition is of utmost importance as it directly impacts a company's productivity, morale, and overall success. The field of Human Resources (HR) has long been striving to develop strategies and measures to address this challenge effectively. In the modern business landscape, where data-driven decision-making has become a cornerstone, leveraging machine learning techniques to analyze and predict employee attrition has emerged as a promising solution.

Our project, titled "Employee Retention Analysis using Machine Learning," delves into this crucial area by employing advanced data analysis techniques to uncover insights and patterns within employee data. By harnessing the power of machine learning algorithms, we aim to build predictive models that can identify potential employees on the brink of leaving the company. This proactive approach can provide organizations with the means to implement targeted retention strategies, foster employee satisfaction, and ultimately enhance organizational stability and growth.

In this project, we will employ Logistic Regression and Random Forest classifiers to predict attrition risk based on a variety of employee-related features. Our analysis will involve preprocessing the data to ensure its quality, exploring the data through visualizations to discover trends, and building models that can accurately predict employee attrition. By assessing model performance and presenting comprehensive evaluation metrics, we intend to provide organizations with actionable insights that can guide effective decision-making.

The rest of this report is organized as follows. We begin with a discussion on the methodology, outlining the data preprocessing steps, exploratory data analysis, and model-building techniques employed. Subsequently, we present the results of our analysis, showcasing the predictive capabilities of the developed models. We conclude by summarizing our findings, highlighting the significance of our approach in the context of employee retention, and suggesting potential avenues for further research and application.

Through this project, we seek to contribute to the ongoing efforts to manage employee attrition effectively and empower organizations to create a workplace environment that nurtures and retains its valuable human resources.

2.OVERVIEW

The purpose of the project is to analyze employee retention factors and compare the performance of two different machine learning models in predicting whether an employee will leave the organization ('Attrition' label) based on various features. It provides insights into feature importance and model accuracy, allowing for a data-driven approach to understand and mitigate employee turnover.

Its primary goal is to analyze employee data to gain insights into factors that contribute to employee attrition (employees leaving the organization) and to build predictive models that can classify whether an employee is likely to leave or stay based on various features. It is divided into several sections, each with a specific purpose.

The code begins by importing necessary libraries such as pandas, numpy, seaborn, and matplotlib.pyplot. These libraries are used for data manipulation, visualization, and analysis.The code loads employee data from a CSV file using pandas (pd.read_csv).Certain columns like 'Attrition', 'Over18', and 'OverTime' are converted into binary values.Columns 'EmployeeCount', 'EmployeeNumber', 'Over18', and 'StandardHours' are dropped as they likely do not contribute to the analysis.

By loading and preprocessing employee data, the code helps us to understand the characteristics of employees who have left the organization (attrited) compared to those who have stayed. It visualizes and explores various features to identify potential trends and patterns related to attrition.

The code includes visualization techniques such as histograms, count plots, and heatmaps to provide visual insights into the data. These visualizations help us to identify relationships between different features and attrition, helping you understand which factors might contribute to employee turnover. Count plots and heatmaps are used to explore the relationships between different features and employee attrition.

Categorical features are one-hot encoded, and numerical features are scaled using MinMaxScaler to ensure all features are on the same scale.

The code trains two types of machine learning models: Logistic Regression and Random Forest Classifier. These models learn from the provided data and are used to predict whether an employee is likely to leave the organization based on their features.

A Logistic Regression model is created and trained on the training data using model.fit.Predictions are made on the test data using the trained model (model.predict).The accuracy of the model is calculated using accuracy_score.A confusion matrix and classification report are generated and displayed using seaborn's heatmap and classification_report.Similar steps are repeated for a Random Forest Classifier.The model is trained, predictions are made, and the classification report is generated.

The code evaluates the performance of the trained models using metrics like accuracy, precision, recall, and F1-score. The confusion matrix and classification report provides a detailed breakdown of the model's predictions and their quality.

Both models provide insights into the importance of different features in predicting attrition. This analysis can help us to understand which factors have the most significant impact on whether an employee stays or leaves.

The results and insights gained from this analysis can be used by HR departments and organizational leaders to make informed decisions about employee retention strategies. By understanding the key factors that influence attrition, companies can implement targeted interventions to improve employee satisfaction and reduce turnover.

This code can be considered a starting point for ongoing employee retention analysis. As new data becomes available or circumstances change, we can revisit the analysis, fine-tune the models, and adjust retention strategies accordingly.

In summary, the code allows organizations to take a data-driven approach to employee retention. By leveraging machine learning and data analysis techniques, it helps businesses gain insights, optimize strategies, and make informed decisions to retain valuable employees and create a more stable workforce.

3.MODELS

Logistic Regression:

Logistic Regression is a popular statistical model used for binary classification, that is for predictions of the type this or that, yes or no, A or B, etc. Logistic regression can, however, be used for multiclass classification. This type of statistical model (also known as *logit model*) is often used for classification and predictive analytics. Logistic regression estimates the probability of an event occurring, such as voted or not voted, based on a given dataset of independent variables. Since the outcome is a probability, the dependent variable is bounded between 0 and 1. In logistic regression, a logit transformation is applied to the odds—that is, the probability of success divided by the probability of failure. This is also commonly known as the log odds or the natural logarithm of odds, and this logistic function is represented by the following formulas:

Logit(pi) = 1/(1+ exp(-pi))

ln(pi/(1-pi)) = Beta_0 + Beta_1*X_1 + … + B_k*K_k

Logistic Regression working:

In this logistic regression equation, logit(pi) is the dependent or response variable and x is the independent variable. The beta parameter, or coefficient, in this model is commonly estimated via maximum likelihood estimation (MLE). This method tests different values of beta through multiple iterations to optimize for the best fit of log odds. All of these iterations produce the log likelihood function, and logistic regression seeks to maximize this function to find the best parameter estimate. Once the optimal coefficient (or coefficients if there is more than one independent variable) is found, the conditional probabilities for each observation can be calculated, logged, and summed together to yield a predicted probability. For binary classification, a probability less than .5 will predict 0 while a probability greater than 0 will predict 1. After the model has been computed, it's best practice to evaluate how well the model predicts the dependent variable, which is called goodness of fit.

Random Forest Classifier:

The random forest classifier is a supervised learning algorithm that you can use for regression and classification problems. It is among the most popular machine learning algorithms due to its high flexibility and ease of implementation.

Why is the random forest classifier called the random forest?

That's because it consists of multiple decision trees just as a forest has many trees. On top of that, it uses randomness to enhance its accuracy and combat overfitting, which can be a huge issue for such a sophisticated algorithm. These algorithms make decision trees based on a random selection of data samples and get predictions from every tree. After that, they select the best viable solution through votes.

It has numerous applications in our daily lives such as feature selectors, recommender systems, and image classifiers. Some of its real-life applications include fraud detection, classification of loan applications, and disease prediction.

How does it work?

Assuming the dataset has "m" features, the random forest will randomly choose "k" features where k < m. Now, the algorithm will calculate the root node among the k features by picking a node that has the highest information gain.

After that, the algorithm splits the node into child nodes and repeats this process "n" times. Now you have a forest with n trees. Finally, you'll perform bootstrapping, ie, combine the results of all the decision trees present in your forest.

It is certainly one of the most sophisticated algorithms as it builds on the functionality of decision trees.

Technically, it is an ensemble algorithm. The algorithm generates the individual decision trees through an attribute selection indication. Every tree relies on an independent random sample. In a classification problem, every tree votes and the most popular class is the end result. On the other hand, in a regression problem, you'll compute the average of all the tree outputs and that would be your end result.

A random forest Python implementation is much simpler and more robust than other non-linear algorithms used for classification problems.

4. DATA PREPROCESSING

4.1. Importing Libraries:

Importing essential libraries for data analysis and visualization in Python is the first step. We have imported some libraries such as Pandas, NumPy, Seaborn and Matplotlib.

Code -

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

Pandas: Pandas is a powerful library for data manipulation and analysis. It provides data structures like DataFrame, which allows to store and manipulate tabular data efficiently. With Pandas, you can load data from various sources, clean and preprocess it, perform transformations, and more.

NumPy: NumPy is a fundamental library for numerical computations in Python. It provides support for large, multi-dimensional arrays and matrices, along with various mathematical functions to operate on these arrays. NumPy is often used in conjunction with Pandas to handle and manipulate numerical data efficiently.We can also perform statistical operations.

Seaborn: Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for creating attractive and informative statistical graphics. Seaborn simplifies the process of creating various types of plots, such as scatter plots, bar plots, histograms, and more, with minimal code. We can create categorical plots to analyze the distribution of data across categories. We can even customize the appearance of plots using color palettes and themes.

Matplotlib: Matplotlib is a widely used plotting library in Python. It provides a versatile way to create a wide range of static, interactive, and animated visualizations. While Matplotlib can be a bit low-level for some tasks, it offers great customization options for creating publication-quality plots.

4.2. Data Analyzing:

Loading Data:

This involves loading a CSV file named ‘ EmployeeRetention.csv’ into a pandas DataFrame called employee_df and then performing initial data analysis using the DataFrame.

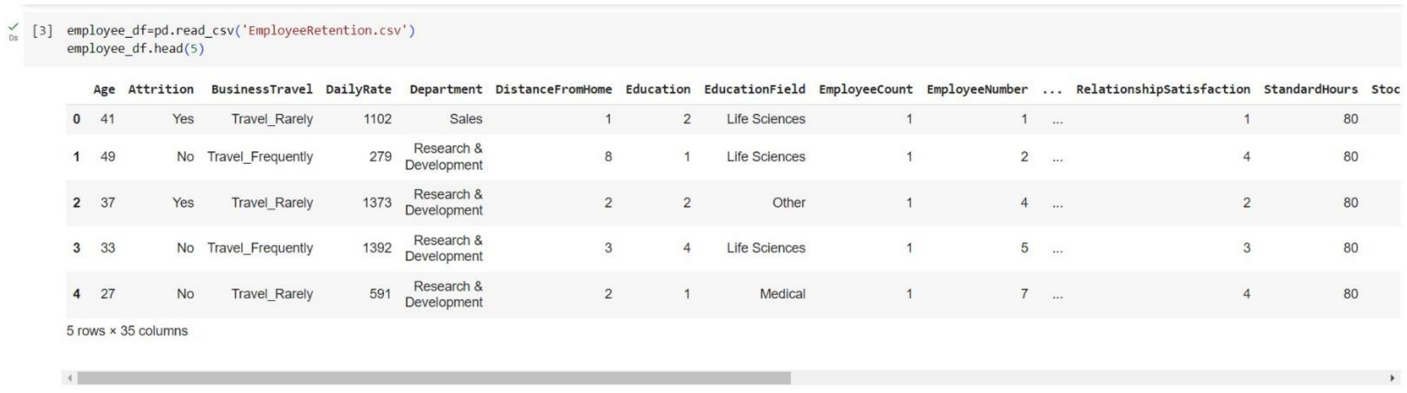
```
➤ employee_df=pd.read_csv('EmployeeRetention.csv')
```

CSV stands for Comma Separated File. This CSV file contains information about employees and their retention status.

Displaying Data:

```
➤ employee_df.head(5)
```

This line of code uses the.head() method to display the first five rows of the DataFrame employee_df. This provides a quick glance at the structure and contents of the dataset.



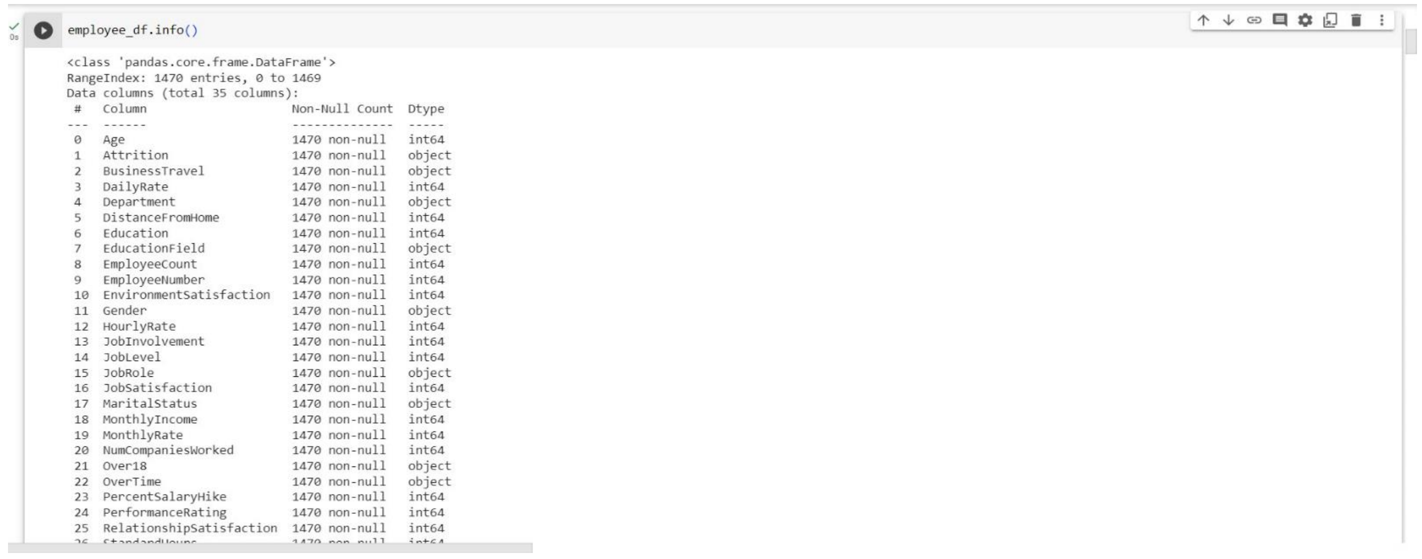
```
[3] employee_df=pd.read_csv('EmployeeRetention.csv')
    employee_df.head(5)
```

```
   Age  Attrition  BusinessTravel     DailyRate  Department     DistanceFromHome  Education  EducationField  EmployeeCount  EmployeeNumber  ...  RelationshipSatisfaction  StandardHours  Stoc
0  41         Yes  Travel_Rarely           1102  Sales                         1          2  Life Sciences               1               1  ...                         1             80
1  49          No  Travel_Frequently        279  Research &                    8          1  Life Sciences               1               2  ...                         4             80
                                                 Development
2  37         Yes  Travel_Rarely           1373  Research &                    2          2  Other                       1               4  ...                         2             80
                                                 Development
3  33          No  Travel_Frequently       1392  Research &                    3          4  Life Sciences               1               5  ...                         3             80
                                                 Development
4  27          No  Travel_Rarely            591  Research &                    2          1  Medical                     1               7  ...                         4             80
                                                 Development
5 rows × 35 columns
```

Data Information:

```
➤ employee_df.info()
```

This line uses the .info() method to display essential information about the DataFrame employee_df. It shows the number of non-null values in each column, the data type of each column, and the memory usage. This information helps you understand the data's integrity and potential data cleaning needs.



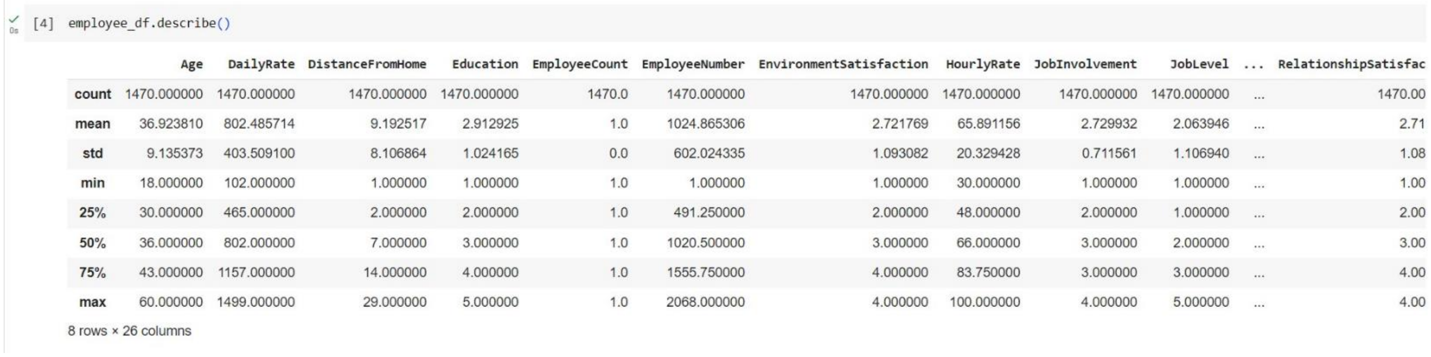
```
employee_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1470 entries, 0 to 1469
Data columns (total 35 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Age                       1470 non-null   int64
 1   Attrition                 1470 non-null   object
 2   BusinessTravel            1470 non-null   object
 3   DailyRate                 1470 non-null   int64
 4   Department                1470 non-null   object
 5   DistanceFromHome          1470 non-null   int64
 6   Education                 1470 non-null   int64
 7   EducationField            1470 non-null   object
 8   EmployeeCount             1470 non-null   int64
 9   EmployeeNumber            1470 non-null   int64
 10  EnvironmentSatisfaction   1470 non-null   int64
 11  Gender                    1470 non-null   object
 12  HourlyRate                1470 non-null   int64
 13  JobInvolvement            1470 non-null   int64
 14  JobLevel                  1470 non-null   int64
 15  JobRole                   1470 non-null   object
 16  JobSatisfaction           1470 non-null   int64
 17  MaritalStatus             1470 non-null   object
 18  MonthlyIncome             1470 non-null   int64
 19  MonthlyRate               1470 non-null   int64
 20  NumCompaniesWorked        1470 non-null   int64
 21  Over18                    1470 non-null   object
 22  OverTime                  1470 non-null   object
 23  PercentSalaryHike         1470 non-null   int64
 24  PerformanceRating         1470 non-null   int64
 25  RelationshipSatisfaction  1470 non-null   int64
```

Summary Statistics:

➢ `employee_df.describe()`

The .describe() method to generate summary statistics for numeric columns in the DataFrame. This includes statistics like mean, standard deviation, percentiles, and more. These statistics offer insights into the distribution and central tendencies of the numeric data.



```
[4] employee_df.describe()
```

	Age	DailyRate	DistanceFromHome	Education	EmployeeCount	EmployeeNumber	EnvironmentSatisfaction	HourlyRate	JobInvolvement	JobLevel	...	RelationshipSatisfac
count	1470.000000	1470.000000	1470.000000	1470.000000	1470.0	1470.000000	1470.000000	1470.000000	1470.000000	1470.000000	...	1470.00
mean	36.923810	802.485714	9.192517	2.912925	1.0	1024.865306	2.721769	65.891156	2.729932	2.063946	...	2.71
std	9.135373	403.509100	8.106864	1.024165	0.0	602.024335	1.093082	20.329428	0.711561	1.106940	...	1.08
min	18.000000	102.000000	1.000000	1.000000	1.0	1.000000	1.000000	30.000000	1.000000	1.000000	...	1.00
25%	30.000000	465.000000	2.000000	2.000000	1.0	491.250000	2.000000	48.000000	2.000000	1.000000	...	2.00
50%	36.000000	802.000000	7.000000	3.000000	1.0	1020.500000	3.000000	66.000000	3.000000	2.000000	...	3.00
75%	43.000000	1157.000000	14.000000	4.000000	1.0	1555.750000	4.000000	83.750000	3.000000	3.000000	...	4.00
max	60.000000	1499.000000	29.000000	5.000000	1.0	2068.000000	4.000000	100.000000	4.000000	5.000000	...	4.00

8 rows × 26 columns

4.3. Data Encoding:

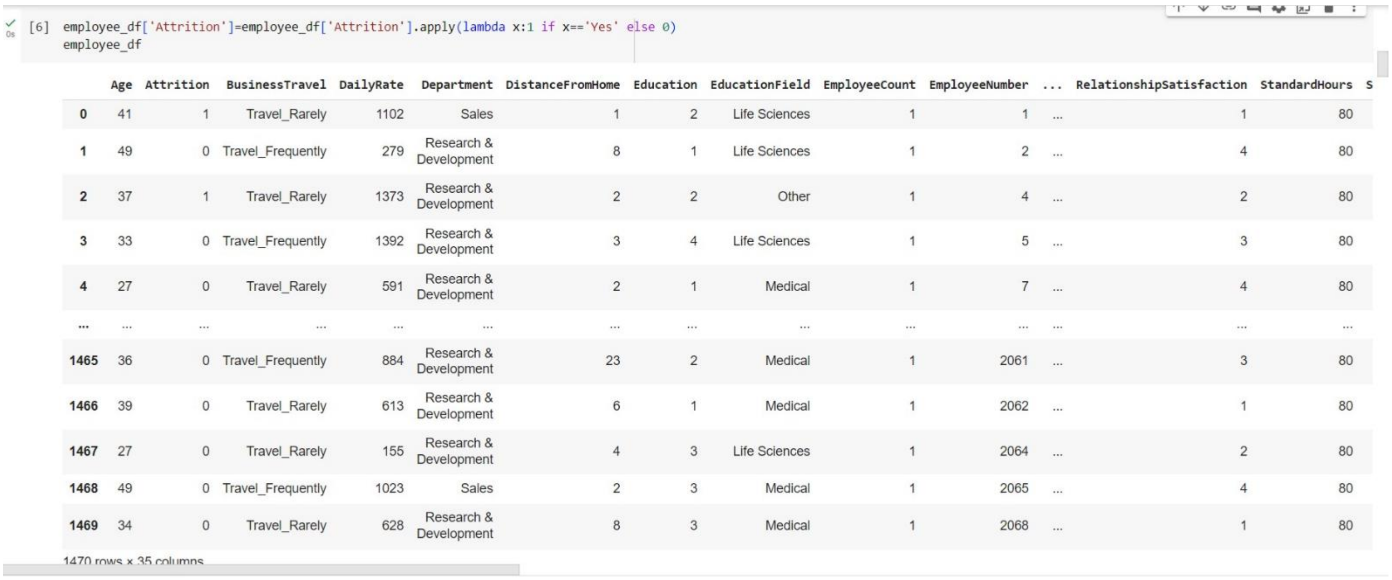
Data encoding is the process of converting categorical variables into a numerical format that machine learning algorithms can understand. Categorical variables cannot be directly used in most machine-learning models, so they need to be encoded. There are different ways to perform data encoding, and the lambda functions in this code are one method.

-
  ```
  employee_df['Attrition']=employee_df['Attrition'].apply(lambda x:1 if x=='Yes' else 0)
  ```

-
  ```
  employee_df['Over18']=employee_df['Over18'].apply(lambda x:1 if x=='Y' else 0)
  ```

-
  ```
  employee_df['OverTime']=employee_df['OverTime'].apply(lambda x:1 if x=='Yes' else 0)
  ```

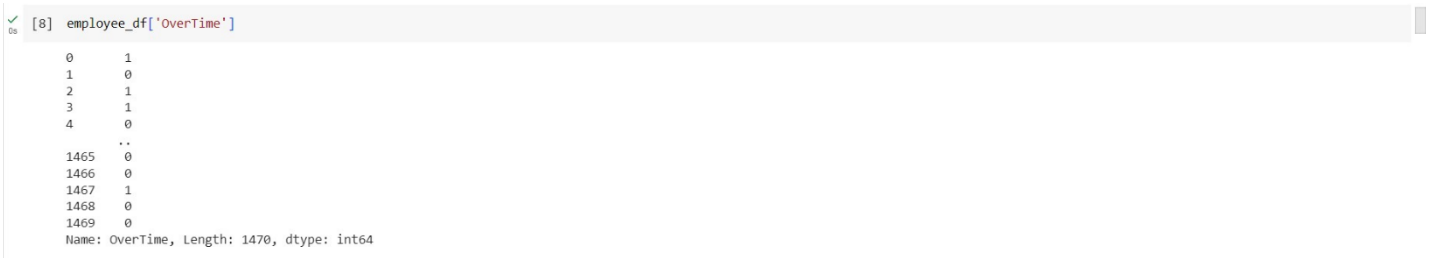
-
  ```
  employee_df['Over18']
  ```



```
[6] employee_df['Attrition']=employee_df['Attrition'].apply(lambda x:1 if x=='Yes' else 0)
    employee_df
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	EmployeeCount	EmployeeNumber	...	RelationshipSatisfaction	StandardHours	S
0	41	1	Travel_Rarely	1102	Sales	1	2	Life Sciences	1	1	...	1	80	
1	49	0	Travel_Frequently	279	Research & Development	8	1	Life Sciences	1	2	...	4	80	
2	37	1	Travel_Rarely	1373	Research & Development	2	2	Other	1	4	...	2	80	
3	33	0	Travel_Frequently	1392	Research & Development	3	4	Life Sciences	1	5	...	3	80	
4	27	0	Travel_Rarely	591	Research & Development	2	1	Medical	1	7	...	4	80	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1465	36	0	Travel_Frequently	884	Research & Development	23	2	Medical	1	2061	...	3	80	
1466	39	0	Travel_Rarely	613	Research & Development	6	1	Medical	1	2062	...	1	80	
1467	27	0	Travel_Rarely	155	Research & Development	4	3	Life Sciences	1	2064	...	2	80	
1468	49	0	Travel_Frequently	1023	Sales	2	3	Medical	1	2065	...	4	80	
1469	34	0	Travel_Rarely	628	Research & Development	8	3	Medical	1	2068	...	1	80	

1470 rows × 35 columns



```
[8] employee_df['OverTime']

0       1
1       0
2       1
3       1
4       0
       ..
1465    0
1466    0
1467    1
1468    0
1469    0
Name: OverTime, Length: 1470, dtype: int64
```

4.4. Data Cleaning:

Data cleaning is an essential step in the data analysis process. It involves identifying and addressing issues such as missing values, duplicate entries, irrelevant columns, and inconsistencies in the dataset. Pandas provide various functions and methods that make data-cleaning tasks more manageable.

Dropping Irrelevant Columns:

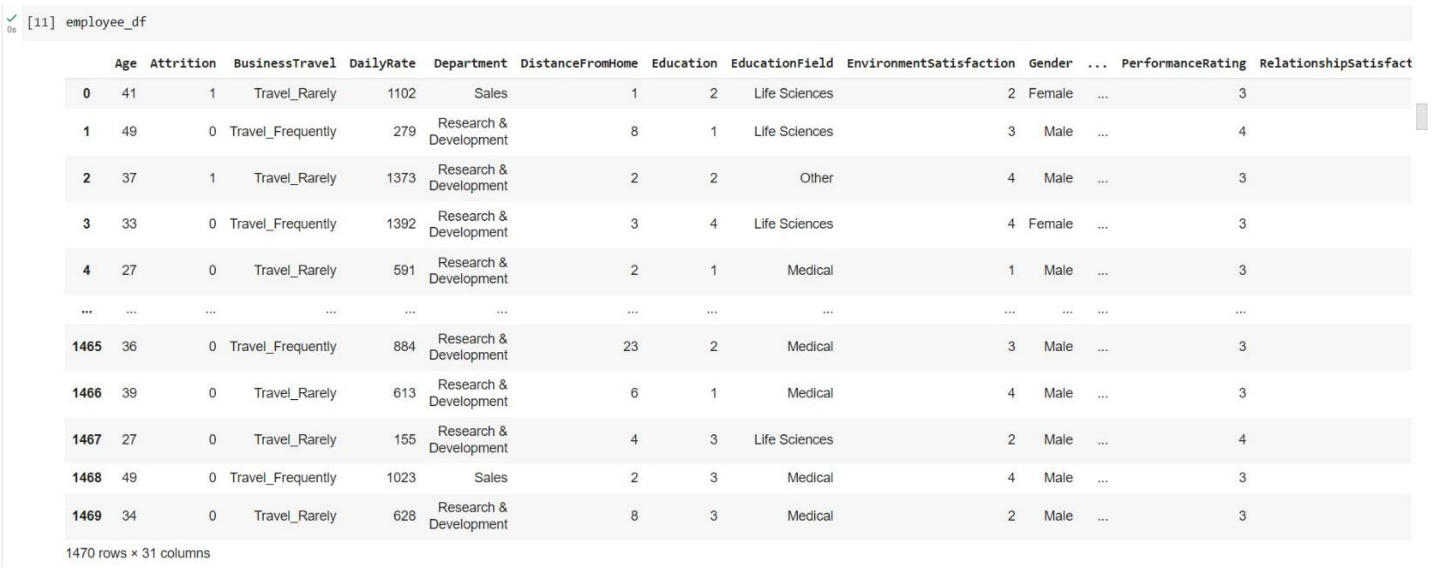
```
➢ employee_df.drop(['EmployeeCount','EmployeeNumber','Over18',
  'StandardHours'],axis=1,inplace=True)
```

In this code, the drop() method is to remove certain columns from the DataFrame employee_df. The columns being dropped are 'EmployeeCount', 'EmployeeNumber', 'Over18', and 'StandardHours'. These columns might contain information that is not relevant to your analysis or modeling.

['EmployeeCount', 'EmployeeNumber', 'Over18', 'StandardHours']: This is a list of column names that we have dropped from the DataFrame.

Axis=1: This specifies that you're dropping columns (as opposed to rows). The value 1 refers to columns, while 0 would refer to rows.

Inplace=True: This parameter modifies the DataFrame directly rather than creating a new DataFrame. When set to True, the DataFrame is updated in place, and no new DataFrame is returned.



```
[11] employee_df
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	EnvironmentSatisfaction	Gender	...	PerformanceRating	RelationshipSatisfact
0	41	1	Travel_Rarely	1102	Sales	1	2	Life Sciences	2	Female	...	3	
1	49	0	Travel_Frequently	279	Research & Development	8	1	Life Sciences	3	Male	...	4	
2	37	1	Travel_Rarely	1373	Research & Development	2	2	Other	4	Male	...	3	
3	33	0	Travel_Frequently	1392	Research & Development	3	4	Life Sciences	4	Female	...	3	
4	27	0	Travel_Rarely	591	Research & Development	2	1	Medical	1	Male	...	3	
...	...	...	...	...	...	...	...	...	...	...	...	...	
1465	36	0	Travel_Frequently	884	Research & Development	23	2	Medical	3	Male	...	3	
1466	39	0	Travel_Rarely	613	Research & Development	6	1	Medical	4	Male	...	3	
1467	27	0	Travel_Rarely	155	Research & Development	4	3	Life Sciences	2	Male	...	4	
1468	49	0	Travel_Frequently	1023	Sales	2	3	Medical	4	Male	...	3	
1469	34	0	Travel_Rarely	628	Research & Development	8	3	Medical	2	Male	...	3	

1470 rows × 31 columns

Initially, we have 35 columns, after dropping now we have 31 columns.

4.5. Printing Attrition Statistics:

In this section of our code, we are calculating and presenting key numerical insights related to the attrition of employees within a company. Attrition refers to the phenomenon of employees leaving an organization voluntarily or involuntarily. These statistics offer a clear quantitative perspective on the impact of attrition and provide valuable insights for understanding workforce dynamics. The calculations include the count and percentage of employees who left the company compared to those who chose to stay. These statistics play a crucial role in assessing the stability of the workforce and guiding organizational strategies to manage attrition effectively.

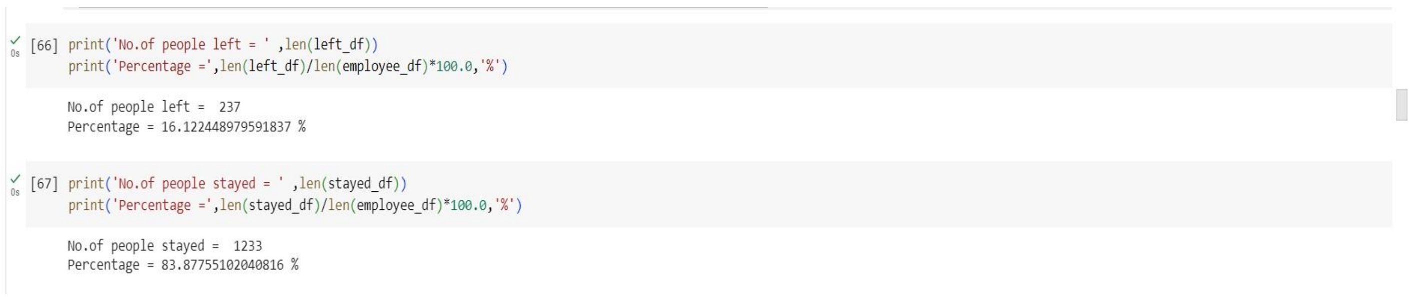
1. No. of people left: This line of code calculates the number of employees who left the company. It does this by determining the length (number of rows) of the DataFrame `left_df`, which contains data only for employees who left the company. The result is the count of employees who left.

2. Percentage of people who left: This line calculates the percentage of employees who left the company in relation to the total number of employees in the entire dataset. It divides the length of `left_df` by the length of the entire `employee_df` and multiplies the result by 100 to express it as a percentage.

3. No. of people stayed: Similar to the first point, this line calculates the number of employees who chose to stay with the company. It does this using the length of the DataFrame `stayed_df`, which contains data only for employees who stayed.

4. Percentage of people who stayed: This line calculates the percentage of employees who stayed with the company in relation to the total number of employees in the dataset. It divides the length of `stayed_df` by the length of the entire `employee_df` and multiplies the result by 100 to express it as a percentage.

These calculations and print statements provide insights into the distribution of attrition within the organization, indicating how many employees left, how many stayed, and the corresponding percentages. It helps to quantify the scale of attrition and retention in relation to the entire employee population.



```
[66] print('No.of people left = ' ,len(left_df))
     print('Percentage =',len(left_df)/len(employee_df)*100.0,'%')

     No.of people left =  237
     Percentage = 16.122448979591837 %

[67] print('No.of people stayed = ' ,len(stayed_df))
     print('Percentage =',len(stayed_df)/len(employee_df)*100.0,'%')

     No.of people stayed =  1233
     Percentage = 83.87755102040816 %
```

4.6. Data Visualisation:

Data visualization is the graphical representation of data to help people understand its significance. It involves using charts, graphs, and other visual elements to present data in a way that is easy to comprehend and interpret. Data visualization is a powerful tool for exploring data, identifying patterns, trends, correlations, and outliers, as well as communicating insights effectively to others.

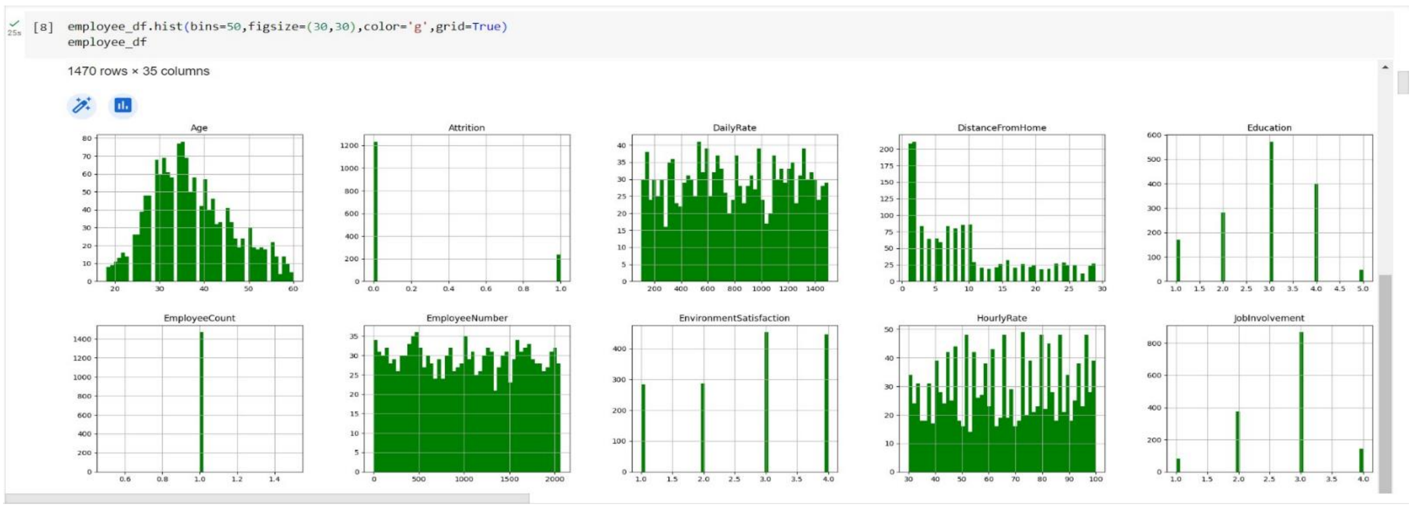
Histograms:

```
➢ employee_df.hist(bins=50,figsize=(30,30),color='g',grid=True
  )
➢ employee_df
```

The code is using the DataFrame to generate the histogram.

The **hist()** function in Pandas is used to create histograms, which are graphical representations of the distribution of data points within a specific numerical range. Histograms are especially useful for understanding the frequency or count of data points that fall into different bins or intervals. They allow you to see patterns, central tendencies, and potential outliers in your data.

Bins Parameter: This parameter specifies the number of bins or intervals that the data range will be divided into. More bins can provide finer details about the distribution, but too few bins might oversimplify the information. In your example, bins=50 divides the data into 50 intervals.

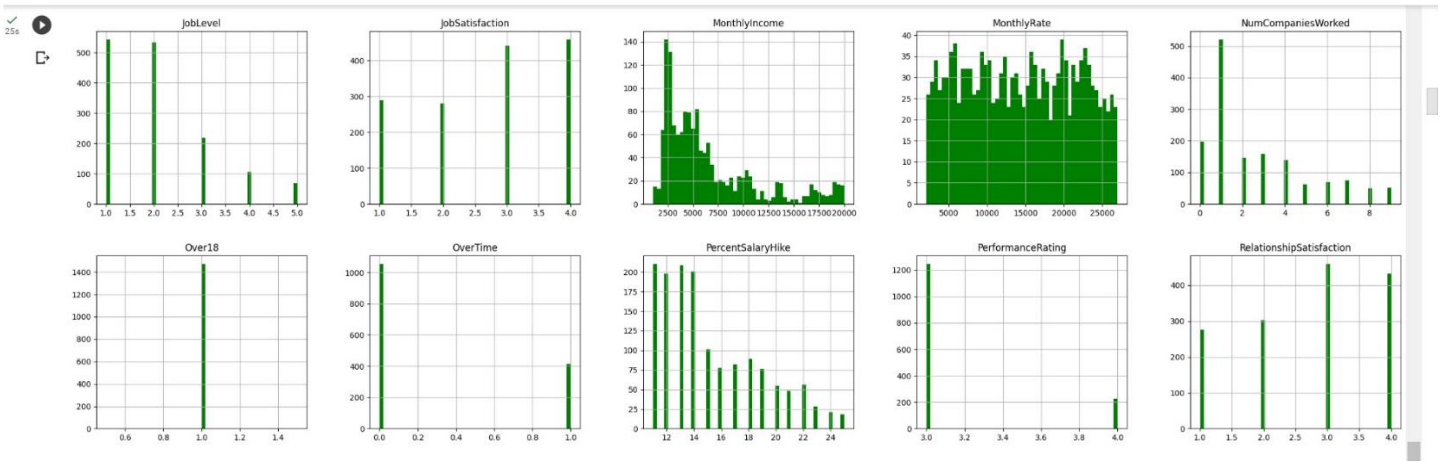


```
[8] employee_df.hist(bins=50,figsize=(30,30),color='g',grid=True)
    employee_df
```

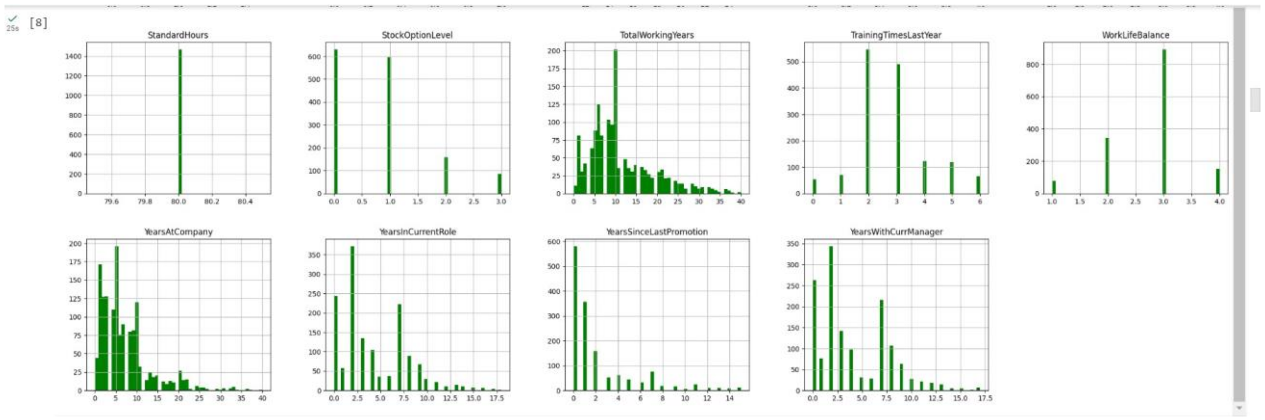
1470 rows × 35 columns

Figsize Parameter: This parameter sets the size of the figure that contains the histogram. It takes a tuple of two values, representing the width and height of the figure in inches. Larger figures can be more informative and visually appealing.

Color Parameter: This parameter specifies the color of the bars in the histogram. You provided 'g', which stands for green. You can use various color representations such as 'b' for blue, 'r' for red, 'c' for cyan, and so on.



Grid Parameter: This parameter determines whether a grid is displayed in the background of the histogram. Setting it to True adds a grid, making it easier to read the values of the bars and intervals. Setting it to False would remove the grid lines.



Heatmaps:

Heatmaps are a type of data visualization that is used to display the correlation between different variables in a tabular form. They are particularly useful when you want to visualize the relationships and patterns among multiple variables.

A heatmap is a graphical representation of data in a matrix format, where individual values are represented using colors. Heatmaps are particularly useful for visualizing relationships, patterns, and distributions within two-dimensional data sets. Here's a brief overview of heatmaps:

Matrix Representation: Heatmaps organize data in a tabular form, where rows and columns correspond to different categories, variables, or dimensions.

Color Encoding: Each cell in the heatmap is filled with a color that corresponds to the value of the data point at the intersection of the corresponding row and column. Colors can be chosen to represent different ranges or magnitudes of values.

Data Patterns: Heatmaps help reveal patterns, trends, and correlations within the data. By using colors to represent values, you can quickly identify high and low values or clusters of similar values.

Correlation Visualization: One common use of heatmaps is for displaying correlation matrices. Positive and negative correlations are visually distinguishable through color variations.

```
➢ correlation=employee_df.corr()
➢ ax=plt.subplots(figsize=(20,20))
➢ sns.heatmap(correlation,annot=True,cmap='viridis')
```

This will generate a heatmap where each cell represents the correlation coefficient between two variables in employee_df, and annotations in each cell will display the actual correlation values. The color of the cells will indicate the strength and direction of the correlation, with warmer colors (reds) representing positive correlations, and cooler colors (blues) representing negative correlations. The diagonal elements will be 1.0 since they represent the correlation of a variable with itself.

The correlation_matrix will be a new DataFrame containing the correlation coefficients between all pairs of numerical columns in employee_df. Each cell in the DataFrame will represent the correlation value between the corresponding variables. The diagonal elements will be 1.0, as they represent the correlation of a variable with itself. This will create a heatmap visualization of the correlation matrix, where each cell represents the correlation coefficient between two variables, and the annotations in each cell show the actual correlation values.

The correlation matrix can be calculated practically using the formula given below:

r = (Σ[(Xi - X_mean) * (Yi - Y_mean)]) / (n * StdDev(X) * StdDev(Y))

Where:

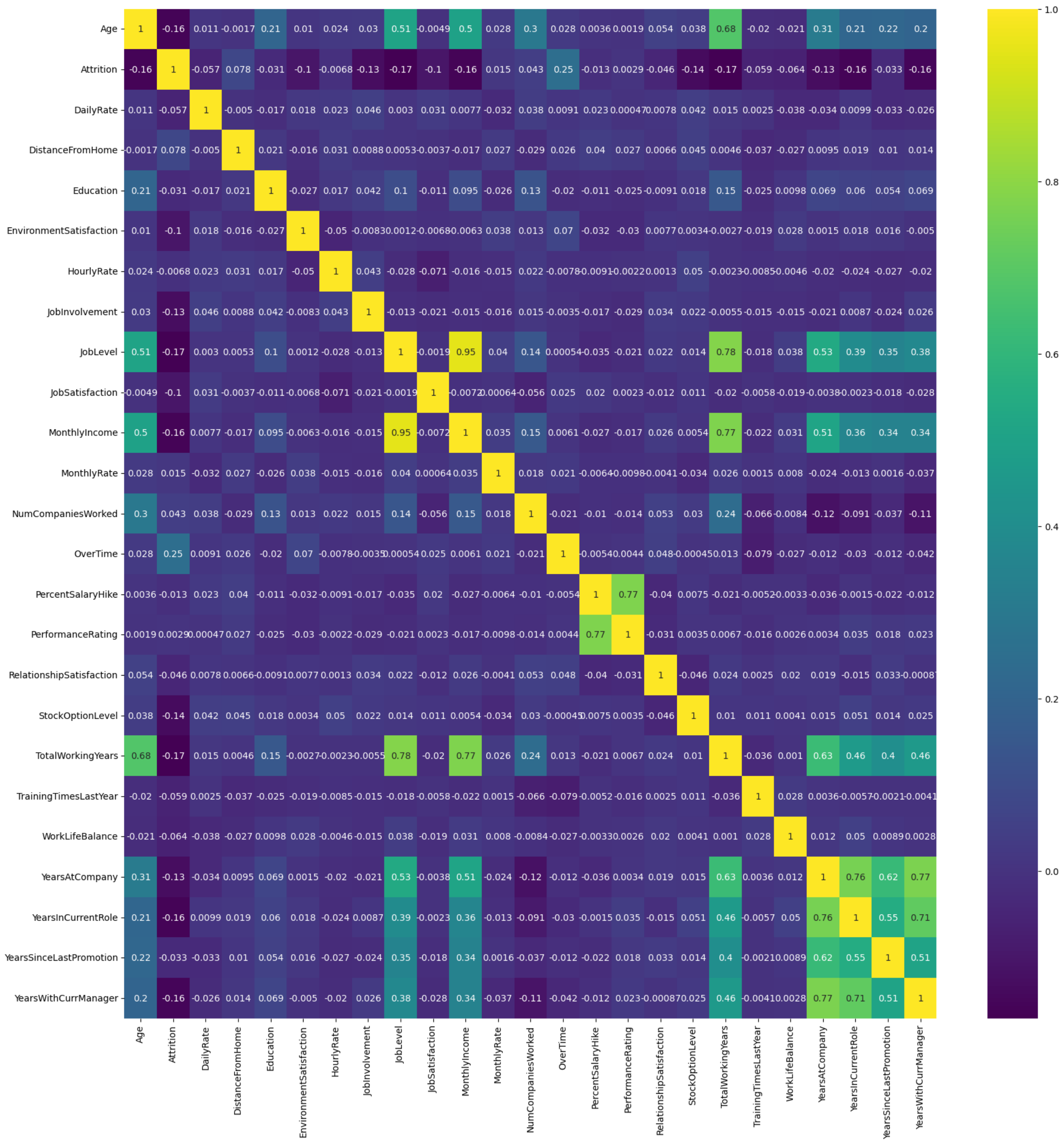
Xi and Yi are individual data points of X and Y, respectively.

X_mean and Y_mean are the means of X and Y, respectively.

n is the number of data points (sample size).

StdDev(X) and StdDev(Y) are the standard deviations of X and Y, respectively.

For each pair of numerical columns (X, Y), we can calculate the correlation coefficient using the formula above, resulting in a correlation matrix.



The default colormap of seaborn will be applied to the heatmap if cmap is not specified. The default colormap is usually a diverging colormap called "viridis."

In matplotlib and seaborn, there are various colormaps (cmap) available to use in visualizations. Each colormap has its own unique set of colors that represent different ranges of data values. Choosing the right colormap can greatly impact the effectiveness of data visualization. Here are some of the commonly used cmap names: Sequential Colormaps: 'viridis', 'plasma', 'inferno', 'magma', 'cividis'.

Grouped countplot:

The countplot is useful for visualizing the distribution of categorical data and comparing different categories within a single variable. It helps to understand how the 'Attrition' status is distributed across different age groups in the employee dataset, giving insights into potential relationships between age and attrition.

```
➢ plt.figure(figsize=(25,12))
➢ sns.countplot(x = 'Age',hue = 'Attrition',  data =
  employee_df,palette='magma')
➢ obInvolvement',hue = 'Attrition',  data =
  employee_df,palette='magma')
➢ sns.countplot(x = 'JobLevel',hue = 'Attrition',  data =
  employee_df,palette='magma')
➢ sns.countplot(x = 'JobRole',hue = 'Attrition',  data =
  employee_df,palette='magma')
➢ sns.countplot(x = 'JobSatisfaction',hue = 'Attrition',  data
  = employee_df,palette='magma')
➢ sns.countplot(x = 'MaritalStatus',hue = 'Attrition',  data =
  employee_df,palette='magma')
```

The parameter x specifies the column from the DataFrame that will be plotted along the x-axis of the countplot. In this case, x='Age' indicates that the 'Age' column from the employee_df DataFrame will be plotted.

In data visualization, the term "hue" refers to a visual encoding that allows you to group and differentiate data points or categories using different colors. When used in the context of Seaborn's plotting functions, the hue parameter is often used to apply this color grouping to the plot.

The hue parameter is used to map a categorical variable to different colors in a plot. When you set hue to a column name from your DataFrame, seaborn will automatically use different colors for each unique category or level of that column.

The hue parameter is set to 'Attrition', which means that Seaborn will use different colors to represent each level of the 'Attrition' column. If the 'Attrition' column has two levels, say 'Yes' and 'No', then the countplot will have bars for each unique age value, and the bars will be grouped and colored based on whether the 'Attrition' status is 'Yes' or 'No'.

Using hue is particularly useful when we have categorical data that one wants to compare across different levels or categories. It allows us to see how the distribution of one variable (age in this case) varies across different groups defined by the 'Attrition' status.

The hue parameter in Seaborn's plotting functions adds an extra level of visual grouping to the plot based on a categorical variable, enhancing the insights and making it easier to compare different categories within the data.

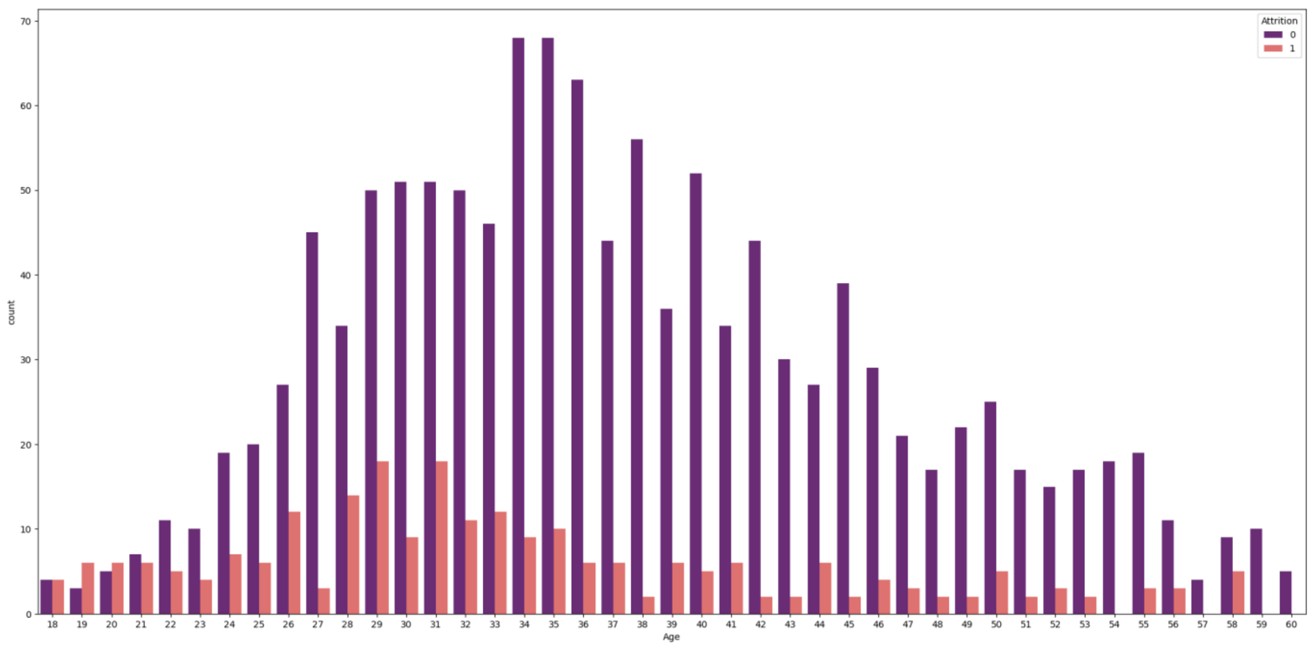
data=employee_df: Specifies the DataFrame containing the data to be visualized.

The resulting grouped countplot will have:

The x-axis represents different age categories.

The y-axis represents the count of occurrences for each age category.

Bars are grouped by the 'Attrition' status, with each 'Age' category split into segments for 'Yes' and 'No' using different colors.



In the output bar graph with two colored bars, each pair of bars corresponds to a specific age group ('Age') for employees. The two different colors represent the two possible 'Attrition' statuses: 'Yes' and 'No'.

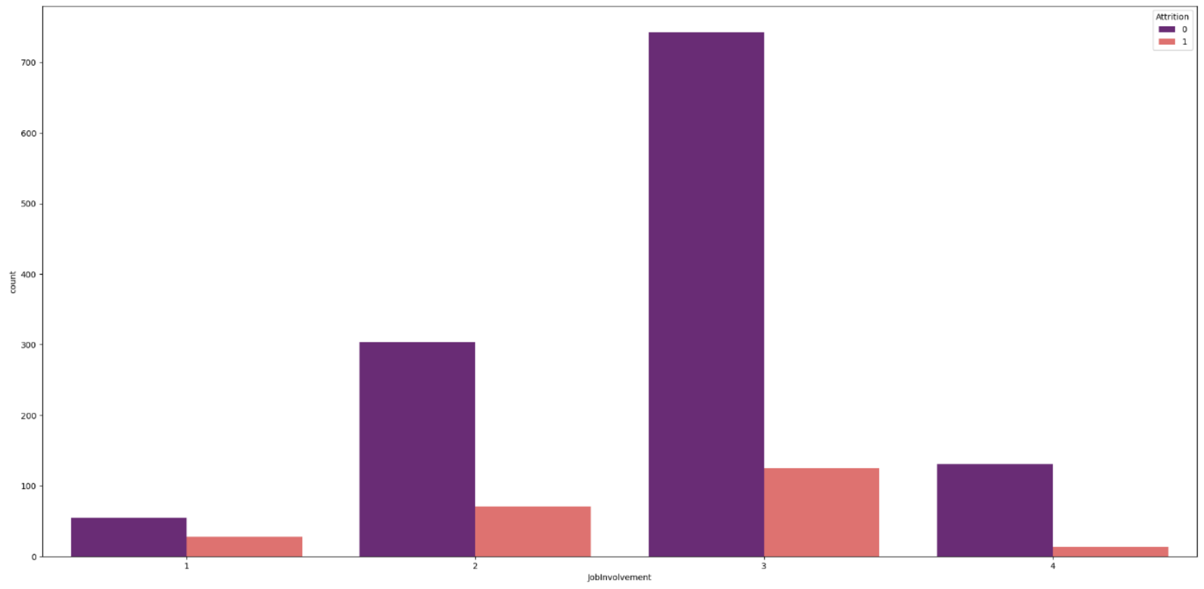
First Bar in Pair: This bar represents the count of employees within the specified age group who have 'Attrition' status 'Yes', meaning they left the company.

Second Bar in Pair: This bar represents the count of employees within the specified age group who have 'Attrition' status 'No', meaning they did not leave the company.

The comparison between the two bars in each pair provides insights into how attrition status is distributed across different age groups. We can identify which age groups have a higher proportion of employees leaving ('Yes') and which have a higher proportion staying ('No'). The colors allow you to quickly differentiate between these two attrition statuses in the visualization.

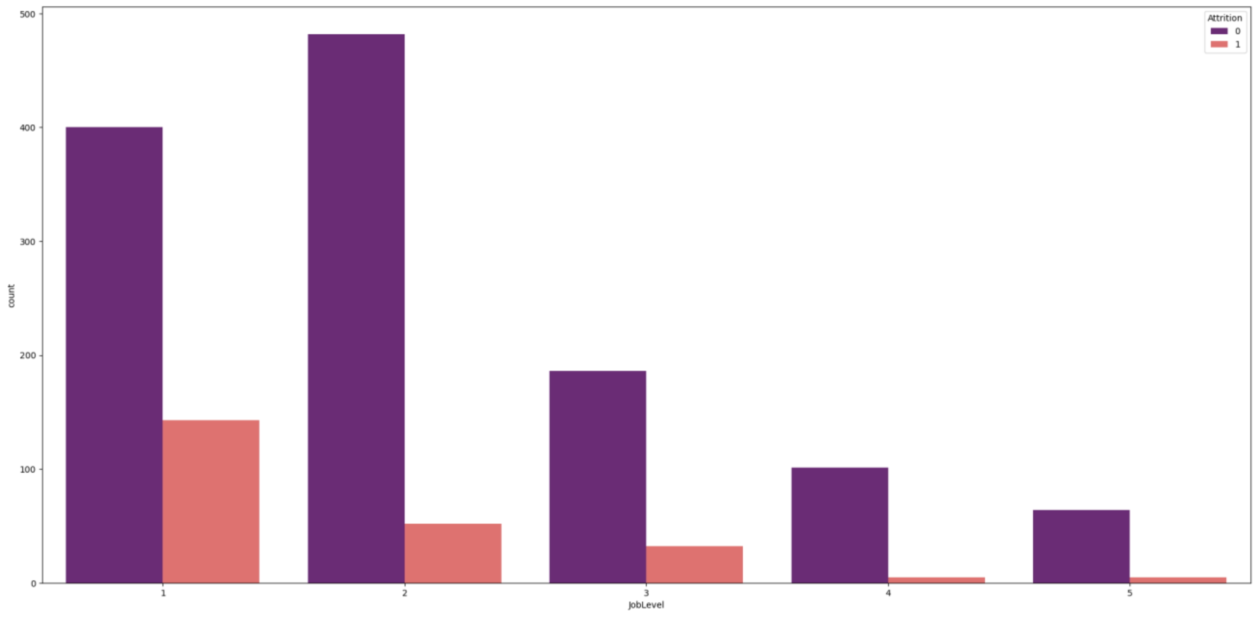
Now, for comparison between the distribution of different levels of job involvement (JobInvolvement) with respect to the 'Attrition' status of employees in the dataset:

We can identify that job involvement levels have a higher proportion of employees leaving ('Yes') and which have a higher proportion of staying ('No'). The colors allow you to quickly differentiate between these two attrition statuses in the visualization.



Comparison between 'Job level' and 'Attrition':

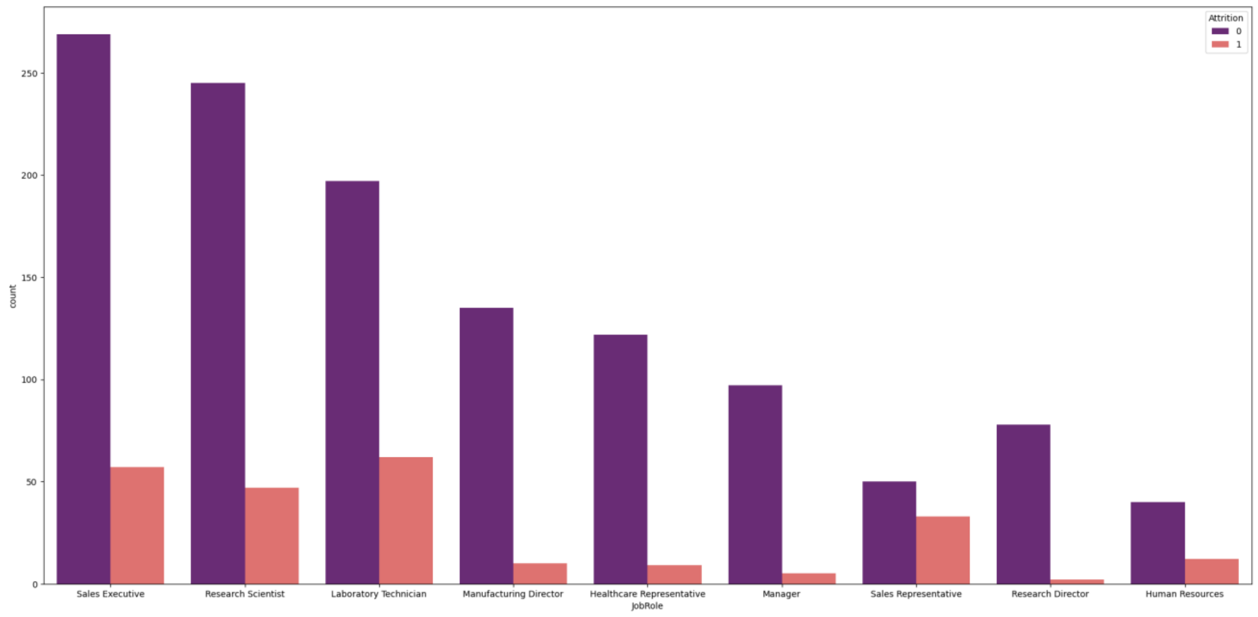
Employees at lower job levels (e.g., Job Level 1) tend to have a higher likelihood of attrition ('Yes') compared to higher job levels. As job level increases, the proportion of attrition ('Yes') appears to decrease.



Comparison between 'Job Role' and 'Attrition':

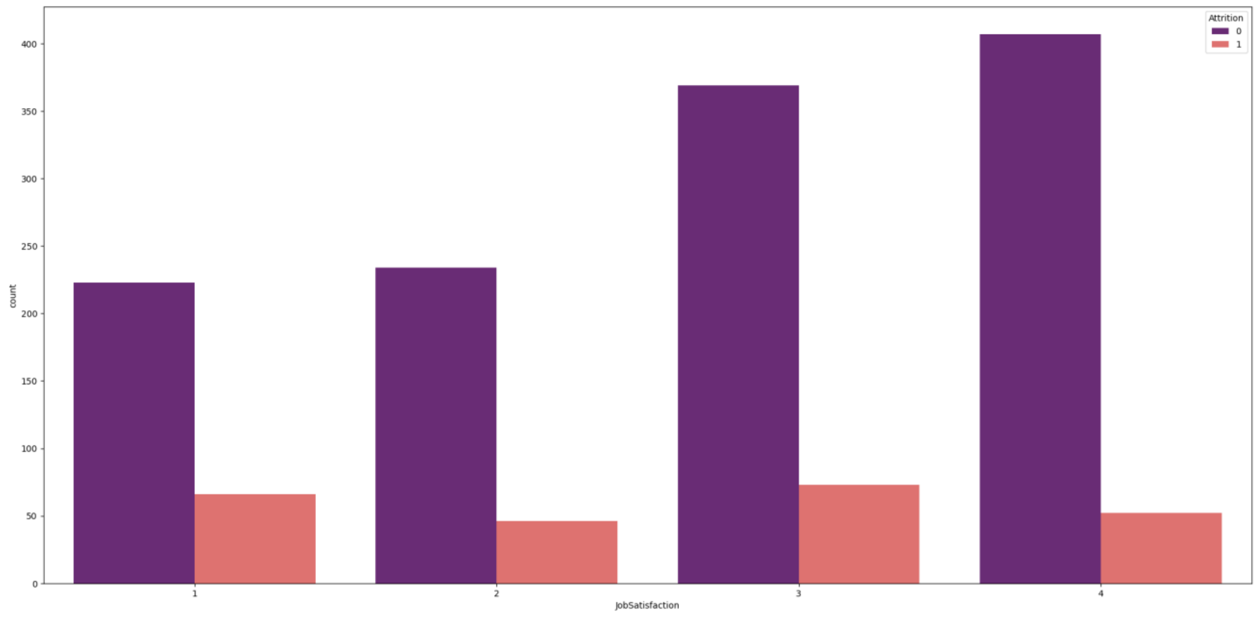
Different job roles exhibit varying levels of attrition. For instance, employees with roles such as 'Sales Executive' and 'Laboratory Technician' seem to have a higher proportion of attrition ('Yes') compared to roles like 'Manager' and 'Research Director'.

Certain job roles may have factors that contribute to increased attrition, possibly related to job responsibilities or growth opportunities



Comparison between 'Job Satisfaction' and 'Attrition':

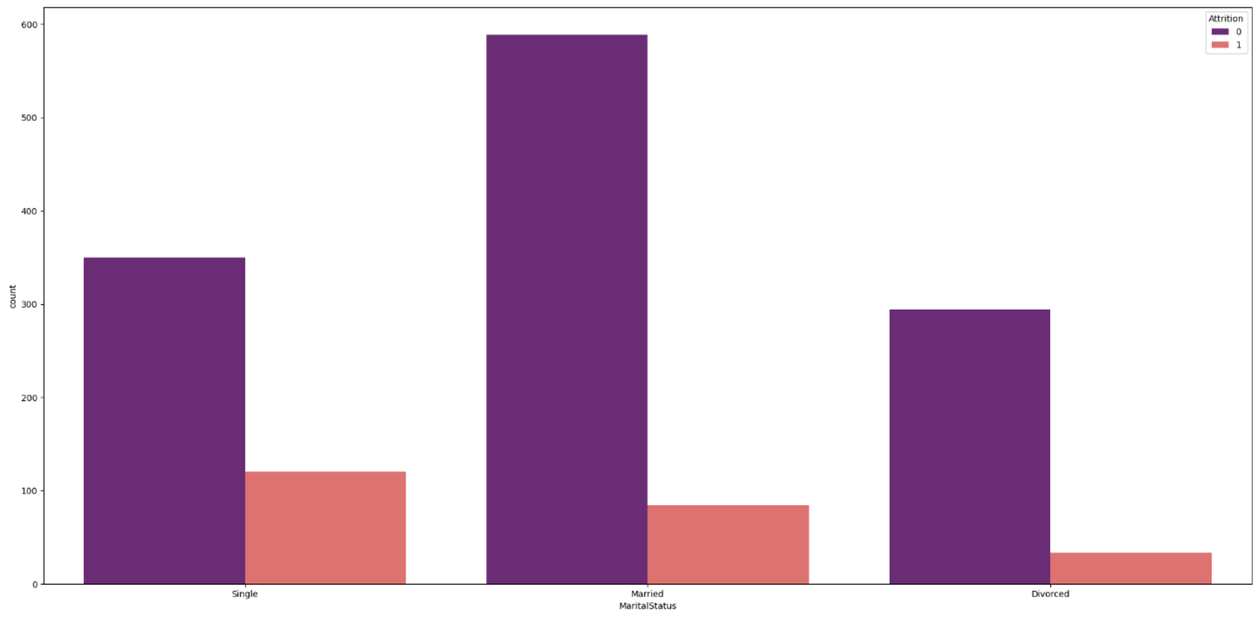
Employees with lower job satisfaction tend to have a higher likelihood of attrition ('Yes'). Higher job satisfaction levels appear to correlate with a lower proportion of attrition ('Yes').



Comparison between ‘Marital Status’ and ‘Attrition’:

Employees who are 'Single' have a relatively higher likelihood of attrition ('Yes') compared to employees who are 'Married' or 'Divorced'.

'Married' employees tend to have a lower proportion of attrition compared to other marital status categories.



Kernel Density Estimate (KDE) overlaid plot:

- **sns.kdeplot(left_df['DistanceFromHome'],label='employees who left', fill=True, color='r')**
  sns.kdeplot(stayed_df['DistanceFromHome'],label='employees who stayed', fill=True, color='g')

In this visualization, two KDE plots are overlaid on the same axes to compare the distributions of the 'DistanceFromHome' variable for employees who left ('employees who left') and employees who stayed ('employees who stayed').

In this type of visualization:

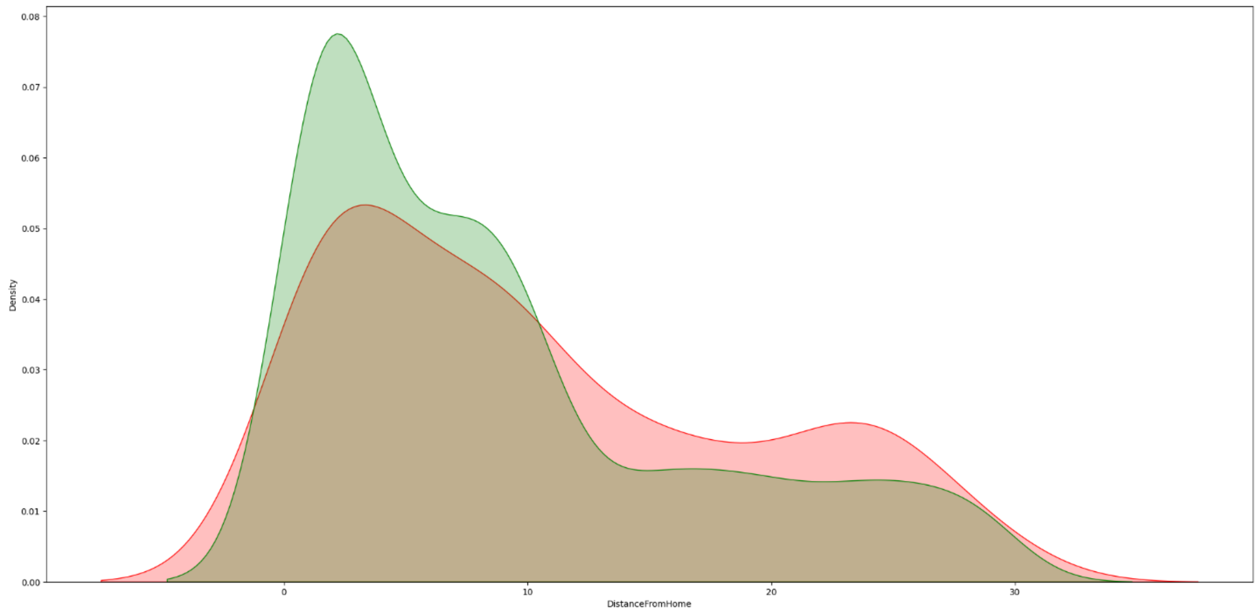
- The x-axis represents the values of the 'DistanceFromHome' variable.

- The y-axis represents the estimated density of data points at each value along the x-axis.

- Different KDE plots are shown using different colors ('r' for red and 'g' for green in your case).

The KDE plots give you insights into the distribution of the 'DistanceFromHome' variable within each group. We can observe where the data is concentrated and where it is more spread out, as well as any overlaps or differences between the distributions of employees who left and those who stayed.

The use of filled areas under the KDE curves emphasizes the density of the data at different points along the x-axis, making it easier to compare the distributions visually. Overlaid KDE plots are particularly effective when comparing continuous distributions for different groups.



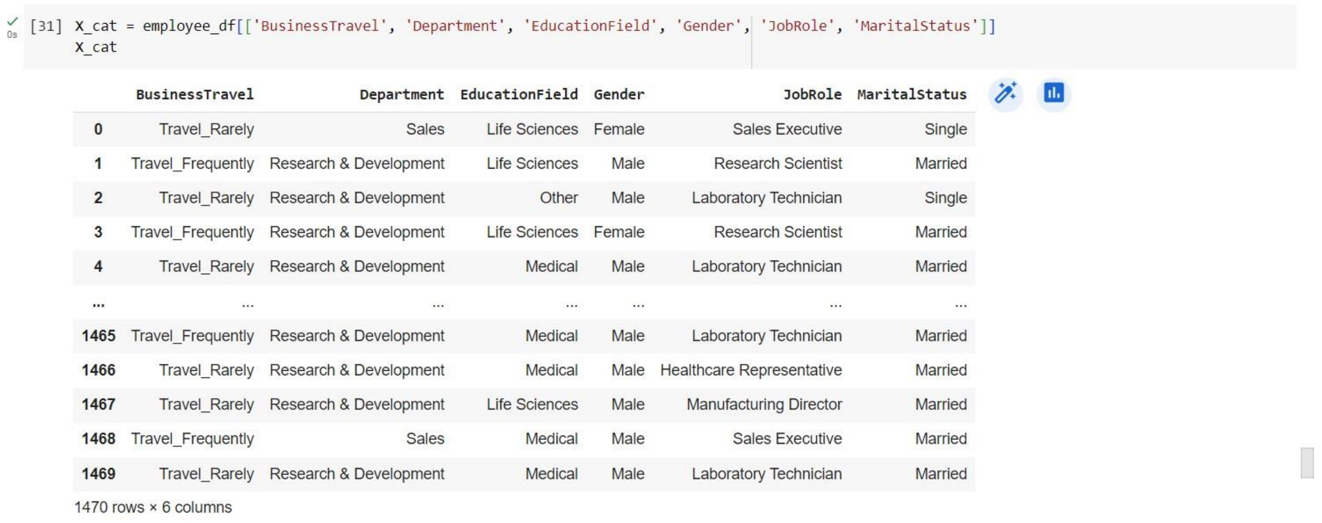
4.7. Data preprocessing and setup:

This section encompasses the necessary tasks and processes required to prepare the raw data for machine learning analysis. Each step serves a specific purpose in transforming the data into a suitable format for training and evaluating machine learning models.

```
➢ X_cat = employee_df[['BusinessTravel', 'Department',
  'EducationField', 'Gender', 'JobRole', 'MaritalStatus']]
  X_cat
```

This step is done to isolate the categorical features that will be encoded separately from the numerical features.

The reason for this separation is that categorical variables need to be transformed into a numerical format before they can be used as input for machine learning algorithms like Logistic Regression or Random Forest. One-hot encoding is a common technique used to convert categorical variables into a numerical format that these algorithms can understand.



```
[31] X_cat = employee_df[['BusinessTravel', 'Department', 'EducationField', 'Gender', 'JobRole', 'MaritalStatus']]
     X_cat
```

	BusinessTravel	Department	EducationField	Gender	JobRole	MaritalStatus
0	Travel_Rarely	Sales	Life Sciences	Female	Sales Executive	Single
1	Travel_Frequently	Research & Development	Life Sciences	Male	Research Scientist	Married
2	Travel_Rarely	Research & Development	Other	Male	Laboratory Technician	Single
3	Travel_Frequently	Research & Development	Life Sciences	Female	Research Scientist	Married
4	Travel_Rarely	Research & Development	Medical	Male	Laboratory Technician	Married
...	...	...	...	...	...	...
1465	Travel_Frequently	Research & Development	Medical	Male	Laboratory Technician	Married
1466	Travel_Rarely	Research & Development	Medical	Male	Healthcare Representative	Married
1467	Travel_Rarely	Research & Development	Life Sciences	Male	Manufacturing Director	Married
1468	Travel_Frequently	Sales	Medical	Male	Sales Executive	Married
1469	Travel_Rarely	Research & Development	Medical	Male	Laboratory Technician	Married

1470 rows × 6 columns

We select the columns that contain categorical features that you want to include in your model. These categorical features are 'BusinessTravel', 'Department', 'EducationField', 'Gender', 'JobRole', and 'MaritalStatus'.

```
➢ from sklearn.preprocessing import OneHotEncoder
➢ onehotencoder = OneHotEncoder()
➢ X_cat = onehotencoder.fit_transform(X_cat).toarray()
```

One-Hot Encoding:

After selecting the categorical features,we perform one-hot encoding on the X_cat DataFrame. This step converts the categorical variables into a binary (0/1) format, where each category becomes a separate binary column.

OneHotEncoder is a popular technique used in machine learning and data preprocessing to encode categorical data into a binary format that can be more easily used for various algorithms. It is commonly employed when dealing with categorical features that do not have an inherent ordinal relationship.

Here's how data is encoded using the OneHotEncoder:

Input Data, Creating Binary Columns, Encoding Process

The encoding process involves assigning a value of 1 to the corresponding binary column for the category that a data point belongs to, and 0 to all the other binary columns. Essentially, each category is represented as a binary vector where only one element is 1 and the rest are 0.

Once the data has been one-hot encoded, it can be used as input for machine learning algorithms. Many algorithms work better with numerical data, and OneHotEncoder transforms categorical data into a format that algorithms can process more effectively.

Scikit-learn is a popular Python library that provides an implementation of OneHotEncoder, and using it is quite straightforward

The number of columns we will get after applying OneHotEncoder depends on the number of unique categories across all the categorical columns.

BusinessTravel: 3 unique categories

Department: 3 unique categories

EducationField: 6 unique categories

Gender: 2 unique categories

JobRole: 9 unique categories

MaritalStatus: 3 unique categories

The total number of columns after applying OneHotEncoder would be

3 + 3 + 6 + 2 + 9 + 3 = 26 columns.

So, with 1470 rows and 26 columns, the X_cat transformed using OneHotEncoder would have a shape of (1470, 26).

Fit and Transform:

In machine learning, the process of "fitting" and "transforming" data is a common pattern used to prepare data for training and modeling

1. Fit:

The "fit" step involves learning or estimating parameters from the data. When you "fit" a model or a preprocessing technique to data, you are calculating internal parameters based on the characteristics of the data. These parameters are learned to capture patterns, distributions, or relationships within the data.

In the case of preprocessing techniques like normalization, scaling, and encoding, "fitting" refers to calculating parameters necessary for the transformation. For example, when using the `MinMaxScaler`, the "fit" step calculates the minimum and maximum values of each feature so that the scaling can be applied properly.

In the context of a machine learning algorithm, "fitting" refers to the process of adjusting the model's internal parameters (like weights in a neural network or coefficients in a linear regression) to learn patterns from the training data.

2. Transform:

The "transform" step applies the learned parameters to the data to perform the actual transformation. This step uses the calculated parameters to modify or process the data in a specific way.

In preprocessing, the "transform" step is where the data is actually transformed based on the calculated parameters. For example, using the `transform` method of a scaler applies the scaling formula using the previously calculated minimum and maximum values.

In the context of a machine learning algorithm, "transforming" data means using the learned model to make predictions or infer information about new, unseen data.

3. `fit_transform` Method:

The `fit_transform` method combines both the "fit" and "transform" steps into a single operation. It first learns the necessary parameters from the data (the "fit" step), and then it immediately applies those parameters to transform the data (the "transform" step).

Using `fit_transform` can be more efficient than separately calling `fit` and `transform`, especially for certain preprocessing techniques where the intermediate parameters are needed in the transformation step.

In the above code, `onehotencoder.fit_transform(X_cat)` is using the `fit_transform` method of the `OneHotEncoder` object to both fit the encoder to the data (learn the categories) and then immediately transform the data into a one-hot encoded format.

This helps to streamline the process of preparing categorical data for machine learning models.

Toarray:

The result of the transformation is a sparse matrix, which is a memory-efficient way to represent the data. However, in this case, `.toarray()` is used to convert the sparse matrix into a NumPy array. This makes the data more accessible and easier to work with in subsequent steps.

The combined effect of `fit_transform` and `toarray` is that it takes the categorical features in `X_cat`, applies one-hot encoding to them, and returns a NumPy array containing the one-hot encoded values. Each categorical variable is converted into several binary columns, where each column corresponds to one category of the original variable. The presence (1) or absence (0) of each category is represented in these binary columns.

For example, let's say you have the 'Gender' categorical variable with two categories: 'Male' and 'Female'. After one-hot encoding, this single 'Gender' column would be transformed into two separate columns: 'Gender_Male' and 'Gender_Female'. If a data point originally had 'Male' as its gender, the 'Gender_Male' column would have a value of 1 and the 'Gender_Female' column would have a value of 0 for that data point.

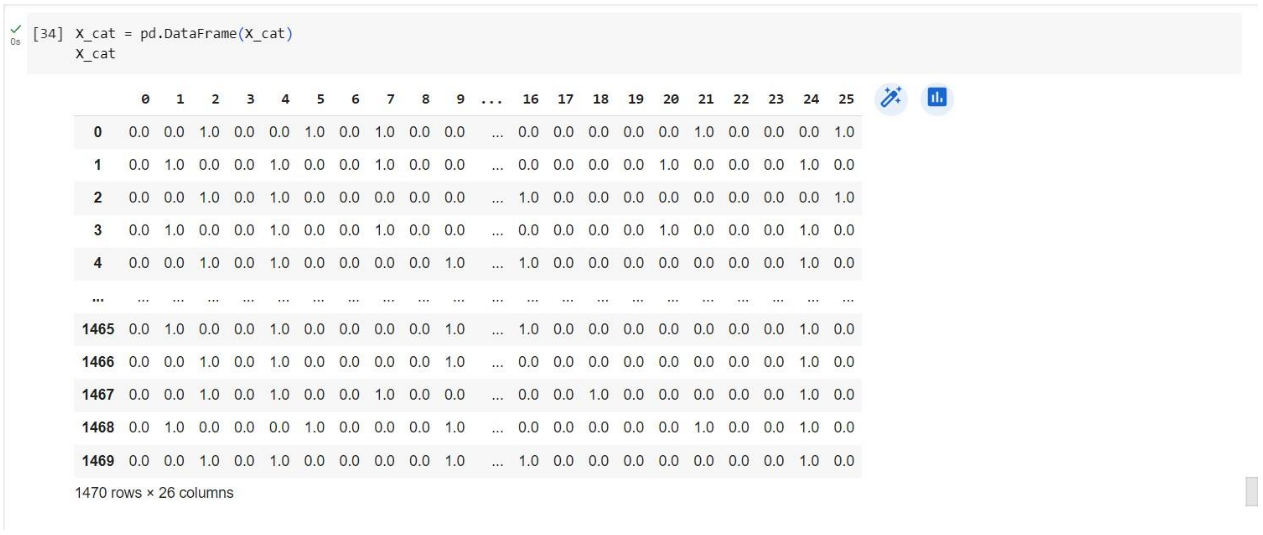
This is an essential step in preprocessing categorical data, converting it into a format that can be used as input for machine learning algorithms that require numerical data.

```
➢ X_cat = pd.DataFrame(X_cat)
➢ X_cat
```

Now, after performing one-hot encoding using the `OneHotEncoder`, the result is transformed into a NumPy array. However, often it's useful to have this encoded data in a DataFrame format, especially for further analysis or combining it with other data. The code you provided achieves this by converting the one-hot encoded NumPy array back into a DataFrame using the `pd.DataFrame()` function:

By passing `X_cat` as an argument to `pd.DataFrame()`, we create a new DataFrame containing the one-hot encoded data.

The variable name `X_cat` is then assigned to the newly created DataFrame, which now holds the one-hot encoded categorical features in a tabular format.This DataFrame will have columns corresponding to the one-hot encoded categories of the original categorical variables.



```
[34] X_cat = pd.DataFrame(X_cat)
     X_cat
```

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21	22	23	24	25
0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
1	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
3	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
4	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1465	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
1466	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
1467	0.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
1468	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
1469	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0

1470 rows × 26 columns

This step is useful because it allows us to work with the encoded categorical features in a familiar DataFrame format, making it easier to combine with other data, perform visualizations, or prepare for training machine learning models.

```
➢ X_numerical = employee_df[['Age', 'DailyRate',
  'DistanceFromHome',  'Education', 'EnvironmentSatisfaction',
  'HourlyRate', 'JobInvolvement', 'JobLevel',
  'JobSatisfaction',  'MonthlyIncome',  'MonthlyRate',  'NumCompaniesW
  orked', 'OverTime', 'PercentSalaryHike', 'PerformanceRating',
  'RelationshipSatisfaction', 'StockOptionLevel', 'TotalWorkingYears'
  ,'TrainingTimesLastYear'  ,
  'WorkLifeBalance',  'YearsAtCompany'  ,'YearsInCurrentRole',
  'YearsSinceLastPromotion', 'YearsWithCurrManager']]
➢ X_numerical
```

Here we created a DataFrame named `X_numerical` by selecting specific columns from the original DataFrame `employee_df`. These selected columns contain numerical features that we intend to use for model training and analysis.

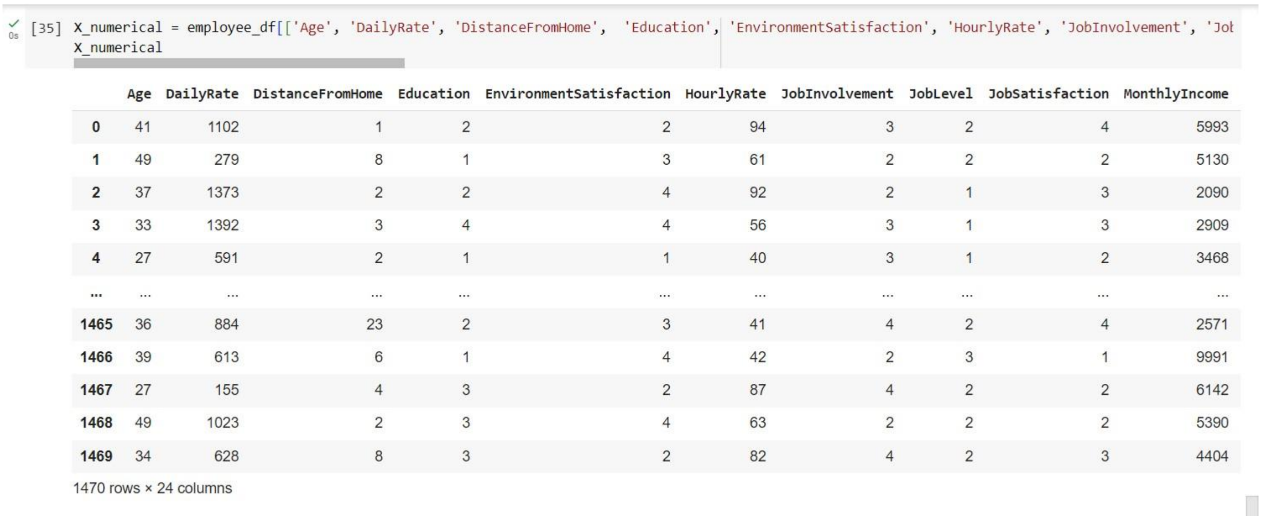
1. `employee_df[['Age', 'DailyRate', 'DistanceFromHome', ...]]:

This part of the code selects multiple columns from the DataFrame `employee_df`.The columns being selected are: 'Age', 'DailyRate', 'DistanceFromHome', 'Education', and so on. These columns likely contain numerical attributes related to employee characteristics, such as age, income, job-related metrics, and more.

2. `X_numerical`:

The result of the column selection operation is assigned to the DataFrame variable `X_numerical`. This new DataFrame `X_numerical` will contain only the columns that were selected from the original DataFrame.

The purpose of creating `X_numerical` is to isolate and organize the numerical features in the data. Numerical features are often treated differently during preprocessing (e.g., scaling) compared to categorical features. By separating them into their own DataFrame, you can apply specific preprocessing steps that are appropriate for numerical data.



```
[35] X_numerical = employee_df[['Age', 'DailyRate', 'DistanceFromHome',  'Education', 'EnvironmentSatisfaction', 'HourlyRate', 'JobInvolvement', 'Jol
     X_numerical
```

	Age	DailyRate	DistanceFromHome	Education	EnvironmentSatisfaction	HourlyRate	JobInvolvement	JobLevel	JobSatisfaction	MonthlyIncome
0	41	1102	1	2	2	94	3	2	4	5993
1	49	279	8	1	3	61	2	2	2	5130
2	37	1373	2	2	4	92	2	1	3	2090
3	33	1392	3	4	4	56	3	1	3	2909
4	27	591	2	1	1	40	3	1	2	3468
...	...	...	...	...	...	...	...	...	...	...
1465	36	884	23	2	3	41	4	2	4	2571
1466	39	613	6	1	4	42	2	3	1	9991
1467	27	155	4	3	2	87	4	2	2	6142
1468	49	1023	2	3	4	63	2	2	2	5390
1469	34	628	8	3	2	82	4	2	3	4404

1470 rows × 24 columns

`X_numerical` is a DataFrame that holds the selected numerical features from the original dataset, making it easier to apply preprocessing steps and integrate them into the overall model-building process.

```
➢ X_all = pd.concat([X_cat, X_numerical], axis = 1)
➢ X_all.columns=X_all.columns.astype(str)
➢ X_all
```

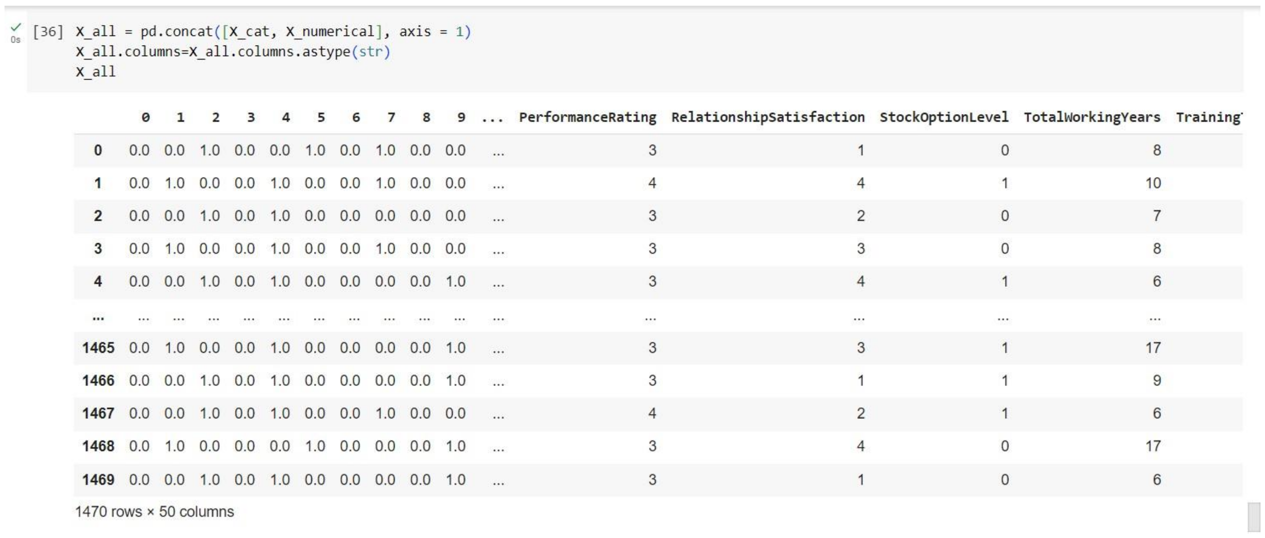
In this step, we combined two sets of features, one-hot encoded categorical features (`X_cat`) and numerical features (`X_numerical`), into a single DataFrame called `X_all`. Let's break down the process step by step:

`pd.concat()` is a function provided by the pandas library in Python used to concatenate or combine DataFrames along a specified axis.`X_cat` is the DataFrame containing the one-hot encoded categorical features.`X_numerical` is the DataFrame containing the numerical features.

`axis=1` specifies that we want to concatenate the DataFrames side by side (horizontally), combining their columns.

`X_all`:

Finally, the concatenated DataFrame `X_all` is assigned to the variable `X_all`. This DataFrame contains both the one-hot encoded categorical features and the numerical features in a single tabular format.The resulting DataFrame, `X_all`, is a comprehensive dataset that includes all the features you have prepared for further analysis or for training machine learning models.



```
[36] X_all = pd.concat([X_cat, X_numerical], axis = 1)
     X_all.columns=X_all.columns.astype(str)
     X_all
```

```
        0    1    2    3    4    5    6    7    8    9  ...  PerformanceRating  RelationshipSatisfaction  StockOptionLevel  TotalWorkingYears  Training
   0  0.0  0.0  1.0  0.0  0.0  1.0  0.0  1.0  0.0  0.0  ...                  3                         1                 0                  8
   1  0.0  1.0  0.0  0.0  1.0  0.0  0.0  1.0  0.0  0.0  ...                  4                         4                 1                 10
   2  0.0  0.0  1.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  ...                  3                         2                 0                  7
   3  0.0  1.0  0.0  0.0  1.0  0.0  0.0  1.0  0.0  0.0  ...                  3                         3                 0                  8
   4  0.0  0.0  1.0  0.0  1.0  0.0  0.0  0.0  0.0  1.0  ...                  3                         4                 1                  6
 ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...                ...                       ...               ...                ...
1465  0.0  1.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  1.0  ...                  3                         3                 1                 17
1466  0.0  0.0  1.0  0.0  1.0  0.0  0.0  0.0  0.0  1.0  ...                  3                         1                 1                  9
1467  0.0  0.0  1.0  0.0  1.0  0.0  0.0  1.0  0.0  0.0  ...                  4                         2                 1                  6
1468  0.0  1.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  1.0  ...                  3                         4                 0                 17
1469  0.0  0.0  1.0  0.0  1.0  0.0  0.0  0.0  0.0  1.0  ...                  3                         1                 0                  6
1470 rows × 50 columns
```

It is important to have all the features combined into a single DataFrame before feeding them into a machine learning algorithm, as most algorithms expect a single input matrix where each row corresponds to an instance and each column corresponds to a feature.

```
➢ from sklearn.preprocessing import MinMaxScaler
➢ scaler=MinMaxScaler()
➢ X=scaler.fit_transform(X_all)
```

Min-Max Scaler:

The Min-Max Scaler is a preprocessing technique used in machine learning to scale numerical features to a specific range, usually between 0 and 1. It is designed to ensure that all features have the same scale, which can be important for certain algorithms that are sensitive to the scale of the input data.

The Min-Max Scaler scales features by subtracting the minimum value from each feature and then dividing by the range (difference between the maximum and minimum values). The formula for scaling a feature `x` using the Min-Max Scaler is:

scaled_x = x - min(x) / max(x) - min(x)

Where:
- (x) is the original value of the feature.
- (min(x)) is the minimum value of the feature.
- (max(x)) is the maximum value of the feature.
- (scaled_x) is the scaled value of the feature.

Advantages of Min-Max Scaling:

1. Preserves Relationships: Min-Max scaling preserves the relationships between the values in the original data while ensuring they all fall within a defined range. This can be important for certain algorithms that rely on distance metrics or require features to be on the same scale.

2. Good for Algorithms: Algorithms like support vector machines (SVM), k-means clustering, and neural networks often perform better when features are scaled to a similar range.

3. Interpretability: Min-Max scaling doesn't change the distribution of the data, making it relatively interpretable. The scaled values are still proportional to the original values.

In the above code, `data` is the matrix of numerical features we wanted to scale. The `fit_transform` method both computes the scaling parameters from the data and transforms the data using these parameters.

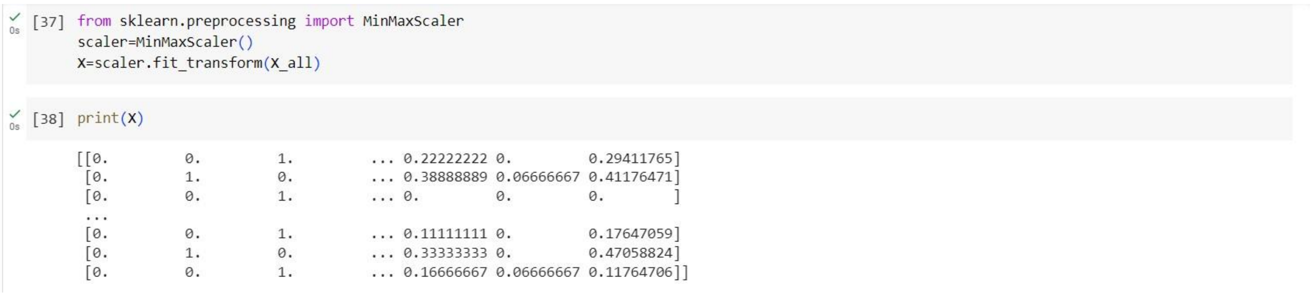
scaler=MinMaxScaler()`:

This line creates an instance of the `MinMaxScaler` class and assigns it to the variable `scaler`.The `MinMaxScaler` scales features to a specified range (usually between 0 and 1) by subtracting the minimum value and dividing by the range.

X=scaler.fit_transform(X_all)`:

This line applies the scaling transformation to the entire combined feature matrix `X_all` using the `fit_transform` method of the `scaler`.The `fit_transform` method first learns the scaling parameters (minimum and maximum values) from the data in `X_all` and then scales the data using those parameters.

The purpose of using the `MinMaxScaler` is to ensure that all features are on the same scale, which can be important for many machine learning algorithms. It prevents features with larger numeric ranges from dominating those with smaller ranges during the modeling process.



```
[37] from sklearn.preprocessing import MinMaxScaler
     scaler=MinMaxScaler()
     X=scaler.fit_transform(X_all)

[38] print(X)

     [[0.         0.         1.         ... 0.22222222 0.         0.29411765]
      [0.         1.         0.         ... 0.38888889 0.06666667 0.41176471]
      [0.         0.         1.         ... 0.         0.         0.        ]
      ...
      [0.         0.         1.         ... 0.11111111 0.         0.17647059]
      [0.         1.         0.         ... 0.33333333 0.         0.47058824]
      [0.         0.         1.         ... 0.16666667 0.06666667 0.11764706]]
```

The code using `MinMaxScaler` scales the combined feature matrix `X_all` so that all features are in a consistent numeric range, which is a common preprocessing step in machine learning to improve the model's performance and convergence.

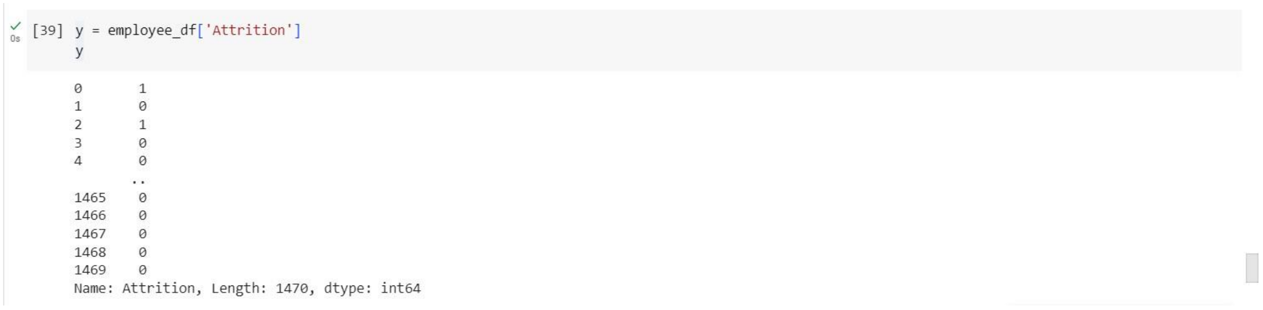
```
➢ y = employee_df['Attrition']
```

Here, `y` is a variable that represents the target variable or the outcome you want to predict using your machine learning model. Here is a breakdown of what's happening:

y = employee_df['Attrition']:

`employee_df` is a DataFrame that contains your dataset, likely representing information about employees.`'Attrition'` is a column name in the `employee_df` DataFrame. It represents whether an employee has left the company or not. This column holds the target variable that you're trying to predict.The code `y = employee_df['Attrition']` extracts the data from the 'Attrition' column and assigns it to the variable `y`.

After executing the above code, the variable `y` will hold the values of the 'Attrition' column. Depending on the content of the column, these values could be binary (e.g., 'Yes' or 'No') or encoded (e.g., 1 or 0) to indicate whether an employee left the company (`1`) or not (`0`).



```
[39] y = employee_df['Attrition']
     y

     0       1
     1       0
     2       1
     3       0
     4       0
            ..
     1465    0
     1466    0
     1467    0
     1468    0
     1469    0
     Name: Attrition, Length: 1470, dtype: int64
```

The variable `y` is often used as the target variable during the model training process. In supervised learning tasks, our machine learning model learns patterns in the input features (`X`) to predict or classify the target variable (`y`). By splitting the dataset into features (`X`) and target (`y`), one can effectively train and evaluate the model's performance on new, unseen data.

```
➢ from sklearn.model_selection import train_test_split
➢ X_train, X_test, y_train, y_test = train_test_split(X, y,
  test_size = 0.25)
```

`from sklearn.model_selection import train_test_split`: This line imports the `train_test_split` function from scikit-learn's `model_selection` module. This function is used to split datasets into training and testing subsets.

The `train_test_split` function is a crucial step in the process of building and evaluating machine learning models. It is used to divide your dataset into separate subsets for training and testing.

Why Train-Test Splitting?

When building a machine learning model, you want to ensure that your model can perform well on new, unseen data. Simply training your model on the entire dataset and evaluating its performance on the same data it was trained on can lead to overfitting, where the model memorizes the training data but doesnot generalize well to new data. To avoid this, you split your dataset into two parts: one for training the model and one for testing its performance on new data.

The Purpose of Train and Test Sets:

1. Training Set (`X_train`, `y_train`):

The training set is used to train our machine learning model. It contains both the input features (`X_train`) and the corresponding target labels or values (`y_train`).The model learns patterns, relationships, and associations from this data during training.

2. Test Set (`X_test`, `y_test`):

The test set is used to evaluate the performance of the trained model on new, unseen data. It simulates how well your model will perform in real-world scenarios.It contains input features (`X_test`) and their corresponding true target labels or values (`y_test`).We use the trained model to make predictions on the test set and then compare those predictions to the true values to assess how well your model generalizes.

Key Concepts and Benefits:

Generalization: The main goal of machine learning is to create models that can generalize well to new, unseen data. Train-test splitting helps you evaluate this generalization ability.

Avoiding Overfitting: By assessing your model's performance on a separate test set, you can detect whether it's overfitting (performing well on training data but poorly on test data) or underfitting (performing poorly on both training and test data).

We can use the test set to fine-tune hyperparameters or make decisions about model complexity and configuration. However, it is important to be cautious with this step to prevent data leakage.

Model Selection: Train-test splitting allows you to compare the performance of different models and select the one that performs best on unseen data.

Splitting the Data:

`X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)`: This line splits the input data (`X`) and the corresponding target variable (`y`) into four subsets: `X_train`, `X_test`, `y_train`, and `y_test`.

`X`: This is the feature matrix that we have prepared earlier (after one-hot encoding and scaling).

`y`: This is the target variable that we want to predict or classify.

Parameters:

`test_size=0.25`: This parameter specifies the proportion of the data that should be reserved for testing. In this case, it is set to 0.25, which means 25% of the data will be used for testing, and 75% will be used for training.

Outputs:

X_train: This is the subset of your feature data that will be used for training your model.

X_test: This is the subset of your feature data that will be used for testing the trained model's performance.

y_train: This is the subset of your target variable that corresponds to the training data.

y_test: This is the subset of your target variable that corresponds to the testing data.

By splitting the data into training and testing sets, we ensure that wehave separate subsets for building and training your machine learning model and for evaluating its performance on unseen data. This helps us to assess how well your model generalizes to new data and prevents overfitting to the training data.

```
➤ X_train.shape
➤ X_test.shape
```

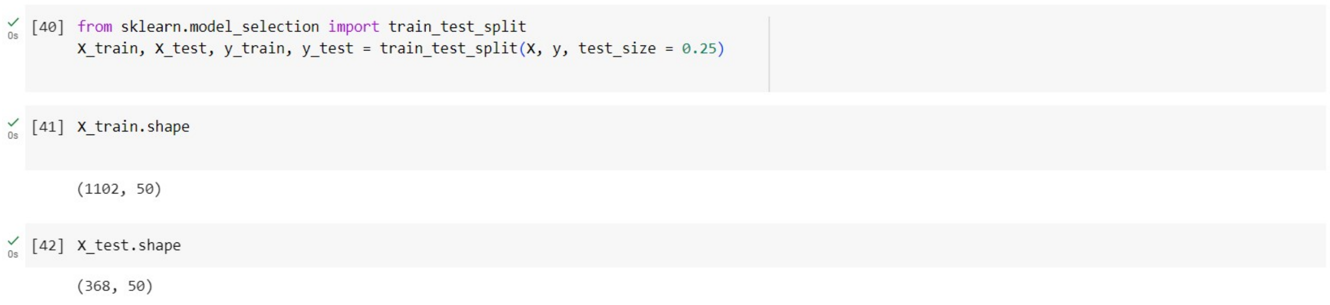
The `shape` attribute checks the dimensions (number of rows and columns) of the `X_train` and `X_test` datasets after the train-test split. Let's break down the code and its purpose:

1. `X_train.shape`:

`X_train` is the dataset that contains the features for training your machine learning model.The `.shape` attribute returns a tuple representing the dimensions of the array or DataFrame. For a two-dimensional object, it returns the number of rows followed by the number of columns.Therefore, `X_train.shape` returns the dimensions of the training dataset, which is the number of samples (rows) and the number of features (columns) in the training data.

2. `X_test.shape`:

`X_test` is the dataset that contains the features for testing your machine learning model's performance on new, unseen data.Similarly to `X_train.shape`,



```
[40] from sklearn.model_selection import train_test_split
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

```
[41] X_train.shape

     (1102, 50)
```

```
[42] X_test.shape

     (368, 50)
```

`X_test.shape` gives you the dimensions of the test dataset, which is the number of samples (rows) and the number of features (columns) in the test data.

By checking the dimensions of `X_train` and `X_test`, we can get an understanding of the size of the datasets that your model will be trained and tested on. This information can be useful for verifying that the train-test split has been performed correctly and that we are feeding the appropriate amount of data to your machine learning algorithms during training and evaluation.

5. MODEL BUILDING

Model building is a crucial step in machine learning where you create predictive models from your data. Model building involves a series of steps aimed at creating a predictive model that can learn patterns from data and make accurate predictions on new, unseen data.

5.1.Logistic regression:

Logistic Regression is a widely used algorithm for binary classification problems. It is used when the dependent variable is binary (two classes) and you want to predict the probability that a given input point belongs to a particular class. Despite its name, logistic regression is used for classification, not regression.

The logistic regression model uses a logistic function to model the probability of the binary outcome based on the input features. The logistic function maps any input to a value between 0 and 1, which represents the probability. The model is trained to adjust the weights of the input features to best fit the observed data.

Logistic Regression is relatively simple and interpretable, making it a good starting point for many classification tasks.

```
➢ from sklearn.linear_model import LogisticRegression
➢ from sklearn.metrics import accuracy_score
➢ model = LogisticRegression()
➢ model.fit(X_train, y_train)
➢ y_pred = model.predict(X_test)
```

`from sklearn.linear_model import LogisticRegression`: This line imports the `LogisticRegression` class from scikit-learn's `linear_model` module. Logistic regression is a commonly used classification algorithm.

`from sklearn.metrics import accuracy_score`: This line imports the `accuracy_score` function from scikit-learn's `metrics` module. This function is used to calculate the accuracy of the model's predictions.

Initializing and Training the Model:

Initializing the Model:

Initializing and training a machine learning model is a fundamental step in the process of building predictive or classification models

Initialization involves creating an instance of the chosen machine learning algorithm's class. In scikit-learn, this is often done by creating an object of the corresponding class.The initialization step sets up the model with default hyperparameters (configuration settings). You can further customize these settings based on your specific problem.

Training the Model:

Training (also called fitting) is the process of teaching the model to learn patterns and relationships in the training data.During training, the model adjusts its internal parameters to minimize the difference between its predictions and the actual target values.The process typically involves an optimization algorithm that finds the best parameter values for the model based on the training data.In supervised learning, the model learns the relationship between the input features and the target variable.

Training Data: During training, the model uses the training data (features and target values) to learn patterns and relationships. The quality and quantity of the training data significantly affect the model's performance.

`model = LogisticRegression()`: This line creates an instance of the `LogisticRegression` class, which represents the logistic regression model.

`model.fit(X_train, y_train)`: This line trains the logistic regression model using the training data. It takes two arguments:

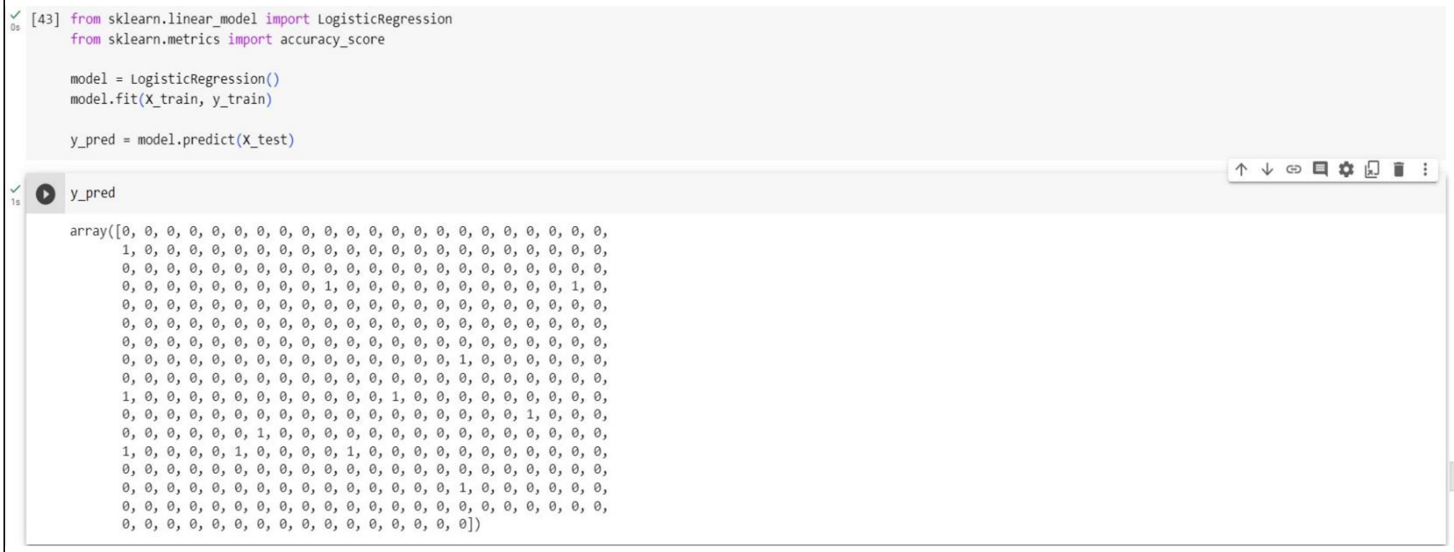
`X_train`: The training data features (input variables).`

y_train`: The corresponding target labels for the training data.

Making Predictions:

`y_pred = model.predict(X_test)`: This line uses the trained logistic regression model to make predictions on the test data. It takes the following arguments:

`X_test`: The test data features (input variables).The predictions are stored in the variable `y_pred`.



```
[43] from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import accuracy_score

     model = LogisticRegression()
     model.fit(X_train, y_train)

     y_pred = model.predict(X_test)
```

```
y_pred
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

5.2. Random Forest Classifier:

The Random Forest Classifier is a powerful ensemble learning algorithm used for classification tasks. It belongs to the category of decision tree-based algorithms, but it combines multiple decision trees to create a more robust and accurate model.

Ensemble Learning:

Ensemble learning involves combining the predictions of multiple individual models (base models) to produce a stronger final prediction. Random Forest is an ensemble technique because it combines multiple decision trees.

How Random Forest Works:

Random Forest starts by creating multiple subsets of the original dataset through a process called bootstrapping. Each subset is of the same size as the original dataset but may contain duplicate instances.Each subset is used to train a separate decision tree.

Random Feature Selection:

For each decision tree, a random subset of features (attributes) is selected from the full set of features. This introduces diversity among the trees and helps prevent overfitting.

Building Decision Trees:

Each decision tree in the Random Forest is built using the bootstrapped subset of the data and the randomly selected subset of features.The decision tree is constructed by recursively splitting the data based on the selected features, using criteria like Gini impurity or information gain.

Voting or Averaging:

During prediction, each decision tree in the forest generates its own prediction. For classification tasks, the final prediction is determined by majority voting: the class that receives the most votes among the trees. For regression tasks, the final prediction is the average prediction of all the trees.

Advantages of Random Forest:

1. By combining multiple decision trees and introducing randomness, Random Forest reduces the risk of overfitting which is common in individual decision trees.

2. Random Forest tends to produce accurate predictions due to the aggregation of multiple diverse models.

3. They can handle missing values and outliers

The Random Forest Classifier is a versatile and powerful algorithm for classification tasks. Its ability to reduce overfitting, handle non-linearity, and provide insights into feature importance makes it a popular choice for various machine learning problems.

Certainly! This code snippet involves using scikit-learn's `RandomForestClassifier` to build and train a random forest model. Let's break down each part of the code:

Random forest is an ensemble learning technique that combines multiple decision trees to create a more robust and accurate model.

Initializing and Training the Model:

`model = RandomForestClassifier()`: This line creates an instance of the `RandomForestClassifier` class. The random forest model is initialized with default hyperparameters.

`model.fit(X_train, y_train)`: This line trains the random forest model using the training data.

`X_train`: The training data features (input variables).

`y_train`: The corresponding target labels for the training data.

This code initializes and trains a random forest classifier using the training data. The trained random forest model is then ready to make predictions on new, unseen data.

```
➢ y_pred = model.predict(X_test)
```

The line `y_pred = model.predict(X_test)` is used to generate predictions for the target variable (labels) based on the features (input data) using the trained machine learning model.

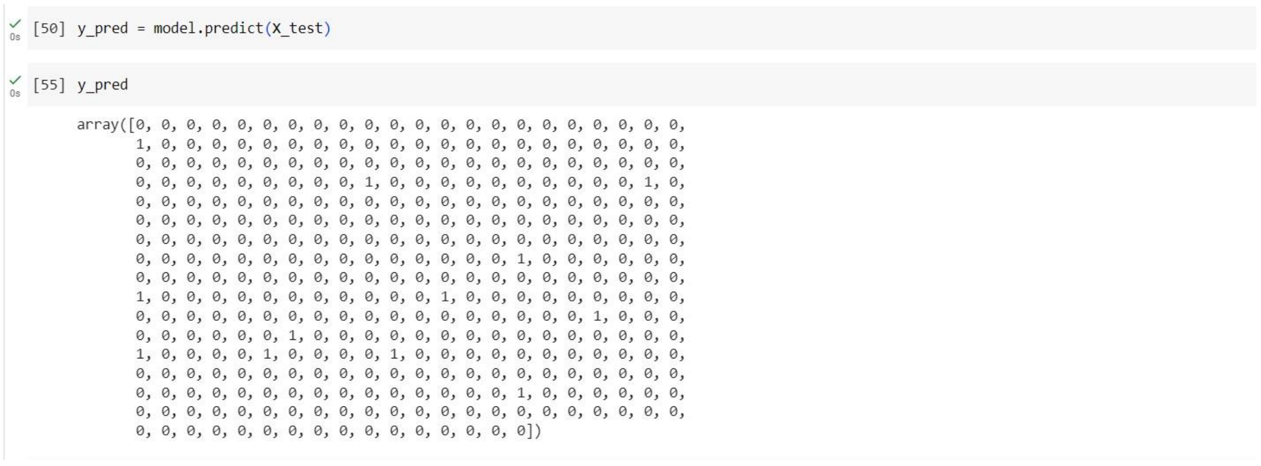
`model.predict(X_test)`:

`model`: This is the instance of the trained machine learning model, in this case, a Random Forest Classifier, that we have previously initialized and trained.

`.predict(X_test)`: This is a method of the model object. It takes the test data (`X_test`) as input and generates predictions for the target variable based on the learned patterns in the training data.

`X_test` represents the test data features (input variables) on which we want to make predictions. It is a subset of the dataset that we have set aside for evaluating the model's performance on unseen data.

`y_pred` is the variable where the predicted target values (labels) are stored.After executing `y_pred = model.predict(X_test)`, `y_pred` will hold the predicted labels for each instance in the test set.



```
[50] y_pred = model.predict(X_test)
```

```
[55] y_pred
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

How `predict` Works:

For each instance in the test set, the trained model applies its learned decision rules (based on the ensemble of decision trees in the case of a Random Forest) to make a prediction.The predicted label corresponds to the class that the majority of decision trees in the ensemble vote for (classification) or the average prediction (regression).

This step generates predictions for the target variable based on the learned patterns in the trained model. These predictions can then be used to assess the model's performance and make informed decisions about its usefulness for real-world applications.

6. MODEL EVALUATION

Model evaluation is a critical step in the machine learning process where we can assess the performance of the trained model using various metrics and techniques. The goal is to understand how well your model is performing, identify potential issues, and make informed decisions about its deployment or further improvements.

Training vs. Test Data:

Models are trained on a subset of our data known as the training dataset. Model evaluation is performed on a separate subset, the test dataset, which the model hasn't seen during training. This helps assess how well the model generalizes to new, unseen data.

➢
```
print("Accuracy {} %".format( 100 * accuracy_score(y_pred,
y_test)))
```

6.1.Evaluating Model Accuracy:

`accuracy_score(y_pred, y_test)`: This line calculates the accuracy of the model's predictions by comparing `y_pred` (predicted labels) with `y_test` (true labels from the test data).The accuracy score is a common metric used to evaluate classification models. It represents the proportion of correctly predicted labels.

`y_true: This is the array of true target values (ground truth). It represents the actual labels or outcomes that you're trying to predict.

`y_pred`: This is the array of predicted target values generated by the model.

What Does the `accuracy_score` Function Do?

The `accuracy_score` function compares the predicted labels (`y_pred`) with the true labels (`y_true`) and calculates the proportion of correctly predicted instances. It returns a value between 0 and 1, where 1 indicates perfect accuracy (all predictions are correct) and 0 indicates no accuracy (all predictions are wrong).

When to Use Accuracy:

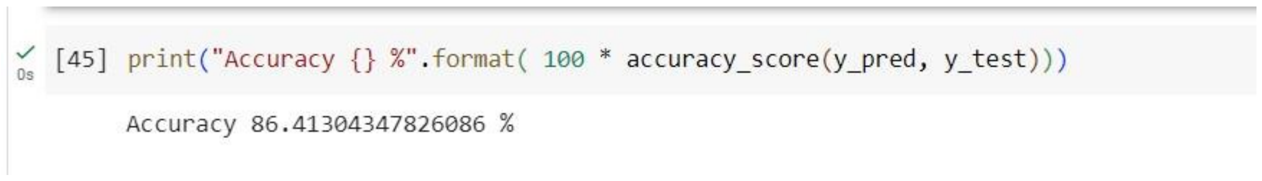
It is a quick and easy way to evaluate the overall performance of a classification model.

This step initializes, trains, and evaluates a logistic regression model. After training, the model is used to predict outcomes for the test data, and the accuracy of these predictions is calculated using the `accuracy_score` function. This accuracy score provides an indication of how well the model is performing on the unseen test data.

```
➢ print("Accuracy {} %".format( 100 * accuracy_score(y_pred,
  y_test))
```

This line of code is printing out the accuracy of the model's predictions on the test data in a human-readable format.

This part calculates the accuracy of the model's predictions (`y_pred`) on the test data compared to the true target values (`y_test`).The `accuracy_score` function computes the proportion of correctly predicted instances among all instances in the test set.



```
[45] print("Accuracy {} %".format( 100 * accuracy_score(y_pred, y_test)))

     Accuracy 86.41304347826086 %
```

The `format()` method is used to insert a value into a string. In this case, it's used to insert the calculated accuracy percentage into the string "Accuracy {} %".This line of code is a common way to display the accuracy of a machine learning model's predictions in a user-friendly format, making it easy to interpret the model's performance at a glance.

```
➢ from sklearn.metrics import confusion_matrix,
  classification_report
```

The `confusion_matrix` and `classification_report` functions are part of scikit-learn's `metrics` module and are used for evaluating the performance of classification models. They provide valuable insights into how well your model is making predictions and help you understand the types of errors it is making. Let's break down each of these functions:

6.2.Confusion Matrix:

A table showing the true positive, true negative, false positive, and false negative predictions.

```
➢ cm = confusion_matrix(y_pred, y_test)
➢ sns.heatmap(cm, annot=True,cmap='Blues')
```

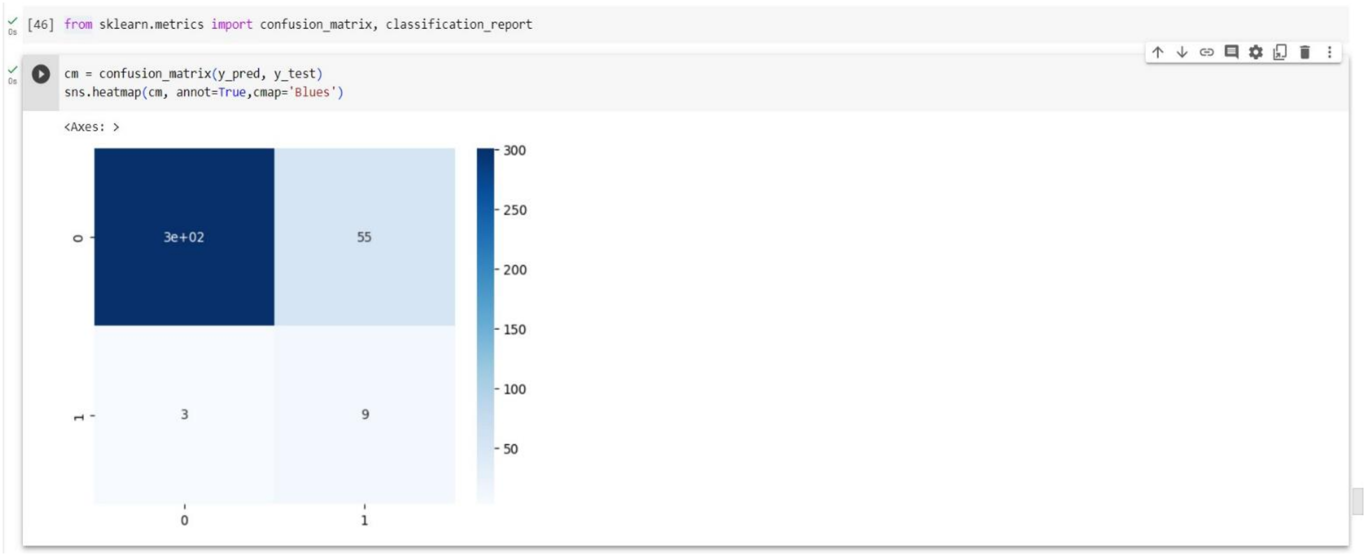
confusion_matrix(y_true, y_pred):

The `confusion_matrix` function creates a confusion matrix, which is a table used to describe the performance of a classification model on a set of data.

It takes two arguments:

`y_true`: The true target values (ground truth) of the data.

`y_pred`: The predicted target values generated by the model.



```
[46] from sklearn.metrics import confusion_matrix, classification_report
```

```
cm = confusion_matrix(y_pred, y_test)
sns.heatmap(cm, annot=True,cmap='Blues')
```

```
<Axes: >
```

It provides a visual representation of the model's performance, showing true positives, true negatives, false positives, and false negatives.Each cell of the matrix represents the count of instances for a particular combination of true and predicted class labels.This helps us to understand the types of errors the model is making (false positives and false negatives) and their magnitudes.

6.3.Classification Report:

Common Evaluation Metrics:

Accuracy: Measures the proportion of correct predictions. It's suitable when classes are balanced.

Precision: Measures the proportion of true positive predictions among all positive predictions. It is important when false positives are costly.

Recall (Sensitivity): Measures the proportion of true positive predictions among all actual positive instances. It's important when false negatives are costly.

F1-Score: The harmonic mean of precision and recall. It balances precision and recall.

➢ `print(classification_report(y_test, y_pred))`

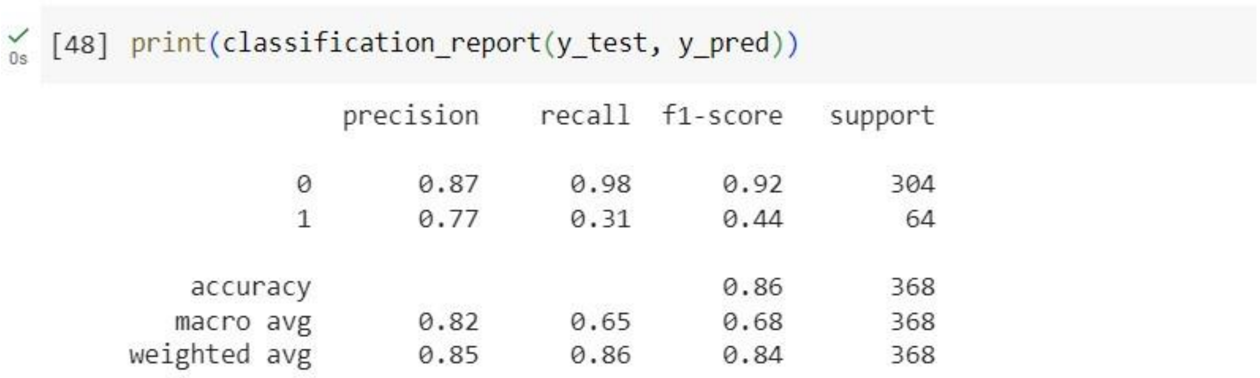
The `classification_report` function generates a detailed report of various classification metrics for each class and overall averages.

It takes two arguments:

`y_true`: The true target values (ground truth) of the data.

`y_pred`: The predicted target values generated by the model.

The classification report includes metrics such as precision, recall, F1-score, and support for each class, as well as the macro and weighted averages across all classes.



```
[48] print(classification_report(y_test, y_pred))

              precision    recall  f1-score   support

           0       0.87      0.98      0.92       304
           1       0.77      0.31      0.44        64

    accuracy                           0.86       368
   macro avg       0.82      0.65      0.68       368
weighted avg       0.85      0.86      0.84       368
```

The classification report is a comprehensive summary of the model's performance metrics for each class and overall. It helps us to understand how well the model is performing in terms of precision (accuracy of positive predictions), recall (sensitivity), and the harmonic mean F1-score (balance between precision and recall). It provides a more holistic view of the model's performance, giving you insights into precision, recall, and F1-score for each class.

In summary, `confusion_matrix` and `classification_report` are powerful tools for understanding and evaluating the performance of your classification model, going beyond simple accuracy to provide a detailed view of how well your model is doing on different aspects of prediction.

7.CONCLUSION

In this project, we aimed to predict employee attrition using machine learning techniques. Employee attrition, the phenomenon of employees leaving a company, can have significant impacts on an organization's productivity and operations. Therefore, predicting attrition can help organizations proactively take steps to retain valuable employees and maintain a stable workforce.

Data Preparation and Feature Engineering:

We began by loading and exploring the employee dataset, which contained various features related to employees' demographic information, work environment, job roles, and more. To prepare the data for modeling, we performed the following steps:

We handled missing values, converted categorical variables into numerical format using one-hot encoding, and transformed binary categorical variables into numeric values (e.g., 'Yes' to 1 and 'No' to 0).

We selected relevant features that could potentially influence employee attrition, such as business travel frequency, department, education field, gender, job role, and marital status.

We scaled the numerical features to ensure that they were on a similar scale, preventing any particular feature from dominating the model due to its magnitude.

For our predictive modeling, we used two different algorithms: Logistic Regression and Random Forest Classifier.

We trained a logistic regression model using the preprocessed dataset. Logistic regression is a linear model that estimates the probability of binary outcomes.We evaluated the model's performance using accuracy and generated a confusion matrix to analyze the predictions' true positive, true negative, false positive, and false negative counts.

We also trained a random forest classifier, an ensemble model combining multiple decision trees. The random forest provides robustness against overfitting and captures complex relationships in the data. We evaluated the model's performance using accuracy, and we further analyzed the predictions using a confusion matrix and classification report to assess precision, recall, and F1-score.

In conclusion, both the logistic regression and random forest models demonstrated promising performance in predicting employee attrition. The classification reports provided insights into the precision and recall for each class, allowing us to evaluate the models' effectiveness in capturing different types of attrition cases.

Our project highlights the significance of predictive modeling in HR analytics, where data-driven insights can help organizations identify employees at risk of attrition and implement targeted retention strategies. Moving forward, we recommend exploring additional techniques such as hyperparameter tuning, feature engineering, and cross-validation to further enhance the models' performance.

Predicting employee attrition not only contributes to better human resource management but also showcases the potential of machine learning in addressing real-world business challenges.

REFERENCES

https://www.w3schools.com/python/

https://www.geeksforgeeks.org/libraries-in-python/

https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/