IITK AIML Core: Machine Learning

Course End Project Name: ***Employee Turnover Analytics***

Submitted by: ***Saurav Ganguly***

Package imports

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score, roc_curve, auc
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.inspection import permutation_importance
```

Importing/reading the csv and load as dataframe

In [2]:
```
emp_turnover_df = pd.read_csv('HR_comma_sep.csv')
```

=========== Data Wrangling =============

In [3]:
```
# Using the len() we first find the length or the total number of rows of the dataframe
df_length = len(emp_turnover_df)
print(f'Number of records in the dataframe is: {df_length}')
```

```
Number of records in the dataframe is: 14999
```

Inspecting whether the csv data has been properly converted to dataframe, by checking the first and last 5 rows of the dataframe

In [4]:
```
emp_turnover_df.head()
```

Out[4]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	sales	salary
0	0.38	0.53	2	157	3	0	1	0	sales	low
1	0.80	0.86	5	262	6	0	1	0	sales	medium
2	0.11	0.88	7	272	4	0	1	0	sales	medium
3	0.72	0.87	5	223	5	0	1	0	sales	low
4	0.37	0.52	2	159	3	0	1	0	sales	low

In [5]:
```
emp_turnover_df.tail()
```

Out[5]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	sales	salary
14994	0.40	0.57	2	151	3	0	1	0	support	low
14995	0.37	0.48	2	160	3	0	1	0	support	low
14996	0.37	0.53	2	143	3	0	1	0	support	low
14997	0.11	0.96	6	280	4	0	1	0	support	low
14998	0.37	0.52	2	158	3	0	1	0	support	low

Checking for null or NaN values in all the columns of the data frame

In [6]:
```
emp_turnover_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14999 entries, 0 to 14998
Data columns (total 10 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   satisfaction_level     14999 non-null  float64
 1   last_evaluation        14999 non-null  float64
 2   number_project         14999 non-null  int64  
 3   average_montly_hours   14999 non-null  int64  
 4   time_spend_company     14999 non-null  int64  
 5   Work_accident          14999 non-null  int64  
 6   left                   14999 non-null  int64  
 7   promotion_last_5years  14999 non-null  int64  
 8   sales                  14999 non-null  object 
 9   salary                 14999 non-null  object 
dtypes: float64(2), int64(6), object(2)
memory usage: 1.1+ MB
```

From the **emp_turnover_df.info()** we found that out of all 14999 records there is no null or NaN values

We can also do it programatically like below

In [7]:
```python
# W check for any null values in the data frame programatically
# Here we can use any of isna() or isnull() to check for any null or NaN values in the columns
df_cols_null_info = emp_turnover_df.isna().sum()

columns_with_null = []

# Looping through the df_null_info to find the column names that have null values
for index, value in df_cols_null_info.items():
    if value > 0:
        columns_with_null.append(index)

if len(columns_with_null) == 0 :
    print(f'No columns of the dataframes have null or NaN values in {df_length} rows')
else :
    print('The following columns have null values in {df_length} rows\n')
    for column in columns_with_null :
        print(f'{column} has {df_cols_null_info[column]} null or NaN value(s)')
```

```
No columns of the dataframes have null or NaN values in 14999 rows
```

Here we can see that the dataframe contains no missing values (null or NaN) value for any of the columns

- Handling missing data is crucial for maintaining data integrity. Various approaches include **imputation** (replacing missing values with estimated values), **using default values** for missing values, or the removal of records with missing values.
- For replacing the missing values with either imputed or default values can be done by using **`fillna()`** method (with mean, median or mode), or the records with the missing values can be dropped using **`dropna()`** method
- Either of the two approaches is dependent on the requirement of the analysis

Finding out the relevant and irrelevant factors that contributed most to employee turnover at EDA

As per me the most important factors that contributed to the employee turn over at EDA are **`salary`** , **`satisfaction_level`** , **`Work_accident`** , **`last_evaluation`** , **`number_project`** , **`average_monthly_hours`** , **`time_spend_company`**

The **`sales`** column which has the department information is not of any relevence to determine employee turn over at EDA

The **`left`** column is the output column which is the dependent variable here.

Removing the **`sales`** column from the dataframe

In [8]:
```python
emp_turnover_df.drop(columns=['sales'], inplace=True)
emp_turnover_df.head()
```

Out[8]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	salary
0	0.38	0.53	2	157	3	0	1	0	low
1	0.80	0.86	5	262	6	0	1	0	medium
2	0.11	0.88	7	272	4	0	1	0	medium
3	0.72	0.87	5	223	5	0	1	0	low
4	0.37	0.52	2	159	3	0	1	0	low

Using **Encoder** to encode the categorical string values of **`salary`** column to numrical classifications

Helper function to find unique values in any dataframe columns

In [9]:
```
def findUniqueColumnValues(column_name):
    unique_values = emp_turnover_df[column_name].unique()
    unique_values
    print(f'Uniques \"{column_name}\" column values in the dataframe: {unique_values}')
```

Before applying encoder the `salary` column have categorical string data

In [10]:
```
unique_salary_values = findUniqueColumnValues('salary')
```

```
Uniques "salary" column values in the dataframe: ['low' 'medium' 'high']
```

After applying ordinal encoding to the `salary` column as it has non-numeric categorical data

In [11]:
```
# Ordinal Encoding:

# specifying the order of the categories
quality_map = {'low': 1, 'medium': 2, 'high': 3}

# performing ordinal encoding on the 'salary' column
emp_turnover_df['salary'] = emp_turnover_df['salary'].map(quality_map)

unique_salary_values = findUniqueColumnValues('salary')
```

```
Uniques "salary" column values in the dataframe: [1 2 3]
```

In [12]:
```
emp_turnover_df.head()
```

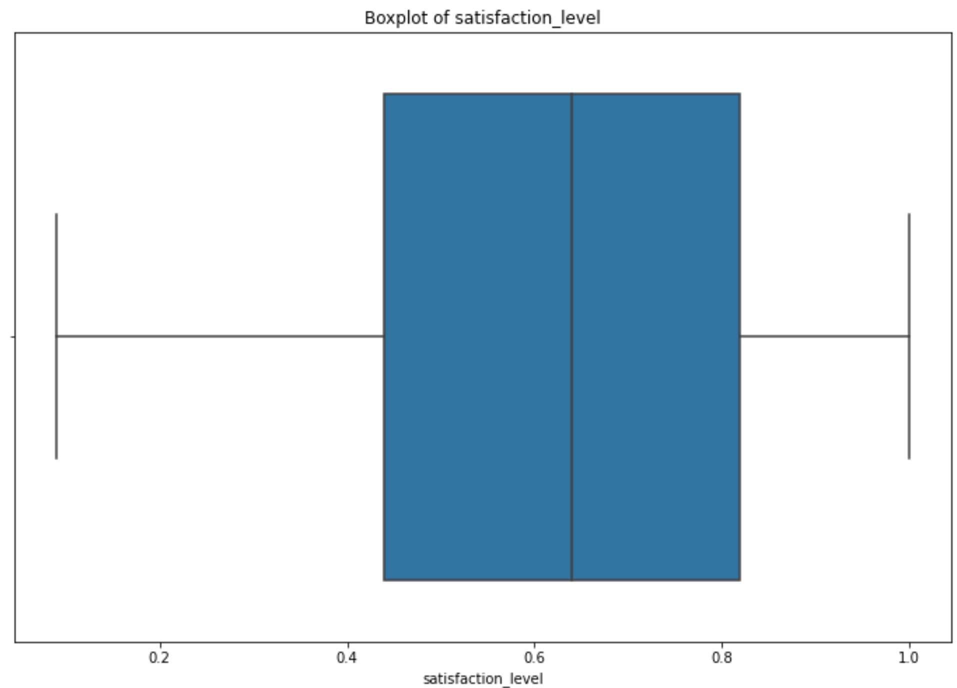
Out[12]:

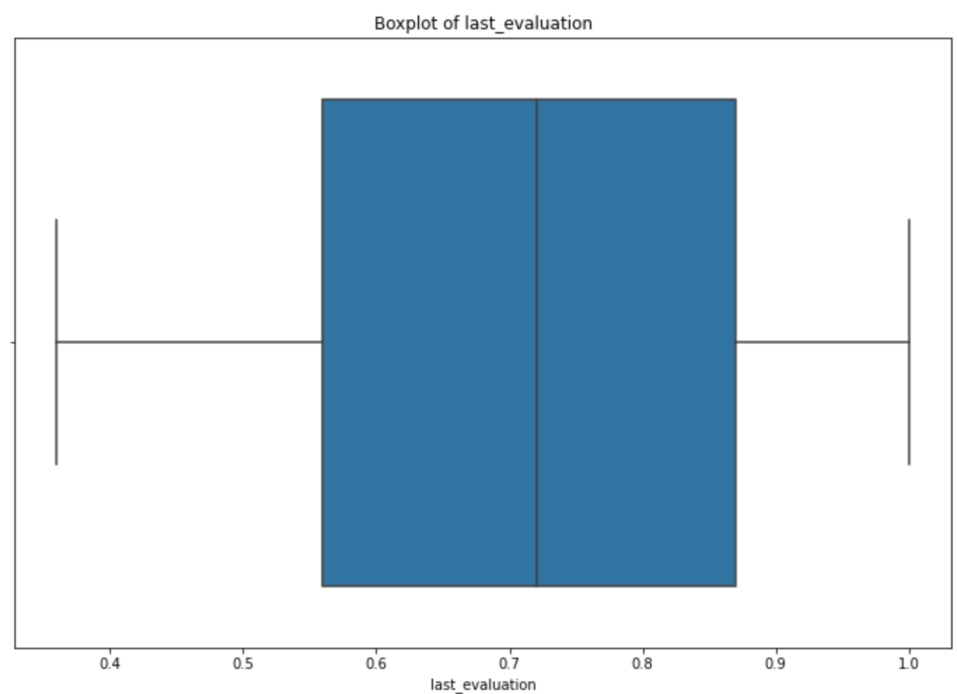
	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	salary
0	0.38	0.53	2	157	3	0	1	0	1
1	0.80	0.86	5	262	6	0	1	0	2
2	0.11	0.88	7	272	4	0	1	0	2
3	0.72	0.87	5	223	5	0	1	0	1
4	0.37	0.52	2	159	3	0	1	0	1

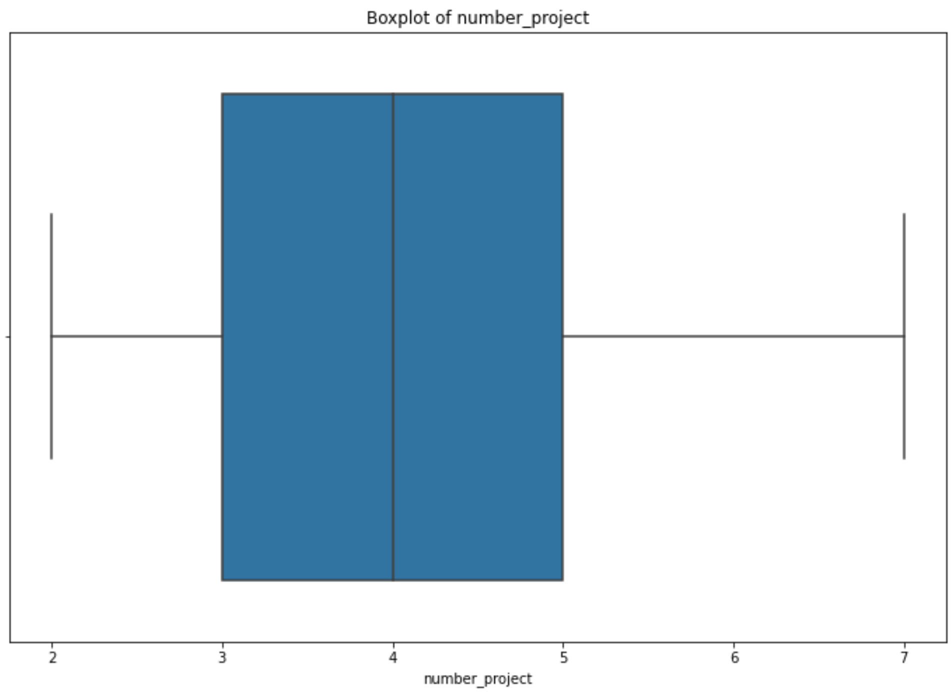
Checking for Outliers

Using Box Plot to visualize the outliers for all the independent variables or features

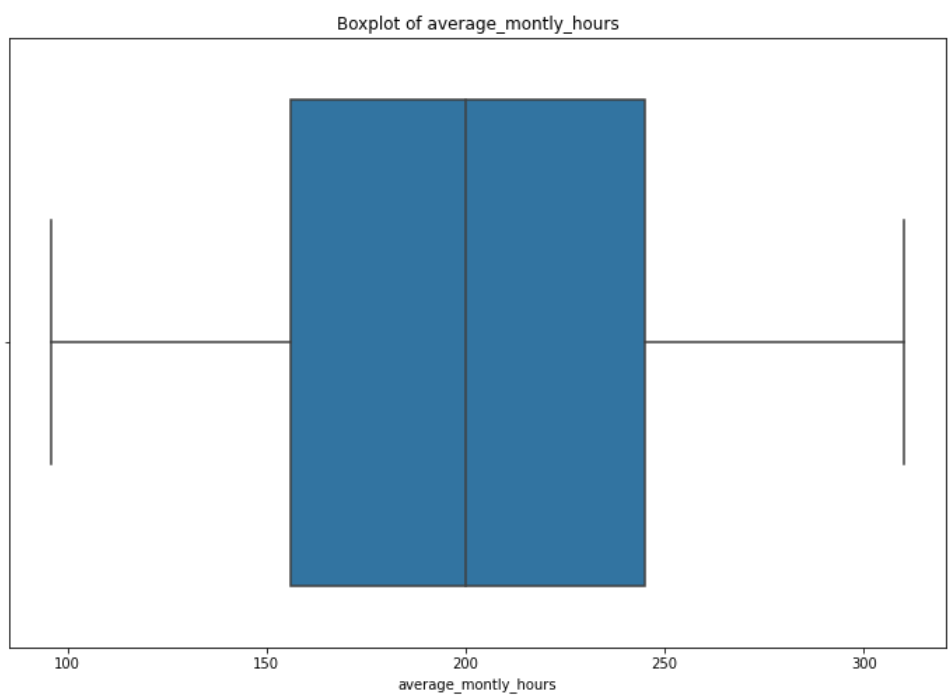
In [13]:
```
for column in emp_turnover_df.columns:
    if column != 'left':
        plt.figure(figsize=(12, 8))  # Adjust figure size as needed
        sns.boxplot(x=emp_turnover_df[column])
        plt.title(f'Boxplot of {column}')
        plt.show()
```


Boxplot of satisfaction_level
0.2
0.4
0.6
0.8
1.0
satisfaction_level

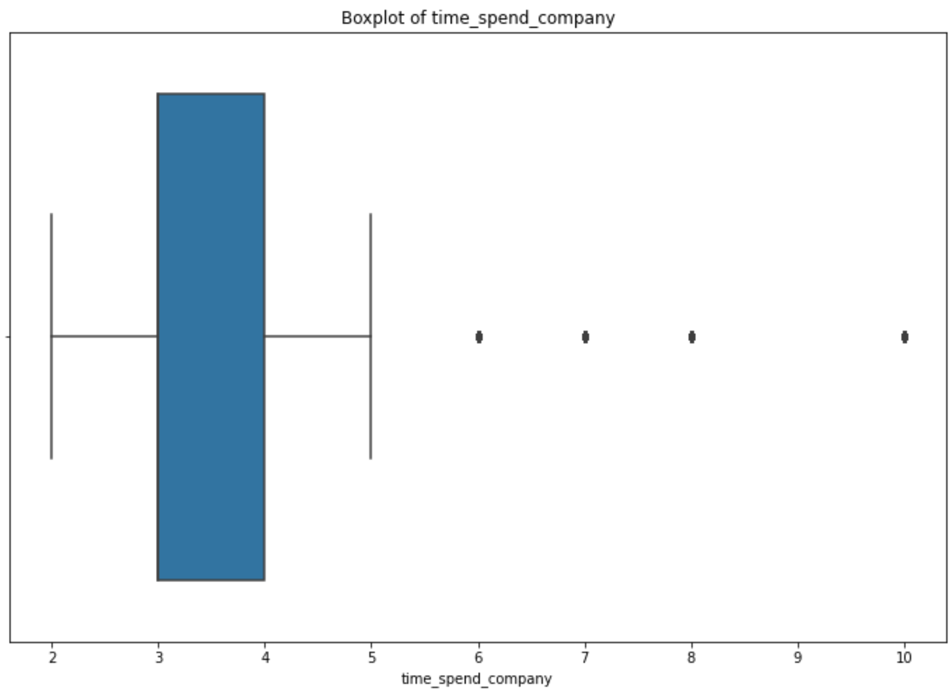

Boxplot of last_evaluation
0.4
0.5
0.6
0.7
0.8
0.9
1.0
last_evaluation

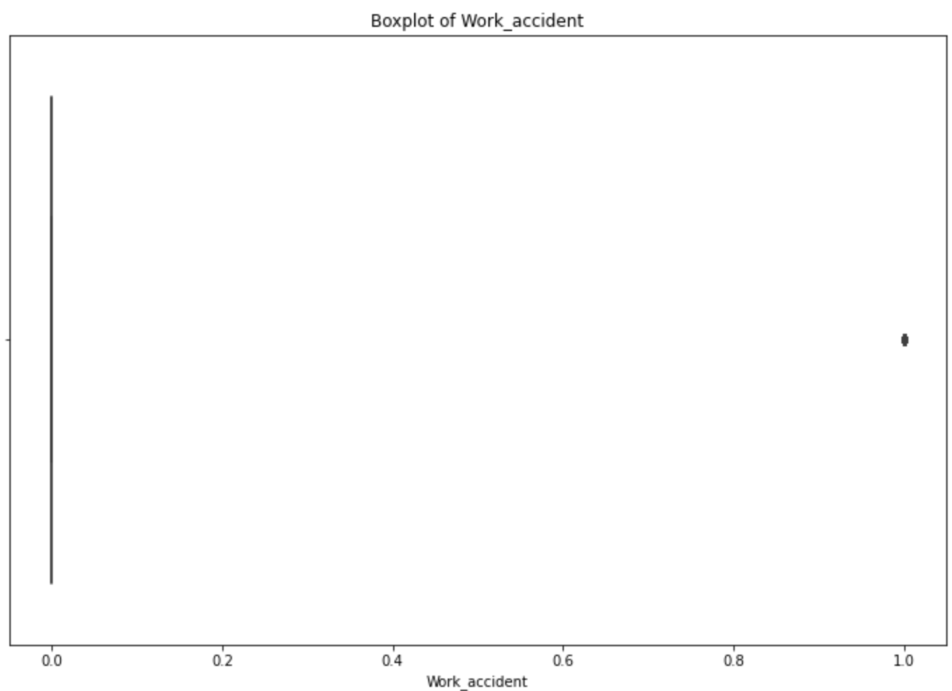


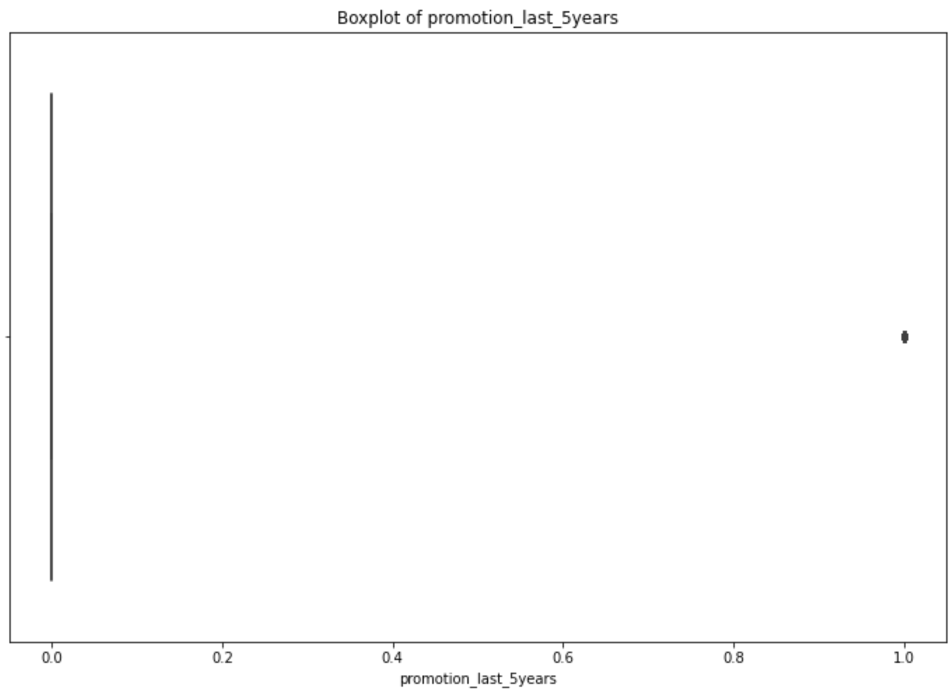
Boxplot of number_project
2
3
4
5
6
7
number_project

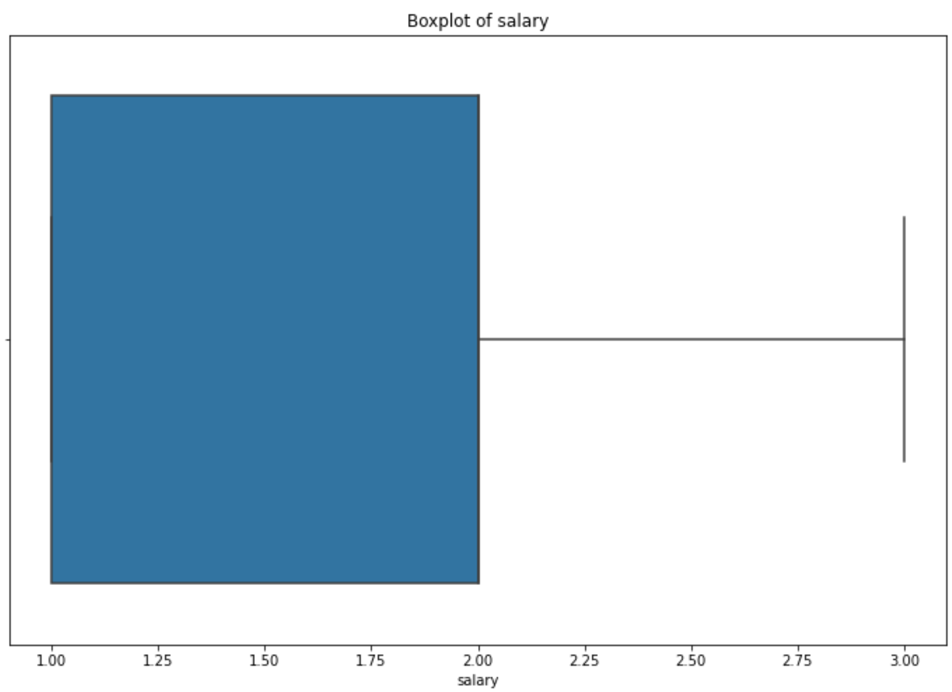


Boxplot of average_montly_hours
100
150
200
250
300
average_montly_hours


Boxplot of time_spend_company
2
3
4
5
6
7
8
9
10
time_spend_company


Boxplot of Work_accident
0.0
0.2
0.4
0.6
0.8
1.0
Work_accident


Boxplot of promotion_last_5years
0.0
0.2
0.4
0.6
0.8
1.0
promotion_last_5years


Boxplot of salary
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
salary

Observation

From the above box plot visualization it is observed that **`time_spend_company`** and **`promotion_last_5years`** has outliers

Finding actual outliers count for the dependent variables

In [14]:
```
for column in emp_turnover_df.columns:
    if column != 'left':
        Q1 = emp_turnover_df[column].quantile(0.25)
        Q3 = emp_turnover_df[column].quantile(0.75)
        QR = Q3 - Q1
        IQR = Q3 - Q1
        outliers = emp_turnover_df[(emp_turnover_df[column] < (Q1 - 1.5 * IQR)) | (emp_turnover_df[column] > (Q3 + 1.5 * IQR))]
        print(f'Number of outliers in {column}:', len(outliers))
```

```
Number of outliers in satisfaction_level: 0
Number of outliers in last_evaluation: 0
Number of outliers in number_project: 0
Number of outliers in average_montly_hours: 0
Number of outliers in time_spend_company: 1282
Number of outliers in Work_accident: 2169
Number of outliers in promotion_last_5years: 319
Number of outliers in salary: 0
```

The above data shows that **`time_spend_company`** , **`Work_accident`** , **`promotion_last_5years`** are the independent features in the dataset that has outliers

Checking `time_spend_company` class

In [15]:
```
emp_turnover_df['time_spend_company'].unique()
```

Out[15]:
```
array([ 3,  6,  4,  5,  2,  8, 10,  7])
```

Observation: `time_spend_company` has outliers but this feature have only 8 unique values so removal is not required **

Checking `Work_accident` class

In [16]:
```
emp_turnover_df['Work_accident'].unique()
```

Out[16]:
```
array([0, 1])
```

Observation: `Work_accident` has outliers but it is a binary type class have only 0 and 1 as the values so removal of outliers is not required

Checking `promotion_last_5years` class

In [17]:
```
emp_turnover_df['promotion_last_5years'].unique()
```

Out[17]:
```
array([0, 1])
```

Observation: `promotion_last_5years` has outliers but it is a binary type class have only 0 and 1 as the values so removal of outliers is not required

Applying K-Means clustering on **`satisfaction_level`** and **`last_evaluation`**

In [18]:
```
# Extacting 'satisfaction_level' and 'last_evaluation' of the employees who left from the the original dataframe

# Extracting rows where 'left' is 1
left_df = emp_turnover_df[emp_turnover_df['left'] == 1]

# Extracting 'satisfaction_level' and 'last_evaluation' for those rows
satisfaction_and_evaluation = left_df[['satisfaction_level', 'last_evaluation']]

satisfaction_and_evaluation
```

Out[18]:

	satisfaction_level	last_evaluation
0	0.38	0.53
1	0.80	0.86
2	0.11	0.88
3	0.72	0.87
4	0.37	0.52
...	...	...
14994	0.40	0.57
14995	0.37	0.48
14996	0.37	0.53
14997	0.11	0.96
14998	0.37	0.52

3571 rows × 2 columns

Elbow method to find the optimal K

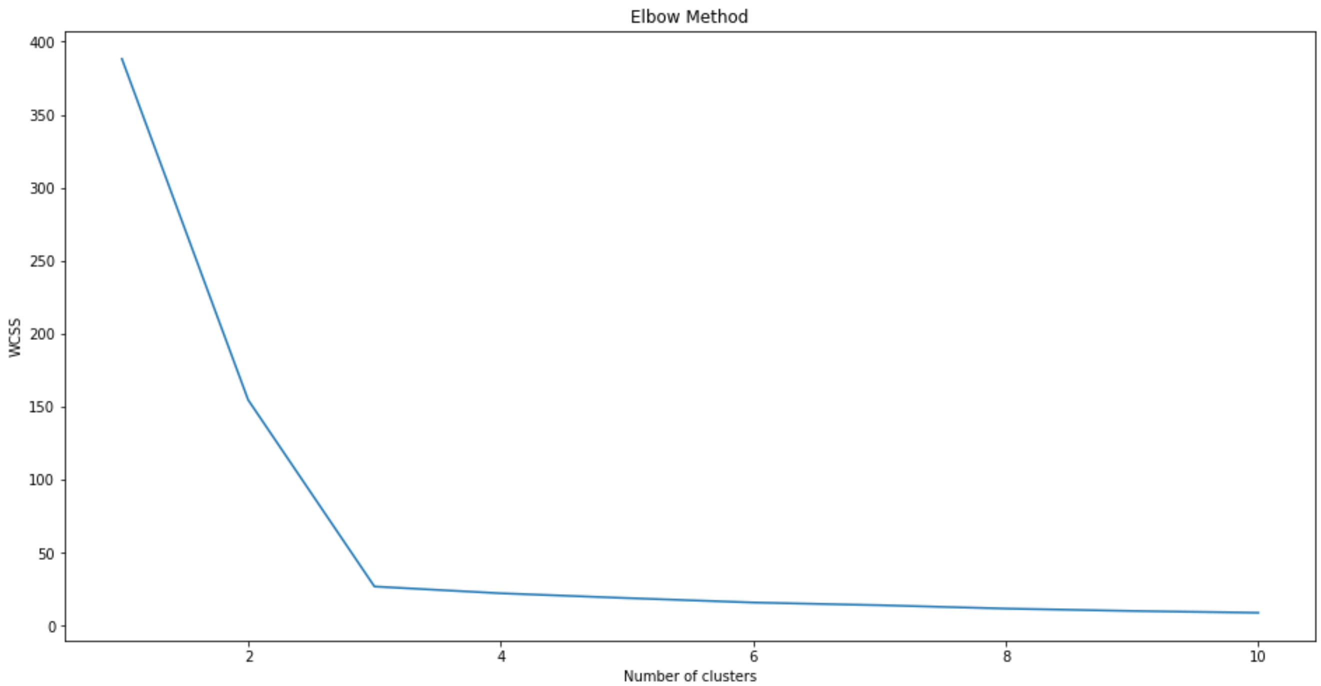
- The elbow method involves plotting the number of clusters against the distortion or inertia to identify a significant flattening point, known as the elbow point.
- The elbow point represents a trade-off between capturing meaningful patterns and avoiding excessive complexity, indicating the optimal number of clusters.
- By choosing the value of k at the elbow point, you strike a balance between cluster quality and simplicity, resulting in a reasonable number of clusters.

Calculating the WCSS (within-cluster sum of squares) for different numbers of clusters.

- WCSS measures how compact a cluster is in k-means clustering. It calculates the total squared distance of all points within a cluster to their cluster's centroid. In simpler terms, it tells you how spread out the points are within a cluster.
- The lower the WCSS, the closer the points are to their cluster's center.

- Plot the WCSS values to find the optimal number of clusters.

In [19]:
```
# Determining optimal number of clusters (using the Elbow method)
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=123)
    kmeans.fit(satisfaction_and_evaluation)
    wcss.append(kmeans.inertia_)

plt.figure(figsize=(16, 8))
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



Observation

In the plotted graph, identify where the WCSS(within-cluster sum of squares) graph starts to flatten out. The plot flattens at **3** . Hence this number is chosen as the Optimal **k**

Training the K-means model with the optimal number of clusters.

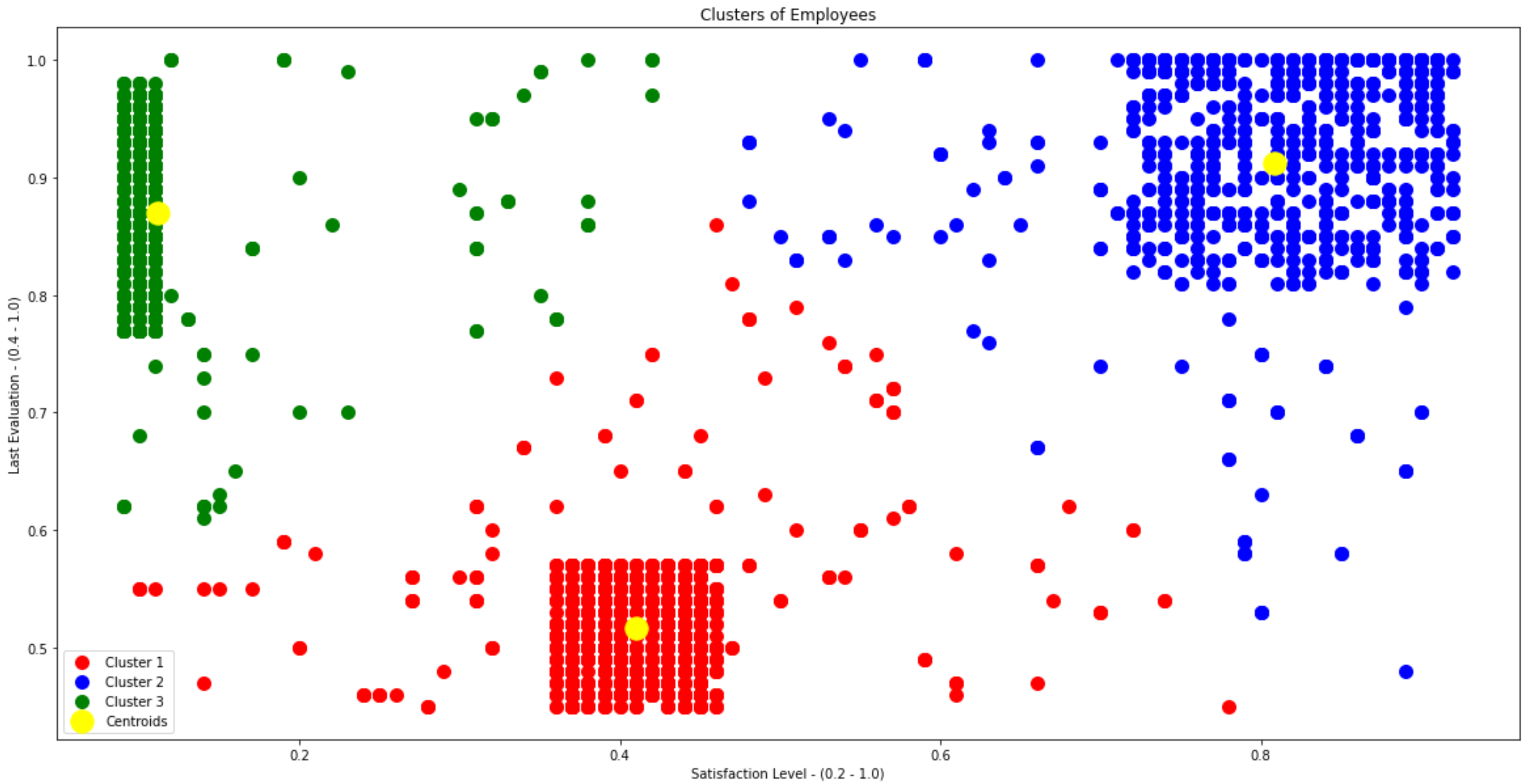
In [20]:
```
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, n_init=10, random_state=0)
y_kmeans = kmeans.fit_predict(satisfaction_and_evaluation)
```

In [21]:
```
y_kmeans
```

Out[21]:
```
array([0, 1, 2, ..., 0, 2, 0], dtype=int32)
```

Plotting the clusters on a scatter plot

In [22]:
```
plt.figure(figsize=(20, 10))
plt.scatter(satisfaction_and_evaluation[y_kmeans == 0]['satisfaction_level'], satisfaction_and_evaluation[y_kmeans == 0]['last_evaluation'], s=100, c='red', la
plt.scatter(satisfaction_and_evaluation[y_kmeans == 1]['satisfaction_level'], satisfaction_and_evaluation[y_kmeans == 1]['last_evaluation'], s=100, c='blue', l
plt.scatter(satisfaction_and_evaluation[y_kmeans == 2]['satisfaction_level'], satisfaction_and_evaluation[y_kmeans == 2]['last_evaluation'], s=100, c='green',

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='yellow', label='Centroids')
plt.title('Clusters of Employees')
plt.xlabel('Satisfaction Level - (0.2 - 1.0)')
plt.ylabel('Last Evaluation - (0.4 - 1.0)')
plt.legend()
plt.show()
```



Observation

K Means Clusters: **`K = 3`**

Interpretation of Each Cluster

`Cluster 1 (Red)` :

Low last evaluation score and moderate satisfaction level. This clusters represents employees who left had low evaluation score and moderate satisfaction level.

`Cluster 2 (Blue)` :

High last evaluation scores and high satisfaction level. This clusters represents employees who left had high evaluation score and high satisfaction level.

`Cluster 3 (Green)` :

High last evaluation but low satisfaction level. This clusters represents employees who left had high evaluation score and low satisfaction level.

Corelation Heat Map

In [23]:
```
emp_turnover_df_corr = emp_turnover_df.copy()
cols_to_scale = emp_turnover_df_corr.columns
cols_to_scale
```

Out[23]:
```
Index(['satisfaction_level', 'last_evaluation', 'number_project',
       'average_montly_hours', 'time_spend_company', 'Work_accident', 'left',
       'promotion_last_5years', 'salary'],
      dtype='object')
```

In [24]:
```
for col in cols_to_scale:
    emp_turnover_df_corr[col] = (emp_turnover_df[col] - emp_turnover_df[col].min()) / (emp_turnover_df[col].max() - emp_turnover_df[col].min())

emp_turnover_df_corr.head()
```

Out[24]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	salary
0	0.318681	0.265625	0.0	0.285047	0.125	0.0	1.0	0.0	0.0
1	0.780220	0.781250	0.6	0.775701	0.500	0.0	1.0	0.0	0.5
2	0.021978	0.812500	1.0	0.822430	0.250	0.0	1.0	0.0	0.5
3	0.692308	0.796875	0.6	0.593458	0.375	0.0	1.0	0.0	0.0
4	0.307692	0.250000	0.0	0.294393	0.125	0.0	1.0	0.0	0.0

In [25]:
```
# Calculating the correlation matrix
correlation_matrix = emp_turnover_df_corr.corr()
# Create the correlation heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f")
plt.title('Correlation Matrix of Employee Turnover Features')
plt.show()
```

Observation:

- High Correlations: Some of the pairs of features exhibit strong correlations (positive and negetive).

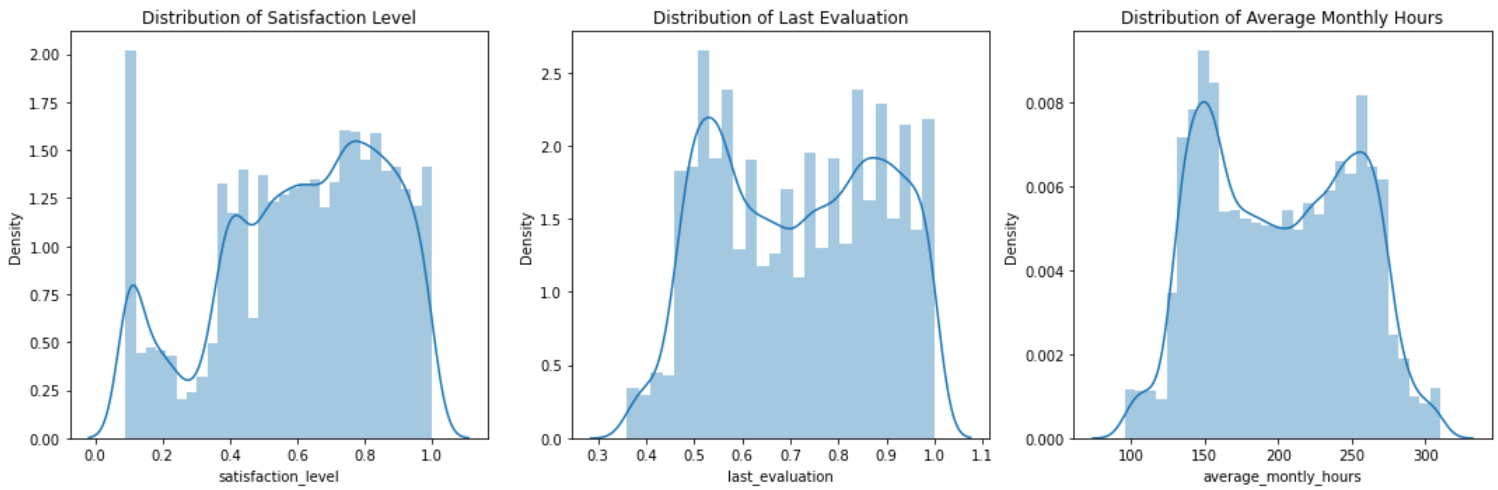
For example, `average_montly_hours` , `last_evaluation` , and `number_project` are highly correlated with each other. This is expected as these features determines the employee turn over.

- Feature Groups: The class `left` is negetively correlated to `satisfacton_level` , `Work_accident` and `salary` which means as satisfaction level goes down and work accident increase the probability of an employee to leave the organization becomes more.

Distribution plot for satisfaction_level, last_evaluation, average_montly_hours

In [26]:
```
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
sns.distplot(emp_turnover_df['satisfaction_level'])
plt.title('Distribution of Satisfaction Level')

plt.subplot(1, 3, 2)
sns.distplot(emp_turnover_df['last_evaluation'])
plt.title('Distribution of Last Evaluation')

plt.subplot(1, 3, 3)
sns.distplot(emp_turnover_df['average_montly_hours'])
plt.title('Distribution of Average Monthly Hours')

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

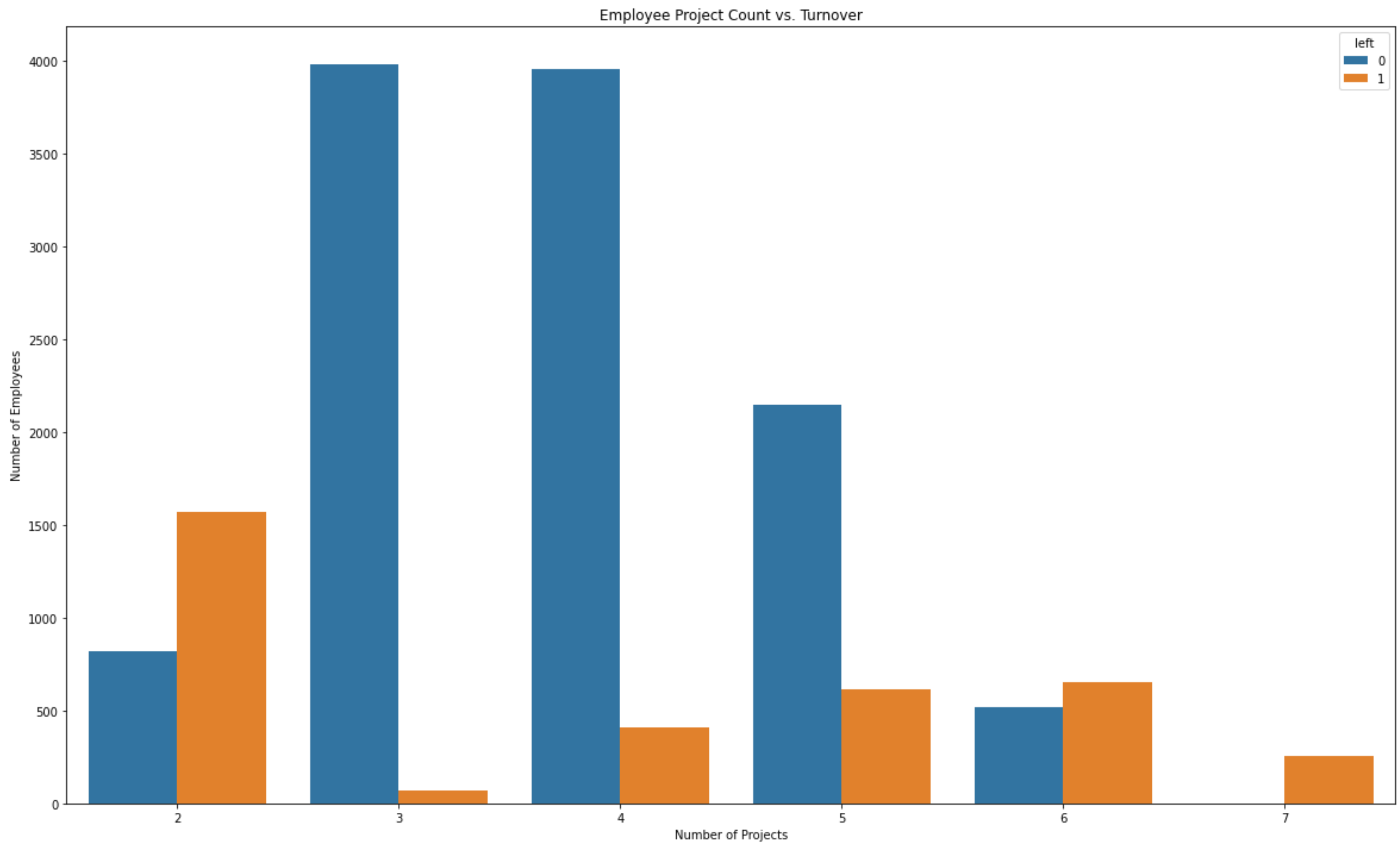
```
/tmp/ipykernel_318/1210454309.py:4: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(emp_turnover_df['satisfaction_level'])
/tmp/ipykernel_318/1210454309.py:8: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(emp_turnover_df['last_evaluation'])
/tmp/ipykernel_318/1210454309.py:12: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(emp_turnover_df['average_montly_hours'])
```



Bar plot of the employee project count of both employees who left and stayed in the organization (use column number_project and hue column left)

In [27]:

```
plt.figure(figsize=(20, 12))
sns.countplot(x='number_project', hue='left', data=emp_turnover_df)
plt.title('Employee Project Count vs. Turnover')
plt.xlabel('Number of Projects')
plt.ylabel('Number of Employees')
plt.show()
```



Observations:

From the above bar plot distribution the followings can be concluded:

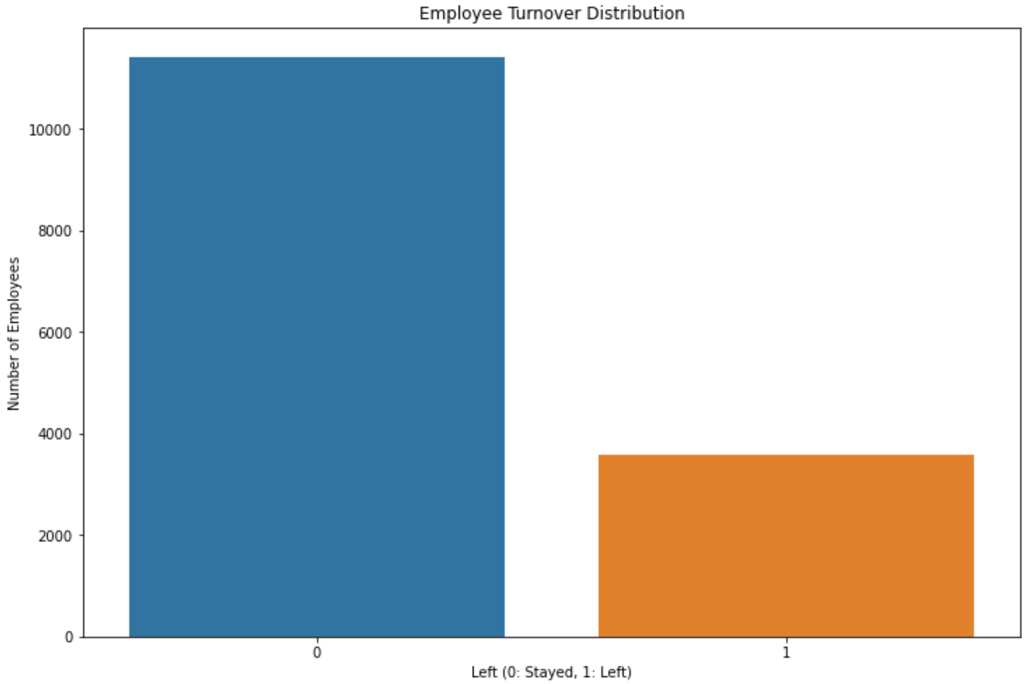
- No employee has stayed in the organization if there is no project assigned to them.
- Most of the employees have left the company when the number of project assigned is 2
- Most of the employees having project count 6 have also left the organization
- Whent the project count is 5, almost 50% of the employees have left the organization
- Having not project or excessive project pressure have both triggred employees to leave the organization.

Removing imbalance in data for the target class `left`

First checking the imbalance in data for the `left` class using `box plot` and `scatter plot`

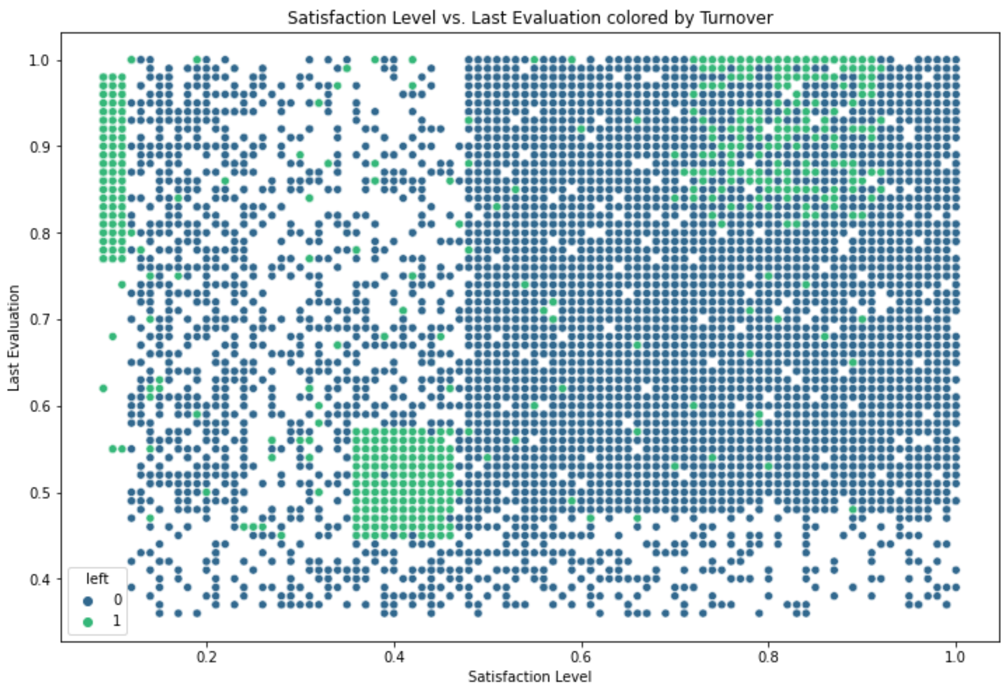
In [28]:

```
plt.figure(figsize=(12, 8))
sns.countplot(x='left', data=emp_turnover_df)
plt.title('Employee Turnover Distribution')
plt.xlabel('Left (0: Stayed, 1: Left)')
plt.ylabel('Number of Employees')
plt.show()
```



In [29]:

```
plt.figure(figsize=(12, 8))
sns.scatterplot(x='satisfaction_level', y='last_evaluation', hue='left', data=emp_turnover_df, palette='viridis')
plt.title('Satisfaction Level vs. Last Evaluation colored by Turnover')
plt.xlabel('Satisfaction Level')
plt.ylabel('Last Evaluation')
plt.show()
```



Observation

From the above plots it is clearly visible that imbalance or bias exists in the data set for the target class **`left`**

Hence we apply **`SMOTE`** to oversample the data to reduce any imbalance

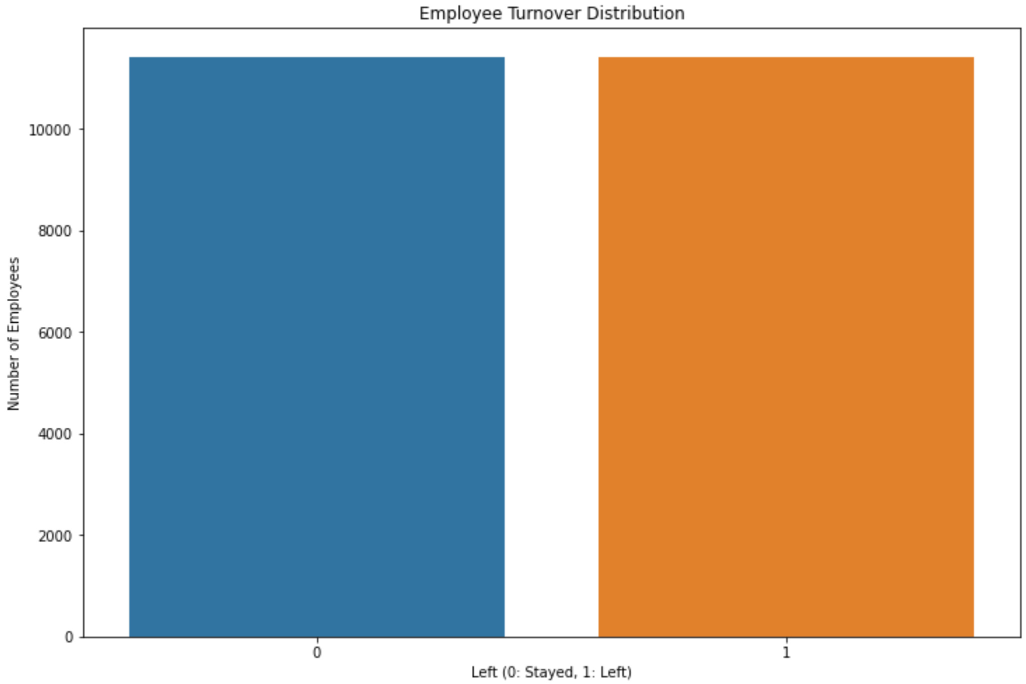
In [30]:
```
# Separate features (X) and target variable (y)
X = emp_turnover_df.drop('left', axis=1)
y = emp_turnover_df['left']

# Applying SMOTE to oversample the minority class
smote = SMOTE(random_state=123)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Creating a new DataFrame with the resampled data
emp_turnover_resampled = pd.DataFrame(X_resampled, columns=X.columns)
emp_turnover_resampled['left'] = y_resampled
```

In [31]:
```
# Now emp_turnover_resampled has a balanced class distribution and the data count increases as new data are added to counter the imbalance
emp_turnover_resampled.shape
```

Out[31]:
```
(22856, 9)
```

After applying SMOTE checking the imbalance in data for the `left` class using `box plot` and `scatter plot`

In [32]:
```
plt.figure(figsize=(12, 8))
sns.countplot(x='left', data=emp_turnover_resampled)
plt.title('Employee Turnover Distribution')
plt.xlabel('Left (0: Stayed, 1: Left)')
plt.ylabel('Number of Employees')
plt.show()
```



In [33]:
```
plt.figure(figsize=(12, 8))
sns.scatterplot(x='satisfaction_level', y='last_evaluation', hue='left', data=emp_turnover_resampled, palette='viridis')
plt.title('Satisfaction Level vs. Last Evaluation colored by Turnover')
plt.xlabel('Satisfaction Level')
plt.ylabel('Last Evaluation')
plt.show()
```

Observation

After applying `SMOTE` we can see that the target class `left` becomes balanced

`Defining features and target variable`

In [34]:
```
X = emp_turnover_resampled.drop(columns=['left'])
y = emp_turnover_resampled['left']
```

Spliting the dataset into training and testing sets 80% training data and 20% testing data

In [35]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
```

In [36]:
```
X_train
```

Out[36]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	promotion_last_5years	salary
12263	0.110000	0.780000	6	260	4	0	0	2
16998	0.423069	0.479901	2	149	3	0	0	3
8874	0.590000	0.990000	5	254	3	1	0	1
16777	0.100000	0.840000	6	261	4	0	0	2
10044	0.980000	0.500000	3	251	3	0	0	2
...	...	...	...	...	...	...	...	...
15377	0.890000	1.000000	5	246	5	0	0	1
21602	0.362527	0.477473	2	137	3	0	0	1
17730	0.834929	0.944976	5	236	5	0	0	1
15725	0.361285	0.512571	2	143	3	0	0	1
19966	0.440000	0.550000	2	135	3	0	0	1

18284 rows × 8 columns

In [37]:
```
X_test
```

Out[37]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	promotion_last_5years	salary
1370	0.740000	0.990000	5	263	5	0	0	1
21489	0.373279	0.498361	2	142	3	0	0	1
12599	0.110000	0.920000	7	307	4	0	0	1
20434	0.864092	0.962046	5	245	5	0	0	1
13031	0.480000	0.580000	3	194	3	0	0	2
...	...	...	...	...	...	...	...	...
1392	0.390000	0.570000	2	157	3	0	0	2
8229	0.650000	0.560000	3	230	2	0	0	3
18930	0.750000	0.810000	5	227	5	0	0	2
8389	0.660000	0.850000	6	165	5	0	0	2
14269	0.380000	0.540000	2	128	3	0	0	1

4572 rows × 8 columns

In [38]:
```
y_train
```

```
Out[38]: 12263    1
16998    1
8874     0
16777    1
10044    0
        ..
15377    1
21602    1
17730    1
15725    1
19966    1
Name: left, Length: 18284, dtype: int64
```

```
In [39]: y_test
```

```
Out[39]: 1370     1
21489    1
12599    1
20434    1
13031    0
        ..
1392     1
8229     0
18930    1
8389     0
14269    1
Name: left, Length: 4572, dtype: int64
```

Model Training

As this is binary classification problem which requires the identification of the target variale `left` as either `0` or `1`

I have choosen the following Machine Learning Algorithms:

- `Logistic Regression`
- `Naive Bayes Classifier`
- `K-Nearest Neighbor (KNN)`
- `Decision Tree`
- `Random Forest`
- `Support Vector Machine (SVM)`
- `Gradient Boosting Classifier`

Logistic Regression

In [40]:

```
# Creating a pipeline with Standard Scaler and Logistic Regression
pipeline_lr = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression(solver='liblinear')) # Using 'liblinear' for small datasets
])


# Defining the parameter grid for grid search
param_grid_lr = {
    'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100],  # Regularization strength
    'classifier__penalty': ['l1', 'l2'] # Regularization type
}

# Creating KFold cross-validation object
kf_lr = KFold(n_splits=5, shuffle=True, random_state=123)

# Creating GridSearchCV object
grid_search_lr = GridSearchCV(pipeline_lr, param_grid_lr, cv=kf_lr, scoring='accuracy')


# Fitting the grid search to the training data
grid_search_lr.fit(X_train, y_train)


# Printing the best hyperparameters and the corresponding score
print("Best hyperparameters:", grid_search_lr.best_params_)
print(f'Best cross-validation score: {grid_search_lr.best_score_ * 100:.2f}%')

# Evaluating the model on the train set
train_accuracy_lr = grid_search_lr.score(X_train, y_train)
print(f'Training accuracy: {train_accuracy_lr * 100:.2f}%')

# Evaluating the model on the test set
test_accuracy_lr = grid_search_lr.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy_lr * 100:.2f}%')
```

```
Best hyperparameters: {'classifier__C': 100, 'classifier__penalty': 'l2'}
Best cross-validation score: 78.96%
Training accuracy: 79.03%
Test accuracy: 77.60%
```

Classification Report Logistic Regression

In [41]:

```
y_pred_lr = grid_search_lr.predict(X_test)
print("Classification Report Logistic Regression:")
print(classification_report(y_test, y_pred_lr))
```

```
Classification Report Logistic Regression:
              precision    recall  f1-score   support

           0       0.81      0.73      0.77      2309
           1       0.75      0.82      0.78      2263

    accuracy                           0.78      4572
   macro avg       0.78      0.78      0.78      4572
weighted avg       0.78      0.78      0.78      4572
```

`Observations: Logistic Regression Classification Report`

Class 0 (Negative Class)

- **Precision: 0.81**

  81% of the instances predicted as class 0 are actually class 0.

- **Recall: 0.73**

  73% of the actual class 0 instances are correctly predicted as class 0.

- **F1-Score (harmonic mean of precision and recall): 0.77**

  Here, the F1-score is 0.77, indicating moderate performance.

- **Support: 2323**

  There are 2323 actual instances of class 0 in the test set.

Class 1 (Positive Class)

- **Precision: 0.75**

  75% of the instances predicted as class 1 are actually class 1.

- **Recall: 0.82**

  82% of the actual class 1 instances are correctly predicted as class 1.

- **F1-Score (harmonic mean of precision and recall): 0.78**

  The F1-score for class 1 is 0.78, indicating moderate performance.

- **Support: 2249**

  There are 2249 actual instances of class 1 in the test set.

Overall Metrics

- **Accuracy: 0.78**

  The overall accuracy of the model, indicating that 78% of the total instances are correctly classified.

- **Macro Average**

  Macro average calculates the metric independently for each class and then takes the average, treating all classes equally. It is useful when you have imbalanced classes.

  Precision: 0.78
  Recall: 0.78
  F1-Score: 0.78

- **Weighted Average**

  Weighted average takes into account the support (the number of true instances for each class) to calculate the average. It is more representative of the performance on imbalanced datasets.

datasets.

Precision: 0.78

Recall: 0.78

F1-Score: 0.78

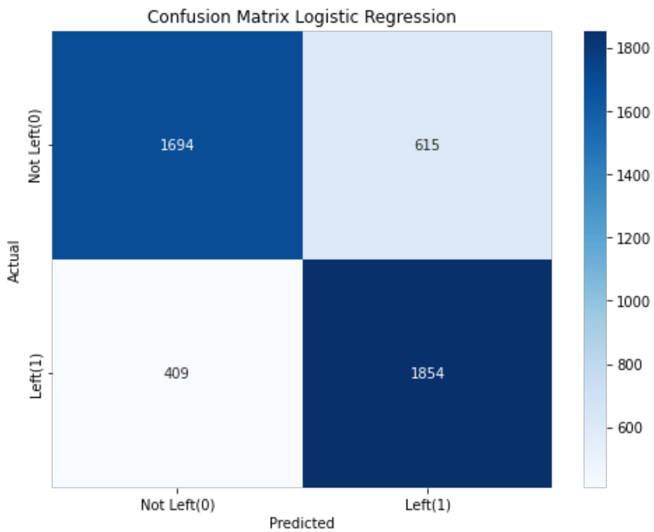
Confusion Matrix Logistic Regression

In [42]:
```
cm_lr = confusion_matrix(y_test, y_pred_lr)
print("Confusion Matrix Logistic Regression:")
cm_lr
```

```
Confusion Matrix Logistic Regression:
```

Out[42]:
```
array([[1694,  615],
       [ 409, 1854]])
```

In [43]:
```
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_lr, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Left(0)', 'Left(1)'], yticklabels=['Not Left(0)', 'Left(1)'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix Logistic Regression')
plt.show()
```



`Observations: Confusion Matrix Logistic Regression`

True Positives (TP): The model correctly predicted `1843` instances as `left=1` when employees indeed left the organization. This indicates that there were `1843` true positive predictions.

True Negatives (TN): The model correctly predicted `1694` instances as `left=0` when the employees indeed did not leave the organization. This shows `1694` true negative predictions.

False Positives (FP): The model incorrectly predicted `615` instance as `left=1` when the employee actually did not leave. This is a false positive, also known as a `Type I error` .

False Negatives (FN): The model incorrectly predicted `409` instances as `left=0` when the employee actually left the organization. This is a false negative, also known as a `Type II error` .

`Precision vs Recall`

`Precision` identifies **False Positive (Employees who stayed being prdicted as Left)**

`Recall` identifies **False Negetives (Employees who left being predicted as Stayed)**

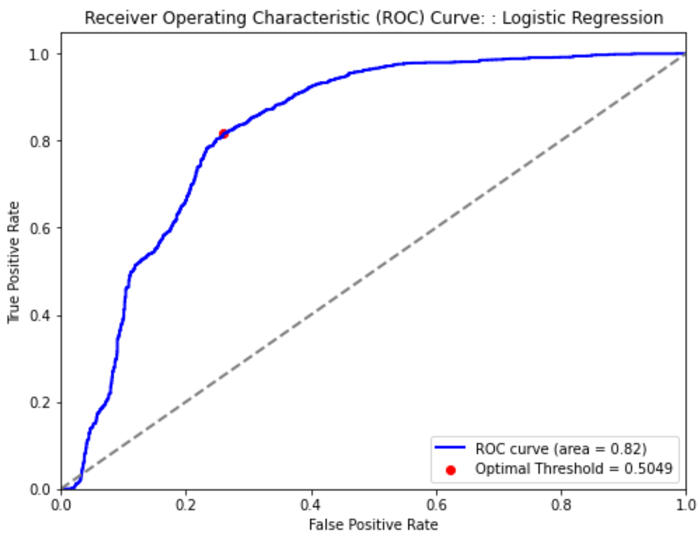
For this problem statement **it is more costly to not identify employees who will leave, therefore, use of `recall` as a primary evaluation metric is suggested**.

AUC-ROC Curve Logistic Regression

In [44]:

```
y_pred_proba_lr = grid_search_lr.predict_proba(X_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_lr)
roc_auc = auc(fpr, tpr)

# Computing Youden's J statistic for each threshold
youden_j_lr = tpr - fpr
optimal_threshold_index_lr = np.argmax(youden_j_lr)
optimal_threshold_lr = thresholds[optimal_threshold_index_lr]

print(f"Optimal Threshold: {optimal_threshold_lr:.4f}")

# Plottingg the ROC curve with the optimal threshold marked
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.scatter(fpr[optimal_threshold_index_lr], tpr[optimal_threshold_index_lr], color='red', marker='o', label=f'Optimal Threshold = {optimal_threshold_lr:.4f}')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve: : Logistic Regression')
plt.legend(loc='lower right')
plt.show()

print("Best parameters found by GridSearchCV (Logistic Regression):")
print(grid_search_lr.best_params_)
```

```
Optimal Threshold: 0.5049
```




```
Best parameters found by GridSearchCV (Logistic Regression):
{'classifier__C': 100, 'classifier__penalty': 'l2'}
```

`Observations: AUC-ROC Curve Logistic Regression`

- The ROC curve in the image reaches the top-left corner **(TPR = 0.8, FPR = ~0.2)**, which indicates a moderate classification performance. The model **moderately** distinguishes between the positive and negative classes at various threshold settings.
- The ROC curve for the logistic regression model has AUC of **0.82**. This indicates that the model **do not have perfect discriminatory power**. An AUC of **0.82** means the model **did not correctly** classifies all positive and negative instances without error.
- The Optimal threshold of **0.5050** defines the decision boundary for the classifier. Probabilities above this value indicate a stronger belief that an instance belongs to the positive class, whereas probabilities below this value indicate a stronger belief that an instance belongs to the negative class.

Feature Importance Coefficients from Logistic Regression Model

In [45]:

```
# Finding the best estimator from the grid search
best_lr_model = grid_search_lr.best_estimator_

# Getting the feature importances (coefficients) for logistic regression
coefficients_lr = best_lr_model.named_steps['classifier'].coef_[0]

# Create a DataFrame to display the coefficients and their corresponding features
feature_importance_lr_df = pd.DataFrame({'Feature': X_train.columns, 'Coefficient': coefficients_lr})

# Sorting the DataFrame by the absolute value of the coefficients to see the most important features
feature_importance_lr_df = feature_importance_lr_df.reindex(feature_importance_lr_df['Coefficient'].abs().sort_values(ascending=False).index).reset_index(drop=

feature_importance_lr_df
```

Out [45]:

	Feature	Coefficient
0	satisfaction_level	-1.230685
1	number_project	-0.691340
2	Work_accident	-0.678641
3	time_spend_company	0.643082
4	salary	-0.495604
5	promotion_last_5years	-0.294951
6	average_montly_hours	0.282575
7	last_evaluation	0.245178

`Observations: Feature Importance Coefficients Logistic Regression`

- It is observed that **`satisfaction_level`** , **`number_project`** , **`Work_accident`** , **`time_spend_company`** , **`salary`** are the top 5 features that determined the probability of an employee leaving the company.

Observation: After applying `Linear Regression Model` on the data

Test Accuracy: **`77.60%`** and Best Cross-Validation Score: **`78.96%`**

`=================================================================================================`

Naive Bayes Classifier

In [46]:
```
# Creating a pipeline with Standard Scaler and Naive Bayes Classifier
pipeline_nb = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', GaussianNB())
])

# Defining the parameter grid for grid search
param_grid_nb = {
    'classifier__var_smoothing': [1e-9, 1e-8, 1e-7, 1e-6, 1e-5], # Smoothing parameter for variance estimation
    'classifier__priors': [None, [0.5, 0.5], [0.4, 0.6]], # Prior probabilities for the classes
}

# Creating KFold cross-validation object
kf_nb = KFold(n_splits=5, shuffle=True, random_state=123)

# Creating GridSearchCV object
grid_search_nb = GridSearchCV(pipeline_nb, param_grid_nb, cv=kf_nb, scoring='accuracy')

# Fitting the grid search to the training data
grid_search_nb.fit(X_train, y_train)

# Printing the best hyperparameters and the corresponding score
print(f'Best cross-validation score: {grid_search_nb.best_score_ * 100:.2f}%')

# Evaluating the model on the train set
train_accuracy_nb = grid_search_nb.score(X_train, y_train)
print(f'Training accuracy: {train_accuracy_nb * 100:.2f}%')

# Evaluating the model on the test set
test_accuracy_nb = grid_search_nb.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy_nb * 100:.2f}%')
```

```
Best cross-validation score: 69.51%
Training accuracy: 69.60%
Test accuracy: 69.23%
```

Classification Report Naive Bayes Classifier

In [47]:
```
y_pred_nb = grid_search_nb.predict(X_test)
print("Classification Report Naive Bayes Classifier:")
print(classification_report(y_test, y_pred_nb))
```

```
Classification Report Naive Bayes Classifier:
              precision    recall  f1-score   support

           0       0.91      0.43      0.59      2309
           1       0.62      0.96      0.76      2263

    accuracy                           0.69      4572
   macro avg       0.77      0.69      0.67      4572
weighted avg       0.77      0.69      0.67      4572
```

`Observations: Naive Bayes Classifier Classification Report`

Class 0 (Negative Class)

- **Precision: 0.91**

  91% of the instances predicted as class 0 are actually class 0.

- **Recall: 0.43**

  43% of the actual class 0 instances are correctly predicted as class 0.

- **F1-Score (harmonic mean of precision and recall): 0.59**

  Here, the F1-score is 0.59, indicating bad performance.

- **Support: 2323**

  There are 2323 actual instances of class 0 in the test set.

Class 1 (Positive Class)

- **Precision: 0.62**

  62% of the instances predicted as class 1 are actually class 1.

- **Recall: 0.96**

  96% of the actual class 1 instances are correctly predicted as class 1.

- **F1-Score (harmonic mean of precision and recall): 0.76**

  The F1-score for class 1 is 0.76, indicating moderate performance.

- **Support: 2249**

  There are 2249 actual instances of class 1 in the test set.

Overall Metrics

- **Accuracy: 0.69**

  The overall accuracy of the model, indicating that 69% of the total instances are correctly classified.

- **Macro Average**

  Macro average calculates the metric independently for each class and then takes the average, treating all classes equally. It is useful when you have imbalanced classes.

  Precision: 0.77
  Recall: 0.69
  F1-Score: 0.67

- **Weighted Average**

  Weighted average takes into account the support (the number of true instances for each class) to calculate the average. It is more representative of the performance on imbalanced

datasets.

Precision: 0.77

Recall: 0.69

F1-Score: 0.67

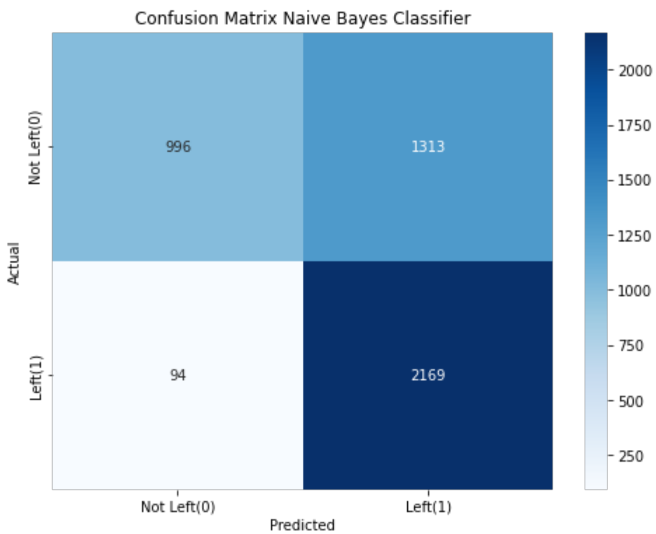
Confusion Matrix Naive Bayes Classifier

In [48]:
```
cm_nb = confusion_matrix(y_test, y_pred_nb)
print("Confusion Matrix Naive Bayes Classifier:")
cm_nb
```

```
Confusion Matrix Naive Bayes Classifier:
```

Out[48]:
```
array([[ 996, 1313],
       [  94, 2169]])
```

In [49]:
```
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_nb, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Left(0)', 'Left(1)'], yticklabels=['Not Left(0)', 'Left(1)'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix Naive Bayes Classifier')
plt.show()
```



`Observations: Confusion Matrix Naive Bayes Classifier`

True Positives (TP): The model correctly predicted `2169` instances as `left=1` when employees indeed left the organization. This indicates that there were `2169` true positive predictions.

True Negatives (TN): The model correctly predicted `996` instances as `left=0` when the employees indeed did not leave the organization. This shows `996` true negative predictions.

False Positives (FP): The model incorrectly predicted `1313` instance as `left=1` when the employee actually did not leave. This is a false positive, also known as a `Type I error`.

False Negatives (FN): The model incorrectly predicted `94` instances as `left=0` when the employee actually left the organization. This is a false negative, also known as a `Type II error`.

`Precision vs Recall`

`Precision` identifies **False Positive (Employees who stayed being prdicted as Left)**

`Recall` identifies **False Negetives (Employees who left being predicted as Stayed)**

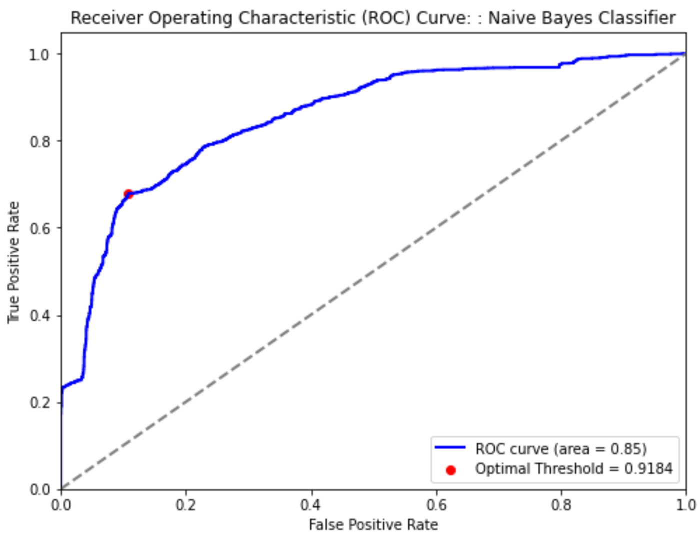
For this problem statement **it is more costly to not identify employees who will leave, therefore, use of `recall` as a primary evaluation metric is suggested**.

AUC-ROC Curve Naive Bayes Classifier

In [50]:

```
y_pred_proba_nb = grid_search_nb.predict_proba(X_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_nb)
roc_auc = auc(fpr, tpr)

# Computing Youden's J statistic for each threshold
youden_j_nb = tpr - fpr
optimal_threshold_index_nb = np.argmax(youden_j_nb)
optimal_threshold_nb = thresholds[optimal_threshold_index_nb]

print(f"Optimal Threshold: {optimal_threshold_nb:.4f}")

# Plotting the ROC curve with the optimal threshold marked
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.scatter(fpr[optimal_threshold_index_nb], tpr[optimal_threshold_index_nb], color='red', marker='o', label=f'Optimal Threshold = {optimal_threshold_nb:.4f}')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve: : Naive Bayes Classifier')
plt.legend(loc='lower right')
plt.show()
```

```
Optimal Threshold: 0.9184
```



`Observations: AUC-ROC Curve Naive Bayes Classifier`

- The ROC curve in the image reaches the top-left corner **(TPR = ~0.7, FPR = 0.1)**, which indicates a moderate classification performance. The model **moderately** distinguishes between the positive and negative classes at various threshold settings.
- The ROC curve for the logistic regression model has AUC of **0.85**. This indicates that the model **do not have perfect discriminatory power**. An AUC of **0.85** means the model **did not correctly** classifies all positive and negative instances without error.
- The Optimal threshold of **0.9184** defines the decision boundary for the classifier. Probabilities above this value indicate a stronger belief that an instance belongs to the positive class, whereas probabilities below this value indicate a stronger belief that an instance belongs to the negative class.

Feature Importance Coefficients from Naive Bayes Classifier Model

In [51]:
```python
# Accessing the best estimator from the grid search
best_nb_model = grid_search_nb.best_estimator_

# **Naive Bayes doesn't have feature importances in the same way as tree-based models or linear models.
# **Therefore, looking into means and standard deviations of each feature for each class.

# Getting the trained Naive Bayes classifier
nb_classifier = best_nb_model.named_steps['classifier']

# Accessing the means and standard deviations
feature_means = nb_classifier.theta_
feature_stds = np.sqrt(nb_classifier.var_)

# Creating a DataFrame to display feature means for each class (0 and 1)
feature_importance_nb_df = pd.DataFrame({'Feature': X_train.columns})
for i, class_label in enumerate(['Not Left', 'Left']):  # Class 0 and Class 1
    feature_importance_nb_df[f'Mean ({class_label})'] = feature_means[i]

# Calculating the difference in means for each feature between the two classes
feature_importance_nb_df['Mean Difference'] = abs(feature_importance_nb_df['Mean (Not Left)'] - feature_importance_nb_df['Mean (Left)'])

# Sorting the DataFrame by Mean Difference to find the most important features
feature_importance_nb_df = feature_importance_nb_df.sort_values(by='Mean Difference', ascending=False).reset_index(drop=True)
print("\nFeature Importance based on Mean Difference:\n")
feature_importance_nb_df
```

```
Feature Importance based on Mean Difference:
```

Out[51]:

	Feature	Mean (Not Left)	Mean (Left)	Mean Difference
0	satisfaction_level	0.430989	-0.428826	0.859815
1	Work_accident	0.248619	-0.247371	0.495990
2	salary	0.222698	-0.221580	0.444279
3	time_spend_company	-0.179343	0.178443	0.357786
4	promotion_last_5years	0.096331	-0.095847	0.192178
5	average_montly_hours	-0.071188	0.070831	0.142019
6	number_project	-0.008556	0.008513	0.017070
7	last_evaluation	0.000295	-0.000294	0.000589

`Observations: Feature Importance Coefficients Naive Bayes Classifier`

- It is observed that **`satisfaction_level`** , **`Work_accident`** , **`salary`** , **`time_spend_company`** , **`promotion_last_5years`** , **`average_montly_hours`** are the top 5 features that determined the probability of an employee leaving the company.

Observation: After applying `Naive Bayes Classifier` on the data

Test Accuracy: **`69.23%`** and Best Cross-Validation Score: **`69.51%`**

==

K-Nearest Neighbors (KNN)

In [52]:
```
# Creating a pipeline with Standard Scaler and KNN classifier
pipeline_knn = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', KNeighborsClassifier())
])

# Defining the parameter grid for grid search
param_grid_knn = {
    'classifier__n_neighbors': [3],  # Number of neighbors
    'classifier__p': [1, 2] # Power parameter for the Minkowski metric
}

# Creating KFold cross-validation object
kf_knn = KFold(n_splits=5, shuffle=True, random_state=123)

# Creating GridSearchCV object
grid_search_knn = GridSearchCV(pipeline_knn, param_grid_knn, cv=kf_knn, scoring='accuracy')

# Fitting the grid search to the training data
grid_search_knn.fit(X_train, y_train)

# Printing the best hyperparameters and the corresponding score
print(f'Best hyperparameters: {grid_search_knn.best_params_}')
print(f'Best cross-validation score: {grid_search_knn.best_score_ * 100:.2f}%')

# Evaluating the model on the train set
train_accuracy_knn = grid_search_knn.score(X_train, y_train)
print(f'Training accuracy: {train_accuracy_knn * 100:.2f}%')

# Evaluating the model on the test set
test_accuracy_knn = grid_search_knn.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy_knn * 100:.2f}%')
```

```
Best hyperparameters: {'classifier__n_neighbors': 3, 'classifier__p': 1}
Best cross-validation score: 96.35%
Training accuracy: 98.25%
Test accuracy: 96.48%
```

Classification Report KNN

In [53]:
```
y_pred_knn = grid_search_knn.predict(X_test)
print("Classification Report KNN:")
print(classification_report(y_test, y_pred_knn))
```

```
Classification Report KNN:
              precision    recall  f1-score   support

           0       0.98      0.95      0.96      2309
           1       0.95      0.98      0.97      2263

    accuracy                           0.96      4572
   macro avg       0.97      0.96      0.96      4572
weighted avg       0.97      0.96      0.96      4572
```

Observations: KNN Classification Report

Class 0 (Negative Class)

- **Precision: 0.98**

  98% of the instances predicted as class 0 are actually class 0.

- **Recall: 0.95**

  95% of the actual class 0 instances are correctly predicted as class 0.

- **F1-Score (harmonic mean of precision and recall): 0.96**

  Here, the F1-score is 0.96, indicating very good performance.

- **Support: 2323**

  There are 2323 actual instances of class 0 in the test set.

Class 1 (Positive Class)

- **Precision: 0.95**

  95% of the instances predicted as class 1 are actually class 1.

- **Recall: 0.98**

  98% of the actual class 1 instances are correctly predicted as class 1.

- **F1-Score (harmonic mean of precision and recall): 0.97**

  The F1-score for class 1 is 0.97, indicating very good performance.

- **Support: 2249**

  There are 2249 actual instances of class 1 in the test set.

Overall Metrics

- **Accuracy: 0.96**

  The overall accuracy of the model, indicating that 96% of the total instances are correctly classified.

- **Macro Average**

  Macro average calculates the metric independently for each class and then takes the average, treating all classes equally. It is useful when you have imbalanced classes.

  Precision: 0.97
  Recall: 0.96
  F1-Score: 0.96

- **Weighted Average**

  Weighted average takes into account the support (the number of true instances for each class) to calculate the average. It is more representative of the performance on imbalanced

datasets.

Precision: 0.97

Recall: 0.96

F1-Score: 0.96

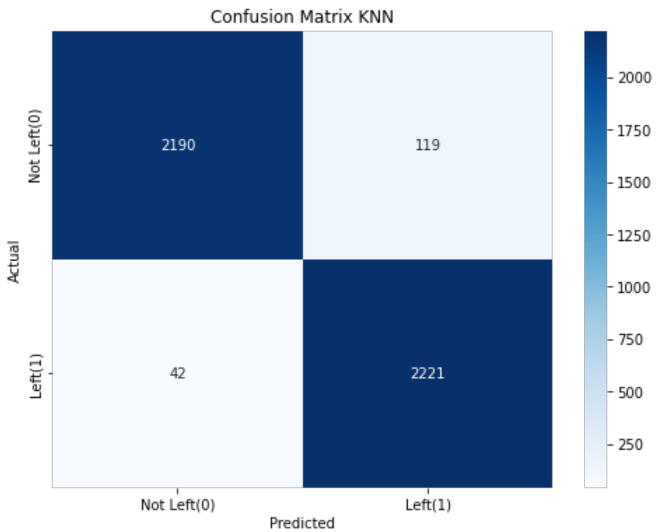
Confusion Matrix KNN

In [54]:
```
cm_knn = confusion_matrix(y_test, y_pred_knn)
print("Confusion Matrix KNN:")
cm_knn
```

```
Confusion Matrix KNN:
```

Out[54]:
```
array([[2190,  119],
       [  42, 2221]])
```

In [55]:
```
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_knn, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Left(0)', 'Left(1)'], yticklabels=['Not Left(0)', 'Left(1)'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix KNN')
plt.show()
```



`Observations: Confusion Matrix KNN`

True Positives (TP): The model correctly predicted `2221` instances as `left=1` when employees indeed left the organization. This indicates that there were `2221` true positive predictions.

True Negatives (TN): The model correctly predicted `2190` instances as `left=0` when the employees indeed did not leave the organization. This shows `2190` true negative predictions.

False Positives (FP): The model incorrectly predicted `119` instance as `left=1` when the employee actually did not leave. This is a false positive, also known as a `Type I error` .

False Negatives (FN): The model incorrectly predicted `42` instances as `left=0` when the employee actually left the organization. This is a false negative, also known as a `Type II error` .

`Precision vs Recall`

`Precision` identifies **False Positive (Employees who stayed being prdicted as Left)**

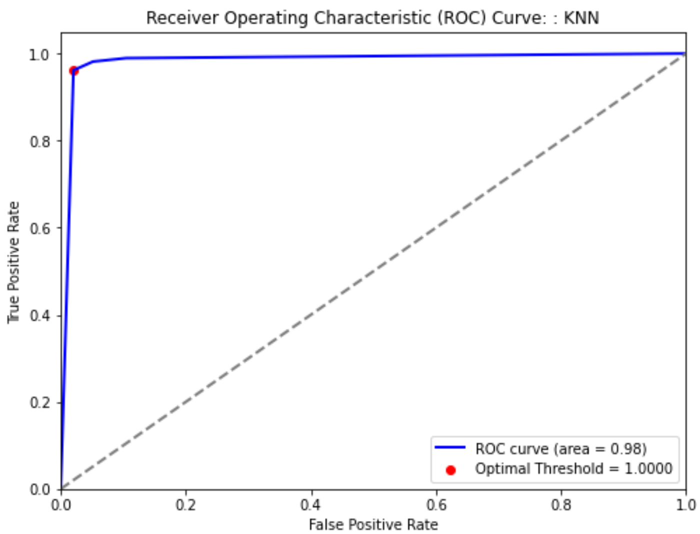
`Recall` identifies **False Negetives (Employees who left being predicted as Stayed)**

For this problem statement **it is more costly to not identify employees who will leave, therefore, use of `recall` as a primary evaluation metric is suggested**.

AUC-ROC Curve KNN

In [56]:
```
y_pred_proba_knn = grid_search_knn.predict_proba(X_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_knn)
roc_auc = auc(fpr, tpr)

# Computing Youden's J statistic for each threshold
youden_j_knn = tpr - fpr
optimal_threshold_index_knn = np.argmax(youden_j_knn)
optimal_threshold_knn = thresholds[optimal_threshold_index_knn]

print(f"Optimal Threshold: {optimal_threshold_knn:.4f}")

# Plotting the ROC curve with the optimal threshold marked
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.scatter(fpr[optimal_threshold_index_knn], tpr[optimal_threshold_index_knn], color='red', marker='o', label=f'Optimal Threshold = {optimal_threshold_knn:.4f
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve: : KNN')
plt.legend(loc='lower right')
plt.show()

print("Best parameters found by GridSearchCV (KNN):")
print(grid_search_knn.best_params_)
```

```
Optimal Threshold: 1.0000
```




```
Best parameters found by GridSearchCV (KNN):
{'classifier__n_neighbors': 3, 'classifier__p': 1}
```

Observations: AUC-ROC Curve KNN

- The ROC curve in the image reaches the top-left corner **(TPR = ~1, FPR = ~0)**, which indicates a almost perfect classification performance. The model **almost perfectly** distinguishes between the positive and negative classes at various threshold settings.
- The ROC curve for the logistic regression model has AUC of **0.98**. This indicates that the model **has almost perfect discriminatory power**. An AUC of **0.98** means the model **almost correctly** classifies all positive and negative instances without error.
- The Optimal threshold of **1.0000** defines the decision boundary for the classifier. Probabilities above this value indicate a stronger belief that an instance belongs to the positive class, whereas probabilities below this value indicate a stronger belief that an instance belongs to the negative class.

Feature Importance Coefficients from KNN Model

In [57]:

```
# Accessing the best estimator from the grid search
best_knn_model = grid_search_knn.best_estimator_

# KNN doesn't have feature importances in the same way as linear models.
# Therefore, using feature permutation to assess feature importance.
importances = []
for col in X_train.columns:
    X_test_copy = X_test.copy()
    X_test_copy[col] = np.random.permutation(X_test_copy[col])  # Permute the feature values
    accuracy_permuted = accuracy_score(y_test, best_knn_model.predict(X_test_copy))
    importance = abs(accuracy_score(y_test, best_knn_model.predict(X_test)) - accuracy_permuted)
    importances.append(importance)


feature_importance_knn_df = pd.DataFrame({'Feature':X_train.columns,'Importance': importances}).sort_values(by='Importance', ascending=False).reset_index(drop=
feature_importance_knn_df
```

Out[57]:

	Feature	Importance
0	number_project	0.208224
1	satisfaction_level	0.204724
2	time_spend_company	0.183946
3	average_montly_hours	0.179571
4	last_evaluation	0.164261
5	salary	0.018373
6	Work_accident	0.003937
7	promotion_last_5years	0.003281

`Observations: Feature Importance Coefficients KNN`

- It is observed that **`number_project`** , **`satisfaction_level`** , **`time_spend_company`** , **`average_montly_hours`** , **`last_evaluation`** are the top 5 features that determined the probability of an employee leaving the company.

Observation: After applying `K-Nearest Neighbor (KNN)` on the data

Test Accuracy: **`96.48%`** and Best Cross-Validation Score: **`96.35%`**

`====================================================================================================`

Decision Tree

In [58]:

```
# Creating a pipeline with Standard Scaler and Decision Tree classifier
pipeline_dt = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', DecisionTreeClassifier())
])

# Defining the parameter grid for grid search
param_grid_dt = {
    'classifier__criterion': ['gini', 'entropy'], # The function to measure the quality of a split
    'classifier__max_depth': [12], # Maximum depth of the tree
}

# Creating KFold cross-validation object
kf_dt = KFold(n_splits=5, shuffle=True, random_state=123)

# Creating GridSearchCV object
grid_search_dt = GridSearchCV(pipeline_dt, param_grid_dt, cv=kf_dt, scoring='accuracy')

# Fitting the grid search to the training data
grid_search_dt.fit(X_train, y_train)

# Printing the best hyperparameters and the corresponding score
print("Best hyperparameters:", grid_search_dt.best_params_)
print(f'Best cross-validation score: {grid_search_dt.best_score_ * 100:.2f}%')

# Evaluating the model on the train set
train_accuracy_dt = grid_search_dt.score(X_train, y_train)
print(f'Training accuracy: {train_accuracy_dt * 100:.2f}%')

# Evaluating the model on the test set
test_accuracy_dt = grid_search_dt.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy_dt * 100:.2f}%')
```

```
Best hyperparameters: {'classifier__criterion': 'gini', 'classifier__max_depth': 12}
Best cross-validation score: 96.81%
Training accuracy: 98.22%
Test accuracy: 97.13%
```

Classification Report Decision Tree

In [59]:

```
y_pred_dt = grid_search_dt.predict(X_test)
print("Classification Report Decision Tree:")
print(classification_report(y_test, y_pred_dt))
```

```
Classification Report Decision Tree:
              precision    recall  f1-score   support

           0       0.97      0.98      0.97      2309
           1       0.98      0.97      0.97      2263

    accuracy                           0.97      4572
   macro avg       0.97      0.97      0.97      4572
weighted avg       0.97      0.97      0.97      4572
```

`Observations: Decision Tree Classification Report`

Class 0 (Negative Class)

- **Precision: 0.97**

  97% of the instances predicted as class 0 are actually class 0.

- **Recall: 0.98**

  98% of the actual class 0 instances are correctly predicted as class 0.

- **F1-Score (harmonic mean of precision and recall): 0.97**

  Here, the F1-score is 0.97, indicating very good performance.

- **Support: 2323**

  There are 2323 actual instances of class 0 in the test set.

Class 1 (Positive Class)

- **Precision: 0.98**

  98% of the instances predicted as class 1 are actually class 1.

- **Recall: 0.97**

  97% of the actual class 1 instances are correctly predicted as class 1.

- **F1-Score (harmonic mean of precision and recall): 0.97**

  The F1-score for class 1 is 0.97, indicating very good performance.

- **Support: 2249**

  There are 2249 actual instances of class 1 in the test set.

Overall Metrics

- **Accuracy: 0.97**

  The overall accuracy of the model, indicating that 97% of the total instances are correctly classified.

- **Macro Average**

  Macro average calculates the metric independently for each class and then takes the average, treating all classes equally. It is useful when you have imbalanced classes.

  Precision: 0.97
  Recall: 0.97
  F1-Score: 0.97

- **Weighted Average**

  Weighted average takes into account the support (the number of true instances for each class) to calculate the average. It is more representative of the performance on imbalanced datasets.

datasets.

Precision: 0.97

Recall: 0.97

F1-Score: 0.97

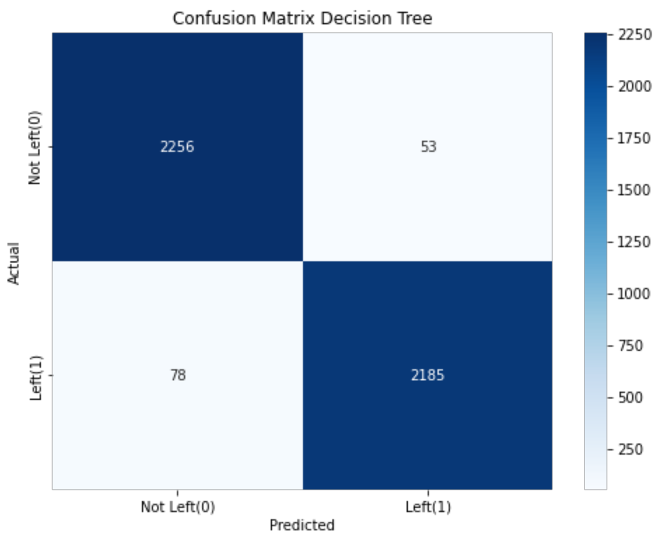
Confusion Matrix Decision Tree

In [60]:
```
cm_dt = confusion_matrix(y_test, y_pred_dt)
print("Confusion Matrix Decision Tree:")
cm_dt
```

```
Confusion Matrix Decision Tree:
```

Out[60]:
```
array([[2256,   53],
       [  78, 2185]])
```

In [61]:
```
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_dt, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Left(0)', 'Left(1)'], yticklabels=['Not Left(0)', 'Left(1)'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix Decision Tree')
plt.show()
```



`Observations: Confusion Matrix Decision Tree`

True Positives (TP): The model correctly predicted `2185` instances as `left=1` when employees indeed left the organization. This indicates that there were `2185` true positive predictions.

True Negatives (TN): The model correctly predicted `2256` instances as `left=0` when the employees indeed did not leave the organization. This shows `2256` true negative predictions.

False Positives (FP): The model incorrectly predicted `53` instance as `left=1` when the employee actually did not leave. This is a false positive, also known as a `Type I error` .

False Negatives (FN): The model incorrectly predicted `78` instances as `left=0` when the employee actually left the organization. This is a false negative, also known as a `Type II error` .

`Precision vs Recall`

`Precision` identifies **False Positive (Employees who stayed being prdicted as Left)**

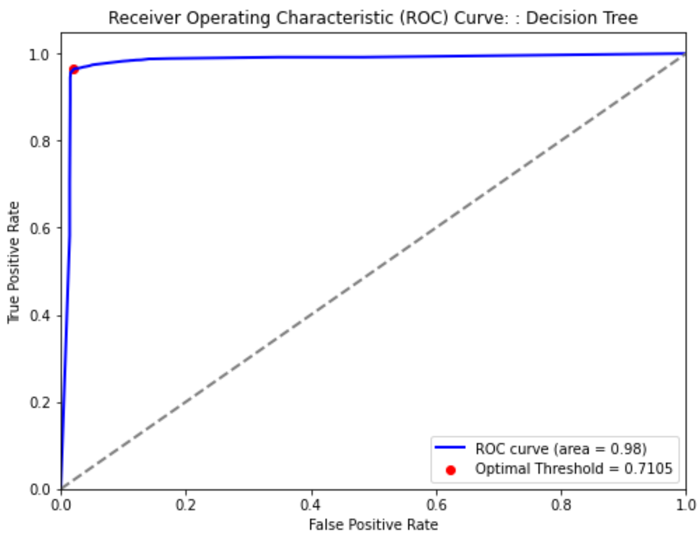
`Recall` identifies **False Negetives (Employees who left being predicted as Stayed)**

For this problem statement **it is more costly to not identify employees who will leave, therefore, use of `recall` as a primary evaluation metric is suggested**.

AUC-ROC Curve Decision Tree

In [62]:
```
y_pred_proba_dt = grid_search_dt.predict_proba(X_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_dt)
roc_auc = auc(fpr, tpr)

# Computing Youden's J statistic for each threshold
youden_j_dt = tpr - fpr
optimal_threshold_index_dt = np.argmax(youden_j_dt)
optimal_threshold_dt = thresholds[optimal_threshold_index_dt]

print(f"Optimal Threshold: {optimal_threshold_dt:.4f}")

# Plotting the ROC curve with the optimal threshold marked
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.scatter(fpr[optimal_threshold_index_dt], tpr[optimal_threshold_index_dt], color='red', marker='o', label=f'Optimal Threshold = {optimal_threshold_dt:.4f}')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve: : Decision Tree')
plt.legend(loc='lower right')
plt.show()

print("Best parameters found by GridSearchCV (Decision Tree):")
print(grid_search_dt.best_params_)
```

```
Optimal Threshold: 0.7105
```




```
Best parameters found by GridSearchCV (Decision Tree):
{'classifier__criterion': 'gini', 'classifier__max_depth': 12}
```

Observations: AUC-ROC Curve Decision Tree

- The ROC curve in the image reaches the top-left corner **(TPR = ~1, FPR = ~0)**, which indicates a almost perfect classification performance. The model **almost perfectly** distinguishes between the positive and negative classes at various threshold settings.
- The ROC curve for the logistic regression model has AUC of **0.98**. This indicates that the model **has perfect discriminatory power**. An AUC of **0.98** means the model **almost correctly** classifies all positive and negative instances without error.
- The Optimal threshold of **0.7105** defines the decision boundary for the classifier. Probabilities above this value indicate a stronger belief that an instance belongs to the positive class, whereas probabilities below this value indicate a stronger belief that an instance belongs to the negative class.

Feature Importance Coefficients from Decision Tree Model

In [63]:
```python
# Accessing the best estimator from the grid search
best_dt_model = grid_search_dt.best_estimator_

# Getting feature importances from the decision tree
importances_dt = best_dt_model.named_steps['classifier'].feature_importances_

# Create a DataFrame to display feature importances
feature_importance_dt_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': importances_dt})

# Sort the DataFrame by importance in descending order
feature_importance_dt_df = feature_importance_dt_df.sort_values(by='Importance', ascending=False).reset_index(drop=True)

# Display the DataFrame
feature_importance_dt_df
```

Out [63]:

	Feature	Importance
0	satisfaction_level	0.442046
1	time_spend_company	0.330761
2	last_evaluation	0.117036
3	average_montly_hours	0.080810
4	number_project	0.022531
5	salary	0.003645
6	Work_accident	0.002747
7	promotion_last_5years	0.000424

`Observations: Feature Importance Coefficients Decision Tree`

- It is observed that **`satisfaction_level`** , **`time_spend_company`** , **`last_evaluation`** , **`average_montly_hours`** , **`number_project`** are the top 5 features that determined the probability of an employee leaving the company.

Observation: After applying `Decision Tree` on the data

Test Accuracy: **`97.13%`** and Best Cross-Validation Score: **`96.81%`**

==

Random Forest Classifier

In [64]:
```python
#Creating a pipeline with scaling and Random Forest classifier
pipeline_rf = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=123))
])

# Defining the parameter grid for RandomForestClassifier
param_grid_rf = {
    'classifier__n_estimators': [10],  # Number of trees in the forest
    'classifier__max_depth': [12],  # Maximum depth of the tree
    # 'classifier__min_samples_split': [2, 5, 10], # Minimum number of samples required to split an internal node
    # 'classifier__min_samples_leaf': [1, 2, 4] # Minimum number of samples required to be at a leaf node
}

# Creating KFold cross-validation object
kf_rf = KFold(n_splits=5, shuffle=True, random_state=123)

# Creating GridSearchCV object
grid_search_rf = GridSearchCV(pipeline_rf, param_grid_rf, cv=kf_rf, scoring='accuracy')

# Fitting the grid search to the training data
grid_search_rf.fit(X_train, y_train)

# Printing the best hyperparameters and the corresponding score
print(f'Best hyperparameters: {grid_search_rf.best_params_}')
print(f'Best cross-validation score: {grid_search_rf.best_score_ * 100:.2f}%')

# Evaluating the model on the train set
train_accuracy_rf = grid_search_rf.score(X_train, y_train)
print(f'Training accuracy: {train_accuracy_rf * 100:.2f}%')

# Evaluating the model on the test set
test_accuracy_rf = grid_search_rf.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy_rf * 100:.2f}%')
```

```
Best hyperparameters: {'classifier__max_depth': 12, 'classifier__n_estimators': 10}
Best cross-validation score: 97.29%
Training accuracy: 98.16%
Test accuracy: 97.73%
```

Classification Report Random Forest

In [65]:
```python
y_pred_rf = grid_search_rf.predict(X_test)
print("Classification Report Random Forest:")
print(classification_report(y_test, y_pred_rf))
```

```
Classification Report Random Forest:
              precision    recall  f1-score   support

           0       0.96      0.99      0.98      2309
           1       0.99      0.96      0.98      2263

    accuracy                           0.98      4572
   macro avg       0.98      0.98      0.98      4572
weighted avg       0.98      0.98      0.98      4572
```

Observations: Random Forest Classification Report

Class 0 (Negative Class)

- **Precision: 0.96**

  96% of the instances predicted as class 0 are actually class 0.

- **Recall: 0.99**

  99% of the actual class 0 instances are correctly predicted as class 0.

- **F1-Score (harmonic mean of precision and recall): 0.98**

  Here, the F1-score is 0.98, indicating very good performance.

- **Support: 2323**

  There are 2323 actual instances of class 0 in the test set.

Class 1 (Positive Class)

- **Precision: 0.99**

  99% of the instances predicted as class 1 are actually class 1.

- **Recall: 0.96**

  96% of the actual class 1 instances are correctly predicted as class 1.

- **F1-Score (harmonic mean of precision and recall): 0.96**

  The F1-score for class 1 is 0.98, indicating very good performance.

- **Support: 2249**

  There are 2249 actual instances of class 1 in the test set.

Overall Metrics

- **Accuracy: 0.98**

  The overall accuracy of the model, indicating that 98% of the total instances are correctly classified.

- **Macro Average**

  Macro average calculates the metric independently for each class and then takes the average, treating all classes equally. It is useful when you have imbalanced classes.

  Precision: 0.98
  Recall: 0.98
  F1-Score: 0.98

- **Weighted Average**

  Weighted average takes into account the support (the number of true instances for each class) to calculate the average. It is more representative of the performance on imbalanced datasets.

datasets.

Precision: 0.98

Recall: 0.98

F1-Score: 0.98

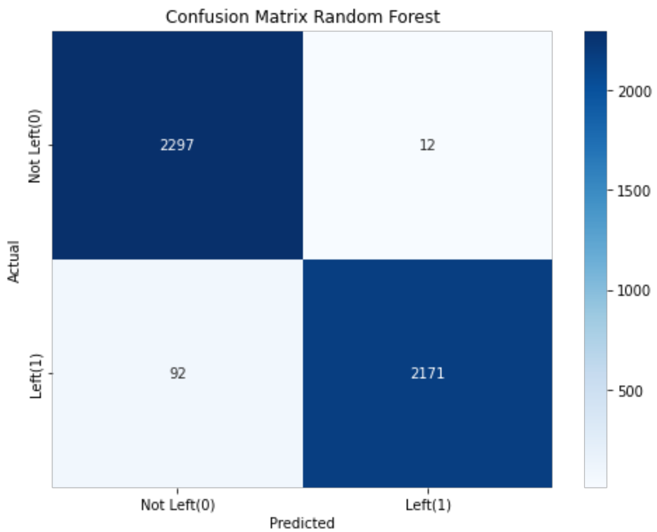
Confusion Matrix Random Forest

In [66]:
```
cm_rf = confusion_matrix(y_test, y_pred_rf)
print("Confusion Matrix Random Forest:")
cm_rf
```

```
Confusion Matrix Random Forest:
```

Out[66]:
```
array([[2297,   12],
       [  92, 2171]])
```

In [67]:
```
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Left(0)', 'Left(1)'], yticklabels=['Not Left(0)', 'Left(1)'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix Random Forest')
plt.show()
```



`Observations: Confusion Matrix Random Forest`

True Positives (TP): The model correctly predicted `2171` instances as `left=1` when employees indeed left the organization. This indicates that there were `2171` true positive predictions.

True Negatives (TN): The model correctly predicted `2297` instances as `left=0` when the employees indeed did not leave the organization. This shows `2297` true negative predictions.

False Positives (FP): The model incorrectly predicted `12` instance as `left=1` when the employee actually did not leave. This is a false positive, also known as a `Type I error`.

False Negatives (FN): The model incorrectly predicted `92` instances as `left=0` when the employee actually left the organization. This is a false negative, also known as a `Type II error`.

`Precision vs Recall`

`Precision` identifies **False Positive (Employees who stayed being prdicted as Left)**

`Recall` identifies **False Negetives (Employees who left being predicted as Stayed)**

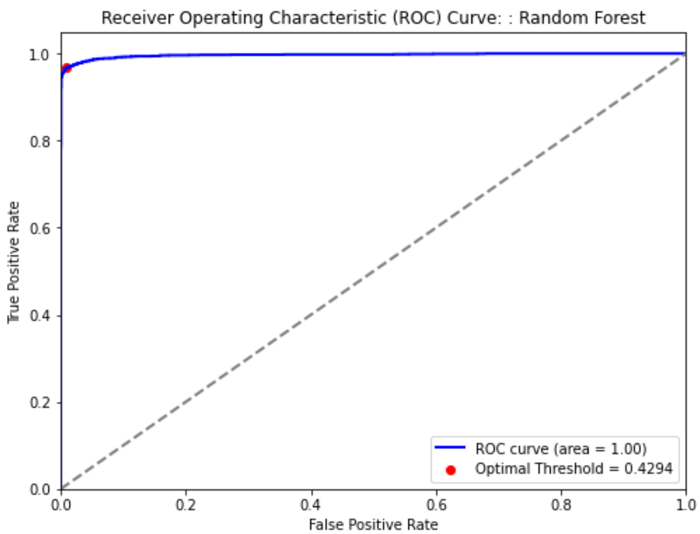
For this problem statement **it is more costly to not identify employees who will leave, therefore, use `recall` as a primary evaluation metric is suggested**.

AUC-ROC Curve Random Forest

In [68]:

```
y_pred_proba_rf = grid_search_rf.predict_proba(X_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_rf)
roc_auc = auc(fpr, tpr)

# Computing Youden's J statistic for each threshold
youden_j_rf = tpr - fpr
optimal_threshold_index_rf = np.argmax(youden_j_rf)
optimal_threshold_rf = thresholds[optimal_threshold_index_rf]

print(f"Optimal Threshold: {optimal_threshold_rf:.4f}")

# Plotting the ROC curve with the optimal threshold marked
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.scatter(fpr[optimal_threshold_index_rf], tpr[optimal_threshold_index_rf], color='red', marker='o', label=f'Optimal Threshold = {optimal_threshold_rf:.4f}')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve: : Random Forest')
plt.legend(loc='lower right')
plt.show()

print("Best parameters found by GridSearchCV (Random Forest):")
print(grid_search_rf.best_params_)
```

```
Optimal Threshold: 0.4294
```




```
Best parameters found by GridSearchCV (Random Forest):
{'classifier__max_depth': 12, 'classifier__n_estimators': 10}
```

Observations: AUC-ROC Curve Random Forest

- The ROC curve in the image reaches the top-left corner **(TPR = ~1, FPR = 0)**, which indicates a almost perfect classification performance. The model **almost perfectly** distinguishes between the positive and negative classes at various threshold settings.
- The ROC curve for the logistic regression model has AUC of **1.00**. This indicates that the model **has almost perfect discriminatory power**. An AUC of **1.00** means the model **almost correctly** classifies all positive and negative instances without error.
- The Optimal threshold of **0.4294** defines the decision boundary for the classifier. Probabilities above this value indicate a stronger belief that an instance belongs to the positive class, whereas probabilities below this value indicate a stronger belief that an instance belongs to the negative class.

Feature Importance Coefficients from Random Forest Classifier

In [69]:
```
# Get feature importances
importances_rf = grid_search_rf.best_estimator_.named_steps['classifier'].feature_importances_

# Create a DataFrame for visualization
feature_importances_rf_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': importances_rf})

# Sort by importance
feature_importances_rf_df = feature_importances_rf_df.sort_values('Importance', ascending=False).reset_index(drop=True)

feature_importances_rf_df
```

Out [69]:

	Feature	Importance
0	time_spend_company	0.263146
1	satisfaction_level	0.225941
2	number_project	0.184306
3	average_montly_hours	0.162282
4	last_evaluation	0.137341
5	Work_accident	0.013535
6	salary	0.012418
7	promotion_last_5years	0.001032

`Observations: Feature Importance Coefficients Random Forest Classifier`

- It is observed that **`time_spend_company`** , **`satisfaction_level`** , **`number_project`** , **`average_montly_hours`** , **`last_evaluation`** are the top 5 features that determined the probability of an employee leaving the company.

Observation: After applying `Random Forest Classifier` model on the data

Test Accuracy: **`97.73%`** and Best Cross-Validation Score: **`97.29%`**

==

Support Vector Machine (SVM)

In [70]:
```
#Creating a pipeline with scaling and SVM classifier
pipeline_svm = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', SVC(probability=True)) # probability=True for ROC curve
])

# Defining the parameter grid for grid search
param_grid_svm = {
    'classifier__C': [0.1, 1, 10], # Regularization parameter
    'classifier__kernel': ['rbf'], # Kernel type
}

# Creating KFold cross-validation object
kf_svm = KFold(n_splits=5, shuffle=True, random_state=123)

# Creating GridSearchCV object
grid_search_svm = GridSearchCV(pipeline_svm, param_grid_svm, cv=kf_svm, scoring='accuracy')

# Fitting the grid search to the training data
grid_search_svm.fit(X_train, y_train)

# Printing the best hyperparameters and the corresponding score
print(f'Best hyperparameters: { grid_search_svm.best_params_}')
print(f'Best cross-validation score: {grid_search_svm.best_score_ * 100:.2f}%')

# Evaluating the model on the train set
train_accuracy_svm = grid_search_svm.score(X_train, y_train)
print(f'Training accuracy: {train_accuracy_svm * 100:.2f}%')

# Evaluating the model on the test set
test_accuracy_svm = grid_search_svm.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy_svm * 100:.2f}%')
```

```
Best hyperparameters: {'classifier__C': 10, 'classifier__kernel': 'rbf'}
Best cross-validation score: 95.60%
Training accuracy: 96.06%
Test accuracy: 95.84%
```

Classification Report SVM

In [71]:
```
y_pred_svm = grid_search_svm.predict(X_test)
print("Classification Report SVM:")
print(classification_report(y_test, y_pred_svm))
```

```
Classification Report SVM:
              precision    recall  f1-score   support

           0       0.95      0.97      0.96      2309
           1       0.97      0.94      0.96      2263

    accuracy                           0.96      4572
   macro avg       0.96      0.96      0.96      4572
weighted avg       0.96      0.96      0.96      4572
```

`Observations: SVM Classification Report`

Class 0 (Negative Class)

- **Precision: 0.95**

  95% of the instances predicted as class 0 are actually class 0.

- **Recall: 0.97**

  97% of the actual class 0 instances are correctly predicted as class 0.

- **F1-Score (harmonic mean of precision and recall): 0.96**

  Here, the F1-score is 0.96, indicating very good performance.

- **Support: 2323**

  There are 2323 actual instances of class 0 in the test set.

Class 1 (Positive Class)

- **Precision: 0.97**

  97% of the instances predicted as class 1 are actually class 1.

- **Recall: 0.94**

  94% of the actual class 1 instances are correctly predicted as class 1.

- **F1-Score (harmonic mean of precision and recall): 0.96**

  The F1-score for class 1 is 0.96, indicating very good performance.

- **Support: 2249**

  There are 2249 actual instances of class 1 in the test set.

Overall Metrics

- **Accuracy: 0.96**

  The overall accuracy of the model, indicating that 96% of the total instances are correctly classified.

- **Macro Average**

  Macro average calculates the metric independently for each class and then takes the average, treating all classes equally. It is useful when you have imbalanced classes.

  Precision: 0.96
  Recall: 0.96
  F1-Score: 0.96

- **Weighted Average**

  Weighted average takes into account the support (the number of true instances for each class) to calculate the average. It is more representative of the performance on imbalanced datasets.

datasets.

Precision: 0.96

Recall: 0.96

F1-Score: 0.96

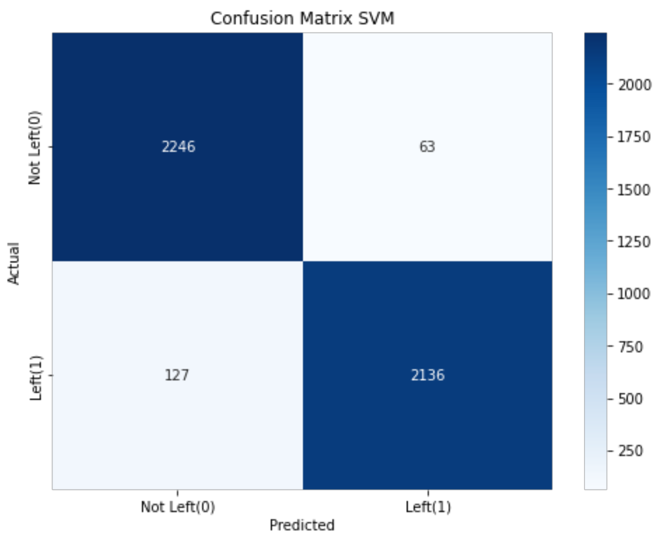
Confusion Matrix SVM

In [72]:
```
cm_svm = confusion_matrix(y_test, y_pred_svm)
print("Confusion Matrix SVM:")
cm_svm
```

```
Confusion Matrix SVM:
```

Out[72]:
```
array([[2246,   63],
       [ 127, 2136]])
```

In [73]:
```
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_svm, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Left(0)', 'Left(1)'], yticklabels=['Not Left(0)', 'Left(1)'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix SVM')
plt.show()
```



Observations: Confusion Matrix SVM

True Positives (TP): The model correctly predicted **2136** instances as **left=1** when employees indeed left the organization. This indicates that there were **2136** true positive predictions.

True Negatives (TN): The model correctly predicted **2246** instances as **left=0** when the employees indeed did not leave the organization. This shows **2246** true negative predictions.

False Positives (FP): The model incorrectly predicted **63** instance as **left=1** when the employee actually did not leave. This is a false positive, also known as a **Type I error** .

False Negatives (FN): The model incorrectly predicted **127** instances as **left=0** when the employee actually left the organization. This is a false negative, also known as a **Type II error** .

Precision vs Recall

Precision identifies **False Positive (Employees who stayed being prdicted as Left)**

Recall identifies **False Negetives (Employees who left being predicted as Stayed)**

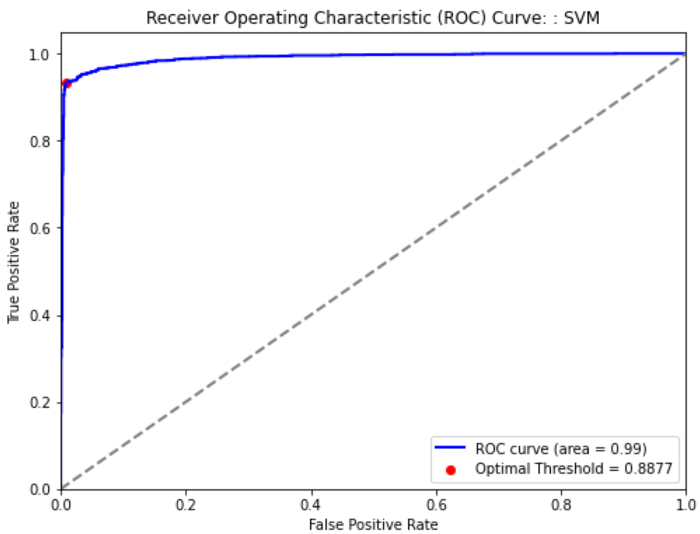
For this problem statement **it is more costly to not identify employees who will leave, therefore, use recall as a primary evaluation metric is suggested**.

AUC-ROC Curve SVM

In [74]:

```
y_pred_proba_svm = grid_search_svm.predict_proba(X_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_svm)
roc_auc = auc(fpr, tpr)

# Computing Youden's J statistic for each threshold
youden_j_svm = tpr - fpr
optimal_threshold_index_svm = np.argmax(youden_j_svm)
optimal_threshold_svm = thresholds[optimal_threshold_index_svm]

print(f"Optimal Threshold: {optimal_threshold_svm:.4f}")

# Plotting the ROC curve with the optimal threshold marked
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.scatter(fpr[optimal_threshold_index_svm], tpr[optimal_threshold_index_svm], color='red', marker='o', label=f'Optimal Threshold = {optimal_threshold_svm:.4f
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve: : SVM')
plt.legend(loc='lower right')
plt.show()

print("Best parameters found by GridSearchCV (SVM):")
print(grid_search_svm.best_params_)
```

```
Optimal Threshold: 0.8877
```




```
Best parameters found by GridSearchCV (SVM):
{'classifier__C': 10, 'classifier__kernel': 'rbf'}
```

`Observations: AUC-ROC Curve SVM`

- The ROC curve in the image reaches the top-left corner **(TPR = ~1, FPR = ~0)**, which indicates a almost perfect classification performance. The model **almost perfectly** distinguishes between the positive and negative classes at various threshold settings.
- The ROC curve for the logistic regression model has AUC of **0.99**. This indicates that the model **has almost perfect discriminatory power**. An AUC of **0.99** means the model **almost correctly** classifies all positive and negative instances without error.
- The Optimal threshold of **0.8876** defines the decision boundary for the classifier. Probabilities above this value indicate a stronger belief that an instance belongs to the positive class, whereas probabilities below this value indicate a stronger belief that an instance belongs to the negative class.

Feature Importance Coefficients from SVM Model

In [75]:

```
# Access the best estimator from the grid search
best_svm_model = grid_search_svm.best_estimator_

# Get feature importances from the SVM model (if available)
# ** SVM models don't directly provide feature importances like decision trees.
# Therefore, I have used coefficients for linear SVM, or other methods for non-linear kernels.

# If using a linear kernel:
if best_svm_model.named_steps['classifier'].kernel == 'linear':
    coefficients_svm = best_svm_model.named_steps['classifier'].coef_
    feature_importance_svm_df = pd.DataFrame({'Feature': X_train.columns, 'Coefficient': abs(coefficients_svm[0])})
    feature_importance_svm_df = feature_importance_svm_df.sort_values(by='Coefficient', ascending=False).reset_index(drop=True)

# For non-linear kernels (e.g., 'rbf', 'poly'), use other methods like permutation feature importance:
else:
    result = permutation_importance(best_svm_model, X_test, y_test, n_repeats=10, random_state=123, n_jobs=2)
    feature_importance_svm_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': result.importances_mean})
    feature_importance_svm_df = feature_importance_svm_df.sort_values(by='Importance', ascending=False).reset_index(drop=True)

feature_importance_svm_df
```

Out[75]:

	Feature	Importance
0	satisfaction_level	0.202843
1	time_spend_company	0.198206
2	number_project	0.198185
3	last_evaluation	0.195035
4	average_montly_hours	0.182568
5	salary	0.007480
6	promotion_last_5years	0.002056
7	Work_accident	0.002056

`Observations: Feature Importance Coefficients SVM`

- It is observed that **`satisfaction_level`** , **`time_spend_company`** , **`number_project`** , **`last_evaluation`** , **`average_montly_hours`** are the top 5 features that determined the probability of an employee leaving the company.

Observation: After applying `Support Vector Machine (SVM)` model on the data

Test Accuracy: **`95.84%`** and Best Cross-Validation Score: **`95.60%`**

==

Gradient Boosting Classifier

In [76]:

```
# Create a pipeline with scaling and Gradient Boosting Classifier
pipeline_gbc = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', GradientBoostingClassifier( random_state=123))
])

# Defining the parameter grid for grid search
param_grid_gbc = {
    'classifier__n_estimators': [10, 50, 100],  # Number of boosting stages
    'classifier__learning_rate': [0.01, 0.1, 1],  # Step size shrinkage used in update to prevent overfitting
}

# Creating KFold cross-validation object
kf_gbc = KFold(n_splits=5, shuffle=True, random_state=123)

# Creating GridSearchCV object
grid_search_gbc = GridSearchCV(pipeline_gbc, param_grid_gbc, cv=kf_gbc, scoring='accuracy')

# Fitting the grid search to the training data
grid_search_gbc.fit(X_train, y_train)

# Printing the best hyperparameters and the corresponding score
print(f'Best hyperparameters:, {grid_search_gbc.best_params_}')
print(f'Best cross-validation score: {grid_search_gbc.best_score_ * 100:.2f}%')

# Evaluating the model on the train set
train_accuracy_gbc = grid_search_gbc.score(X_train, y_train)
print(f'Training accuracy: {train_accuracy_gbc * 100:.2f}%')

# Evaluating the model on the test set
test_accuracy_gbc = grid_search_gbc.score(X_test, y_test)
print(f"Test accuracy: {test_accuracy_gbc * 100:.2f}%")
```

```
Best hyperparameters:, {'classifier__learning_rate': 1, 'classifier__n_estimators': 100}
Best cross-validation score: 97.46%
Training accuracy: 99.11%
Test accuracy: 97.86%
```

Classification Report Gradient Boosting Classifier

In [77]:

```
y_pred_gbc = grid_search_gbc.predict(X_test)
print("Classification Report Gradient Boosting Classifier:")
print(classification_report(y_test, y_pred_gbc))
```

```
Classification Report Gradient Boosting Classifier:
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      2309
           1       0.98      0.98      0.98      2263

    accuracy                           0.98      4572
   macro avg       0.98      0.98      0.98      4572
weighted avg       0.98      0.98      0.98      4572
```

`Observations: Gradient Boosting Classifier Classification Report`

Class 0 (Negative Class)

- **Precision: 0.98**

  98% of the instances predicted as class 0 are actually class 0.

- **Recall: 0.98**

  98% of the actual class 0 instances are correctly predicted as class 0.

- **F1-Score (harmonic mean of precision and recall): 0.98**

  Here, the F1-score is 0.98, indicating very good performance.

- **Support: 2323**

  There are 2323 actual instances of class 0 in the test set.

Class 1 (Positive Class)

- **Precision: 0.98**

  97% of the instances predicted as class 1 are actually class 1.

- **Recall: 0.98**

  95% of the actual class 1 instances are correctly predicted as class 1.

- **F1-Score (harmonic mean of precision and recall): 0.98**

  The F1-score for class 1 is 0.98, indicating very good performance.

- **Support: 2249**

  There are 2249 actual instances of class 1 in the test set.

Overall Metrics

- **Accuracy: 0.98**

  The overall accuracy of the model, indicating that 96% of the total instances are correctly classified.

- **Macro Average**

  Macro average calculates the metric independently for each class and then takes the average, treating all classes equally. It is useful when you have imbalanced classes.

  Precision: 0.98
  Recall: 0.98
  F1-Score: 0.98

- **Weighted Average**

  Weighted average takes into account the support (the number of true instances for each class) to calculate the average. It is more representative of the performance on imbalanced
  datasets

datasets.

Precision: 0.98

Recall: 0.98

F1-Score: 0.98

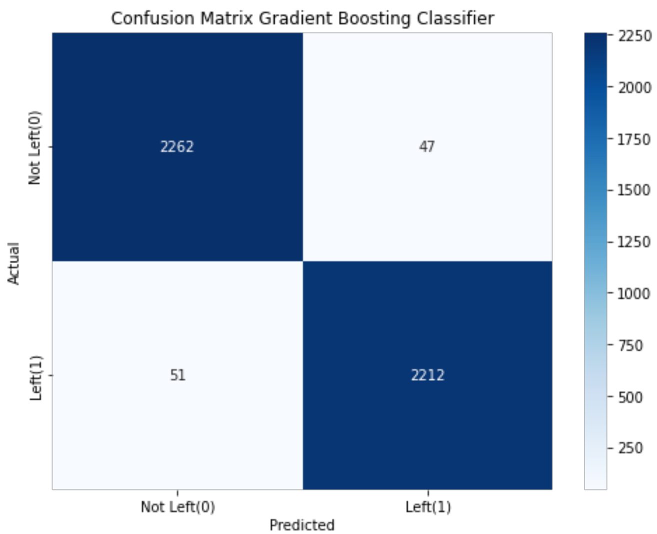
Confusion Matrix Gradient Boosting Classifier

In [78]:
```
cm_gbc = confusion_matrix(y_test, y_pred_gbc)
print("Confusion Matrix Gradient Boosting Classfier:")
cm_gbc
```

```
Confusion Matrix Gradient Boosting Classfier:
```

Out[78]:
```
array([[2262,   47],
       [  51, 2212]])
```

In [79]:
```
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_gbc, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Left(0)', 'Left(1)'], yticklabels=['Not Left(0)', 'Left(1)'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix Gradient Boosting Classifier')
plt.show()
```



Observations: Confusion Matrix Gradient Boosting Classifier

True Positives (TP): The model correctly predicted **2212** instances as **left=1** when employees indeed left the organization. This indicates that there were **2212** true positive predictions.

True Negatives (TN): The model correctly predicted **2262** instances as **left=0** when the employees indeed did not leave the organization. This shows **2262** true negative predictions.

False Positives (FP): The model incorrectly predicted **47** instance as **left=1** when the employee actually did not leave. This is a false positive, also known as a **Type I error** .

False Negatives (FN): The model incorrectly predicted **51** instances as **left=0** when the employee actually left the organization. This is a false negative, also known as a **Type II error** .

Precision vs Recall

Precision identifies **False Positive (Employees who stayed being prdicted as Left)**

Recall identifies **False Negetives (Employees who left being predicted as Stayed)**

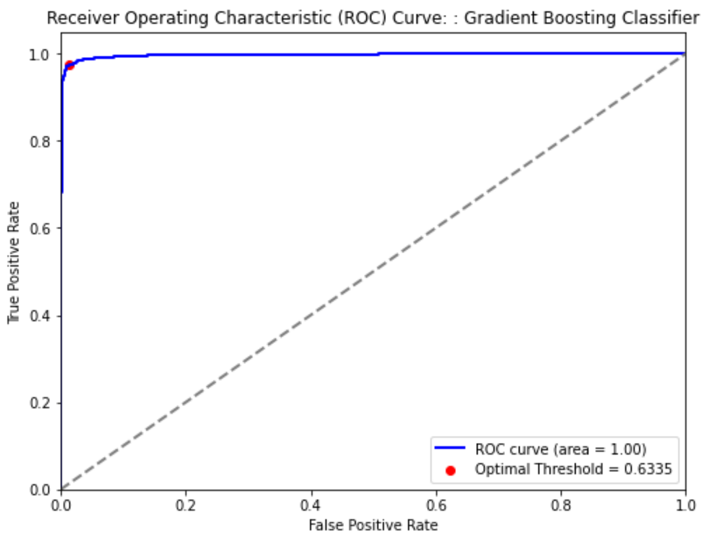
For this problem statement **it is more costly to not identify employees who will leave, therefore, use recall as a primary evaluation metric is suggested**.

AUC-ROC Curve Gradient Boosting Classifier

In [80]:

```
y_pred_proba_gbc = grid_search_gbc.predict_proba(X_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba_gbc)
roc_auc = auc(fpr, tpr)

# Computing Youden's J statistic for each threshold
youden_j_gbc = tpr - fpr
optimal_threshold_index_gbc = np.argmax(youden_j_gbc)
optimal_threshold_gbc = thresholds[optimal_threshold_index_gbc]

print(f"Optimal Threshold: {optimal_threshold_gbc:.4f}")

# Plotting the ROC curve with the optimal threshold marked
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.scatter(fpr[optimal_threshold_index_gbc], tpr[optimal_threshold_index_gbc], color='red', marker='o', label=f'Optimal Threshold = {optimal_threshold_gbc:.4f
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve: : Gradient Boosting Classifier')
plt.legend(loc='lower right')
plt.show()

print("Best parameters found by GridSearchCV (Gradient Boosting Classifier):")
print(grid_search_gbc.best_params_)
```

```
Optimal Threshold: 0.6335
```




```
Best parameters found by GridSearchCV (Gradient Boosting Classifier):
{'classifier__learning_rate': 1, 'classifier__n_estimators': 100}
```

Observations: AUC-ROC Curve Gradient Boosting Classifier

- The ROC curve in the image reaches the top-left corner **(TPR = ~1, FPR = 0)**, which indicates a almost perfect classification performance. The model **almost perfectly** distinguishes between the positive and negative classes at various threshold settings.
- The ROC curve for the logistic regression model has AUC of **1.00**. This indicates that the model **has perfect discriminatory power**. An AUC of **1.00** means the model **correctly** classifies all positive and negative instances without error.
- The Optimal threshold of **0.6335** defines the decision boundary for the classifier. Probabilities above this value indicate a stronger belief that an instance belongs to the positive class, whereas probabilities below this value indicate a stronger belief that an instance belongs to the negative class.

Feature Importance Coefficients from SVM Model

In [81]:
```
# Getting feature importances from the trained Gradient Boosting Classifier
importances = grid_search_gbc.best_estimator_.named_steps['classifier'].feature_importances_

# Creating a DataFrame to display feature importances
feature_importances_gbc_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': importances})

# Sorting the DataFrame by importance in descending order
feature_importances_gbc_df = feature_importances_gbc_df.sort_values(by='Importance', ascending=False).reset_index(drop = True)

# Printing or display the feature importances
feature_importances_gbc_df
```

Out[81]:

	Feature	Importance
0	satisfaction_level	0.432656
1	time_spend_company	0.364428
2	last_evaluation	0.097360
3	average_montly_hours	0.061932
4	number_project	0.034580
5	Work_accident	0.005343
6	salary	0.003487
7	promotion_last_5years	0.000213

`Observations: Feature Importance Coefficients Gradient Boosting Classifier`

- It is observed that **`satisfaction_level`** , **`time_spend_company`** , **`last_evaluation`** , **`average_montly_hours`** , **`number_project`** are the top 5 features that determined the probability of an employee leaving the company.

Observation: After applying `Graident Boosting Classifier` model on the data

Test Accuracy: **`97.86%`** and Best Cross-Validation Score: **`97.46`**

`================================================================================`

Summarizing the above models outputs

| Sl. No.| Model Algorithm | Training Accuracy | Testing Accuracy | Observation | | --- | --- | --- | --- | --- | | 1 | Logistic Regression | 79.03% | 77.60% | Negligible Overfitting | | 2 | Naibe Bayes Classifier | 69.60% | 69.23% | No Overfitting | | 3 | K-Nearest Neighbor (KNN) | 98.25% | 96.48% | Negligible Overfitting | | 4 | Decision Tree | 98.22% | 97.13% | No Overfitting | | 5 | Random Forest Classifier | 98.16% | 97.73% | No Overfitting | | 6 | Support Vector Machine | 96.06% | 95.84% | No Overfitting | | 7 | Gradient Boosting Classifier | 99.11% | 97.86% | No Overfitting |

From the above table it is found that the highest test accuracy is obtained using **`Gradient Boosting Classifier`** amongst all other algorithms.

`Logistic Regression and Naibe Bayes Classifier` yeild significantly lower training and testing accuracies, so they are also not optimal to generalize well for new data and showed minor overfitting.

`Gradient Boosting Classifier` in this case has performed optimally by yeilding highest test accuracy than some other algorithms. The balance between the training and testing accuracy is optimal, as a small gap between the two is a good sign of a model that can generalize well to new data, making the "99% training, 98% testing accuracy" model the better choice in this scenario

Therefore, I choose **`Gradient Boosting Classifier`** for this data set as its output is most generalized of all the others.

In [82]:

```python
# Predicting probabilities on the test set using the best Gradient Boosting Classifier
y_pred_proba_gbc = grid_search_gbc.predict_proba(X_test)
y_pred_proba_gbc
```

Out[82]:

```
array([[2.03171571e-03, 9.97968284e-01],
       [1.13946811e-03, 9.98860532e-01],
       [5.44606005e-06, 9.99994554e-01],
       ...,
       [7.71301691e-02, 9.22869831e-01],
       [7.38435569e-01, 2.61564431e-01],
       [1.13199970e-03, 9.98868000e-01]])
```

In [83]:

```python
# Extracting the probability of leaving (class left = 1)
probability_of_leaving = y_pred_proba_gbc[:, 1]

# Creating a DataFrame with probability scores and employee data
suggestion_df = pd.DataFrame({'Probability_of_Leaving': probability_of_leaving})

# Defining the probability score ranges and zones
def categorize_employees(probability):
    if probability < 0.2:
        return 'Safe Zone (Green Zone)'
    elif 0.2 <= probability < 0.6:
        return 'Low-Risk Zone (Yellow Zone)'
    elif 0.6 <= probability < 0.8:
        return 'Medium-Risk Zone (Orange Zone)'
    else:
        return 'High-Risk Zone (Red Zone)'

# Applying categorization to the probability scores
suggestion_df['Risk_Zone'] = suggestion_df['Probability_of_Leaving'].apply(categorize_employees)


# Setting Retention strategies by zone
def retention_strategy(zone):
    if zone == 'Safe Zone (Green Zone)':
        return 'Proactive engagement, regular feedback, and opportunities for growth.'
    elif zone == 'Low-Risk Zone (Yellow Zone)':
        return 'Increased communication, mentorship, skill development, and addressing concerns.'
    elif zone == 'Medium-Risk Zone (Orange Zone)':
        return 'Immediate action, address root causes of dissatisfaction, negotiate salary/benefits.'
    elif zone == 'High-Risk Zone (Red Zone)':
        return 'Counter-offers, performance incentives, investigate underlying issues, retention bonus.'
    else:
        return 'Unknown'

suggestion_df['Retention_Strategy'] = suggestion_df['Risk_Zone'].apply(retention_strategy).reset_index(drop=True)

# Displaying the results
suggestion_df
```

Out [83]:

	Probability_of_Leaving	Risk_Zone	Retention_Strategy
0	0.997968	High-Risk Zone (Red Zone)	Counter-offers, performance incentives, invest...
1	0.998861	High-Risk Zone (Red Zone)	Counter-offers, performance incentives, invest...
2	0.999995	High-Risk Zone (Red Zone)	Counter-offers, performance incentives, invest...
3	0.995918	High-Risk Zone (Red Zone)	Counter-offers, performance incentives, invest...
4	0.017294	Safe Zone (Green Zone)	Proactive engagement, regular feedback, and op...
...	...	...	...
4567	0.988298	High-Risk Zone (Red Zone)	Counter-offers, performance incentives, invest...
4568	0.000079	Safe Zone (Green Zone)	Proactive engagement, regular feedback, and op...
4569	0.922870	High-Risk Zone (Red Zone)	Counter-offers, performance incentives, invest...
4570	0.261564	Low-Risk Zone (Yellow Zone)	Increased communication, mentorship, skill dev...
4571	0.998868	High-Risk Zone (Red Zone)	Counter-offers, performance incentives, invest...

4572 rows × 3 columns

In [ ]: