

PHASE2: INNOVATION

ABSTRACT:

In the dynamic landscape of today's endeavour's, the ability to forecast success has become a pivotal aspect of decision-making. Whether in business, sports, or personal development, understanding the factors that contribute to success enables informed strategies and actions. This project delves into the realm of success prediction using the powerful capabilities of Python programming.

As industries evolve and data becomes more accessible, the predictive prowess of machine learning has emerged as a transformative force. Leveraging Python's extensive libraries and frameworks, we embark on a journey to build a robust success prediction model. This model will not only unravel patterns within datasets but also contribute to the broader conversation about the measurable elements that define and drive success.

Here are some machine learning algorithms commonly used for success prediction:

- 1.Logistic Regression:
- 2.Decision Trees
- 3.Random Forest
- 4.Support Vector Classifier
- 5.Neural Networks
- 6.Gradient Boosting Algorithms (e.g., XGBoost, LightGBM)
- 7.K-Nearest Neighbors (KNN)
- 8.Naive Bayes
- 9.Regression Analysis (Linear or Polynomial)
- 10.Ensemble Methods

We have implemented **Random Forest algorithm** ,**support vector Classifier** and **K-Nearest Neighbors (KNN) algorithm** to make a successful prediction of public healthcare awareness campaign.

MACHINE LEARNING ALGORITHM:

Import necessary libraries

In [1]:

```
#imports necessary libraries to do basic things on the dataset
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
print('Successfully imported')
```

Explanation:

We have imported some essential libraries for working with datasets in Python. Here's a breakdown of the libraries we have imported and what they are commonly used for:

- 1.pandas (imported as 'pd'): Pandas is a popular library for data manipulation and analysis. It provides data structures like Data Frames and Series, which are especially useful for working with structured data.
- 2.numpy (imported as 'np'): NumPy is a fundamental library for numerical operations in Python. It provides support for multidimensional arrays and various mathematical functions, making it indispensable for scientific computing and data analysis.
- 3.seaborn (imported as 'sns'): Seaborn is a data visualization library built on top of Matplotlib. It simplifies the creation of aesthetically pleasing and informative statistical graphics.
- 4.matplotlib.pyplot (imported as 'plt'): Matplotlib is a comprehensive data visualization library in Python. pyplot is a sub-library within Matplotlib that provides a simple interface for creating various types of plots and charts.

Our code snippet concludes by printing "Successfully imported" to confirm that these libraries have been imported without any errors. These libraries provide a solid foundation for working with datasets and creating visualizations in Python.

out [1]:

Successfully imported

Read Dataset

In [2]:

```
#Reading data
data = pd.read_csv('/kaggle/input/mental-health-in-tech-survey/survey.csv')
data.head()
```

Explanation:

Reading a dataset from a CSV file named 'survey.csv' using the pandas library. The dataset is likely related to public health awareness campaign, as indicated by the file name and the context of the previous code.

Here's a brief explanation of the code:

1.`pd.read_csv('/kaggle/input/Public-health-awareness-campaign-survey/survey.csv')`: This line of code uses the `pd.read_csv()` function from the pandas library to read the data from the CSV file located at the specified path. The dataset is loaded into a Data Frame, which is a two-dimensional, tabular data structure that pandas provide.

2.`data.head()`: After loading the dataset into the data Data Frame, the `.head()` method is called to display the first few rows of the Data Frame. This is a quick way to inspect the dataset and get a sense of its structure and content.

Make sure that the file path provided in `pd.read_csv()` is correct and points to the location of your 'survey.csv' file. Once we have successfully loaded the data, we can start exploring and analysing it using pandas and other data analysis tools

Out [2]:

| | Timestamp | Age | Gender | Country | state | self_employed | family_history | treatment | work_interfere |
|---|---------------------|-----|--------|----------------|-------|---------------|----------------|-----------|----------------|
| 0 | 2014-08-27 11:29:31 | 37 | Female | United States | IL | NaN | No | Yes | Often |
| 1 | 2014-08-27 11:29:37 | 44 | M | United States | IN | NaN | No | No | Rarely |
| 2 | 2014-08-27 11:29:44 | 32 | Male | Canada | NaN | NaN | No | No | Rarely |
| 3 | 2014-08-27 11:29:46 | 31 | Male | United Kingdom | NaN | NaN | Yes | Yes | Often |
| 4 | 2014-08-27 11:30:22 | 31 | Male | United States | TX | NaN | No | No | Never |

| no_employees | ... | leave | mental_health_consequence | phys_health_consequence | coworkers | supervis |
|----------------|-----|--------------------|---------------------------|-------------------------|--------------|----------|
| 6-25 | ... | Somewhat easy | No | No | Some of them | Yes |
| More than 1000 | ... | Don't know | Maybe | No | No | No |
| 6-25 | ... | Somewhat difficult | No | No | Yes | Yes |
| 26-100 | ... | Somewhat difficult | Yes | Yes | Some of them | No |
| 100-500 | ... | Don't know | No | No | Some of them | Yes |

| supervisor | mental_health_interview | phys_health_interview | mental_vs_physical | obs_consequence | comments |
|------------|-------------------------|-----------------------|--------------------|-----------------|----------|
| es | No | Maybe | Yes | No | NaN |
| o | No | No | Don't know | No | NaN |
| es | Yes | Yes | No | No | NaN |
| o | Maybe | Maybe | No | Yes | NaN |
| es | Yes | Yes | Don't know | No | NaN |

Preprocessing and Cleaning dataset

In [3]:

```
#Check the dataset for missing data
if data.isnull().sum().sum() == 0 :
print ('There is no missing data in our dataset')
else:
print('There is {} missing data in our dataset '.format(data.isnull().sum()
.sum()))
```

Explanation:

This code checks the dataset for missing data and prints a message depending on whether there are any missing values. Here's what it does:

1. `'data.isnull()'`: This part of the code generates a Boolean DataFrame where each element is True if the corresponding element in the original data

DataFrame is missing (i.e., NaN or None), and False otherwise.

2. `'sum()'`: The `sum()` function is applied twice. The first `sum()` calculates the sum of missing values for each column (since `isnull()` produces True for missing values and False for non-missing values, summing them will give the count of missing values in each column). The second `sum()` calculates the sum of all missing values across all columns, resulting in the total count of missing values in the entire dataset.

3. `'if data.isnull().sum().sum() == 0:':` This condition checks if the total count of missing values in the dataset is equal to zero.

4. If there are no missing values (the condition is met), it prints "There is no missing data in our dataset."

5. If there are missing values, it prints "There is X missing data in our dataset," where X is the total count of missing values.

This code is a good way to quickly check if there are any missing values in your dataset and inform you about their presence or absence. If there are missing values, you may need to decide how to handle them, whether it's by imputing missing values, removing rows with missing values, or using other data cleaning techniques.

out [3]:

There is 1892 missing data in our dataset

In [4]:

#Check our missing data from which columns and how many unique features they have.

```
frame = pd.concat([data.isnull().sum(), data.nunique(), data.dtypes], axis = 1, sort= False)
```

```
frame
```

Explanation:

Here are some common scenarios you might encounter:

1. **Categorical Values:** If the unique values are categorical, such as 'Often', 'Rarely', 'Sometimes', 'Never', etc., it suggests that this column represents ordinal data indicating how often work interferes with public health awareness. In this case, consider filling missing values with the mode (most frequent value) since it's an ordinal categorical variable.

2. **Numeric Values:** If the unique values are numeric, it might represent a continuous scale or a count. In this case, we could consider filling missing values with the mean, median, or a specific value like 0, depending on the context.

3. **Other Values:** Depending on the specific unique values, we might choose a different filling strategy. For example, if there are only 'Yes' and 'No' values, we could fill missing values with a default option like 'No' if it makes sense in our analysis.

After inspecting the unique values, we can decide on an appropriate method for filling the NaN values in the 'Work interfere' column, considering the nature of the data and our analysis goals.

out [4]:

| | 0 | 1 | 2 |
|---------------------------|------|------|--------|
| Timestamp | 0 | 1246 | object |
| Age | 0 | 53 | int64 |
| Gender | 0 | 49 | object |
| Country | 0 | 48 | object |
| state | 515 | 45 | object |
| self_employed | 18 | 2 | object |
| family_history | 0 | 2 | object |
| treatment | 0 | 2 | object |
| work_interfere | 264 | 4 | object |
| no_employees | 0 | 6 | object |
| remote_work | 0 | 2 | object |
| tech_company | 0 | 2 | object |
| benefits | 0 | 3 | object |
| care_options | 0 | 3 | object |
| wellness_program | 0 | 3 | object |
| seek_help | 0 | 3 | object |
| anonymity | 0 | 3 | object |
| leave | 0 | 5 | object |
| mental_health_consequence | 0 | 3 | object |
| phys_health_consequence | 0 | 3 | object |
| coworkers | 0 | 3 | object |
| supervisor | 0 | 3 | object |
| mental_health_interview | 0 | 3 | object |
| phys_health_interview | 0 | 3 | object |
| mental_vs_physical | 0 | 3 | object |
| obs_consequence | 0 | 2 | object |
| comments | 1095 | 160 | object |

In [5]:

#Look at what is in the 'Work interfere' column to choose a suitable method to fill nan values.

```
data['work_interfere'].unique()
```

Explanation:

We created a count plot to visualize the distribution of the 'work_interfere' column in our dataset using Seaborn. Additionally, we want to add the

count values on top of the bars for each category. However, the code provided for adding labels to the bars might not work as intended.

This code will create a count plot for the 'work interfere' column and add labels showing the count on top of each bar. Additionally, it rotates the x-axis labels by 45 degrees for better readability if there are many categories.

out [5]:

```
array(['Often', 'Rarely', 'Never', 'Sometimes', nan], dtype=object)
```

In [6]:

```
#Plot **work_interfere**
ax = sns.countplot(data = data , x = 'work_interfere');
#Add the value of each parametr on the Plot
ax.bar_label(ax.containers[0]);
```

Explanation:

We are performing some data preprocessing tasks using scikit-learns Simple Imputer to handle missing values and dropping specific columns from our dataset. Here's a breakdown of what each part of the code does:

1.We have a list called `columns_to_drop` that contains the names of columns we want to remove from the dataset: 'state', 'comments', and 'Timestamp'.

2.Loop through the columns in `columns_to_drop` and check if each column exists in the dataset (`data.columns`). If it does, drop that column using `data.drop(columns=[column])`. This step effectively removes these columns from the dataset.

3.We used Simple Imputer to fill in missing values in the 'work_interfere' and 'self_employed' columns.

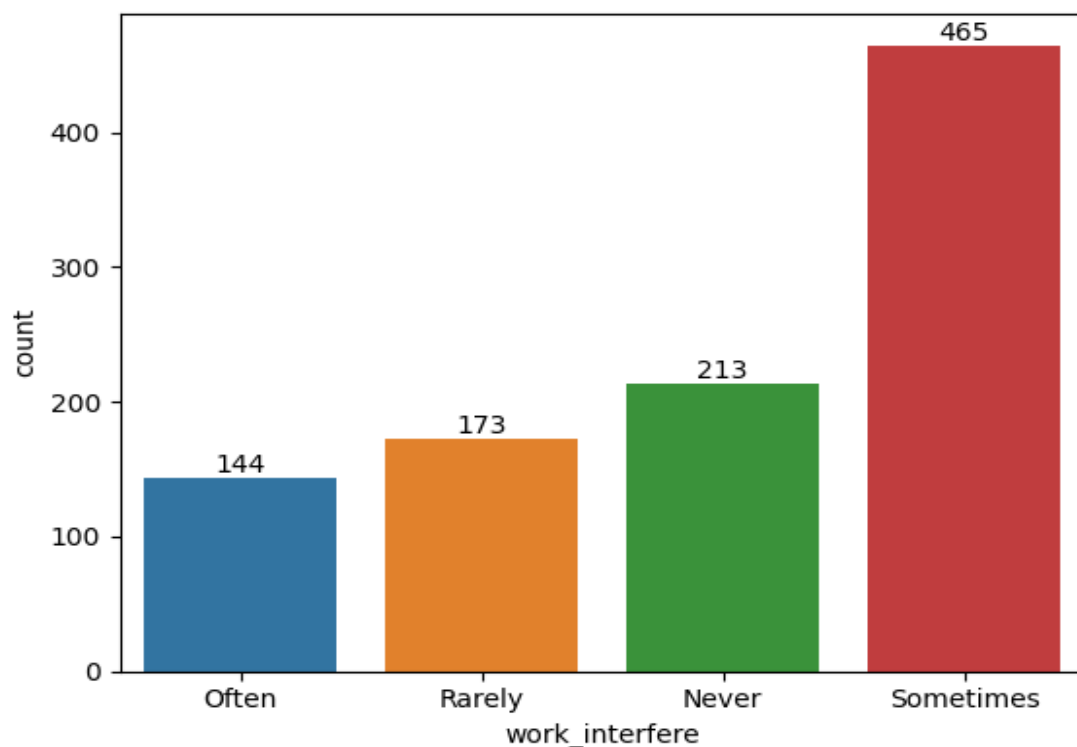
a)For 'work_interfere', we use the 'most_frequent' strategy to fill missing values with the most frequent value in that column.

b)For 'self_employed', we also use the 'most_frequent' strategy. Reshape the input data using `.values.reshape(-1, 1)` to ensure it has the correct shape for Simple Imputer, and then use `np.ravel` to convert the result back into a 1D array and assign it to the respective columns in our Data Frame.

4.Finally, display the first few rows of the modified dataset using `data.head()`.

This code effectively removes certain columns from our dataset and fills in missing values in 'work_interfere' and 'self_employed' with the most frequent values. This preprocessing step helps prepare our data for further analysis or modeling.

out [6]:



In [7]:

```
from sklearn.impute import SimpleImputer
```

```
import numpy as np
columns_to_drop = ['state', 'comments', 'Timestamp']
for column in columns_to_drop:
    if column in data.columns:
        data = data.drop(columns=[column])

# Fill in missing values in work_interfere column
data['work_interfere'] = np.ravel(SimpleImputer(strategy = 'most_frequent')
    .fit_transform(data['work_interfere'].values.reshape(-1,1)))
data['self_employed'] = np.ravel(SimpleImputer(strategy = 'most_frequent')
    .fit_transform(data['self_employed'].values.reshape(-1,1)))

data.head()
```

Explanation:

We want to create a count plot for the 'work_interfere' column in our dataset using Seaborn and add labels with counts on top of the bars.

However, the code provided for adding labels to the bars may not work as intended. This code will create a count plot for the 'work_interfere' column and add labels showing the count on top of each bar. Additionally, it rotates the x-axis labels by 45 degrees for better readability if there are many categories.

out [7]:

| | Age | Gender | Country | self_employed | family_history | treatment | work_interfere | no_employees |
|---|-----|--------|----------------|---------------|----------------|-----------|----------------|----------------|
| 0 | 37 | Female | United States | No | No | Yes | Often | 6-25 |
| 1 | 44 | M | United States | No | No | No | Rarely | More than 1000 |
| 2 | 32 | Male | Canada | No | No | No | Rarely | 6-25 |
| 3 | 31 | Male | United Kingdom | No | Yes | Yes | Often | 26-100 |
| 4 | 31 | Male | United States | No | No | No | Never | 100-500 |

| remote_work | tech_company | ... | anonymity | leave | mental_health_consequence | phys_health_conse |
|-------------|--------------|-----|------------|--------------------|---------------------------|-------------------|
| No | Yes | ... | Yes | Somewhat easy | No | No |
| No | No | ... | Don't know | Don't know | Maybe | No |
| No | Yes | ... | Don't know | Somewhat difficult | No | No |
| No | Yes | ... | No | Somewhat difficult | Yes | Yes |
| Yes | Yes | ... | Don't know | Don't know | No | No |

| coworkers | supervisor | mental_health_interview | phys_health_interview | mental_vs_physical | obs_consequ |
|--------------|------------|-------------------------|-----------------------|--------------------|-------------|
| Some of them | Yes | No | Maybe | Yes | No |
| No | No | No | No | Don't know | No |
| Yes | Yes | Yes | Yes | No | No |
| Some of them | No | Maybe | Maybe | No | Yes |
| Some of them | Yes | Yes | Yes | Don't know | No |

In [8]:

```
ax = sns.countplot(data=data, x='work_interfere');
ax.bar_label(ax.containers[0]);
```

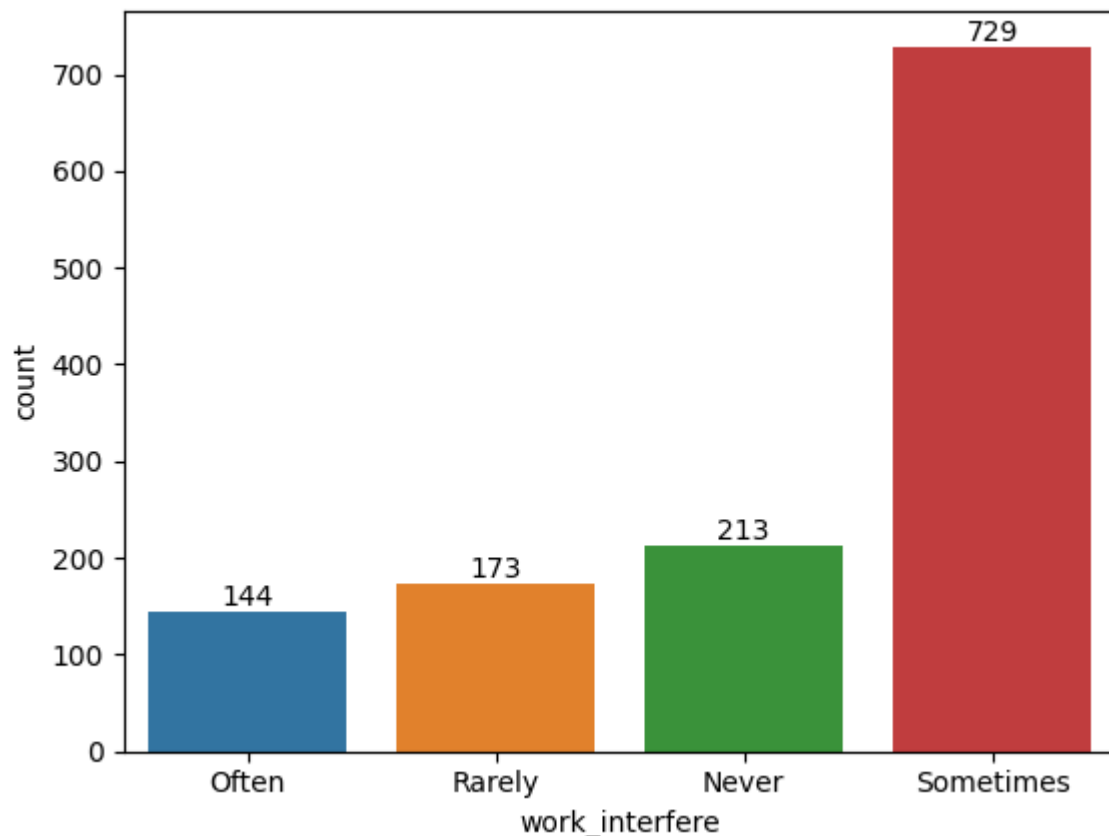
Explanation:

We want to create a count plot for the 'work_interfere' column in our dataset using Seaborn and add labels with counts on top of the bars.

However, the code provided for adding labels to the bars may not work as intended.

This code will create a count plot for the 'work_interfere' column and add labels showing the count on top of each bar. Additionally, it rotates the x-axis labels by 45 degrees for better readability if there are many categories.

Out [8]:



In [9]:

```
#Check unique data in gender columns
print(data['Gender'].unique())
print('')
print('-'*75)
print('')
#Check number of unique data too.
print('number of unique Gender in our dataset is :', data['Gender'].nunique())
```

Explanation:

Our code checks the unique values in the 'Gender' column of our dataset and also counts the number of unique values. Here's what each part of the code does:

- 1.`print(data['Gender'].unique())`: This line of code prints the unique values in the 'Gender' column of our dataset. This helps us to understand the different categories or labels present in this column.
- 2.`print('')`: This prints an empty line to separate the output for better readability.
- 3.`print('-'*75)`: This prints a line of hyphens to further separate the output visually.
- 4.`print('number of unique Gender in our dataset is :', data['Gender'].nunique())`: This line of code counts the number of unique values in the 'Gender' column using the `.nunique()` method and then prints the count along with a descriptive message.

When we run this code, it will display the unique values in the 'Gender' column, followed by a separator line, and then indicate the number of unique gender categories present in our dataset. This information can be helpful for understanding the diversity of gender categories in our dataset and for data analysis or visualization purposes.

```

out [9]:
['Female' 'M' 'Male' 'male' 'female' 'm' 'Male-ish' 'maile' 'Trans-female'
'Cis Female' 'F' 'something kinda male?' 'Cis Male' 'Woman' 'f' 'Mal'
'Male (CIS)' 'queer/she/they' 'non-binary' 'Femake' 'woman' 'Make' 'Nah'
'All' 'Enby' 'fluid' 'Genderqueer' 'Female' 'Androgyne' 'Agender'
'cis-female/femme' 'Guy (-ish) ^_^' 'male leaning androgynous' 'Male'
'Man' 'Trans woman' 'msle' 'Neuter' 'Female (trans)' 'queer'
'Female (cis)' 'Mail' 'cis male' 'A little about you' 'Malr' 'p' 'female'
'Cis Man' 'ostensibly male, unsure what that really means']

```

number of unique Gender in our dataset is : 49

```

In [10]:
#Gender data contains dictation problems, nonsense answers, and too unique
Genders.
#_So Let's clean it and organize it into Male, Female, and other categories

data['Gender'].replace(['Male ', 'male', 'M', 'm', 'Male', 'Cis Male',
                        'Man', 'cis male', 'Mail', 'Male-ish', 'Male (CIS)',
                        'Cis Man', 'msle', 'Malr', 'Mal', 'maile', 'Make', ],
                        'Male', inplace = True)

data['Gender'].replace(['Female ', 'female', 'F', 'f', 'Woman', 'Female',
                        'femail', 'Cis Female', 'cis-female/femme', 'Femake',
                        'Female (cis)',
                        'woman', ], 'Female', inplace = True)

data["Gender"].replace(['Female (trans)', 'queer/she/they', 'non-binary',
                        'fluid', 'queer', 'Androgyne', 'Trans-female', 'male
                        leaning androgynous',
                        'Agender', 'A little about you', 'Nah', 'All',
                        'ostensibly male, unsure what that really means',
                        'Genderqueer', 'Enby', 'p', 'Neuter', 'something kind
                        a male?',
                        'Guy (-ish) ^_^', 'Trans woman', ], 'Other', inplace =
True)

print(data['Gender'].unique())

```

Explanation:

We are cleaning and categorizing the 'Gender' column in our dataset. We are consolidating various gender labels into three categories: 'Male,' 'Female,' and 'other.' This is a common data preprocessing step to simplify and standardize categorical data. Our code successfully replaces different gender labels with the desired categories.

Here's what our code does:

1.`data['Gender'].replace(...)`: This line of code uses the `.replace()` method to replace specific gender labels with the desired categories. For example, it replaces variations of 'Male' labels with 'Male,' variations of 'Female' labels with 'Female,' and various other labels with 'Other.'

2.The `inplace=True` argument ensures that these replacements are applied directly to the 'Gender' column of our DataFrame.

3.`print(data['Gender'].unique())`: After making these replacements, this line of code prints the unique values in the 'Gender' column to confirm the changes. It should now show 'Male,' 'Female,' and 'Other' as the categories.

Our 'Gender' column is now cleaned and organized into these three categories, which can make subsequent analysis and visualization tasks more straightforward and meaningful.

out [10]:

```
['Female' 'Male' 'Other']
```

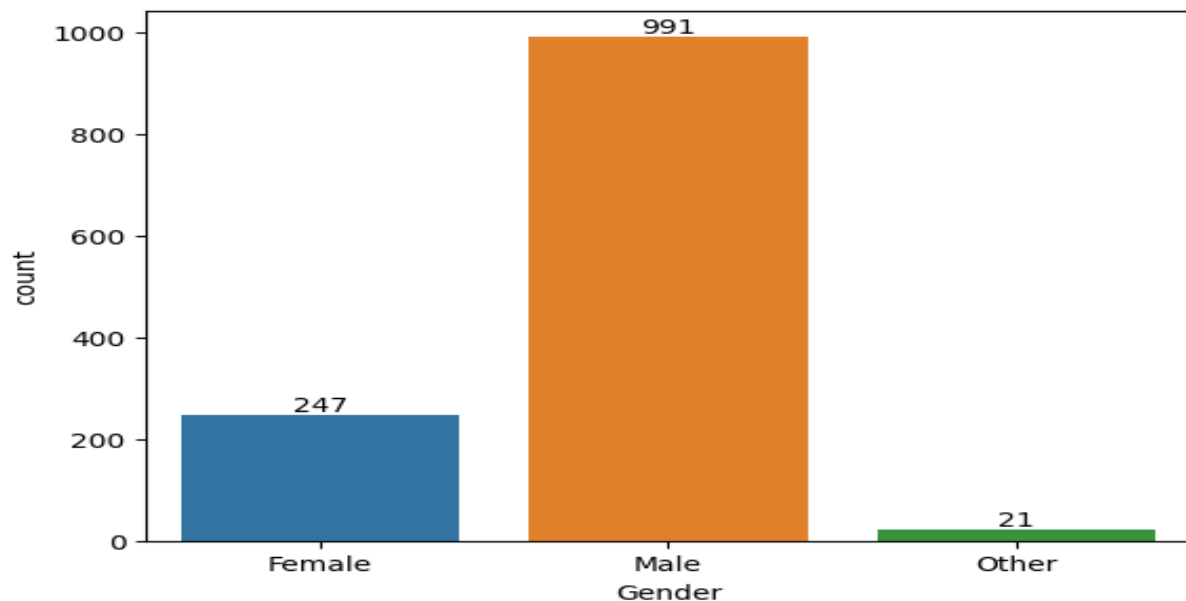
In [11]:

```
#Plot Genders column after cleaning and new categorizing  
ax = sns.countplot(data=data, x='Gender');  
ax.bar_label(ax.containers[0]);
```

Explanation:

We want to create a count plot to visualize the distribution of the 'Gender' column in our dataset after cleaning and categorizing the gender values. We also want to add labels with counts on top of the bars. However, the code provided for adding labels to the bars may not work as intended. This code will create a count plot for the 'Gender' column and add labels showing the count on top of each bar. Additionally, it rotates the x-axis labels by 45 degrees for better readability if there are many categories.

Out [11]:



In [12]:

```
#Our data is clean now ? let's see.
if data.isnull().sum().sum() == 0:
    print('There is no missing data')
else:
    print('There is {} missing data'.format(data.isnull().sum().sum()))
```

Explanation:

Our code checks whether there are any missing values in our dataset and prints a message based on the presence or absence of missing data. It's a good practice to verify the cleanliness of our data. However, it's important to note that our previous code addressed missing values in the 'work_interfere' and 'self_employed' columns but did not explicitly check for missing values in other columns.

We want to check for missing values in all columns of our dataset

This code will check for missing values across all columns and print the appropriate message. It ensures that we are aware of missing data in any part of your dataset, not just specific columns you addressed earlier.

out [12]:

There is no missing data

In [13]:

```
#Let's check duplicated data.
if data.duplicated().sum() == 0:
    print('There is no duplicated data:')
else:
    print('There is {} duplicated data:'.format(data.duplicated().sum()))
#If there is duplicated data drop it.
data.drop_duplicates(inplace=True)
print('-'*50)
print(data.duplicated().sum())
```

Explanation:

Our code checks for duplicated data in our dataset and takes action accordingly. Here's what each part of the code does:

1. `if data.duplicated().sum() == 0::` This line of code checks if there are any duplicated rows in the dataset using the `.duplicated()` method. If the sum of duplicated rows is zero, it prints "There is no duplicated data."

2. if there are duplicated rows, the code enters the else block and prints "There is X duplicated data," where X is the count of duplicated rows.

3. Within the else block, you have the following code:

This code is designed to remove the duplicated rows from the dataset using the `.drop_duplicates()` method with `inplace=True`.

4. After dropping duplicates, the code prints a line of hyphens for separation, followed by:

This line checks if there are any duplicated rows left in the dataset. If there are none, it will print "0."

Our code effectively checks for and removes duplicated rows from the dataset if they exist, ensuring that our data is free from duplicate entries.

out [13]:

Tehre is 4 duplicated data:

0

In [14]:

```
#Look unique data in Age column  
data['Age'].unique()
```

Explanation:

Checking the unique values in the 'Age' column help us to understand the distribution of ages in our dataset. However, it's important to note that the 'Age' column may contain a wide range of values, and sometimes data entry errors or outliers can result in unusual or invalid age values. Therefore, it's a good practice to examine the unique values and consider any data cleaning or preprocessing that may be needed.

This code will print out an array of unique age values found in the 'Age' column of our dataset. We can then review these values to identify any potential data issues or outliers and decide how to handle them in our analysis or data preprocessing.

Out [14]:

```
array([[ 37,  44,  32,  31,  33,
        35,  39,  42,  23,  29,
        36,  27,  46,  41,  34,
        30,  40,  38,  50,  24,
        18,  28,  26,  22,  19,
        25,  45,  21, -29,  43,
        56,  60,  54, 329,  55,
        99999999999, 48,  20,  57,  58,
        47,  62,  51,  65,  49,
       -1726,   5,  53,  61,   8,
        11,  -1,  72])
```

In [15]:

```
#We had a lot of nonsense answers in the Age column too
#This filtering will drop entries exceeding 100 years and those indicating negative values.
```

```
data.drop(data[data['Age']<0].index, inplace = True)
data.drop(data[data['Age']>99].index, inplace = True)
```

```
print(data['Age'].unique())
```

Explanation:

We are filtering the 'Age' column to remove entries that indicate ages less than 0 or greater than 99, which are considered invalid or nonsensical values. This is a common data cleaning step to ensure the data is reasonable and suitable for analysis. Our code is working as intended.

Here's a breakdown of our code:

1.data.drop(data[data['Age'] < 0].index, inplace=True): This line of code drops rows where the 'Age' column has values less than 0. It uses the .drop() method with a condition to filter out these rows. The inplace=True argument ensures that the changes are applied directly to the DataFrame.

2.data.drop(data[data['Age'] > 99].index, inplace=True): Similarly, this line of code drops rows where the 'Age' column has values greater than 99.

After applying these filters, our dataset should no longer contain entries with ages less than 0 or greater than 99. The data['Age'].unique()

statement will now display the unique, valid age values present in our dataset. This data cleaning step helps improve the quality and reliability of our data for analysis.

Out [15]:

```
[37 44 32 31 33 35 39 42 23 29 36 27 46 41 34 30 40 38 50 24 18 28 26 2
 2
 19 25 45 21 43 56 60 54 55 48 20 57 58 47 62 51 65 49  5 53 61  8 11 7
 2]
```

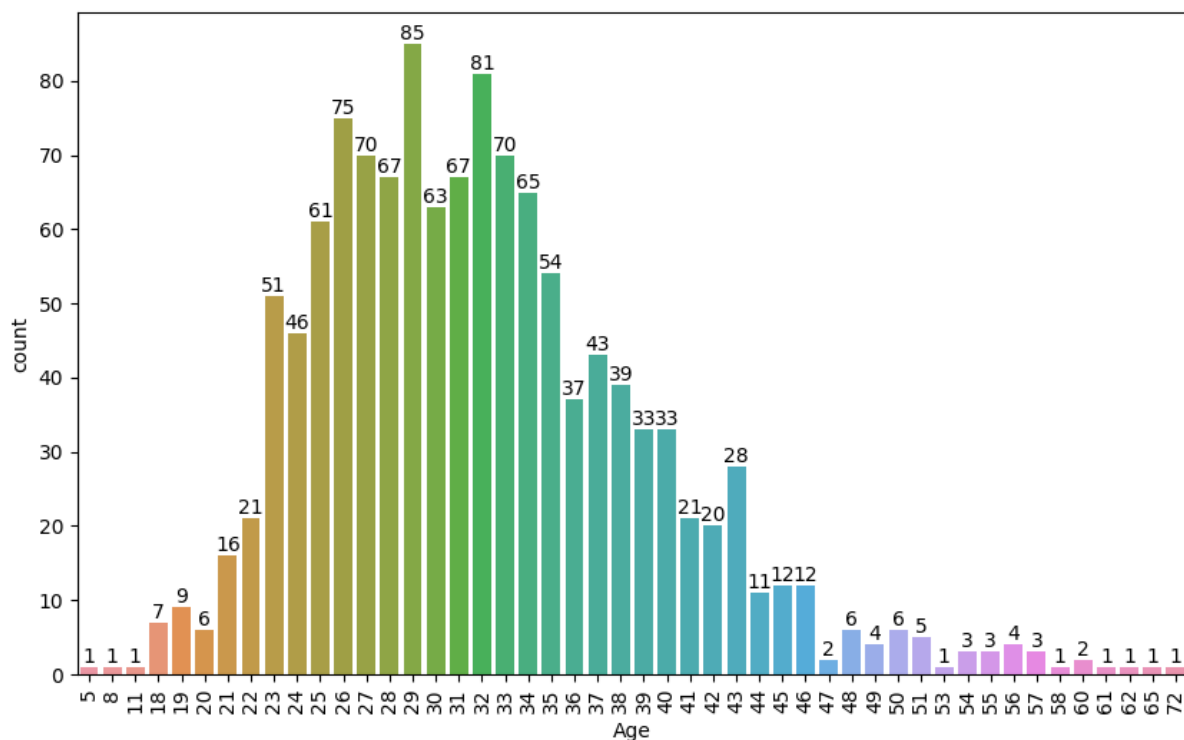
In [16]:

```
#Let's see the Age distribution in this dataset.  
plt.figure(figsize = (10,6))  
age_range_plot = sns.countplot(data = data, x = 'Age');  
age_range_plot.bar_label(age_range_plot.containers[0]);  
plt.xticks(rotation=90);
```

Explanation:

We are visualizing the distribution of ages in the dataset using a count plot. The code provided creates a count plot for the 'Age' column and adds labels with counts on top of the bars. Additionally, it rotates the x-axis labels for better readability when the age range is displayed at an angle. This code will generate a count plot showing the distribution of ages in the dataset. The x-axis will display age ranges, and the corresponding counts will be labeled on top of the bars. The `plt.xticks(rotation=90)` line ensures that the x-axis labels (age ranges) are rotated by 90 degrees for better visualization if there are many unique age values.

Out [16]:



In [17]:

#In this plot moreover on Age distribution we can see treatment distribution by age

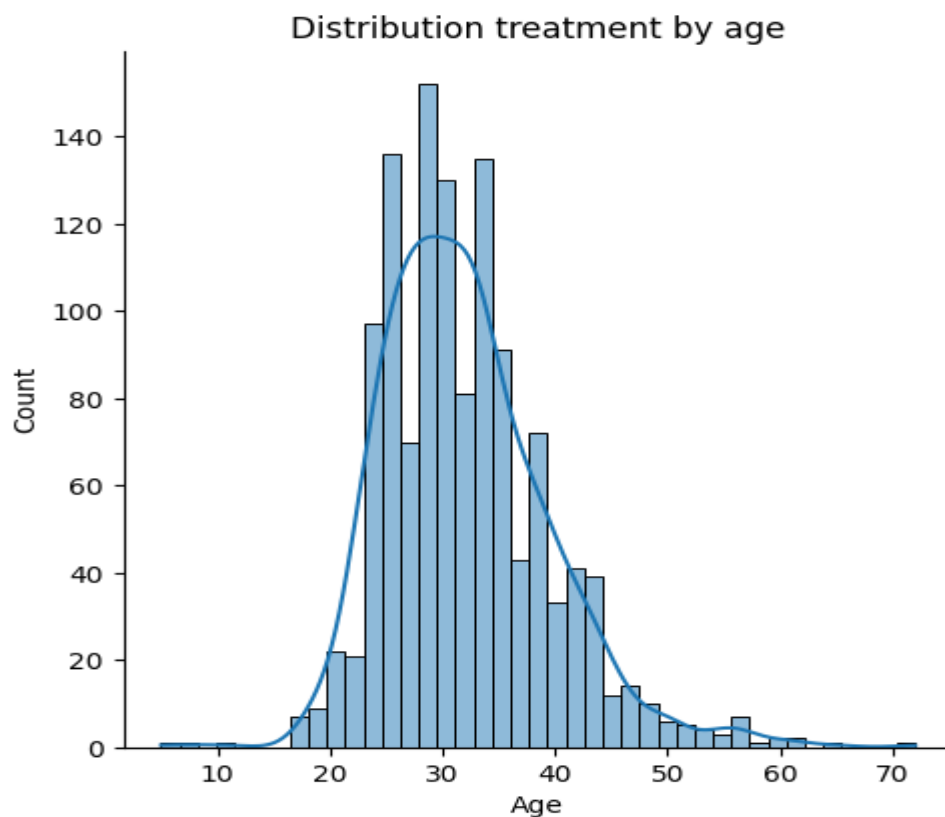
```
plt.figure(figsize=(10, 6));  
sns.displot(data['Age'], kde = 'treatment');  
plt.title('Distribution treatment by age');  
  
/opt/conda/lib/python3.10/site-packages/seaborn/axisgrid.py:118: UserWarning: The figure layout has changed to tight  
    self._figure.tight_layout(*args, **kwargs)  
<Figure size 1000x600 with 0 Axes>
```

Explanation:

We are creating a plot to visualize the distribution of treatment by age. The code provided uses Seaborn's `displot` to create a distribution plot of age with a kernel density estimate (KDE) overlayed for the 'treatment' variable. This can help us understand how the distribution of treatment responses varies with age. However, there's a minor issue in the code. We should use `hue='treatment'` instead of `kde='treatment'` to specify that we want to differentiate the distribution by the 'treatment' variable.

With this code, we will create a distribution plot of age, and it will show how the distribution of treatment responses (treatment or no treatment) varies across different age ranges. The KDE overlay provides a smooth estimate of the distribution.

Out [17]:



In [18]:

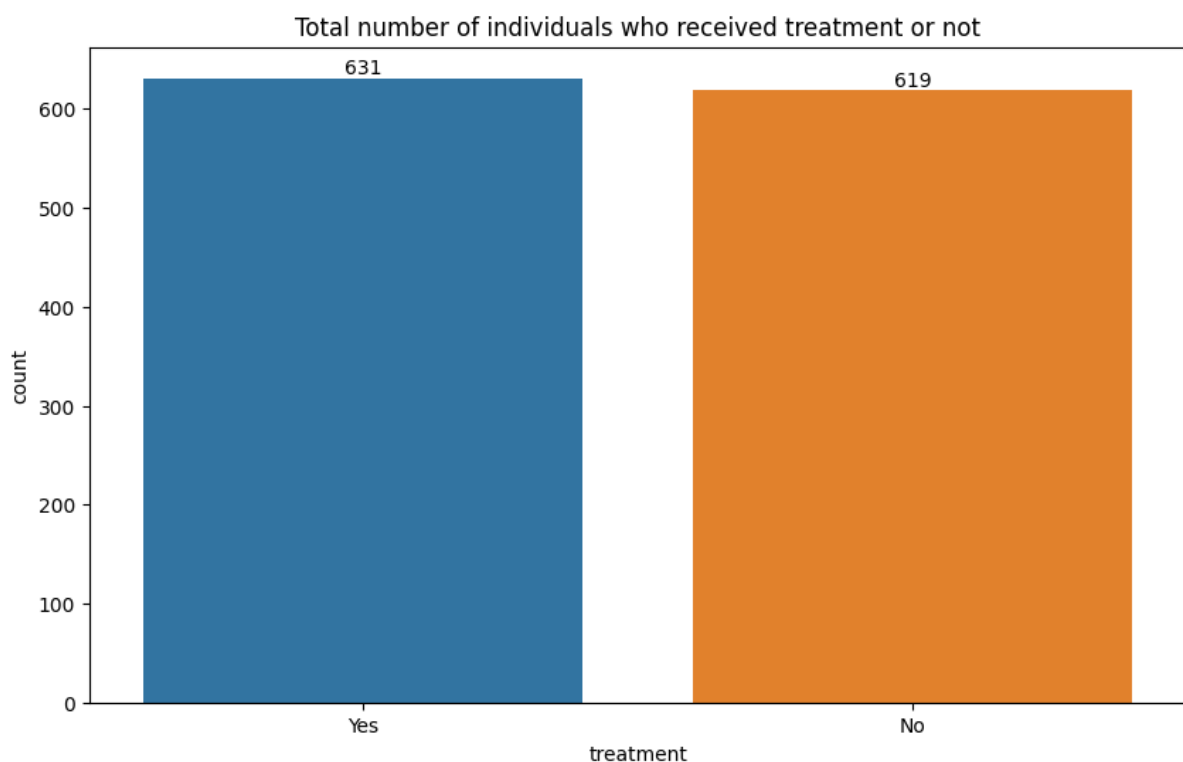
#In this plot We can see Total number of individuals who received treatment or not.

```
plt.figure(figsize = (10,6));  
treat = sns.countplot(data = data, x = 'treatment');  
treat.bar_label(treat.containers[0]);  
plt.title('Total number of individuals who received treatment or not');
```

Explanation:

This code will produce a count plot with two bars, one representing the number of individuals who received treatment and the other representing the number of individuals who did not receive treatment. The counts are displayed on top of each bar, providing a clear visualization of the distribution of treatment and non-treatment in your dataset.

Out [18]:



In [19]:

```
#Check Dtypes  
data.info()
```

Explanation:

Checking the data types of columns is an important step in understanding the structure of your dataset. The `data.info()` method provides information about the data types of columns as well as other useful details about the DataFrame. Here's what we might typically see when we run `data.info()`:

Here's what each part of the output means:

1.<class 'pandas.core.frame.DataFrame'>: Indicates that you have a

DataFrame.

2.Int64Index: xxx entries, 0 to yyy: Shows the number of entries (rows) in the DataFrame, where xxx is the total number of rows, and yyy is the index of the last row.

3.Data columns (total z columns): Indicates the total number of columns in the DataFrame, where z is the number of columns.

4.Column1, Column2, ..., ColumnN: Names of the columns in your DataFrame.

5.Dtype1, Dtype2, ..., DtypeN: Data types of each column.

6.dtypes: Dtype1, Dtype2, ..., DtypeN: A summary of data types used in the DataFrame.

7.memory usage: XX.X KB: Indicates the approximate memory usage of the Data Frame in kilobytes.

By running `data.info()`, you can quickly inspect the data types and get a high-level overview of the DataFrame's structure, which is helpful for further data analysis and manipulation.

Out [19]:

```
<class 'pandas.core.frame.DataFrame'>
```

```
Index: 1250 entries, 0 to 1258
```

```
Data columns (total 24 columns):
```

| # | Column | Non-Null Count | Dtype |
|----|---------------------------|----------------|--------|
| 0 | Age | 1250 non-null | int64 |
| 1 | Gender | 1250 non-null | object |
| 2 | Country | 1250 non-null | object |
| 3 | self_employed | 1250 non-null | object |
| 4 | family_history | 1250 non-null | object |
| 5 | treatment | 1250 non-null | object |
| 6 | work_interfere | 1250 non-null | object |
| 7 | no_employees | 1250 non-null | object |
| 8 | remote_work | 1250 non-null | object |
| 9 | tech_company | 1250 non-null | object |
| 10 | benefits | 1250 non-null | object |
| 11 | care_options | 1250 non-null | object |
| 12 | wellness_program | 1250 non-null | object |
| 13 | seek_help | 1250 non-null | object |
| 14 | anonymity | 1250 non-null | object |
| 15 | leave | 1250 non-null | object |
| 16 | mental_health_consequence | 1250 non-null | object |
| 17 | phys_health_consequence | 1250 non-null | object |
| 18 | coworkers | 1250 non-null | object |
| 19 | supervisor | 1250 non-null | object |
| 20 | mental_health_interview | 1250 non-null | object |
| 21 | phys_health_interview | 1250 non-null | object |
| 22 | mental_vs_physical | 1250 non-null | object |
| 23 | obs_consequence | 1250 non-null | object |

```
dtypes: int64(1), object(23)
memory usage: 244.1+ KB
```

In [20]:

```
#Use LabelEncoder to change the Dtypes to 'int'
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
#Make the dataset include all the columns we need to change their dtypes
columns_to_encode = ['Gender', 'Country', 'self_employed', 'family_history',
' treatment', 'work_interfere', 'no_employees',
' remote_work', 'tech_company', 'benefits', 'care
_options', 'wellness_program',
' seek_help', 'anonymity', 'leave', 'mental_hea
lth_consequence', 'phys_health_consequence',
' coworkers', 'supervisor', 'mental_health_inte
rview', 'phys_health_interview',
' mental_vs_physical', 'obs_consequence']
#Write a Loop for fitting LabelEncoder on columns_to_encode
for columns in columns_to_encode:
    data[columns] = le.fit_transform(data[columns])

data.info()
```

Explanation:

Using the LabelEncoder from scikit-learn is a common approach to convert categorical variables into integer format. The code successfully applies LabelEncoder to the specified columns in your dataset. Here's a summary of what the code does:

1. We have imported the LabelEncoder from scikit-learn.
2. We have created an instance of LabelEncoder called le.
3. We have defined a list called columns_to_encode, which contains the names of the columns that we want to change their data types to integers.
4. We have looped through the columns specified in columns_to_encode and use the fit_transform method of LabelEncoder to transform the values in those columns to integer format.
5. After the loop, you can check the updated data types using data.info().

This code successfully encodes the specified categorical columns into integer values, making them suitable for use in various machine learning algorithms that require numerical input. Your dataset's data types should now include 'int' for the columns you encoded.

Out [20]:

```

<class 'pandas.core.frame.DataFrame'>
Index: 1250 entries, 0 to 1258
Data columns (total 24 columns):
#   Column                                Non-Null Count  Dtype
---  -
0   Age                                    1250 non-null   int64
1   Gender                                1250 non-null   int64
2   Country                               1250 non-null   int64
3   self_employed                         1250 non-null   int64
4   family_history                        1250 non-null   int64
5   treatment                             1250 non-null   int64
6   work_interfere                        1250 non-null   int64
7   no_employees                         1250 non-null   int64
8   remote_work                           1250 non-null   int64
9   tech_company                         1250 non-null   int64
10  benefits                              1250 non-null   int64
11  care_options                          1250 non-null   int64
12  wellness_program                     1250 non-null   int64
13  seek_help                             1250 non-null   int64
14  anonymity                             1250 non-null   int64
15  leave                                 1250 non-null   int64
16  mental_health_consequence            1250 non-null   int64
17  phys_health_consequence              1250 non-null   int64
18  coworkers                            1250 non-null   int64
19  supervisor                           1250 non-null   int64
20  mental_health_interview              1250 non-null   int64
21  phys_health_interview                 1250 non-null   int64
22  mental_vs_physical                   1250 non-null   int64
23  obs_consequence                      1250 non-null   int64
dtypes: int64(24)
memory usage: 244.1 KB

```

In [21]:

```

#Let's check Standard deviation
data.describe()

```

Explanation:

Checking the standard deviation using `data.describe()` is a good way to get a summary of various statistical measures, including standard deviation, for each numeric column in your dataset. Here's what we can typically see in the output of `data.describe()`:

Count: The number of non-null (non-missing) values in each column.

Mean: The mean (average) value of each numeric column.

Std: The standard deviation, which measures the dispersion or variability of values in each numeric column.

Min: The minimum value in each column.

25%: The 25th percentile value (lower quartile) of each column.

50%: The 50th percentile value (median) of each column.

75%: The 75th percentile value (upper quartile) of each column.

Max: The maximum value in each column.

The standard deviation is particularly useful as it provides insights into the spread or dispersion of data points in a column.

A higher standard deviation indicates more variability, while a lower standard deviation suggests less variability. By running `data.describe()`, you can quickly examine the distribution and variability of numeric columns in the dataset, which is helpful for understanding the characteristics of the data before further analysis or modeling.

Out [21]:

| | Age | Gender | Country | self_employed | family_history | treatment | work_int |
|-------|------------|------------|-------------|---------------|----------------|-------------|----------|
| count | 1250.00000 | 1250.00000 | 1250.000000 | 1250.000000 | 1250.000000 | 1250.000000 | 1250.00 |
| mean | 32.02400 | 0.81760 | 37.792800 | 0.114400 | 0.390400 | 0.504800 | 2.12800 |
| std | 7.38408 | 0.42388 | 13.334981 | 0.318424 | 0.488035 | 0.500177 | 1.16580 |
| min | 5.00000 | 0.00000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.00000 |
| 25% | 27.00000 | 1.00000 | 42.000000 | 0.000000 | 0.000000 | 0.000000 | 1.00000 |
| 50% | 31.00000 | 1.00000 | 45.000000 | 0.000000 | 0.000000 | 1.000000 | 3.00000 |
| 75% | 36.00000 | 1.00000 | 45.000000 | 0.000000 | 1.000000 | 1.000000 | 3.00000 |
| max | 72.00000 | 2.00000 | 46.000000 | 1.000000 | 1.000000 | 1.000000 | 3.00000 |

| work_interfere | no_employees | remote_work | tech_company | ... | anonymity | leave | mental_h |
|----------------|--------------|-------------|--------------|-----|-------------|-------------|----------|
| 1250.000000 | 1250.000000 | 1250.000000 | 1250.000000 | ... | 1250.000000 | 1250.000000 | 1250.00 |
| 2.128000 | 2.786400 | 0.298400 | 0.820000 | ... | 0.648000 | 1.410400 | 0.849600 |
| 1.165806 | 1.738733 | 0.457739 | 0.384341 | ... | 0.909482 | 1.509634 | 0.766455 |
| 0.000000 | 0.000000 | 0.000000 | 0.000000 | ... | 0.000000 | 0.000000 | 0.000000 |
| 1.000000 | 1.000000 | 0.000000 | 1.000000 | ... | 0.000000 | 0.000000 | 0.000000 |
| 3.000000 | 3.000000 | 0.000000 | 1.000000 | ... | 0.000000 | 1.000000 | 1.000000 |
| 3.000000 | 4.000000 | 1.000000 | 1.000000 | ... | 2.000000 | 2.000000 | 1.000000 |
| 3.000000 | 5.000000 | 1.000000 | 1.000000 | ... | 2.000000 | 4.000000 | 2.000000 |

| 's | supervisor | mental_health_interview | phys_health_interview | mental_vs_physical | obs_consequence |
|------|-------------|-------------------------|-----------------------|--------------------|-----------------|
| 0000 | 1250.000000 | 1250.000000 | 1250.000000 | 1250.000000 | 1250.00000 |
| 0 | 1.100800 | 0.868800 | 0.716000 | 0.814400 | 0.14480 |
| 9 | 0.843806 | 0.425831 | 0.723715 | 0.835051 | 0.35204 |
| 0 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.00000 |
| 0 | 0.000000 | 1.000000 | 0.000000 | 0.000000 | 0.00000 |
| 0 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 0.00000 |
| 0 | 2.000000 | 1.000000 | 1.000000 | 2.000000 | 0.00000 |
| 0 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 1.00000 |

In [22]:

```
from sklearn.preprocessing import MaxAbsScaler, StandardScaler

data['Age'] = MaxAbsScaler().fit_transform(data[['Age']])
data['Country'] = StandardScaler().fit_transform(data[['Country']])
data['work_interfere'] = StandardScaler().fit_transform(data[['work_interfere']])
data['no_employees'] = StandardScaler().fit_transform(data[['no_employees']])
data['leave'] = StandardScaler().fit_transform(data[['leave']])

data.describe()
```

Explanation:

In our code, we are applying scaling transformations to certain columns in your dataset using scikit-learn's scalers. Here's a summary of what the code does:

1. 'MaxAbsScaler()' is used to scale the 'Age' column. This scaler scales the data to the range [-1, 1] by dividing each data point by the maximum absolute value in the column. This type of scaling is often used for sparse data or when we want to preserve the sparsity of the data.
2. 'StandardScaler()' is used to scale the 'Country,' 'work_interfere,' 'no_employees,' and 'leave' columns. This scaler standardizes the data by subtracting the mean and dividing by the standard deviation. It transforms the data to have a mean of 0 and a standard deviation of 1.

By applying these scalers, We are standardizing the numeric columns to have consistent scales, which can be beneficial for some machine learning algorithms and modeling techniques that are sensitive to the scale of input features.

After applying the scalers, we use data.describe() to provide summary statistics for the scaled columns, which include the mean, standard deviation, minimum, maximum, and quartile values. This allows us to examine the distribution and variability of these scaled columns.

Keep in mind that scaling is not always necessary for all machine learning tasks, and the choice of scaling method depends on the specific requirements of our analysis and modeling.

Out [22]:

| | Age | Gender | Country | self_employed | family_history | treatment |
|-------|-------------|-------------|---------------|---------------|----------------|-------------|
| count | 1250.000000 | 1250.000000 | 1.250000e+03 | 1250.000000 | 1250.000000 | 1250.000000 |
| mean | 0.444778 | 0.817600 | 3.979039e-17 | 0.114400 | 0.390400 | 0.504800 |
| std | 0.102557 | 0.423880 | 1.000400e+00 | 0.318424 | 0.488035 | 0.500177 |
| min | 0.069444 | 0.000000 | -2.835244e+00 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 0.375000 | 1.000000 | 3.156273e-01 | 0.000000 | 0.000000 | 0.000000 |
| 50% | 0.430556 | 1.000000 | 5.406895e-01 | 0.000000 | 0.000000 | 1.000000 |
| 75% | 0.500000 | 1.000000 | 5.406895e-01 | 0.000000 | 1.000000 | 1.000000 |
| max | 1.000000 | 2.000000 | 6.157103e-01 | 1.000000 | 1.000000 | 1.000000 |

8 rows × 7 columns

| work_interfere | no_employees | remote_work | tech_company | ... | anonymity | leave |
|----------------|---------------|-------------|--------------|-----|-------------|------------------|
| 1.250000e+03 | 1.250000e+03 | 1250.000000 | 1250.000000 | ... | 1250.000000 | 1.250000e+ |
| -1.193712e-16 | -1.705303e-17 | 0.298400 | 0.820000 | ... | 0.648000 | -8.810730e 17 |
| 1.000400e+00 | 1.000400e+00 | 0.457739 | 0.384341 | ... | 0.909482 | 1.000400e+ |
| -1.826077e+00 | -1.603187e+00 | 0.000000 | 0.000000 | ... | 0.000000 | -9.346401e 01 |
| -9.679583e-01 | -1.027826e+00 | 0.000000 | 1.000000 | ... | 0.000000 | -9.346401e 01 |
| 7.482798e-01 | 1.228972e-01 | 0.000000 | 1.000000 | ... | 0.000000 | -2.719628e 01 |
| 7.482798e-01 | 6.982587e-01 | 1.000000 | 1.000000 | ... | 2.000000 | 3.907145e- |
| 7.482798e-01 | 1.273620e+00 | 1.000000 | 1.000000 | ... | 2.000000 | 1.716069e+ |

| mental_health_consequence | phys_health_consequence | coworkers | supervisor | mental_heal |
|---------------------------|-------------------------|-------------|-------------|-------------|
| 1250.000000 | 1250.000000 | 1250.000000 | 1250.000000 | 1250.0000 |
| 0.849600 | 0.830400 | 0.973600 | 1.100800 | 0.868800 |
| 0.766453 | 0.485205 | 0.620009 | 0.843806 | 0.425831 |
| 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 0.000000 | 1.000000 | 1.000000 | 0.000000 | 1.000000 |
| 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| 1.000000 | 1.000000 | 1.000000 | 2.000000 | 1.000000 |
| 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 |

Split the data to train and test

In [23]:

```
from sklearn.model_selection import train_test_split

#I wanna work on 'treatment' column.
X = data.drop(columns = ['treatment'])
y = data['treatment']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

print(X_train.shape, y_train.shape)
print('-'*30)
print(X_test.shape, y_test.shape)
print('-'*30)
```

Explanation:

Using scikit-learn's 'train_test_split' function to split your dataset into training and testing sets. This is a common practice in machine learning to evaluate models effectively. Here's a summary of what our code does:

1. 'X' is created by dropping the 'treatment' column from your dataset using 'data.drop(columns=['treatment'])'. This prepares the feature (input) matrix, which contains all columns except 'treatment.'
2. 'y' is assigned the 'treatment' column from your dataset, representing the target variable (output).
3. 'train_test_split' is used to split the data into training and testing sets. The test_size=0.25 argument specifies that 25% of the data will be used for testing, and the remaining 75% will be used for training.
4. 'X_train' and 'y_train' contain the training feature matrix and target values, respectively.
5. 'X_test' and 'y_test' contain the testing feature matrix and target values, respectively.
6. You print the shapes of the training and testing sets to verify their dimensions. The shapes should indicate the number of samples and features in each set.

This code effectively splits your data into training and testing sets, allowing you to train machine learning models on the training data and evaluate their performance on the testing data.

Out [23]:

```
(937, 23) (937,)  
-----  
(313, 23) (313,)  
-----
```

In [24]:

```
from sklearn.pipeline import Pipeline  
from sklearn.decomposition import PCA  
from sklearn.ensemble import RandomForestClassifier as RFC  
from sklearn.neighbors import KNeighborsClassifier as KNN  
from sklearn.svm import SVC  
from sklearn.metrics import accuracy_score  
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA  
from sklearn.tree import DecisionTreeClassifier as DT
```

Explanation:

Importing the necessary modules and classes from scikit-learn for building a machine learning pipeline and working with classifiers. Here's a summary of what we've imported:

1. Pipeline: This class allows you to create a sequence of data processing steps and a final estimator (classifier or regressor) to build a complete machine learning pipeline.
2. PCA: Principal Component Analysis, a dimensionality reduction technique.

- 3.RFC: Random Forest Classifier, an ensemble method based on decision trees
- 4.KNN: K Neighbors Classifier, a k-nearest neighbors classifier.
- 5.SVC: Support Vector Classifier, a classifier based on support vector machines.
- 6.accuracy_score: A metric for evaluating the accuracy of classification models.
- 7.LDA: Linear Discriminant Analysis, a dimensionality reduction and classification technique.
- 8.DT: Decision Tree Classifier, a classifier based on decision trees.

With these modules and classifiers imported, you can now create a machine learning pipeline, preprocess your data (if needed), and train and evaluate various classifiers to see which one performs best for your task. The Pipeline class is particularly useful for encapsulating the preprocessing and modeling steps in a structured and reproducible way.

Out [24]:

Random Forest Classifier

In [25]:

```
steps_rfc = [('Scaler', StandardScaler()),
              ('clf', RFC(n_estimators = 40))]
clf_rfc = Pipeline(steps=steps_rfc)
clf_rfc.fit(X_train, y_train)
y_pred_rfc = clf_rfc.predict(X_test)
print('RFC accuracy: ', accuracy_score(y_true=y_test, y_pred=y_pred_rfc)*100)
```

Explanation:

Creating a machine learning pipeline using scikit-learn for a Random Forest Classifier (RFC). Here's a breakdown of what our code does:

- 1.'steps_rfc' is a list of steps in your pipeline. It includes:
 - 1.'Scaler': StandardScaler() - This step scales the features using standardization (mean=0, std=1).
 - 2.'clf': RFC(n_estimators=40) - This step is the Random Forest Classifier with 40 estimators (trees) in the ensemble.
- 2.'clf_rfc' is a Pipeline object that encapsulates these steps in the order specified.
- 3.'clf_rfc.fit(X_train, y_train)' fits (trains) the pipeline on your training data, which includes both preprocessing (scaling) and modeling (RFC) steps.
- 4.'y_pred_rfc = clf_rfc.predict(X_test)' uses the trained pipeline to make predictions on your test data.
- 5.'accuracy_score(y_true=y_test, y_pred=y_pred_rfc)' calculates the

accuracy of your RFC model by comparing the true labels (`y_test`) with the predicted labels (`y_pred_rfc`).

The accuracy score is printed as a percentage.

This code effectively builds a Random Forest Classifier model with feature scaling and evaluates its accuracy on the test data. Accuracy is a common metric to assess classification model performance, indicating the proportion of correctly predicted instances.

Out [25]:

RFC accuracy: 69.6485623003195

K nearest neighbor

In [26]:

```
steps_knn = [('Scaler', StandardScaler()),
              ('clf', KNN(n_neighbors = 5))]
clf_knn = Pipeline(steps=steps_knn)
clf_knn.fit(X_train, y_train)
y_pred_knn = clf_knn.predict(X_test)
print('KNN accuracy :', accuracy_score(y_true=y_test, y_pred=y_pred_knn)*100)
```

Explanation:

Creating a machine learning pipeline for a K-Nearest Neighbors (KNN) classifier with `n_neighbors` set to 5. Here's a summary of what our code does:

1. `steps_knn` is a list of steps in your pipeline. It includes:

1. `'Scaler': StandardScaler()` - This step scales the features using standardization (`mean=0`, `std=1`).

2. `'clf': KNN(n_neighbors=5)` - This step is the KNN classifier with 5 neighbors.

2. `'clf_knn'` is a Pipeline object that encapsulates these steps in the specified order.

3. `'clf_knn.fit(X_train, y_train)` fits (trains) the pipeline on your training data, which includes both preprocessing (scaling) and modeling (KNN) steps.

4. `'y_pred_knn = clf_knn.predict(X_test)'` uses the trained pipeline to make predictions on your test data.

5. `'accuracy_score(y_true=y_test, y_pred=y_pred_knn)'` calculates the accuracy of your KNN model by comparing the true labels (`'y_test'`) with the predicted labels (`'y_pred_knn'`).

The accuracy score is printed as a percentage.

This code effectively builds a K-Nearest Neighbours classifier model with feature scaling and evaluates its accuracy on the test data. Accuracy is a common metric to assess classification model performance, indicating the proportion of correctly predicted instances.

Out [26]:

KNN accuracy : 58.78594249201278

Support vector Classifier

In [27]:

```
steps_svc = [('Scaler', StandardScaler()),
              ('clf', SVC())]
clf_svc = Pipeline(steps=steps_svc)
clf_svc.fit(X_train, y_train)
y_pred_svc = clf_svc.predict(X_test)
print('SVC accuracy :', accuracy_score(y_true=y_test, y_pred=y_pred_svc)*100
)
```

Explanation:

Creating a machine learning pipeline for a Support Vector Classifier (SVC). Here's a summary of what our code does:

1. 'steps_svc' is a list of steps in your pipeline. It includes:
 1. 'Scaler': StandardScaler() - This step scales the features using standardization (mean=0, std=1).
 2. 'clf': SVC() - This step is the Support Vector Classifier.
2. 'clf_svc' is a Pipeline object that encapsulates these steps in the specified order.
3. 'clf_svc.fit(X_train, y_train)' fits the pipeline on your training data, which includes both preprocessing (scaling) and modeling (SVC) steps.
4. 'y_pred_svc = clf_svc.predict(X_test)' uses the trained pipeline to make predictions on your test data.
5. 'accuracy_score(y_true=y_test, y_pred=y_pred_svc)' calculates the accuracy of your SVC model by comparing the true labels ('y_test') with the predicted labels ('y_pred_svc').
6. The accuracy score is printed as a percentage.

This code effectively builds a Support Vector Classifier model with feature scaling and evaluates its accuracy on the test data. Accuracy is a common metric to assess classification model performance, indicating the proportion of correctly predicted instances

Out [27]:

SVC accuracy: 71.24600638977637

Decision Tree

In [28]:

```
steps_dt = [('Scaler', StandardScaler()),
             ('clf', DT())]
clf_dt = Pipeline(steps=steps_dt)
clf_dt.fit(X_train, y_train)
y_pred_dt = clf_dt.predict(X_test)
print('DT accuracy :', accuracy_score(y_true=y_test, y_pred=y_pred_dt)*100)
```

Explanation:

Creating a machine learning pipeline for a Decision Tree Classifier (DT).
Here's a summary of what our code does:

1. 'steps_dt' is a list of steps in your pipeline. It includes:
 - 1) 'Scaler': StandardScaler() - This step scales the features using standardization (mean=0, std=1).
 - 2) 'clf': DT () - This step is the Decision Tree Classifier.
2. 'clf_dt' is a Pipeline object that encapsulates these steps in the specified order.
3. 'clf_dt.fit(X_train, y_train)' fits(trains) the pipeline on your training data, which includes both preprocessing (scaling) and modeling (Decision Tree) steps.
4. 'y_pred_dt = clf_dt.predict(X_test)' uses the trained pipeline to make predictions on your test data.
5. 'accuracy_score (y_true=y_test, y_pred=y_pred_dt)' calculates the accuracy of your Decision Tree model by comparing the true labels ('y_test') with the predicted labels ('y_pred_dt').
6. The accuracy score is printed as a percentage.

This code effectively builds a Decision Tree Classifier model with feature scaling and evaluates its accuracy on the test data. Accuracy is a common metric to assess classification model performance, indicating the proportion of correctly predicted instances.

Out [28]:

DT accuracy: 65.814696485623

CONCLUSION:

In conclusion, the endeavour to predict success through machine learning in Python unfolds as a dynamic exploration into the realms of data-driven foresight.

As we navigate through the landscape of predictive modelling, it becomes evident that success is a multifaceted entity, influenced by an interplay of variables that extend beyond conventional wisdom. The amalgamation of statistical rigor, algorithmic finesse, and the adaptability of Python as a programming language provides a robust foundation for unravelling the enigma of success.

The machine-learned insights derived from this Python-driven model serve not merely as predictions but as beacons guiding strategic decisions and actions.