# EMPLOYEE CHURN PROJECT

## Building a model to predict whether an employee is churned or not

## Employee Churn: Definition, Importance for Markets and its Use in Data Science

Employee churn, also known as employee turnover, refers to the rate at which employees leave a company and need to be replaced by new hires. It is a significant metric in human resources management and business operations, and it also holds relevance in the field of data science. Here are a few reasons why employee churn is important in the context of data science:

- **Cost and Productivity**: Employee churn can be expensive for organizations. The process of recruiting, hiring, and training new employees requires time and resources. Moreover, when experienced employees leave, they take their knowledge and expertise with them, leading to a loss of productivity. By understanding the factors that contribute to churn, data scientists can help organizations identify areas where interventions can reduce turnover and mitigate the associated costs.

- **Predictive Analytics**: Data scientists can utilize predictive analytics to forecast employee churn. By analyzing historical data on employee behavior, performance, satisfaction, and other relevant factors, they can develop models that predict the likelihood of an employee leaving the organization. These predictive models enable companies to proactively identify individuals who may be at risk of churn and take appropriate actions to retain them.

- **Employee Retention Strategies**: Data science can help identify the key drivers of employee churn, such as job satisfaction, compensation, career growth opportunities, or work-life balance. By analyzing patterns and correlations within the data, data scientists can provide insights to HR departments and management, enabling them to design and implement effective employee retention strategies. For example, by identifying common factors among employees who stay with the company long-term, organizations can develop programs to enhance employee engagement and satisfaction.

- **Talent Management**: Understanding employee churn patterns can provide valuable insights into talent management strategies. Data scientists can analyze the data to identify the characteristics and behaviors of high-performing employees who are less likely to churn. This knowledge can guide the recruitment and selection process, helping organizations attract candidates who possess the attributes associated with long-term success and retention.

- **Organizational Health and Culture**: High employee churn rates may indicate underlying issues within the organization, such as poor management, low employee morale, or a toxic work environment. By analyzing employee data, sentiment analysis, and feedback, data scientists can uncover hidden patterns and correlations that shed light on these factors. This information can be used to address systemic problems, improve organizational health and culture, and ultimately reduce churn.

In summary, employee churn is an essential aspect of data science in the context of human resources and organizational management. By leveraging data analytics and machine learning techniques, data scientists can help organizations understand the causes and consequences of employee churn, predict future turnover, and develop effective strategies to retain talent, improve productivity, and foster a positive work environment.

By using various machine learning and deep learning algorithms, this project aims to build a model to predict the risk of leaving the copany for an employee. The model will be deployed via Streamlit in the end.

# #Determines

In this project you have HR data of a company. A study is requested from you to predict which employee will churn by using this data.

The HR dataset has 14,999 samples. In the given dataset, you have two types of employee one who stayed and another who left the company.

You can describe 10 attributes in detail as:

- ***satisfaction_level:*** It is employee satisfaction point, which ranges from 0-1.
- ***last_evaluation:*** It is evaluated performance by the employer, which also ranges from 0-1.
- ***number_projects:*** How many of projects assigned to an employee?
- ***average_monthly_hours:*** How many hours in averega an employee worked in a month?
- **time_spent_company:** time_spent_company means employee experience. The number of years spent by an employee in the company.
- ***work_accident:*** Whether an employee has had a work accident or not.
- ***promotion_last_5years:*** Whether an employee has had a promotion in the last 5 years or not.
- ***Departments:*** Employee's working department/division.
- ***Salary:*** Salary level of the employee such as low, medium and high.

- ***left:*** Whether the employee has left the company or not.

First of all, to observe the structure of the data, outliers, missing values and features that affect the target variable, you must use exploratory data analysis and data visualization techniques.

Then, you must perform data pre-processing operations such as ***Scaling*** and ***Encoding*** to increase the accuracy score of Gradient Descent Based or Distance-Based algorithms.

You are asked to perform ***Cluster Analysis*** based on the information you obtain during exploratory data analysis and data visualization processes. The purpose of clustering analysis is to cluster data with similar characteristics.

Once the data is ready to be applied to the model, you must ***split the data into train and test***. Then build a model to predict whether employees will churn or not. Train your models with your train set, test the success of your model with your test set.

Try to make your predictions by using the ***Classification Algorithms***. You can use the related modules of the ***scikit-learn*** and ***Tensorflow-Keras*** library. You can use scikit-learn ***Classification Metrics*** module for accuracy calculation.

In the final step, you will deploy your model using Streamlit tool.

# #Tasks

### 1. Exploratory Data Analysis

- EDA is an initial process of analysis, in which you can summarize characteristics of data such as pattern, trends, outliers, and hypothesis testing using descriptive statistics and visualization.
- In the given dataset, you have two types of employee one who stayed and another who left the company. So, you can divide data into two groups and compare their characteristics.

### 2. Data Visualization

- Explore your data via visualizations to find-out:
  - What can be the reason of the churn?
  - Behavioral analysis of churns and not churns ..... etc.

### 3. Cluster Analysis

- Apply ***clustering algorithms*** and writedown your conclusions about the clusters you created.

### 5. Predictive Model Building

- Split Data as Train and Test set
- Built Classification Models(at least four models) and Evaluate Model Performances

### 6. Model Deployement

- Save and Export the Best Model
- Deploy best model via Streamlit

# #Importing Modules and Predefined Functions

In [ ]:
```
# libraries for EDA
import numpy as np
import time
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly
import plotly.express as px
import cufflinks as cf
from termcolor import colored

#Enabling the offline mode for interactive plotting locally
from plotly.offline import download_plotlyjs,init_notebook_mode,plot,iplot
init_notebook_mode(connected=True)
cf.go_offline()
#To display the plots
%matplotlib inline



# sklearn library for machine learning algorithms, data preprocessing, and e
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.model_selection import train_test_split, GridSearchCV, cross_va

from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
import lightgbm as lgb

from sklearn.metrics import accuracy_score, f1_score, recall_score,\
                            precision_score, make_scorer,\
                            classification_report,confusion_matrix,\
                            ConfusionMatrixDisplay, average_precision_score,
                            roc_curve, auc, matthews_corrcoef

# yellowbrick library for visualizing the model performance
from yellowbrick.classifier import ConfusionMatrix
from yellowbrick.cluster import KElbowVisualizer

from sklearn.pipeline import Pipeline
# to get rid of the warnings
import warnings
warnings.filterwarnings("ignore")
warnings.warn("this will not show")
```

### User defined functions

Mainly for the statistical analysis of the data (such as distribution and density of features or comparing various features), 3 functiones will be defined below. See the related docstrings for details of the functions.

In [ ]:
```
def show_distribution(col):

    '''

    This function will prints a Histogram and box plot which are graphical r
    for the frequency of numeric data values. It aims to describe the data a
    the central tendency and variability before using advanced statistical a

    '''
    # Get statistics
    from termcolor import colored

    print(colored('Statistical Calculations :', 'red', attrs=['bold']))
    print(colored('-'*26, 'red', attrs=['bold']))
    min_val = col.min()
    max_val = col.max()
    mean_val = col.mean()
    med_val = col.median()
    mod_val = col.mode()[0]

    print(colored('Minimum:{:>7.2f}\nMean:{:>10.2f}\nMedian:{:>8.2f}\nMode:{

    # Create a figure for 2 subplots (2 rows, 1 column)
    fig, ax = plt.subplots(2, 1, figsize=(15, 15))

    # Plot the histogram
    ax[0].hist(col, bins=30)
    ax[0].set_ylabel('Frequency', fontsize=10)

    # Add lines for the mean, median, and mode
    ax[0].axvline(x=min_val,  color='yellow',     linestyle='dashed', linewi
    ax[0].axvline(x=mean_val, color='lightgreen', linestyle='dashed', linewi
    ax[0].axvline(x=med_val,  color='cyan',       linestyle='dashed', linewi
    ax[0].axvline(x=mod_val,  color='purple',     linestyle='dashed', linewi
    ax[0].axvline(x=max_val,  color='red',        linestyle='dashed', linewi
    ax[0].legend(loc='upper right')

    # Plot the boxplot
    medianprops = dict(linestyle='-', linewidth=3, color='m')
    boxprops=dict(linestyle='-', linewidth=1.5)
    meanprops={"marker":"d", "markerfacecolor":"red", "markeredgecolor":"bla
    flierprops={'marker': 'o', 'markersize': 8, 'markerfacecolor': 'fuchsia'

    ax[1].boxplot(col,
                  vert=False,
                  notch=True,
                  patch_artist=False,
                  medianprops=medianprops,
```

```
                    flierprops=flierprops,
                    showmeans=True,
                    meanprops=meanprops)

    ax[1].set_xlabel('value', fontsize=10)


    # Add a title to the Figure
    fig.suptitle('Data Distribution', fontsize=15)
```

In [ ]:
```
def show_density(col):

    '''

    This function plots a 'density plot' which is a representation of the di
    It uses a kernel density estimate to show the probability density functi

    '''
    from matplotlib import pyplot as plt

    fig = plt.figure(figsize=(15, 5))

    # Plot density
    col.plot.density()

    # Add titles and labels
    plt.title('Data Density', fontsize=15)

    # Show the mean, median, and mode
    plt.axvline(x=col.mean(),    color='cyan',   linestyle='dashed', linewid
    plt.axvline(x=col.median(),  color='red',    linestyle='dashed', linewid
    plt.axvline(x=col.mode()[0], color='yellow', linestyle='dashed', linewid
    plt.legend()

    # Show the figure
    plt.show()
```

In [ ]:
```
def show_compare(df, col1, col2):

    '''
    This function makes comparison among subcategories of target variable ac
    '''
    from matplotlib.patches import Patch
    from matplotlib.lines import Line2D

    # Get statistics
    from termcolor import colored

    print(colored('Statistical Calculations :', 'red', attrs=['bold']))
    print(colored('-'*26, 'red', attrs=['bold']))
    min_val = df[col1].min()
    max_val = df[col1].max()
    mean_val = df[col1].mean()
    med_val = df[col1].median()
    mod_val = df[col1].mode()[0]

    print(colored('Minimum:{:>7.2f}\nMean:{:>10.2f}\nMedian:{:>8.2f}\nMode:{




    fig, ax = plt.subplots(figsize=(12, 6))
```

```
    ax = sns.kdeplot(data=df, x=col1, hue=col2, fill=True)

    plt.title("Data Density", fontsize=20, color="darkblue")
    ax.ticklabel_format(style='plain')

    h, l = ax.get_legend_handles_labels()

    legend_elements1 = [Line2D([0], [0], marker='s', color='lightblue', labe
                        Line2D([0], [0], marker='s', color='orange', label=df
    l1 = plt.legend(handles=legend_elements1, title='Left Type', bbox_to_anc

    legend_elements2 = [Line2D([0], [0], color='green',  label='Overall Mean
                         Line2D([0], [0], color='blue',   label='Group Mean',
                         Line2D([0], [0],  color='orange', label='Group Mean'
                         Line2D([0], [0],  color='red',    label='Median', ma
                         Line2D([0], [0],  color='yellow', label="Mode",   ma
    l2 = plt.legend(handles=legend_elements2,
                    title=f"Overall Mean {round(df[col1].mean(), 2)}\
                    \nGroup Mean {round(df.groupby([col2])[col1].mean()[0],
                    \nGroup Mean {round(df.groupby([col2])[col1].mean()[1],
                    \nOverall Median {round(np.median(df[col1]), 2)}\
                    \nOverall Mode {round(df[col1].mode()[0], 2)}",
                    bbox_to_anchor=(0.9, 0.81))

    plt.axvline(x=df[col1].mean(),    color='green',   linestyle='dashed', l
    plt.axvline(x=df[col1].median(),  color='red',     linestyle='dashed', l
    plt.axvline(x=df[col1].mode()[0], color='yellow',  linestyle='dashed', l


    group_mean1 = df.groupby([col2])[col1].mean()[0]
    group_mean2 = df.groupby([col2])[col1].mean()[1]

    plt.axvline(x=group_mean1, color='blue',  linestyle='-', linewidth=2, la
    plt.axvline(x=group_mean2, color='orange',  linestyle='-', linewidth=2,


    ax.add_artist(l1); # we need this because the 2nd call to legend() erase
```

In [ ]:
```
def measure_single_prediction_time(model, test_data):
    """
    Measure the prediction time for a machine learning model on a test data

    Args:
        model (object): The trained machine learning model.
        test_data (numpy.ndarray): A 2D array of shape (1, num_features) con

    Returns:
        float: The prediction time in seconds.
    """
    start_time = time.time()
    predictions = model.predict(test_data)
    end_time = time.time()
    prediction_time = end_time - start_time
    return prediction_time
```

In [ ]:
```
def measure_prediction_time(model, test_data, num_tests=10, method='mean'):
    """
    Measure the mean or median prediction time for a machine learning model

    Args:
        model (object): The trained machine learning model.
        test_data (numpy.ndarray): A 2D array of shape (1, num_features) con
```

```
        num_tests (int): The number of tests to perform. Defaults to 10.
        method (str): The method used to calculate the prediction time. Can

    Returns:
        float: The mean or median prediction time in seconds.
    """
    prediction_times = np.zeros(num_tests)
    for i in range(num_tests):
        prediction_times[i] = measure_single_prediction_time(model, test_dat

    if method == 'mean':
        # Calculate the mean prediction time
        prediction_time = np.mean(prediction_times)
    elif method == 'median':
        # Calculate the median prediction time
        prediction_time = np.median(prediction_times)
    else:
        raise ValueError(f"Invalid method '{method}'. Valid options are 'mea

    return prediction_time
```

In [ ]:
```
# read the data
df = pd.read_csv('HR_Dataset.csv')
```

In [ ]:
```
# create a deep copy
df0 = df.copy()
```

# 1. Exploratory Data Analysis

In [ ]:
```
# check head
df.head()
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_co |
|---|---|---|---|---|---|
| **0** | 0.38 | 0.53 | 2 | 157 | |
| **1** | 0.80 | 0.86 | 5 | 262 | |
| **2** | 0.11 | 0.88 | 7 | 272 | |
| **3** | 0.72 | 0.87 | 5 | 223 | |
| **4** | 0.37 | 0.52 | 2 | 159 | |

In [ ]:
```
# general overview of the features and data
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   Departments            14999 non-null  object 
 9   salary                 14999 non-null  object 
dtypes: float64(2), int64(6), object(2)
memory usage: 1.1+ MB
```

**DUPLICATES**

As shown below, there are more than 3000 duplicated samples in the dataset, which is a relatively high number for a dataset of 15k samples. It is normal for a dataset to have many samples with similar characteristics. However, upon careful examination of the duplicated observations, it is evident that certain features such as satisfaction level, last evaluation, and average monthly hours are also identical. Hence, these observations will be considered duplicates in this project.

Dropping these observations may potentially result in a decrease in the amount of data, which could lead to lower scores. However, as demonstrated below, we can achieve high scores even without these samples. Therefore, these samples will be excluded from the analysis.

In [ ]:
```
df.duplicated().sum()
```

Out[ ]:
```
3008
```

In [ ]:
```
df.shape
```

Out[ ]:
```
(14999, 10)
```

In [ ]:
```
# drop them
df.drop_duplicates(inplace=True)
```

In [ ]:
```
# reset the index
df.reset_index(drop=True,inplace=True)
```

In [ ]:
```
# check the new shape
df.shape
```

Out[ ]:
```
(11991, 10)
```

In [ ]:
```
# check the new data
df.head(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_co |
|---|---|---|---|---|---|
| **0** | 0.38 | 0.53 | 2 | 157 | |

In [ ]:
```
# Take the target feature to the end to have better insights from the vizs s

target_column = df['left']

# Separate the target column
target = pd.DataFrame(target_column)

# Remove the target column from the original DataFrame
features = df.drop('left', axis=1)

# Append the target column to the end
df = pd.concat([features, target], axis=1)

# Verify the updated DataFrame
df.head(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_co |
|---|---|---|---|---|---|
| **0** | 0.38 | 0.53 | 2 | 157 | |

In [ ]:
```
# check if there are any null values
df.isna().sum()
```

Out[ ]:
```
satisfaction_level       0
last_evaluation          0
number_project           0
average_montly_hours     0
time_spend_company       0
Work_accident            0
promotion_last_5years    0
Departments              0
salary                   0
left                     0
dtype: int64
```

In [ ]:
```
# check main statistics and outliers by comparing mean and std, min and 25%
df.describe().T
# any feature with a higher std than mean. An additional check with boxplots
```

Out[ ]:

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **satisfaction_level** | 11991.0 | 0.629658 | 0.241070 | 0.09 | 0.48 | 0.66 | 0.82 | 1.0 |
| **last_evaluation** | 11991.0 | 0.716683 | 0.168343 | 0.36 | 0.57 | 0.72 | 0.86 | 1.0 |
| **number_project** | 11991.0 | 3.802852 | 1.163238 | 2.00 | 3.00 | 4.00 | 5.00 | 7.0 |
| **average_montly_hours** | 11991.0 | 200.473522 | 48.727813 | 96.00 | 157.00 | 200.00 | 243.00 | 310.0 |
| **time_spend_company** | 11991.0 | 3.364857 | 1.330240 | 2.00 | 3.00 | 3.00 | 4.00 | 10.0 |
| **Work_accident** | 11991.0 | 0.154282 | 0.361234 | 0.00 | 0.00 | 0.00 | 0.00 | 1.0 |
| **promotion_last_5years** | 11991.0 | 0.016929 | 0.129012 | 0.00 | 0.00 | 0.00 | 0.00 | 1.0 |
| **left** | 11991.0 | 0.166041 | 0.372133 | 0.00 | 0.00 | 0.00 | 0.00 | 1.0 |

In [ ]:
```
# create a dataframe from numerical features for statistical analysis
df_numeric =  df.select_dtypes('number')
df_numeric
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spen |
|---|---|---|---|---|---|
| **0** | 0.38 | 0.53 | 2 | 157 | |
| **1** | 0.80 | 0.86 | 5 | 262 | |
| **2** | 0.11 | 0.88 | 7 | 272 | |
| **3** | 0.72 | 0.87 | 5 | 223 | |
| **4** | 0.37 | 0.52 | 2 | 159 | |
| **...** | ... | ... | ... | ... | |
| **11986** | 0.90 | 0.55 | 3 | 259 | |
| **11987** | 0.74 | 0.95 | 5 | 266 | |
| **11988** | 0.85 | 0.54 | 3 | 185 | |
| **11989** | 0.33 | 0.65 | 3 | 172 | |
| **11990** | 0.50 | 0.73 | 4 | 180 | |

11991 rows × 8 columns

In [ ]:
```
# check categorical features
df["Departments "].unique()
```

Out[ ]:
```
array(['sales', 'accounting', 'hr', 'technical', 'support', 'management',
       'IT', 'product_mng', 'marketing', 'RandD'], dtype=object)
```

In [ ]:
```
df["salary"].unique()
```

Out[ ]:
```
array(['low', 'medium', 'high'], dtype=object)
```

## 2. Data Visualization

df.columns

In [ ]:
```
# check columns
df.columns
# attention: the space in the end of Departments
```

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

### What is the distribution of the "left" variable? How many employees left the company and how many stayed?

In [ ]:
```
df['left'].value_counts()
```

Out[ ]:
```
left
0    10000
1     1991
Name: count, dtype: int64
```

In [ ]:
```
df['left'].value_counts(normalize = True)
```

Out[ ]:
```
left
0    0.833959
1    0.166041
Name: proportion, dtype: float64
```

**Important Note** : The data is imbalanced.

The balance or imbalance of the target feature in binary classification refers to the distribution of the two classes within the dataset. A balanced dataset means that the number of instances in each class is roughly equal, while an imbalanced dataset indicates a significant disparity in class frequencies. The importance of the balance or imbalance of the target feature in binary classification lies in the potential impact on the model's performance and the interpretation of its results. Here are a few reasons why it is important:

1) **Model Performance**: Imbalanced classes can introduce bias towards the majority class, making it challenging for the model to accurately predict the minority class. If one class dominates the dataset, the model may become biased towards predicting the majority class, resulting in lower accuracy, precision, recall, and F1 scores for the minority class. Balancing the target feature can help mitigate this issue and improve the overall performance of the model.

2) **Training Bias**: Machine learning algorithms aim to minimize the overall error or loss function during training. In imbalanced datasets, the model can achieve high accuracy by simply predicting the majority class for most instances, which leads to a biased model. Balancing the target feature helps ensure that the model receives sufficient exposure to both classes, preventing it from favoring one class over the other.

3) **Evaluation and Interpretation**: When evaluating a binary classification model, accuracy alone may not provide an accurate representation of its performance, especially in imbalanced datasets. Other metrics like precision, recall, and F1 score become more relevant. With imbalanced classes, the model may achieve high accuracy by primarily predicting the majority class, while failing to detect or accurately classify instances of the minority class. Balancing the target feature allows for a more meaningful evaluation of the model's performance on both classes.

4) **Cost-Sensitive Applications**: In certain domains, misclassification errors can have different costs or consequences for each class. For example, in fraud detection, misclassifying a fraudulent transaction as legitimate may have a higher cost than classifying a legitimate transaction as fraudulent. In such cases, balancing the target feature becomes important to ensure that the model is equally sensitive to both classes and minimizes the overall cost of misclassifications.

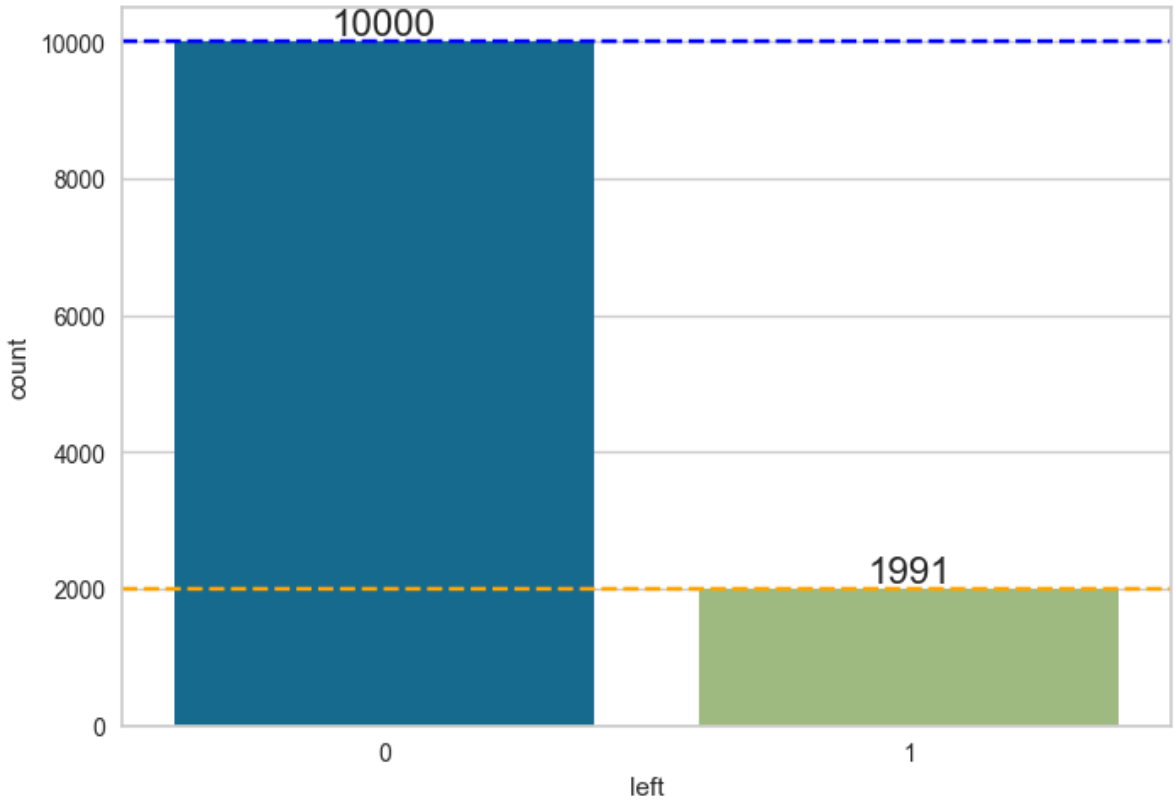
5) **Data Collection and Sampling Bias**: Imbalanced classes can indicate underlying biases in data collection or sampling methods. Addressing class imbalance helps identify and rectify any biases in the dataset, ensuring a fair representation of both classes and reducing the potential for skewed results or discriminatory outcomes. In summary, balancing the target feature in binary classification is important to improve model performance, prevent bias, enable accurate evaluation, handle cost-sensitive scenarios,

and mitigate biases in data collection or sampling. It allows for a more robust and reliable analysis, ensuring that the model is effective for both classes and produces fair and accurate predictions.

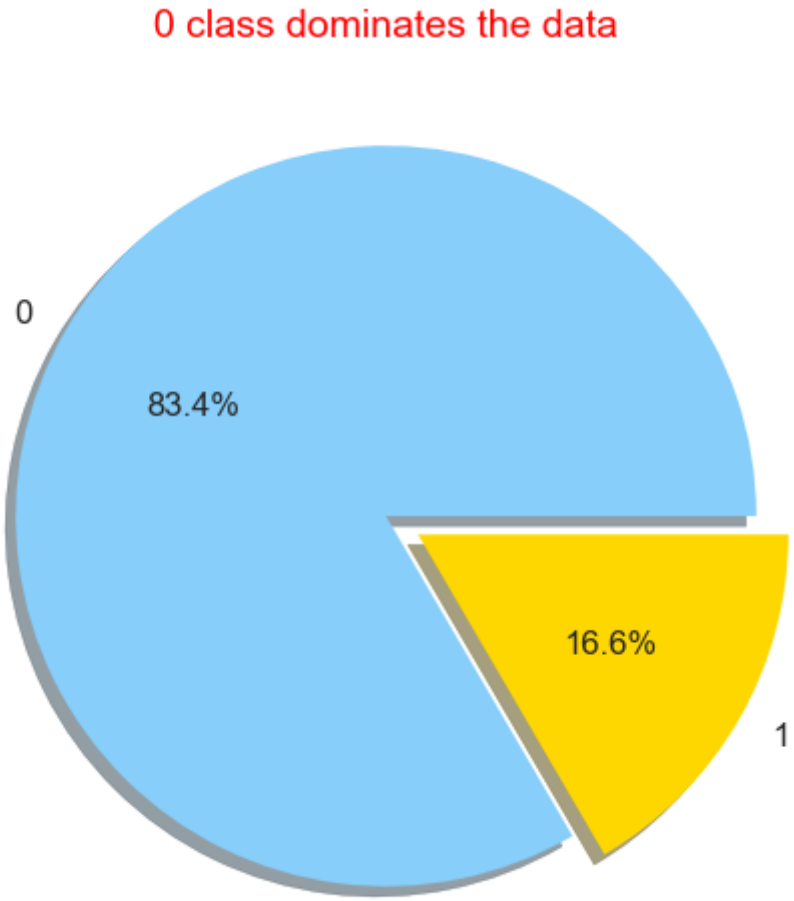
Therefore, we will use class_weight parameter to increase the scores for each class, in particular 1 class forthe target feature "left". If the scores of the class with less samples can't be improved, various techniques such as smote or resampling will be deployed.

In [ ]:
```
# visualize the distribution of the classes for target feature
display(df.left.value_counts())
ax = sns.countplot(x=df["left"])
ax.bar_label(ax.containers[0], size=16)
plt.axhline(y=df.left.value_counts()[1], color='orange', linestyle='--')
plt.axhline(y=df.left.value_counts()[0], color='blue', linestyle='--');
```

```
left
0    10000
1     1991
Name: count, dtype: int64
```




In [ ]:
```
# Visualise the percentage of distribution in the target feature

fig, ax = plt.subplots(figsize=(6, 6))

ax.pie(x=df.left.value_counts().values,
       labels=[0, 1],
       autopct='%.1f%%',
       explode=(0, 0.1),
       colors=['lightskyblue', 'gold'],
       textprops={'fontsize': 12},
       shadow=True
       )
plt.title("0 class dominates the data", fontdict={"color":"red",'fontsize':
plt.show()
```

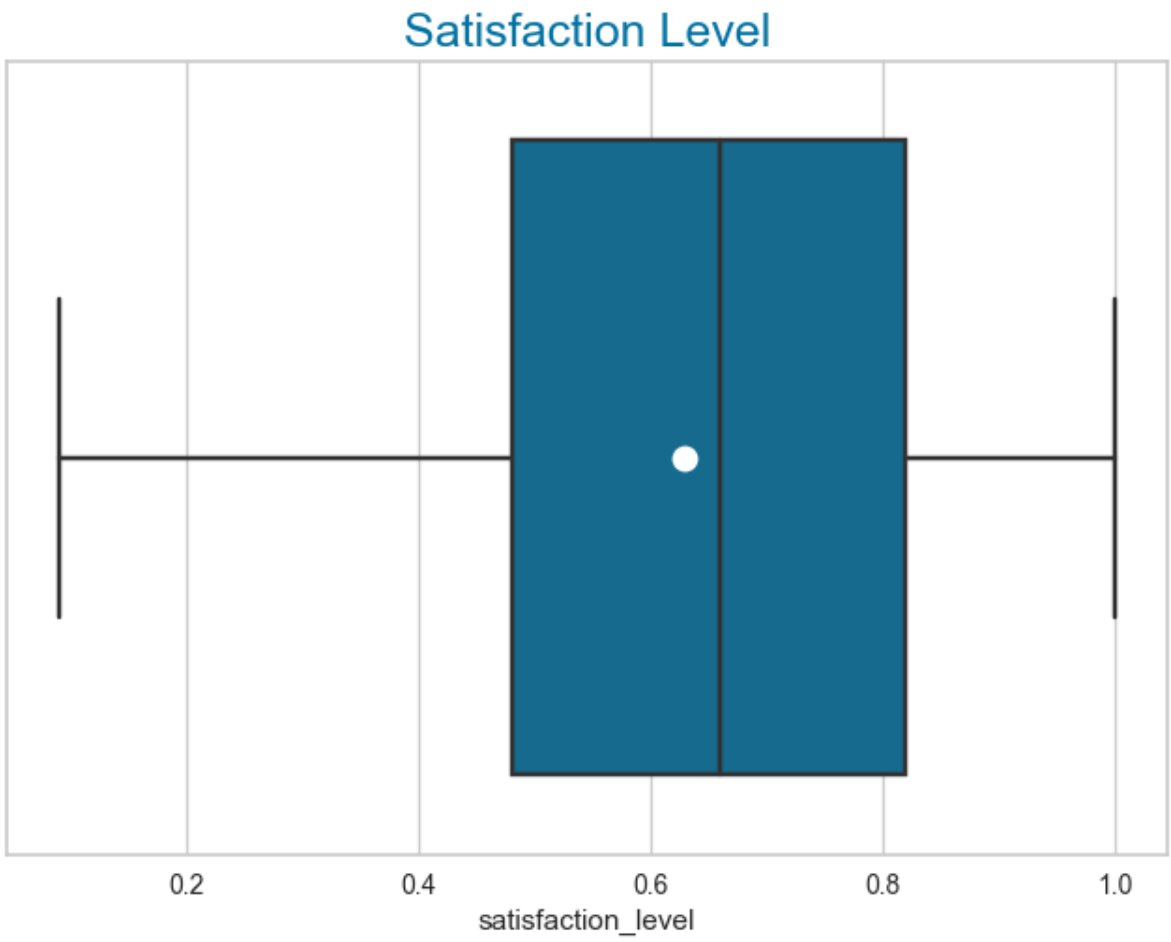



## Numeric Features

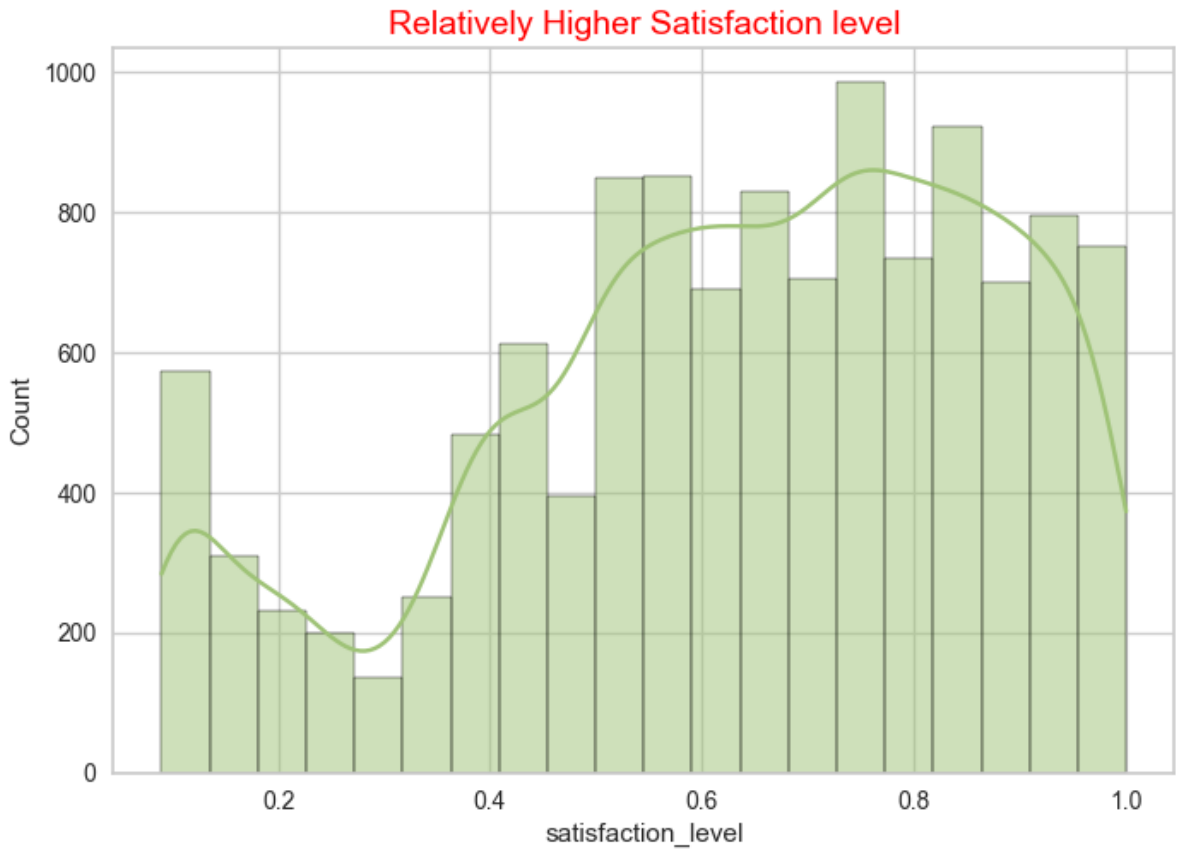
As part of the exploratory data analysis process, both numerical and categorical features will be analyzed below.

### satisfaction level

In [ ]:
```
sns.boxplot(data=df,
            x="satisfaction_level",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

plt.title("Satisfaction Level", fontsize=18, color="b");
```




In [ ]:
```
# Displaying the distribution of satisfaction_level feature with a histogram
sns.histplot(data=df, x="satisfaction_level", bins=20, kde=True, color="g")
plt.title("Relatively Higher Satisfaction level", fontsize=14, color="red");
```



### Check the statistical values

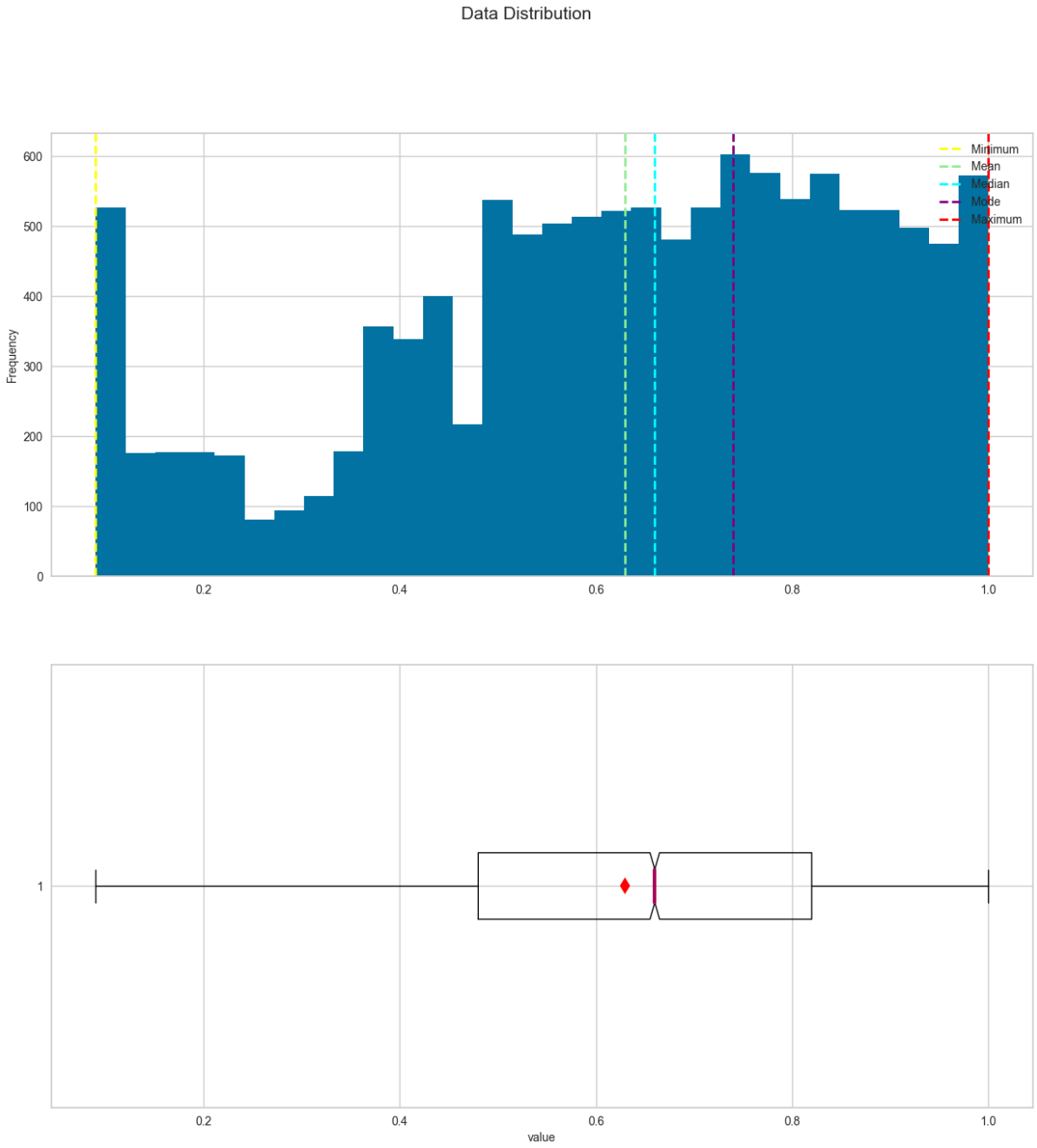
In [ ]:
```
# Descriptive Statistics of "satisfaction_level" Feature

print('Descriptive Statistics of the satisfaction_level Feature:\n')
df.satisfaction_level.describe()
```

```
Descriptive Statistics of the satisfaction_level Feature:
```

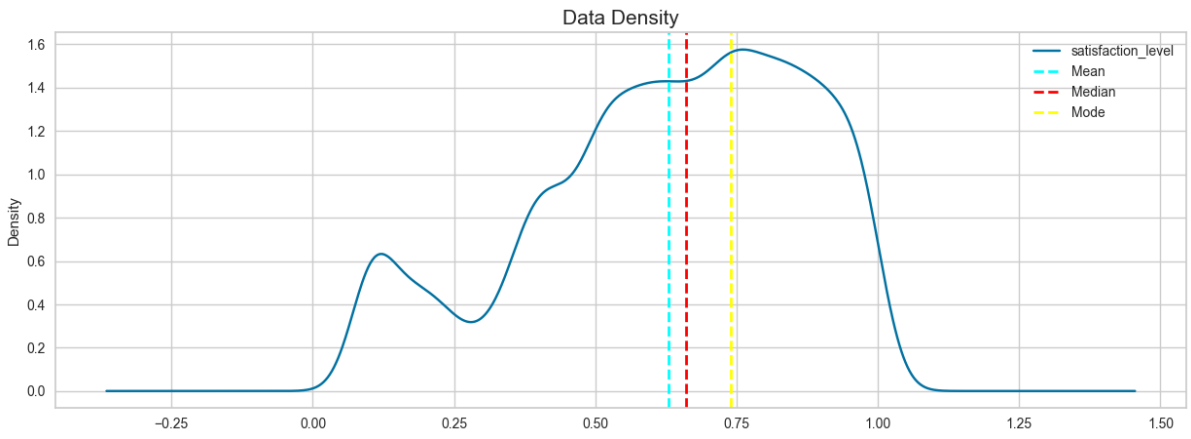
Out[ ]:
```
count    11991.000000
mean         0.629658
std          0.241070
min          0.090000
25%          0.480000
50%          0.660000
75%          0.820000
max          1.000000
Name: satisfaction_level, dtype: float64
```

In [ ]:
```
show_distribution(df["satisfaction_level"])
```

```
Statistical Calculations :
--------------------------
Minimum:   0.09
Mean:      0.63
Median:    0.66
Mode:      0.74
Maximum:   1.00
```



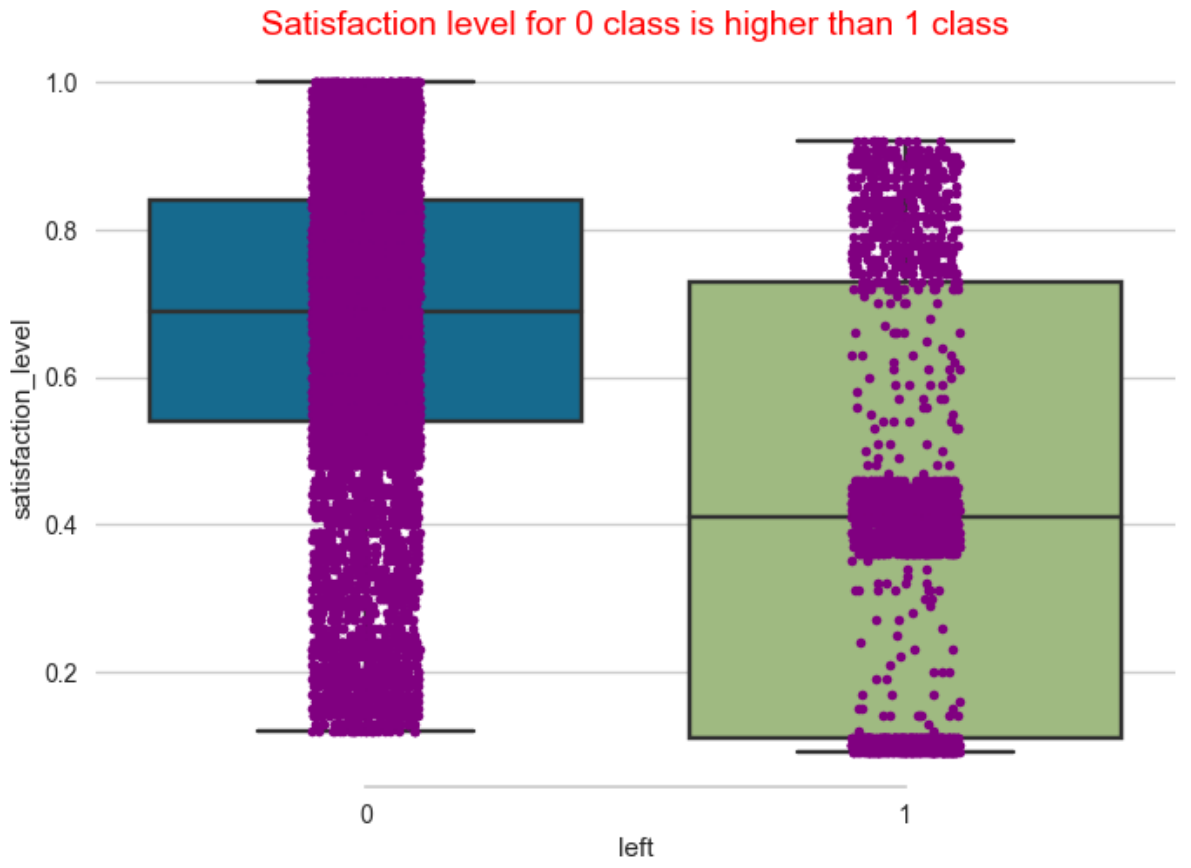

In [ ]:
```
show_density(df["satisfaction_level"])
```



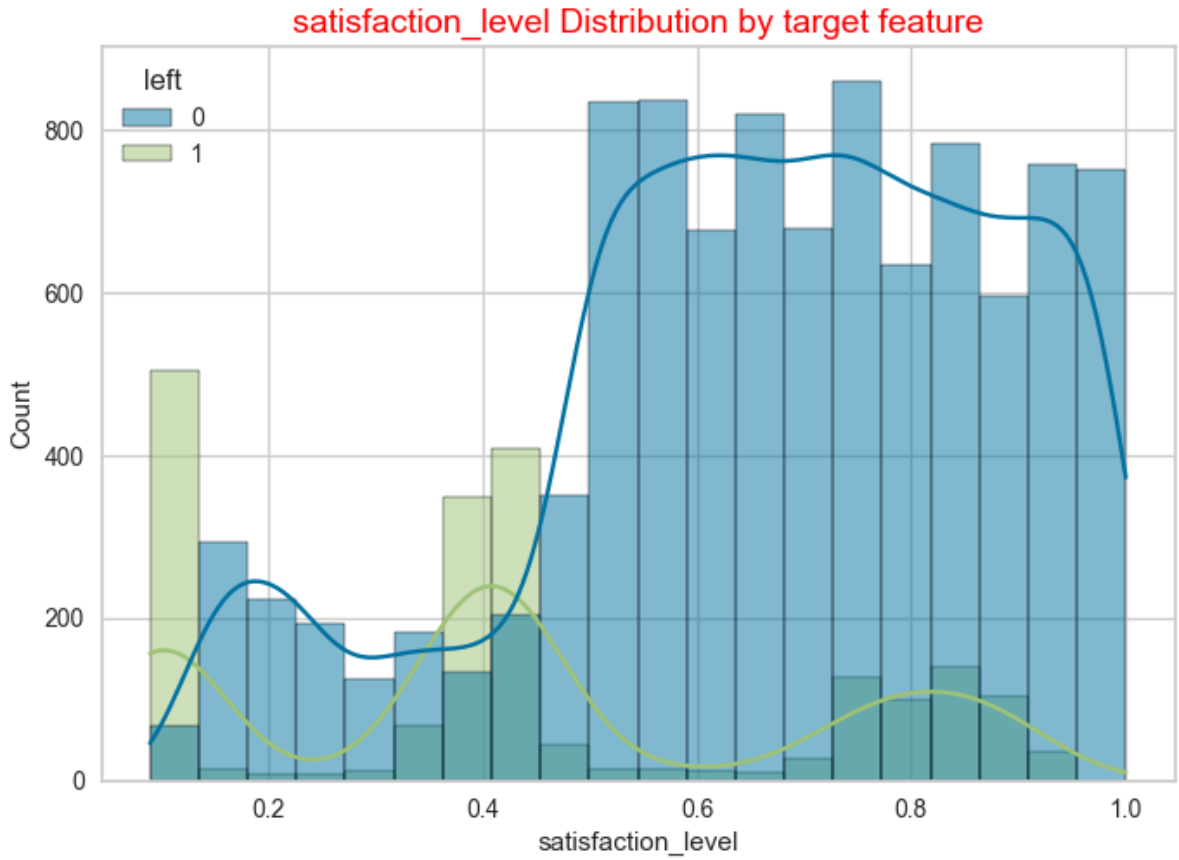

**Check the boxplot and histplot/kdeplot by "left" levels**

In [ ]:
```
# Checking the extreme values in "satisfaction_level" feature by left with b

sns.boxplot(data=df,
```

```
            y="satisfaction_level",
            x="left",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

# Add in points to show each observation
sns.stripplot(data=df,
              y="satisfaction_level",
              x="left",
              size=4,
              color="purple",
              linewidth=0)

# Tweaking the visual presentation
ax.xaxis.grid(True)
ax.set(ylabel="")
sns.despine(trim=True, left=True)

plt.title("Satisfaction level for 0 class is higher than 1 class", fontsize=
```




In [ ]:
```
# Checking Density Distribution of "satisfaction_level" feature by left

sns.histplot(data=df, x="satisfaction_level", bins=20, kde=True, hue="left")

plt.title("satisfaction_level Distribution by target feature", fontsize=14,
```




**Check the statistical values by "left" levels**

In [ ]:
```
# Descriptive Statistics of satisfaction_level with respect to left levels

print(colored('Descriptive Statistics of the satisfaction_level by left:\n',
df.groupby("left").satisfaction_level.describe()
```

**Descriptive Statistics of the satisfaction_level by left:**

Out[ ]:

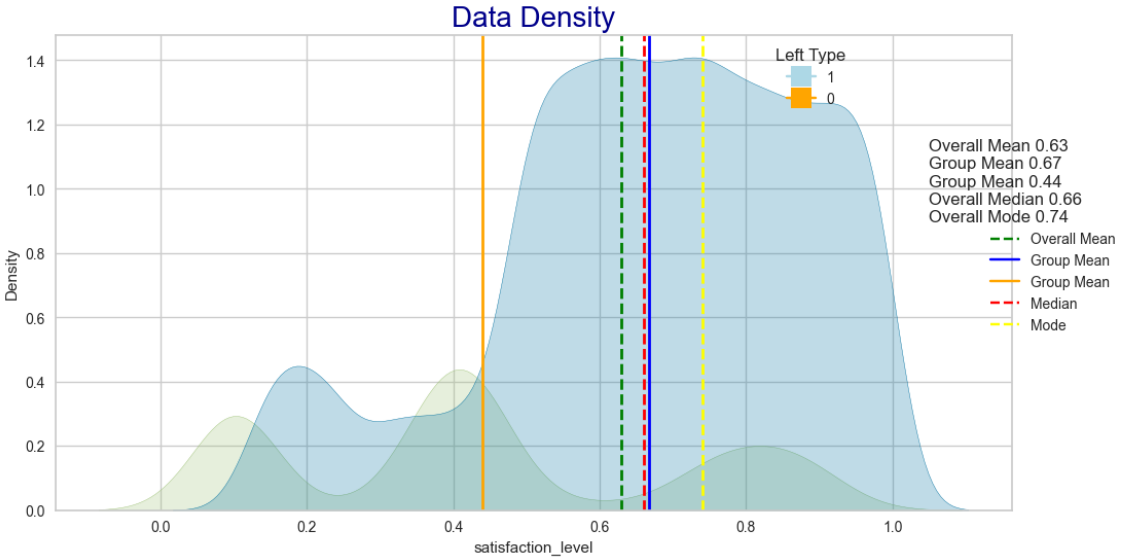
| left | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **0** | 10000.0 | 0.667365 | 0.217082 | 0.12 | 0.54 | 0.69 | 0.84 | 1.00 |
| **1** | 1991.0 | 0.440271 | 0.265207 | 0.09 | 0.11 | 0.41 | 0.73 | 0.92 |

In [ ]:
```
show_compare(df, "satisfaction_level", "left")
```

**Statistical Calculations :**

---

**Minimum:** 0.09
**Mean:** 0.63
**Median:** 0.66
**Mode:** 0.74
**Maximum:** 1.00



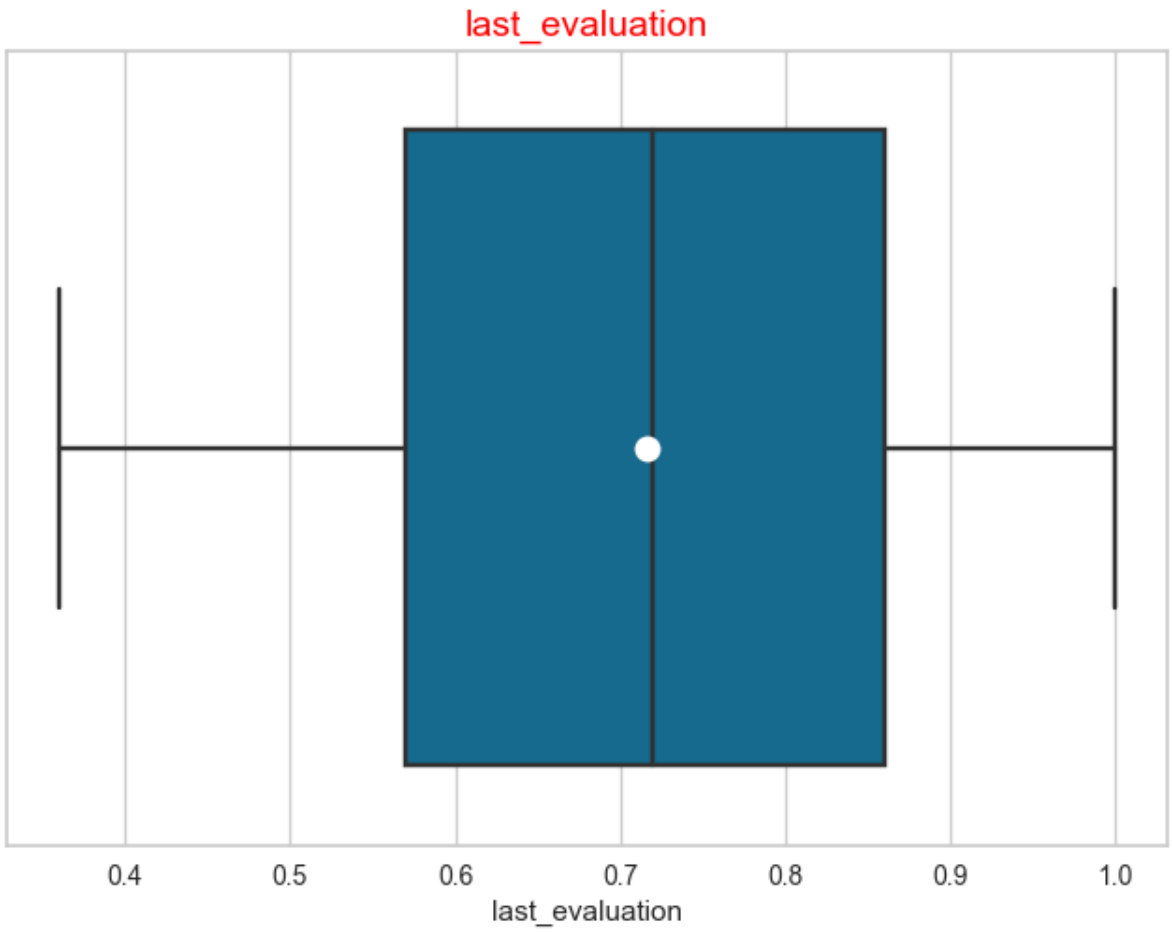

**A general assessment for satisfaction level:**

As observed in the visualization above, the overall satisfaction level of the employees is relatively high, indicating right-skewed data from a technical perspective. In terms of the distribution of the satisfaction level for each class, namely the employee group who left the company (1) and who still continue to work (0 class), it is clear that the general satisfaction level of 0 class is higher than 1 class, which has been emphasized by many scientific empirical studies (For a detailed essay written by Lambert and Hogan that emphasize the importance of employee satisfaction in decreasing the turnover rates click here)

It is also worth noting that this distribution provides valuable insights. However, it is important to acknowledge that without information about the general satisfaction levels of employees in other companies within the same sector, making definitive comments about the overall level of employee satisfaction in this particular company becomes challenging.

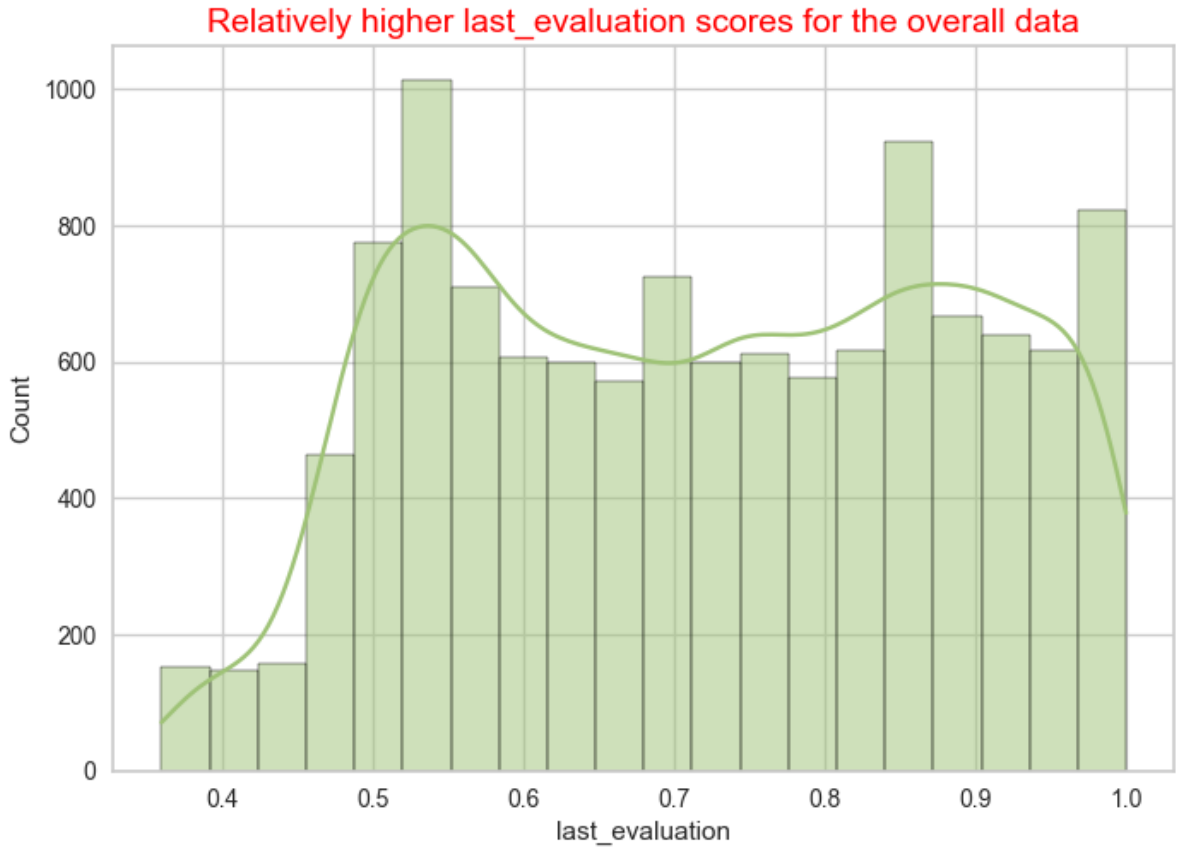
## last_evaluation

In [ ]:
```
sns.boxplot(data=df,
            x="last_evaluation",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

plt.title("last_evaluation", fontsize=14, color="red");
```




In [ ]:

```
# Display the distribution of last_evaluation feature with a histogram

sns.histplot(data=df, x="last_evaluation", bins=20, kde=True, color="g")

plt.title("Relatively higher last_evaluation scores for the overall data", f
```



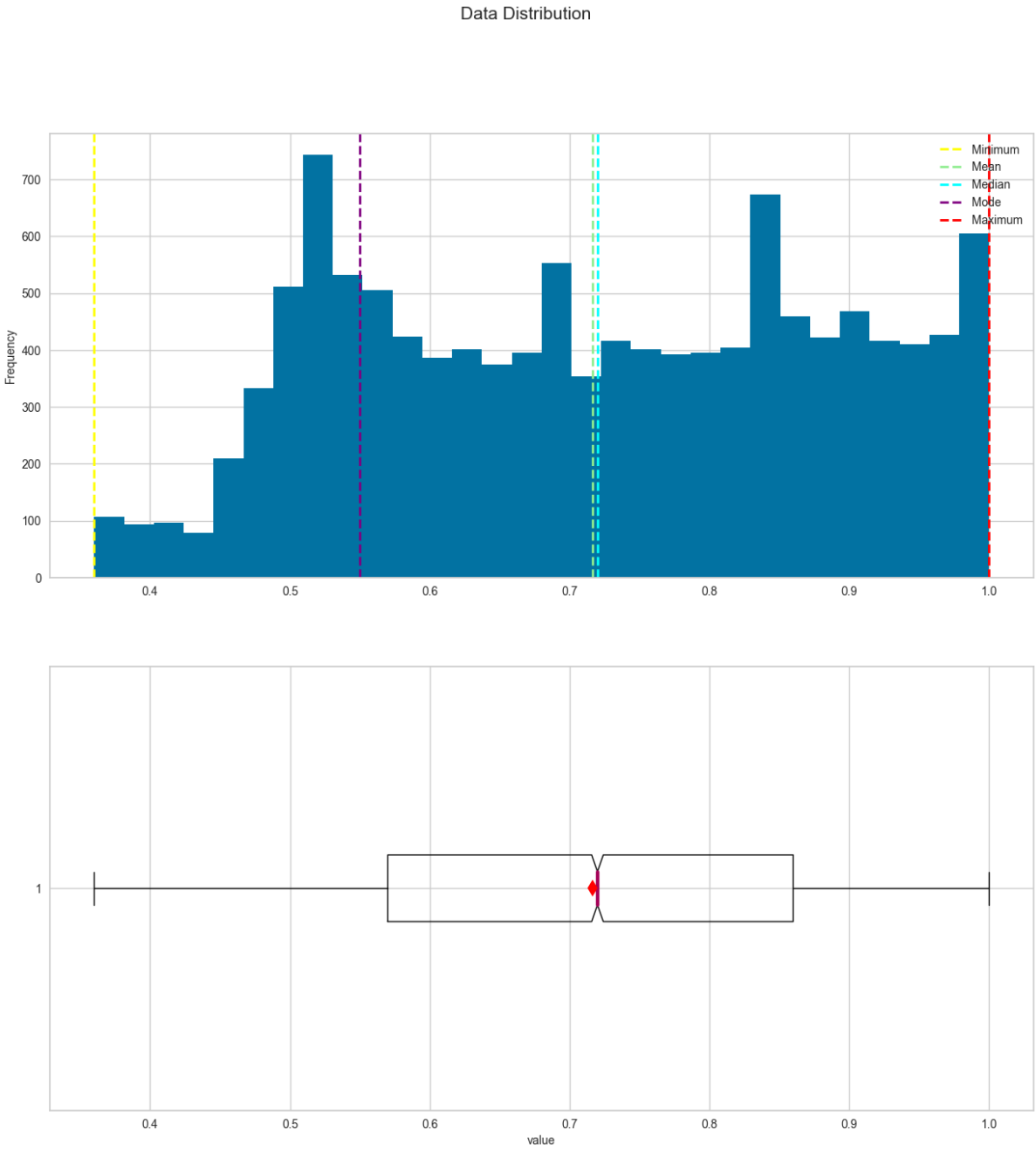
In [ ]:
```
# Descriptive Statistics of "last_evaluation" Feature

print('Descriptive Statistics of the last_evaluation Feature:\n')
df.last_evaluation.describe()
```

```
Descriptive Statistics of the last_evaluation Feature:
```

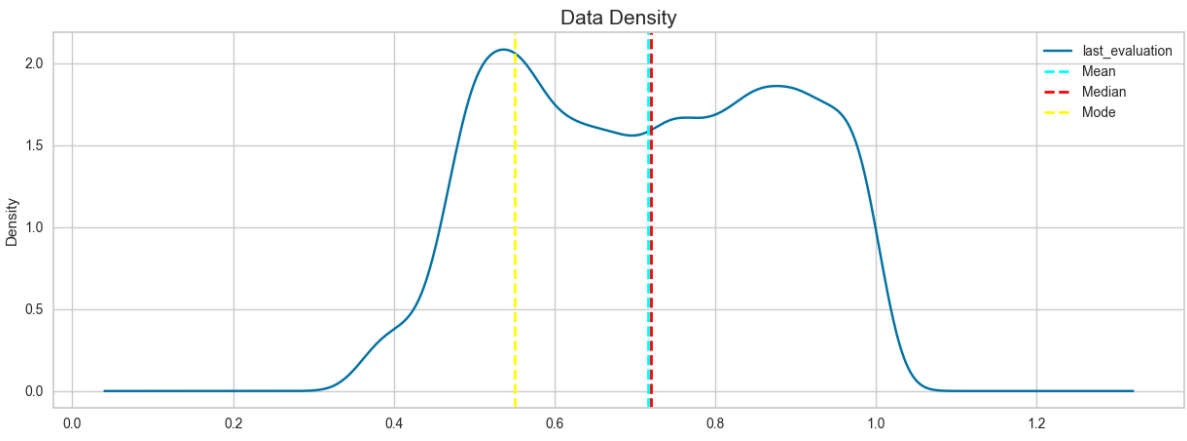
Out[ ]:
```
count    11991.000000
mean         0.716683
std          0.168343
min          0.360000
25%          0.570000
50%          0.720000
75%          0.860000
max          1.000000
Name: last_evaluation, dtype: float64
```

In [ ]:
```
show_distribution(df["last_evaluation"])
```

```
Statistical Calculations :
--------------------------
Minimum:   0.36
Mean:      0.72
Median:    0.72
Mode:      0.55
Maximum:   1.00
```



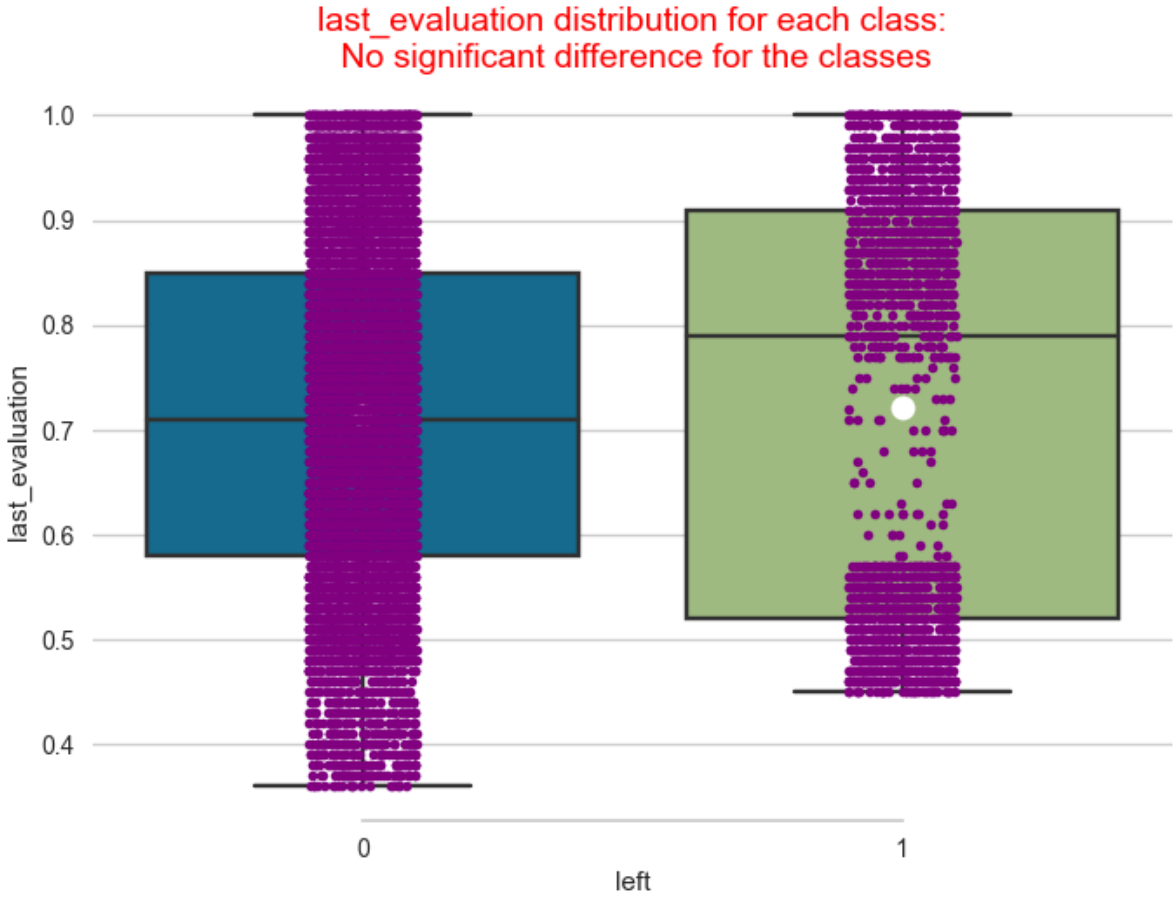

In [ ]:
```
show_density(df["last_evaluation"])
```



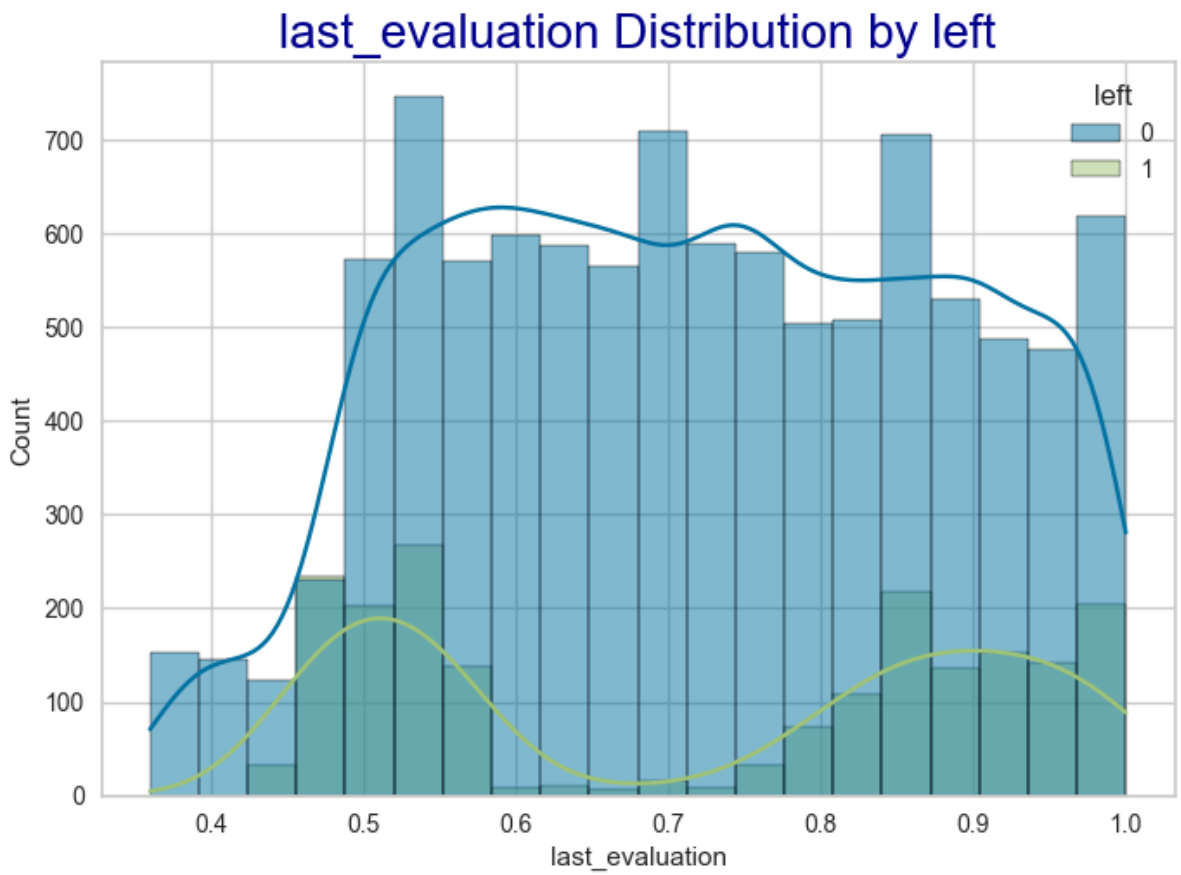

**Check the boxplot and histplot/kdeplot by "left" levels**

In [ ]:
```
# Checking the extreme values in "last_evaluation" feature by left with box

sns.boxplot(data=df,
```

```
            y="last_evaluation",
            x="left",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

# Add in points to show each observation
sns.stripplot(data=df,
              y="last_evaluation",
              x="left",
              size=4,
              color="purple",
              linewidth=0)

# Tweaking the visual presentation
ax.xaxis.grid(True)
ax.set(ylabel="")
sns.despine(trim=True, left=True)

plt.title("last_evaluation distribution for each class:\n No significant dif
          fontsize=14, color="red");
```




In [ ]:
```
# Check Density Distribution of "last_evaluation" feature by left

sns.histplot(data=df, x="last_evaluation", bins=20, kde=True, hue="left")

plt.title("last_evaluation Distribution by left", fontsize=20, color="darkbl
```




**Check the statistical values by "left" levels**

In [ ]:
```
# Descriptive Statistics of last_evaluation with respect to left levels

print(colored('Descriptive Statistics of the last_evaluation by left:\n', 'b
df.groupby("left").last_evaluation.describe()
```

**Descriptive Statistics of the last_evaluation by left:**

Out[ ]:

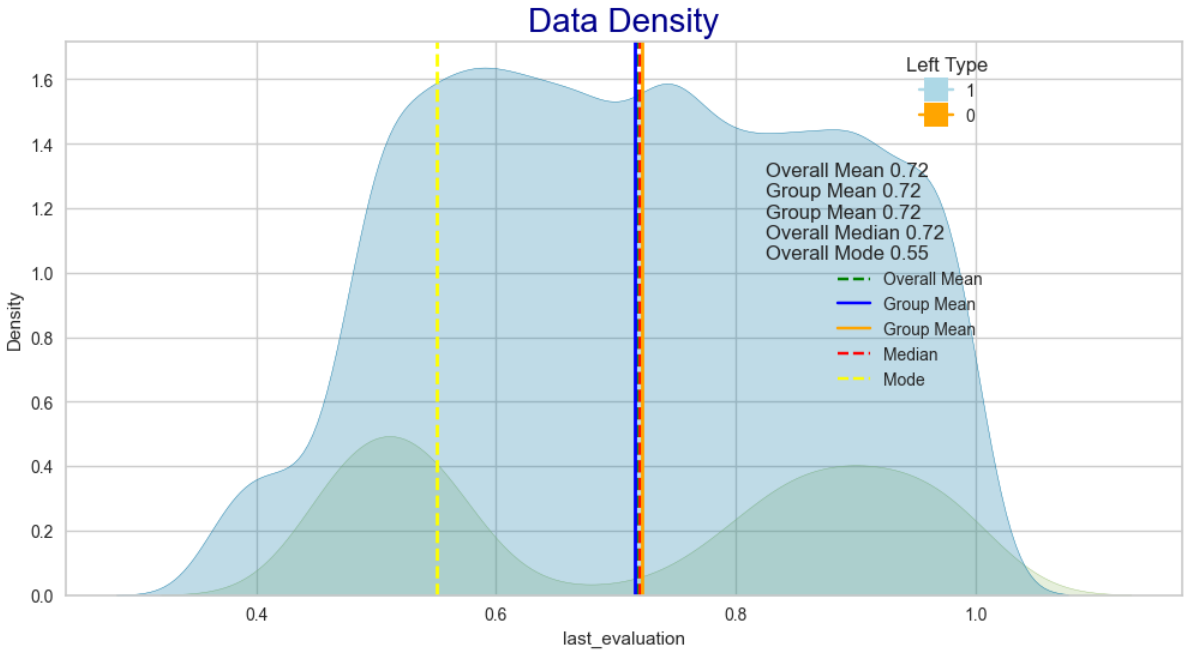
| left | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **0** | 10000.0 | 0.715667 | 0.161919 | 0.36 | 0.58 | 0.71 | 0.85 | 1.0 |
| **1** | 1991.0 | 0.721783 | 0.197436 | 0.45 | 0.52 | 0.79 | 0.91 | 1.0 |

In [ ]:
```
show_compare(df, "last_evaluation", "left")
```

**Statistical Calculations :**

---

**Minimum:** 0.36
**Mean:** 0.72
**Median:** 0.72
**Mode:** 0.55
**Maximum:** 1.00

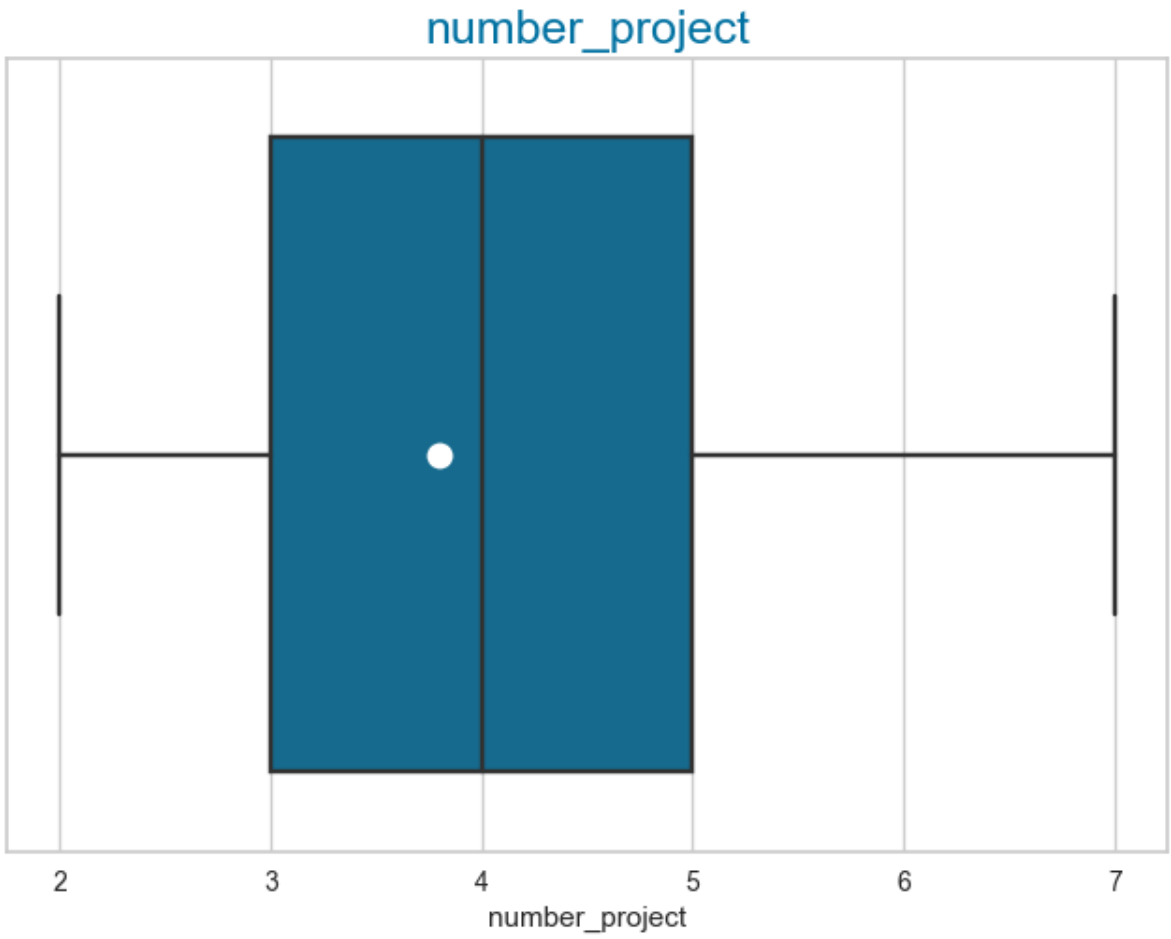



**A general assessment for last_evaluation feature:**

The visualizations above regarding the "last evaluation" feature indicate that this particular feature does not have a significant effect on an employee's decision to stay in or leave the company. However, to confirm this assessment, a statistical significance test will be conducted below.

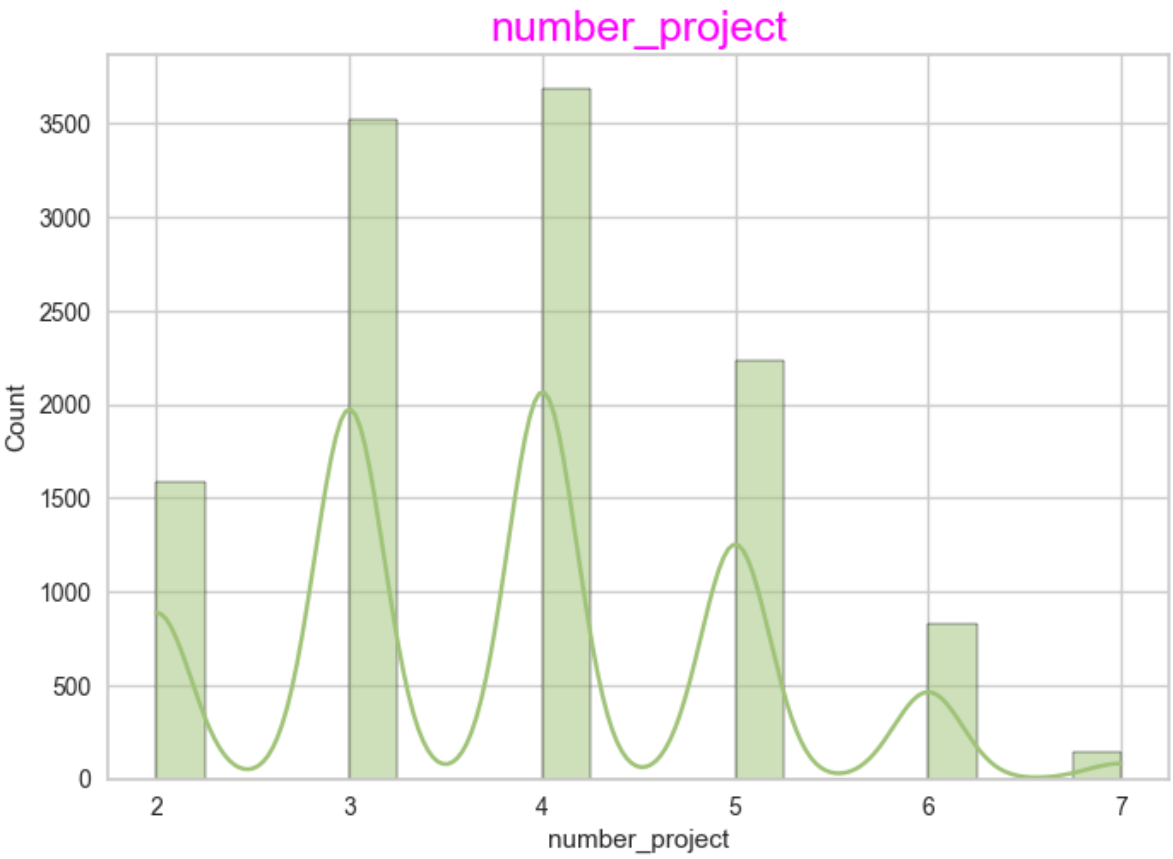
## number_project

In [ ]:
```
sns.boxplot(data=df,
            x="number_project",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

plt.title("number_project", fontsize=18, color="b");
```




In [ ]:

```
# Displaying the distribution of number_project feature with a histogram

sns.histplot(data=df, x="number_project", bins=20, kde=True, color="g")

plt.title("number_project", fontsize=18, color="fuchsia");
```



### Check the statistical values

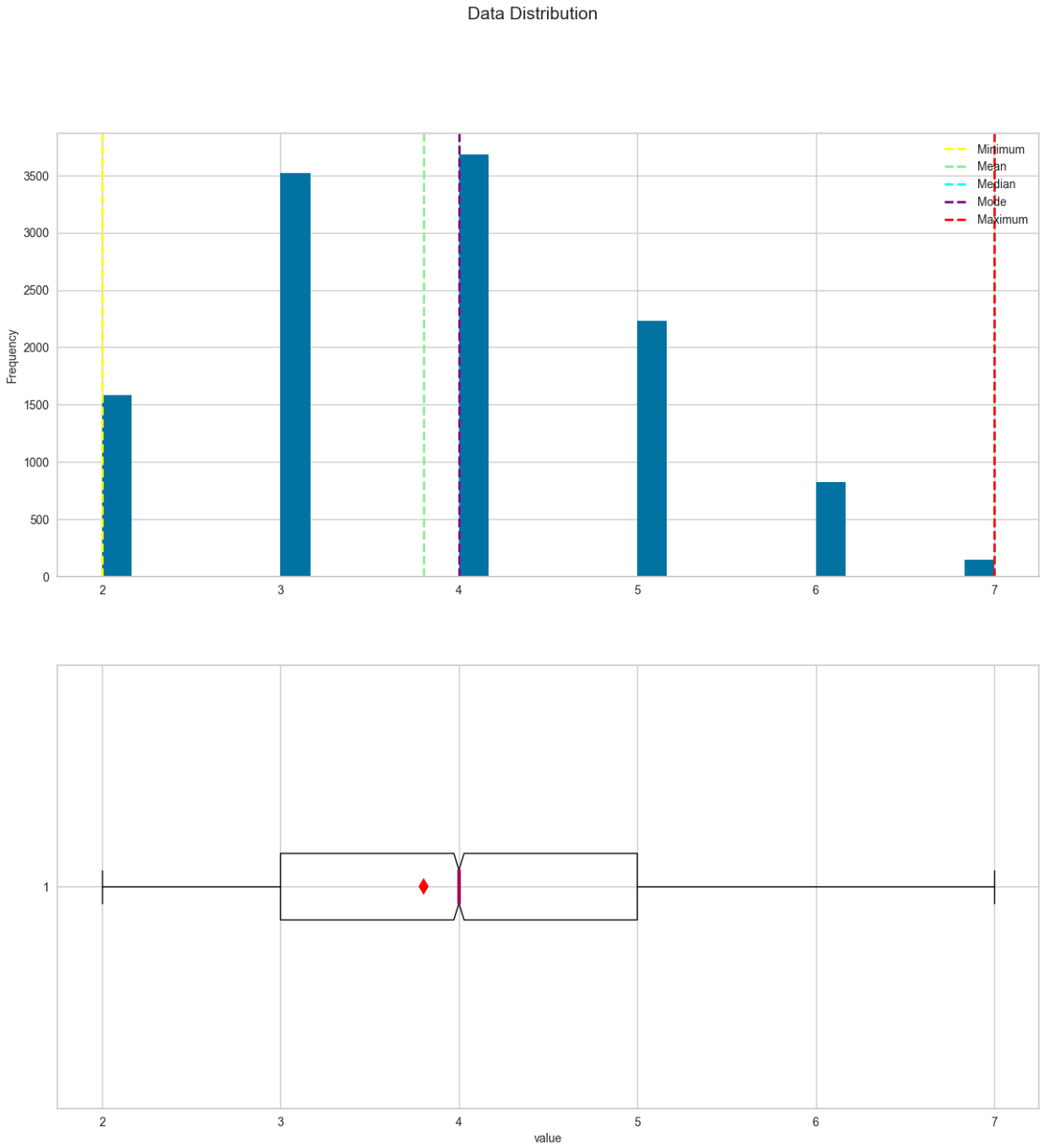
In [ ]:
```
# Descriptive Statistics of "number_project" Feature

print('Descriptive Statistics of the number_project Feature:\n')
df.number_project.describe()
```

```
Descriptive Statistics of the number_project Feature:
```

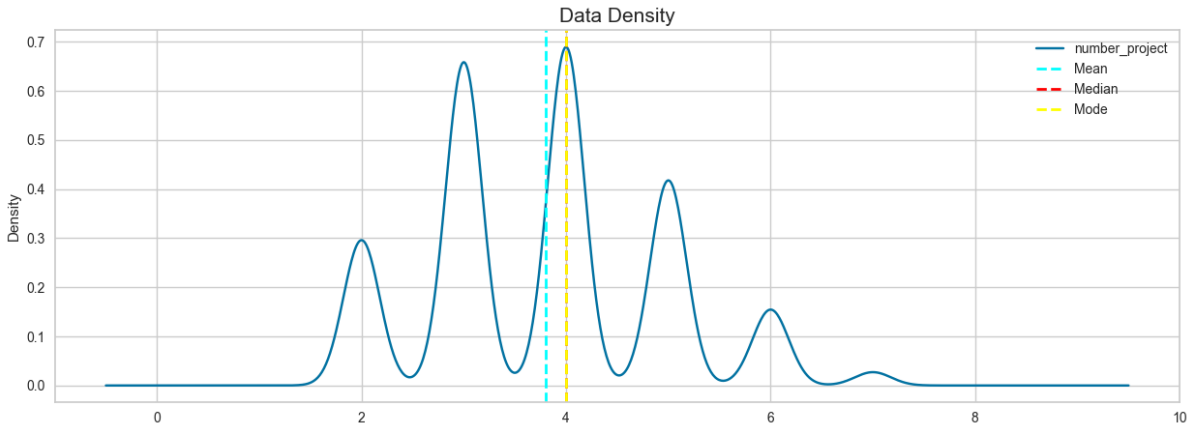
Out[ ]:
```
count    11991.000000
mean         3.802852
std          1.163238
min          2.000000
25%          3.000000
50%          4.000000
75%          5.000000
max          7.000000
Name: number_project, dtype: float64
```

In [ ]:
```
show_distribution(df["number_project"])
```

```
Statistical Calculations :
--------------------------
Minimum:   2.00
Mean:      3.80
Median:    4.00
Mode:      4.00
Maximum:   7.00
```



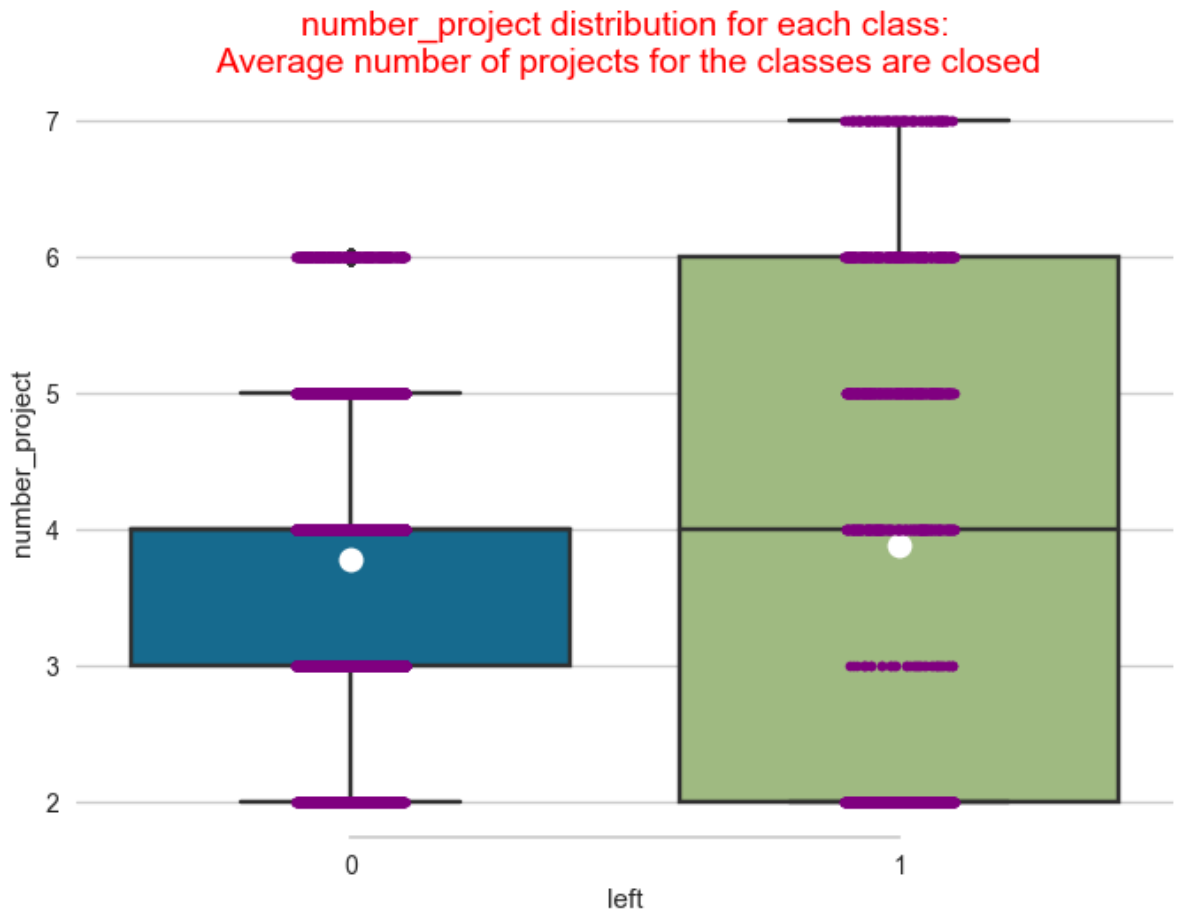

In [ ]:
```
show_density(df["number_project"])
```



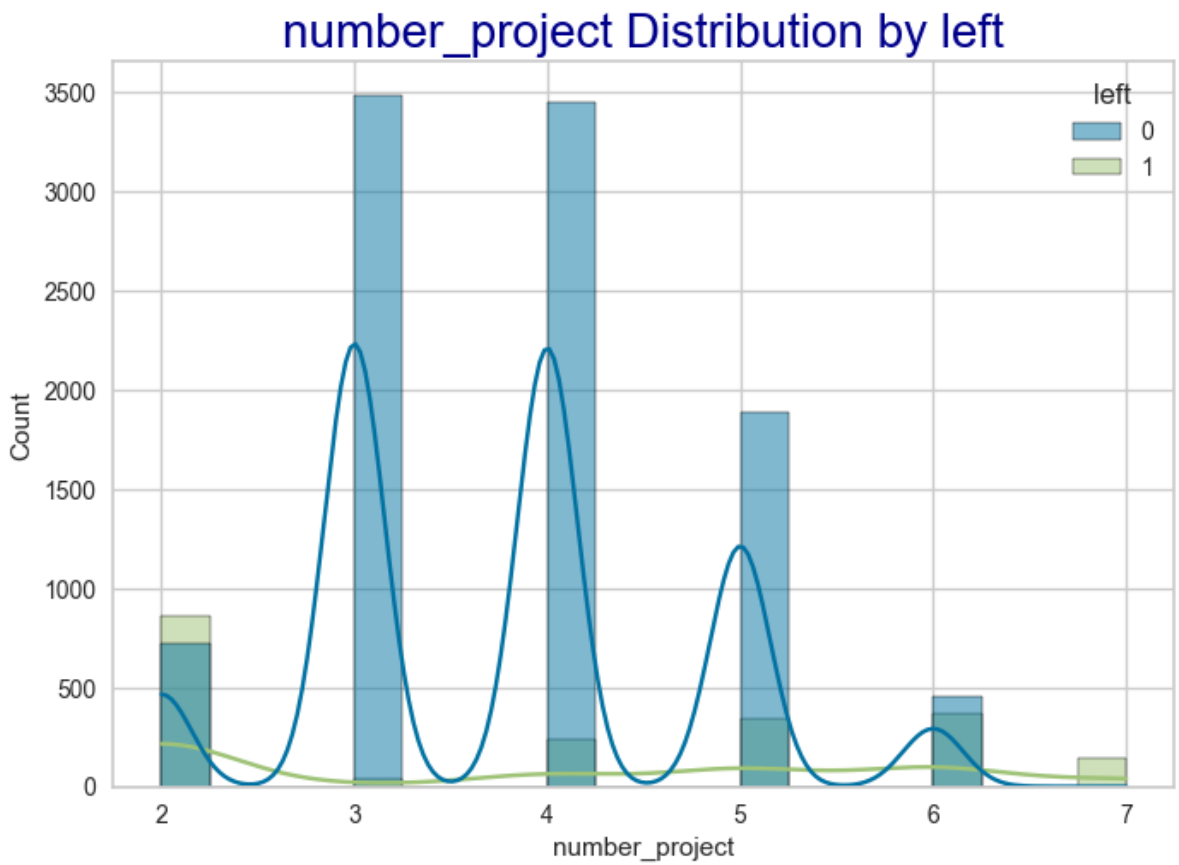

**Check the boxplot and histplot/kdeplot by "left" levels**

In [ ]:
```
# Checking the extreme values in "number_project" feature by left with box p

sns.boxplot(data=df,
```

```
            y="number_project",
            x="left",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

# Add in points to show each observation
sns.stripplot(data=df,
              y="number_project",
              x="left",
              size=4,
              color="purple",
              linewidth=0)

# Tweaking the visual presentation
ax.xaxis.grid(True)
ax.set(ylabel="")
sns.despine(trim=True, left=True)

plt.title("number_project distribution for each class:\n Average number of p
          fontsize= 14, color="red");
```




In [ ]:
```
# Checking Density Distribution of "number_project" feature by left

sns.histplot(data=df, x="number_project", bins=20, kde=True, hue="left")

plt.title("number_project Distribution by left", fontsize=20, color="darkblu
```




**Check the statistical values by "left" levels**

In [ ]:
```
# Descriptive Statistics of number_project with respect to left levels

print(colored('Descriptive Statistics of the number_project by left:\n', 'bl
df.groupby("left").number_project.describe()
```

**Descriptive Statistics of the number_project by left:**

Out[ ]:

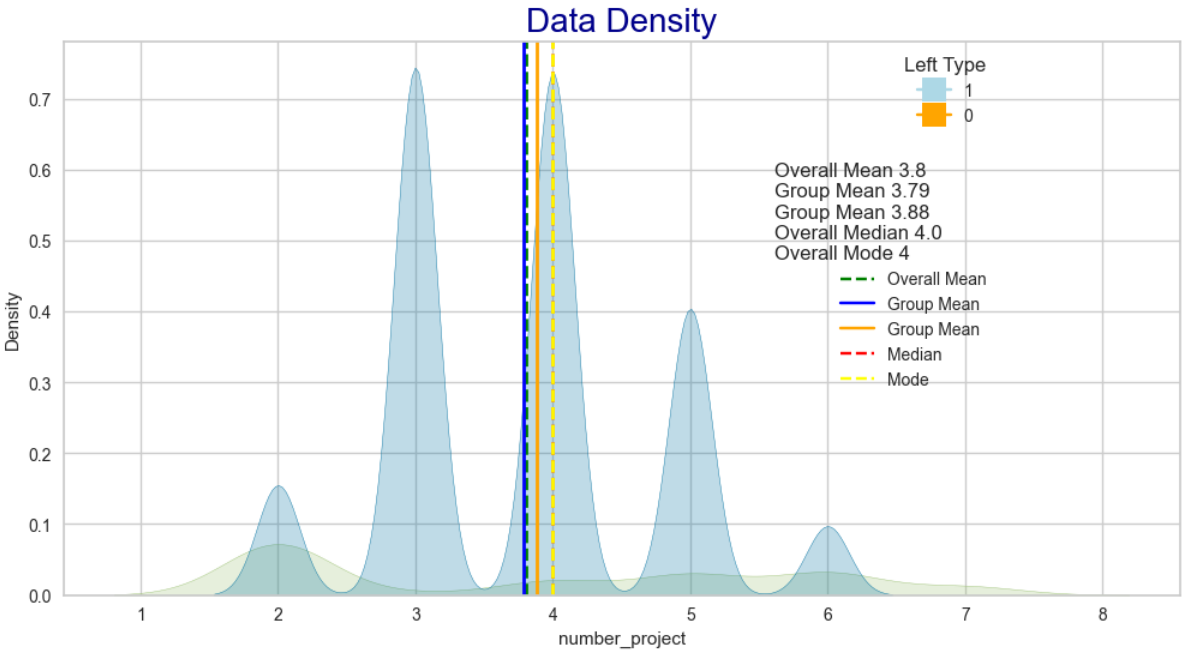
| left | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **0** | 10000.0 | 3.786800 | 0.981755 | 2.0 | 3.0 | 4.0 | 4.0 | 6.0 |
| **1** | 1991.0 | 3.883476 | 1.817139 | 2.0 | 2.0 | 4.0 | 6.0 | 7.0 |

In [ ]:
```
show_compare(df, "number_project", "left")
```

**Statistical Calculations :**

---

**Minimum:** 2.00
**Mean:** 3.80
**Median:** 4.00
**Mode:** 4.00
**Maximum:** 7.00




**A general assesment for number_projects feature**

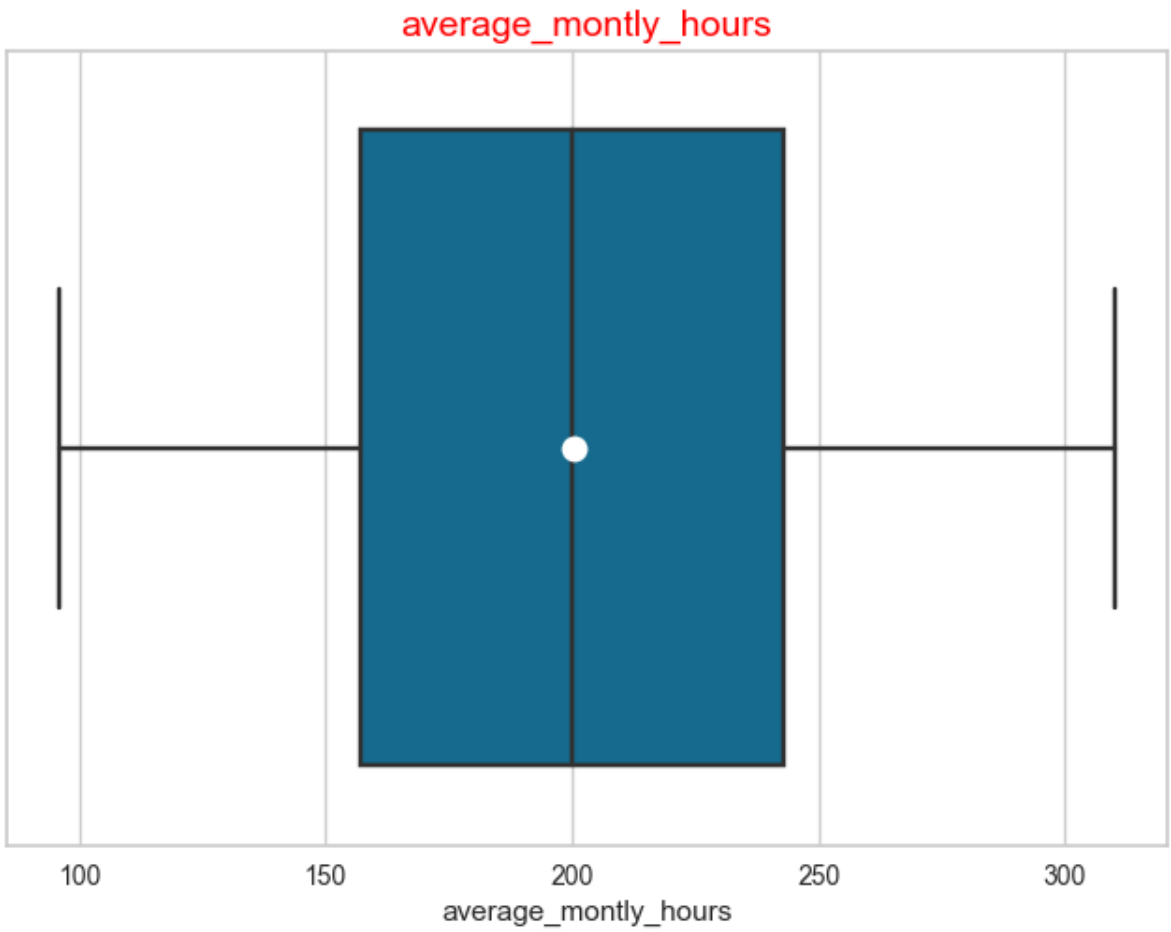
Number of projects that an employee is responsible for is a main indicator of the work load. Workload is a significant determinant of employees' happiness and their decision to continue working in a company. As emphasized by Shahzad et al. in their recent empirical study, "Work overload is often considered the most critical factor that affects employees' commitment and turnover intention towards the organization." The authors suggest that work overload has a negative impact on organizational commitment while positively influencing employees' turnover intentions.

However, workload has a two-way effect. An employee who is given very little work may not want to stay in the company, while an employee who is given excessive work may also want to leave the company. Therefore, extreme values of the number of projects assigned to an employee can be crucial in their decision to leave the job.

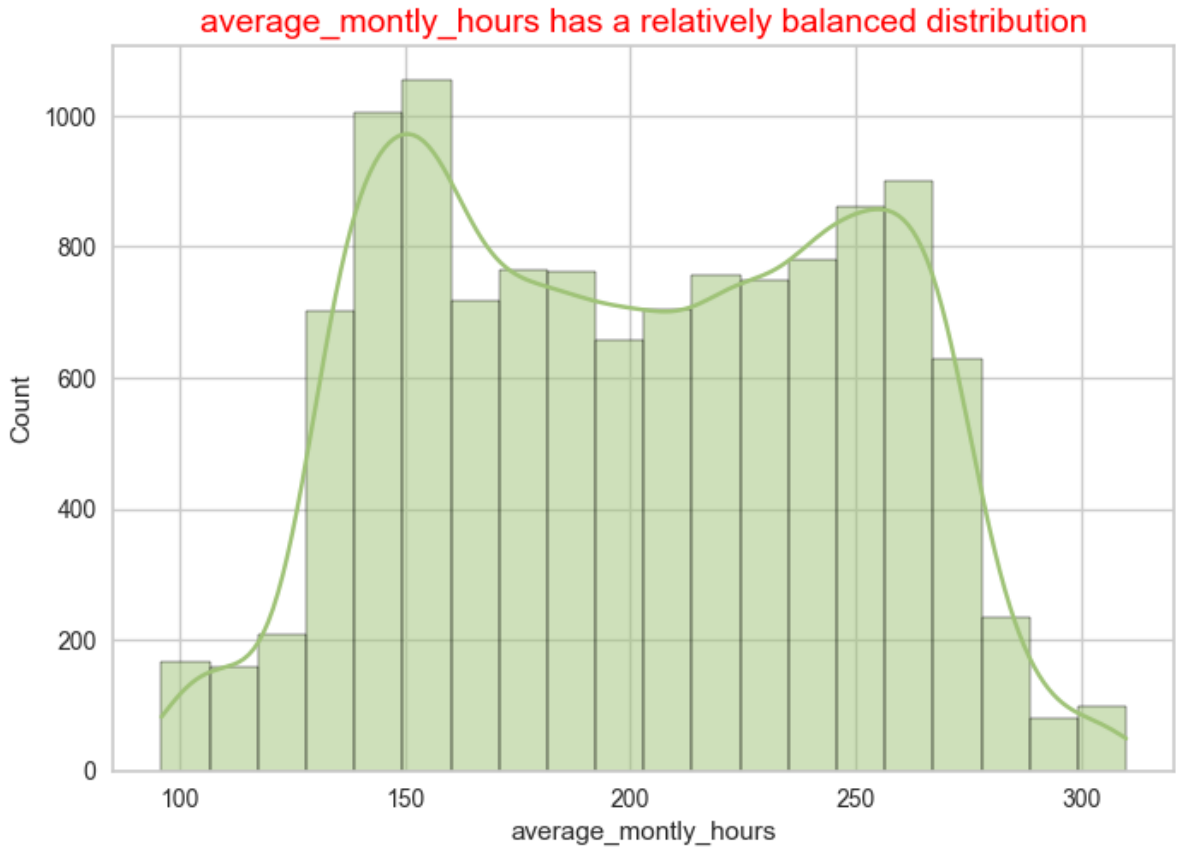
The visualizations above do not show a significant difference in the decisions of employees to leave the company or not. To confirm this observation, a statistical significance test will be conducted below for further assurance.

## average_montly_hours

In [ ]:
```
sns.boxplot(data=df,
            x="average_montly_hours",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

plt.title("average_montly_hours", fontsize=14, color="red");
```




In [ ]:
```
# Displaying the distribution of average_montly_hours feature with a histogr

sns.histplot(data=df, x="average_montly_hours", bins=20, kde=True, color="g"

plt.title("average_montly_hours has a relatively balanced distribution", fon
```



### Check the statistical values

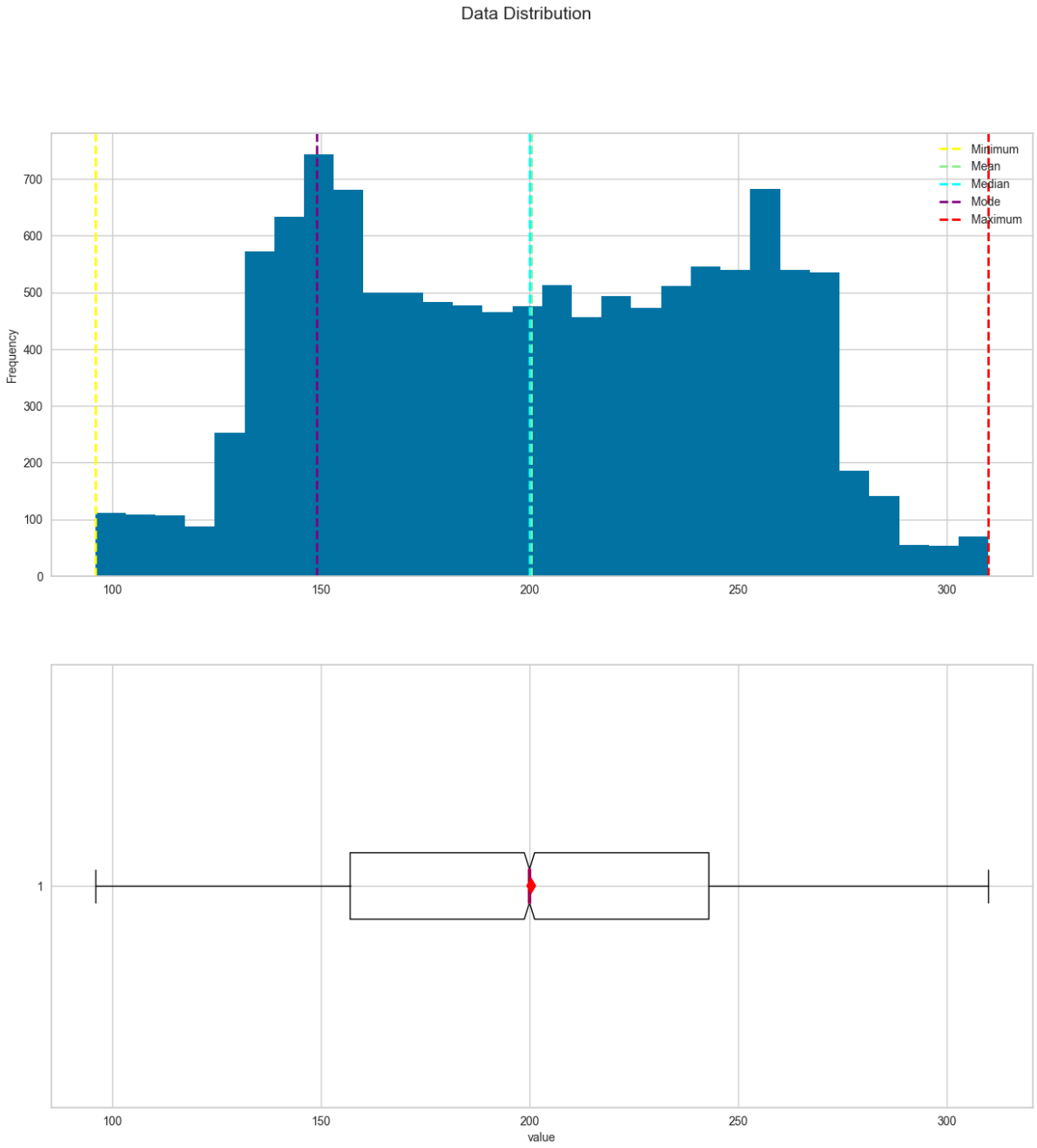
In [ ]:
```
# Descriptive Statistics of "average_montly_hours" Feature

print('Descriptive Statistics of the average_montly_hours Feature:\n')
df.average_montly_hours.describe()
```

```
Descriptive Statistics of the average_montly_hours Feature:
```

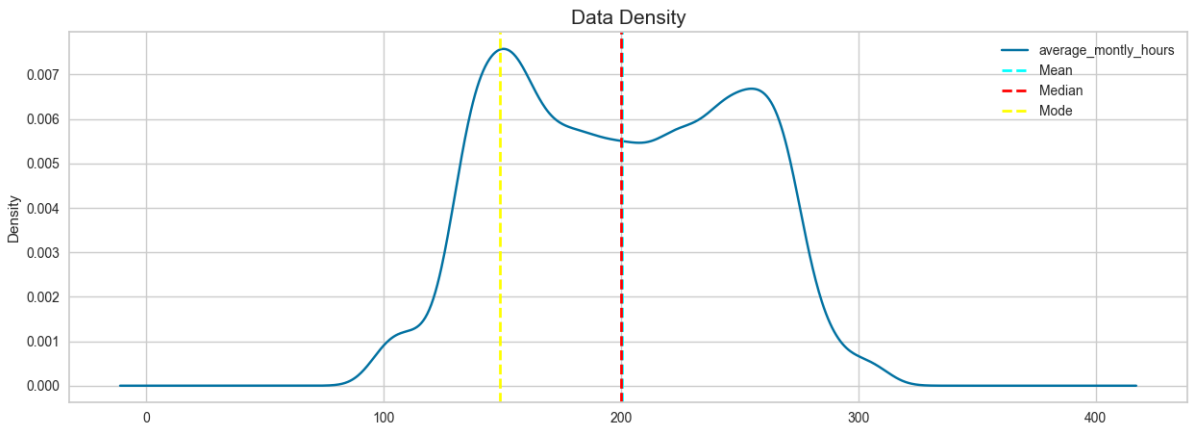
Out[ ]:
```
count    11991.000000
mean       200.473522
std         48.727813
min         96.000000
25%        157.000000
50%        200.000000
75%        243.000000
max        310.000000
Name: average_montly_hours, dtype: float64
```

In [ ]:
```
show_distribution(df["average_montly_hours"])
```

```
Statistical Calculations :
--------------------------
Minimum:  96.00
Mean:    200.47
Median:  200.00
Mode:    149.00
Maximum: 310.00
```



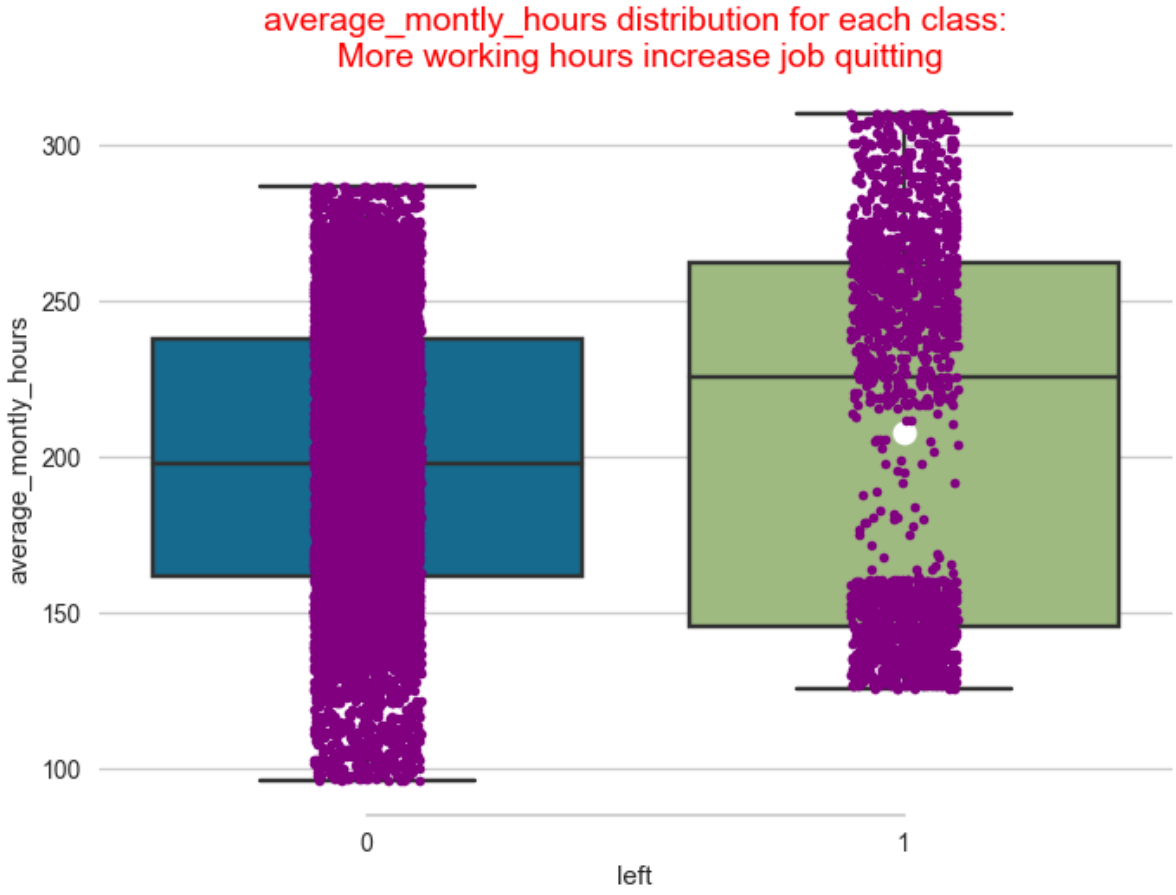

In [ ]:
```
show_density(df["average_montly_hours"])
```




### Check the boxplot and histplot/kdeplot by "left" levels

In [ ]:
```
# Checking the extreme values in "average_montly_hours" feature by left with

sns.boxplot(data=df,
```

```
            y="average_montly_hours",
            x="left",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

# Add in points to show each observation
sns.stripplot(data=df,
              y="average_montly_hours",
              x="left",
              size=4,
              color="purple",
              linewidth=0)

# Tweaking the visual presentation
ax.xaxis.grid(True)
ax.set(ylabel="")
sns.despine(trim=True, left=True)

plt.title("average_montly_hours distribution for each class:\n More working
          fontsize=14, color="red");
```




In [ ]:
```
# Checking Density Distribution of "average_montly_hours" feature by left

sns.histplot(data=df, x="average_montly_hours", bins=20, kde=True, hue="left

plt.title("average_montly_hours distribution for each class:\n Class 1 is mo
```


**Check the statistical values by "left" levels**

In [ ]:
```
# Descriptive Statistics of average_montly_hours with respect to left levels

print(colored('Descriptive Statistics of the average_montly_hours by left:\n
              'blue', attrs=['bold']))
df.groupby("left").average_montly_hours.describe()
```

**Descriptive Statistics of the average_montly_hours by left:**

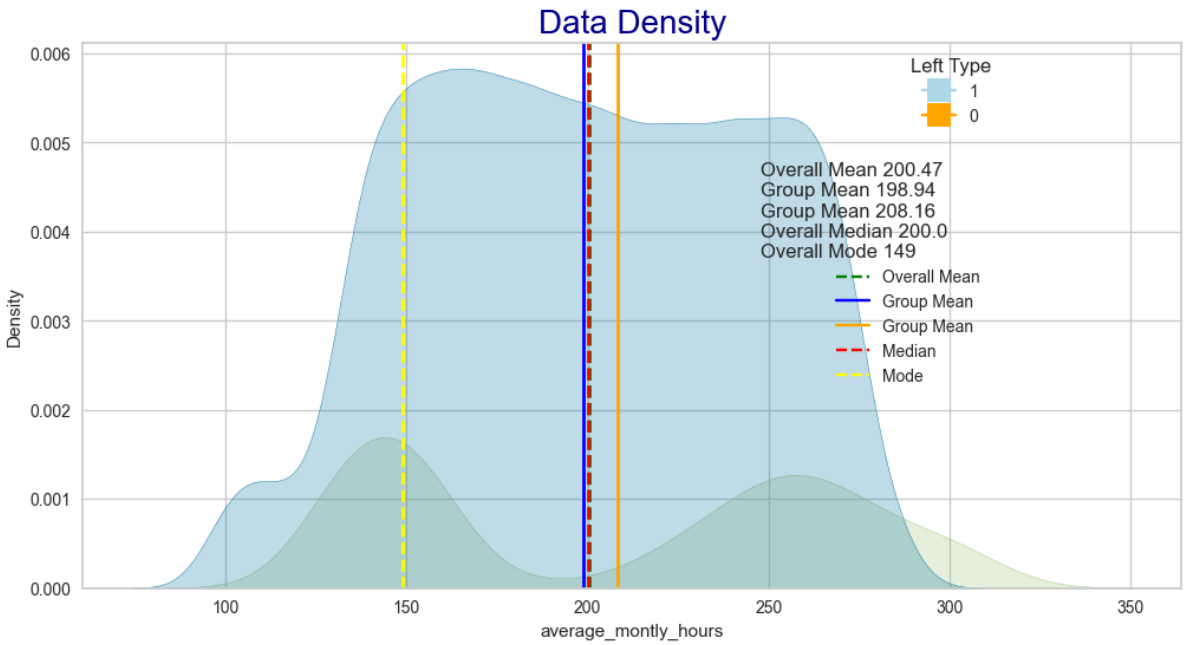
**Mean, min and max are all higher for class 1**

Out[ ]:

| left | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **0** | 10000.0 | 198.94270 | 45.665507 | 96.0 | 162.0 | 198.0 | 238.0 | 287.0 |
| **1** | 1991.0 | 208.16223 | 61.295145 | 126.0 | 146.0 | 226.0 | 262.5 | 310.0 |

In [ ]:
```
show_compare(df, "average_montly_hours", "left")
```

```
Statistical Calculations :
--------------------------
Minimum:  96.00
Mean:    200.47
Median:  200.00
Mode:    149.00
Maximum: 310.00
```

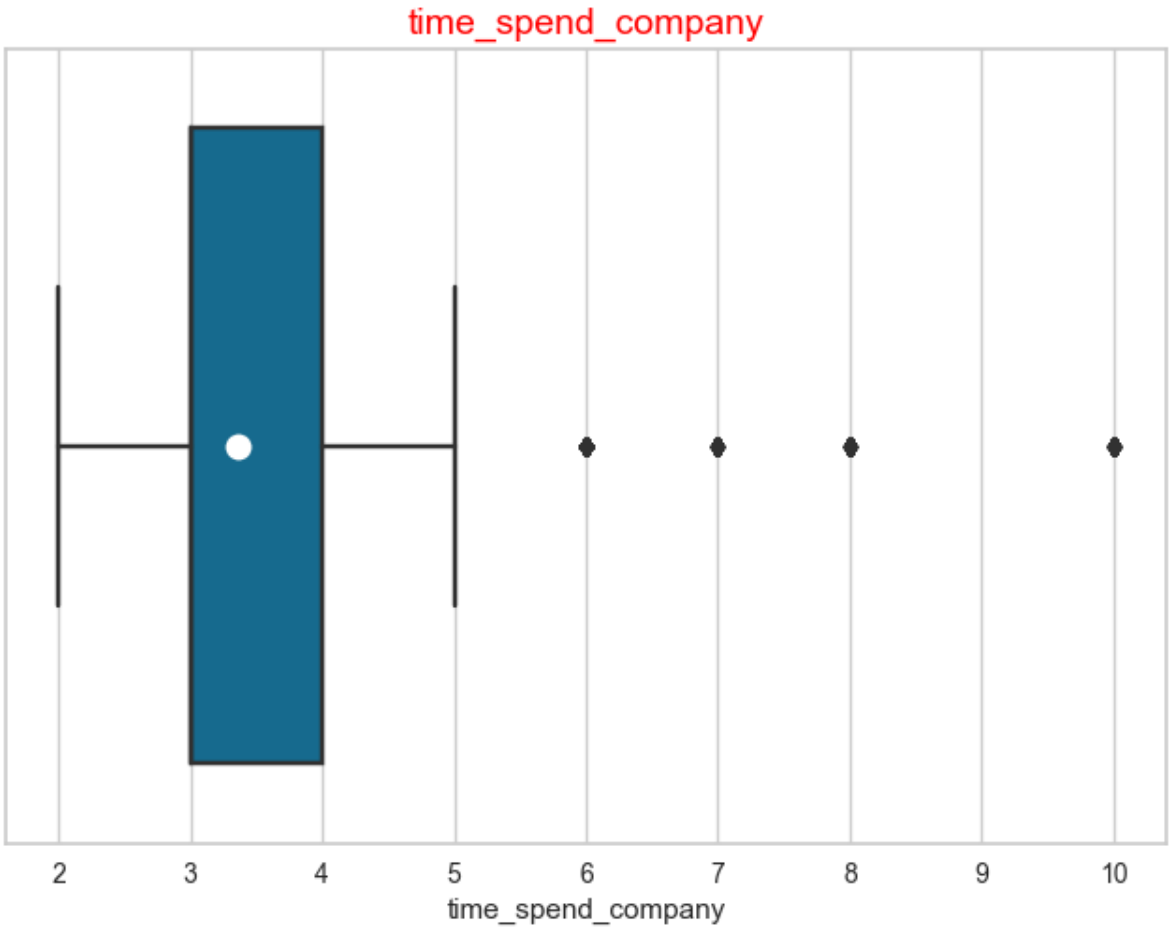



**General assessment of average_working_hours feature**:

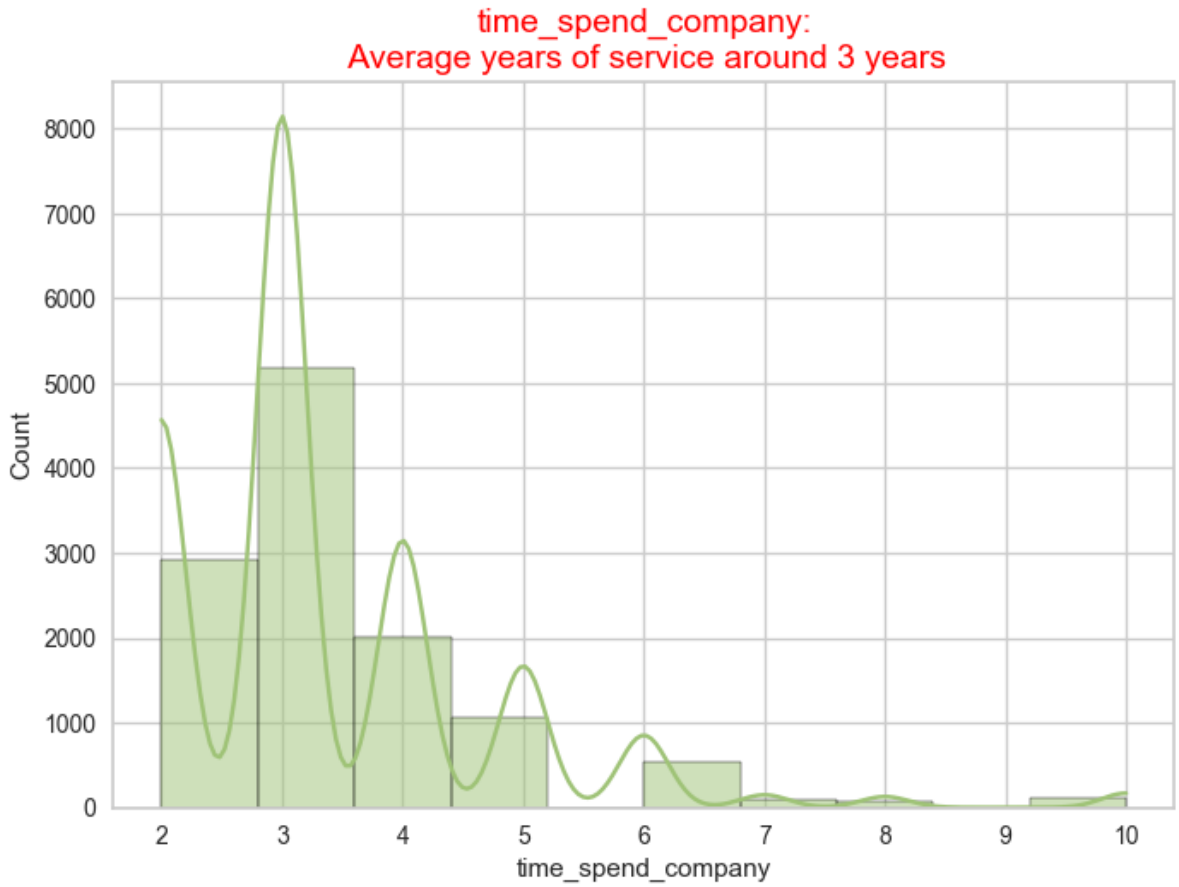
As observed from the visuals and statistical measures presented above, there is a negative effect of increasing working hours on employees' decision to continue working. The negative impact of increasing working hours, as indicated by the higher turnover rate and the tendency towards job quitting (class 1), suggests that employees may be more inclined to leave the company when their average monthly working hours are higher.

## time_spend_company

In [ ]:
```
sns.boxplot(data=df,
            x="time_spend_company",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

plt.title("time_spend_company", fontsize=14, color="red");
```




In [ ]:
```
# Displaying the distribution of time_spend_company feature with a histogram
sns.histplot(data=df, x="time_spend_company", bins=10, kde=True, color="g")
plt.title("time_spend_company:\n Average years of service around 3 years", f
```




### Check the statistical values

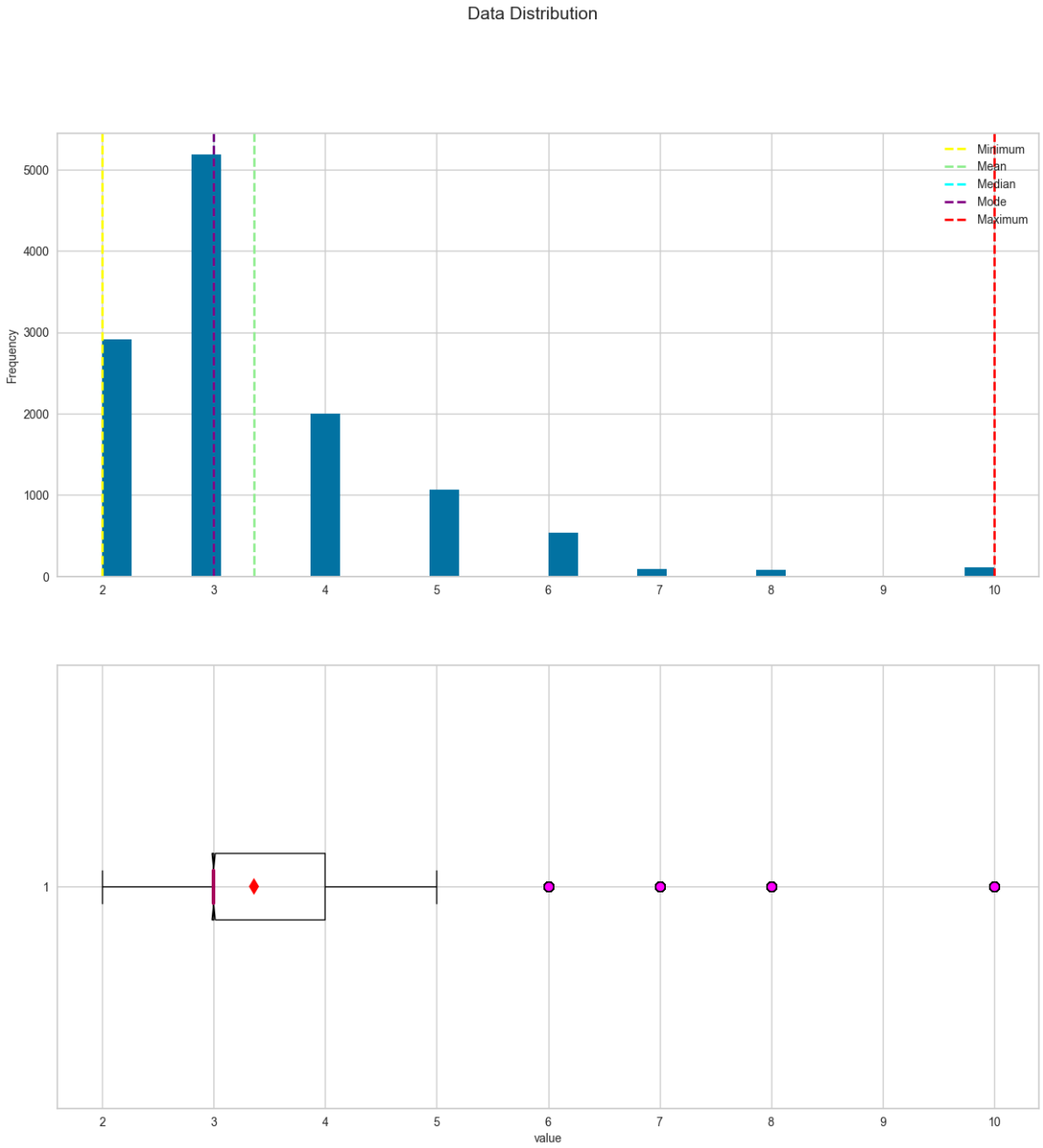
In [ ]:
```
# Descriptive Statistics of "time_spend_company" Feature

print('Descriptive Statistics of the time_spend_company Feature:\n')
df.time_spend_company.describe()
```

```
Descriptive Statistics of the time_spend_company Feature:
```

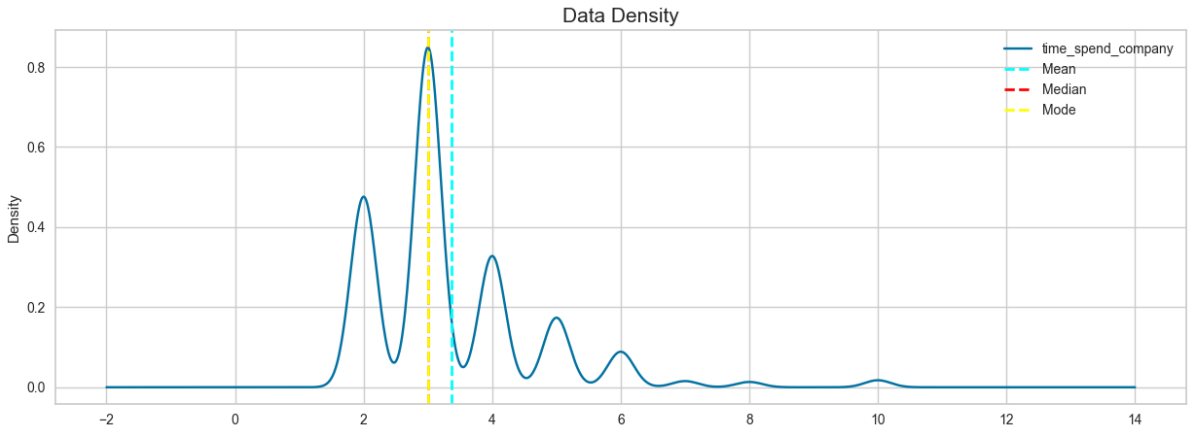
Out[ ]:
```
count    11991.000000
mean         3.364857
std          1.330240
min          2.000000
25%          3.000000
50%          3.000000
75%          4.000000
max         10.000000
Name: time_spend_company, dtype: float64
```

In [ ]:
```
show_distribution(df["time_spend_company"])
```

```
Statistical Calculations :
--------------------------
Minimum:   2.00
Mean:      3.36
Median:    3.00
Mode:      3.00
Maximum:  10.00
```




In [ ]:
```
show_density(df["time_spend_company"])
```

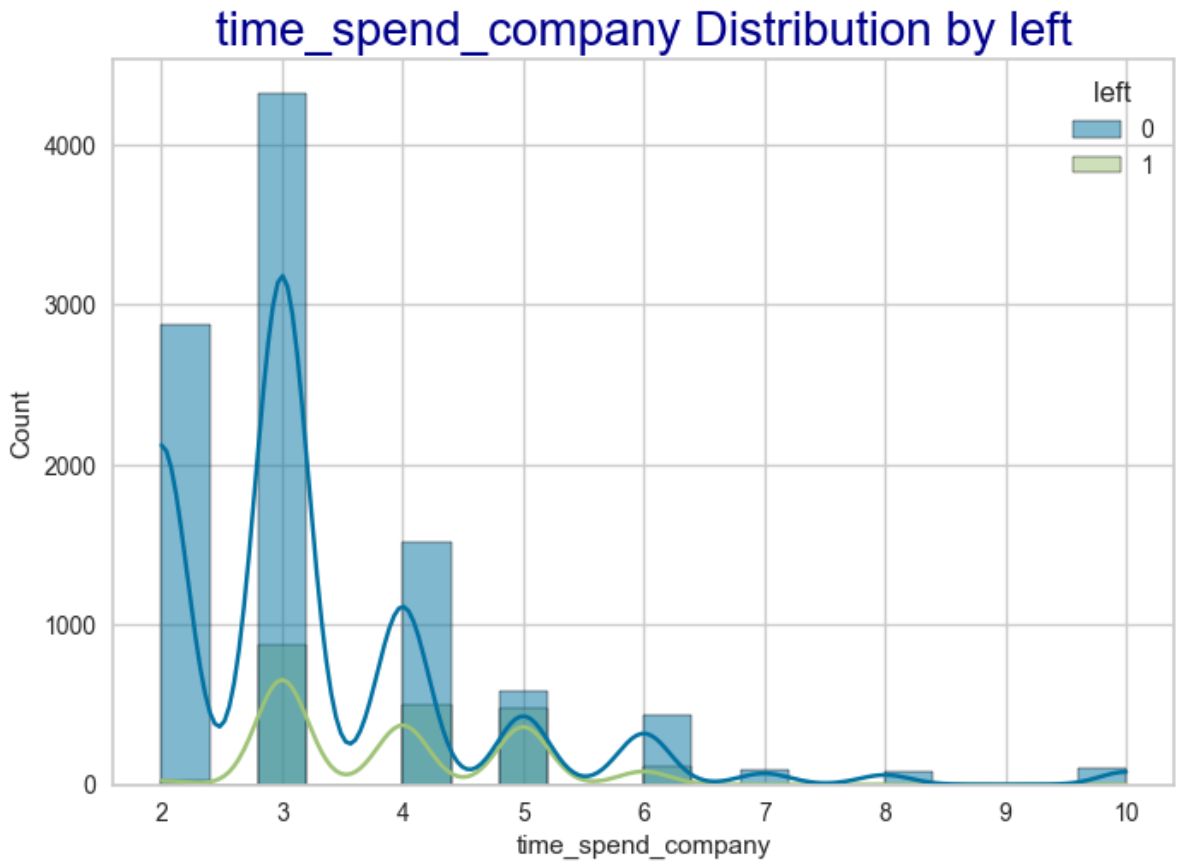



**Check the boxplot and histplot/kdeplot by "left" levels**

In [ ]:
```
# Checking the extreme values in "time_spend_company" feature by left with b

sns.boxplot(data=df,
```

```
            y="time_spend_company",
            x="left",
            showmeans=True,
            meanprops={"marker":"o",
                       "markerfacecolor":"white",
                       "markeredgecolor":"black",
                       "markersize":"10"})

# Add in points to show each observation
sns.stripplot(data=df,
              y="time_spend_company",
              x="left",
              size=4,
              color="purple",
              linewidth=0)

# Tweaking the visual presentation
ax.xaxis.grid(True)
ax.set(ylabel="")
sns.despine(trim=True, left=True)

plt.title("time_spend_company distribution for each class:\n", fontsize=20,
```


In [ ]:
```
# Checking Density Distribution of "time_spend_company" feature by left

sns.histplot(data=df, x="time_spend_company", bins=20, kde=True, hue="left")

plt.title("time_spend_company Distribution by left", fontsize=20, color="dar
```




**Check the statistical values by "left" levels**

In [ ]:
```
# Descriptive Statistics of time_spend_company with respect to left levels

print(colored('Descriptive Statistics of the time_spend_company by left:\n',
df.groupby("left").time_spend_company.describe()
```

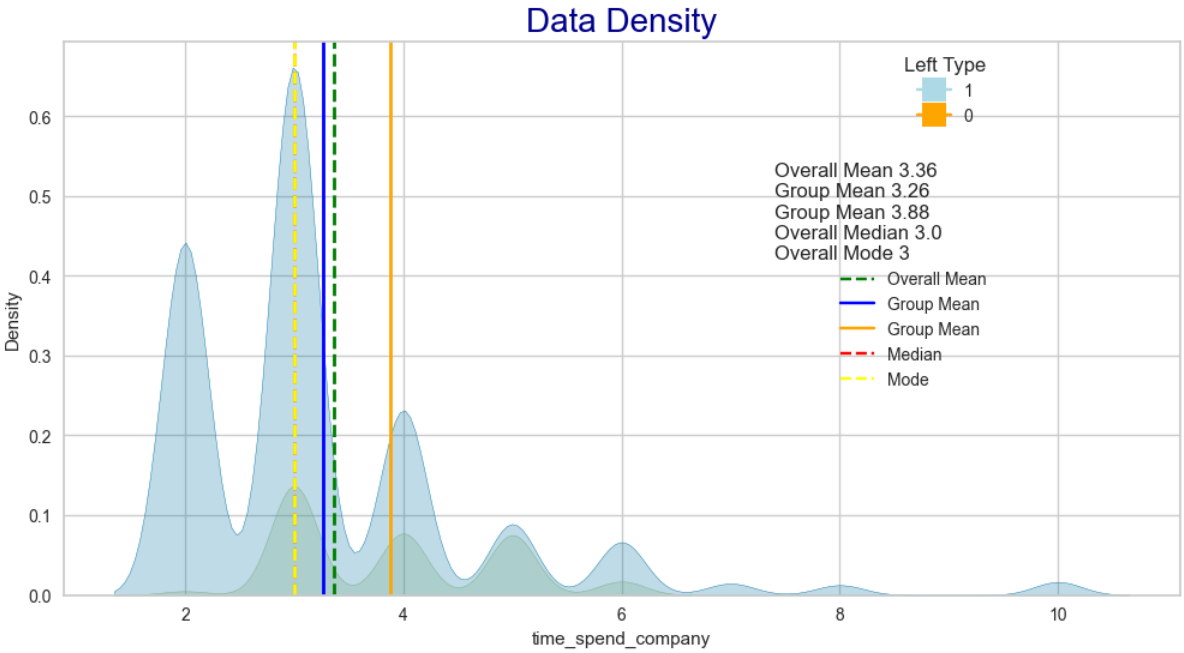
**Descriptive Statistics of the time_spend_company by left:**

Out[ ]:

| left | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **0** | 10000.0 | 3.262000 | 1.367239 | 2.0 | 2.0 | 3.0 | 4.0 | 10.0 |
| **1** | 1991.0 | 3.881467 | 0.974041 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 |

In [ ]:
```
show_compare(df, "time_spend_company", "left")
```

```
Statistical Calculations :
--------------------------
Minimum:   2.00
Mean:      3.36
Median:    3.00
Mode:      3.00
Maximum:  10.00
```




**General assessment for time_spend_company feature**:

The general tendency is that as the average years of service in a company increase, the turnover rate tends to decrease. Employees who have been with a company for a longer duration are generally more likely to have developed a sense of loyalty, job satisfaction, and a stronger commitment to the organization. They may have established relationships, acquired valuable skills, and achieved career progression within the company, which can contribute to their retention.

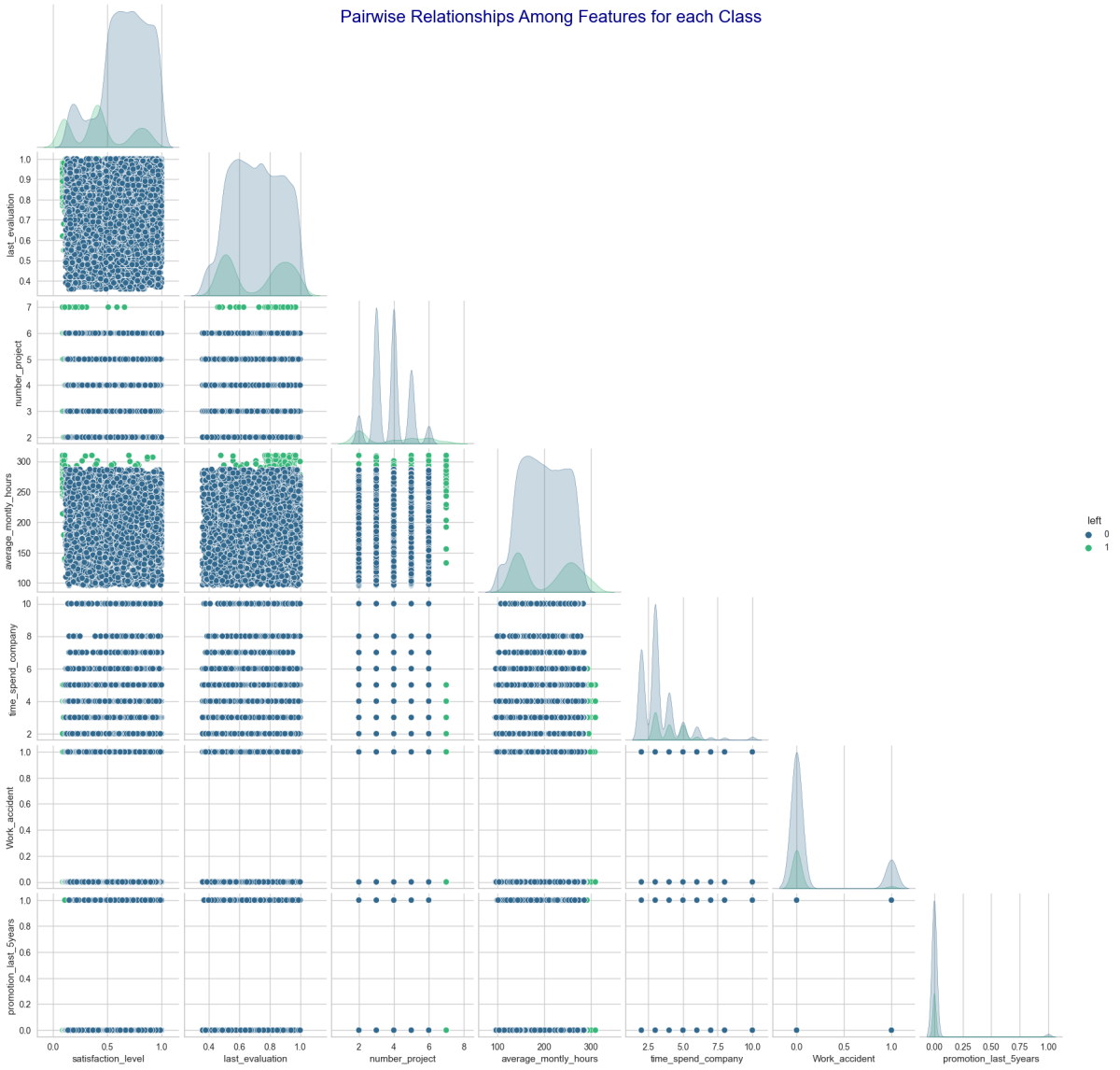
However, it's important to note that individual circumstances and factors specific to each organization can influence the relationship between average years of service and turnover rate. This is what we notice from the visuals and statistical indicators above. Average years of service is longer for the employees who left the company. In order to have more meaningful insights, the main conditions of the company and the sector in which this company operates should be examined more. In such a case there are many questions to be asked, such as:

- Is the competition for the existing labour force in the market high?
- Do the other companies follow a strategy to attrat the employees of rival companies?
- Are the working conditions challenging for the employees emotionally and physically?

## See the relationship between each numeric features by target feature (left) in one plot basically

A pairplot plot a pairwise relationships in a dataset.

In [ ]:
```
g = sns.pairplot(df, hue="left", palette="viridis", corner=True)

g.fig.suptitle("Pairwise Relationships Among Features for each Class", fonts
```




## Categorical Features

In [ ]:
```
df.sample(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend |
|---|---|---|---|---|---|
| **8431** | 0.55 | 0.49 | 5 | 240 | |

### departments

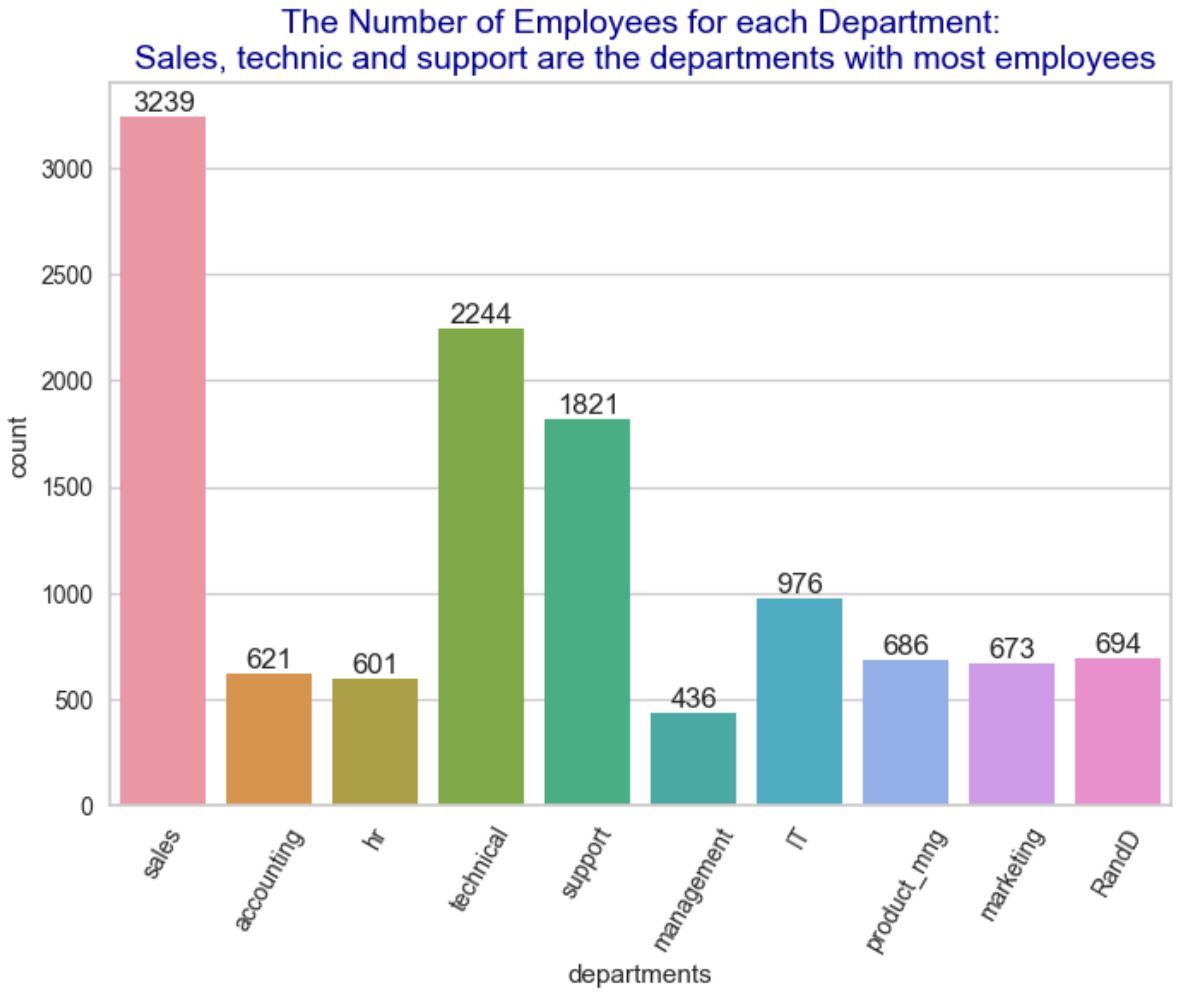
In [ ]:
```
df.rename(columns={'Departments ':'departments'}, inplace = True)
```

In [ ]:
```
# Checking the uniques of "departments" feature and determining their number

df.departments.value_counts(dropna=False)
```

Out[ ]:
```
departments
sales          3239
technical      2244
support        1821
IT              976
RandD           694
product_mng     686
marketing       673
accounting      621
hr              601
management      436
Name: count, dtype: int64
```

**Visualize the count of person in departments**

In [ ]:
```
# Visualization of "departments" feature

ax = sns.countplot(data=df, x="departments")

plt.xticks(rotation=60)

plt.title("The Number of Employees for each Department:\n Sales, technic and
are the departments with most employees", fontsize=14, color="darkblue")

for container in ax.containers:
    ax.bar_label(container);
```

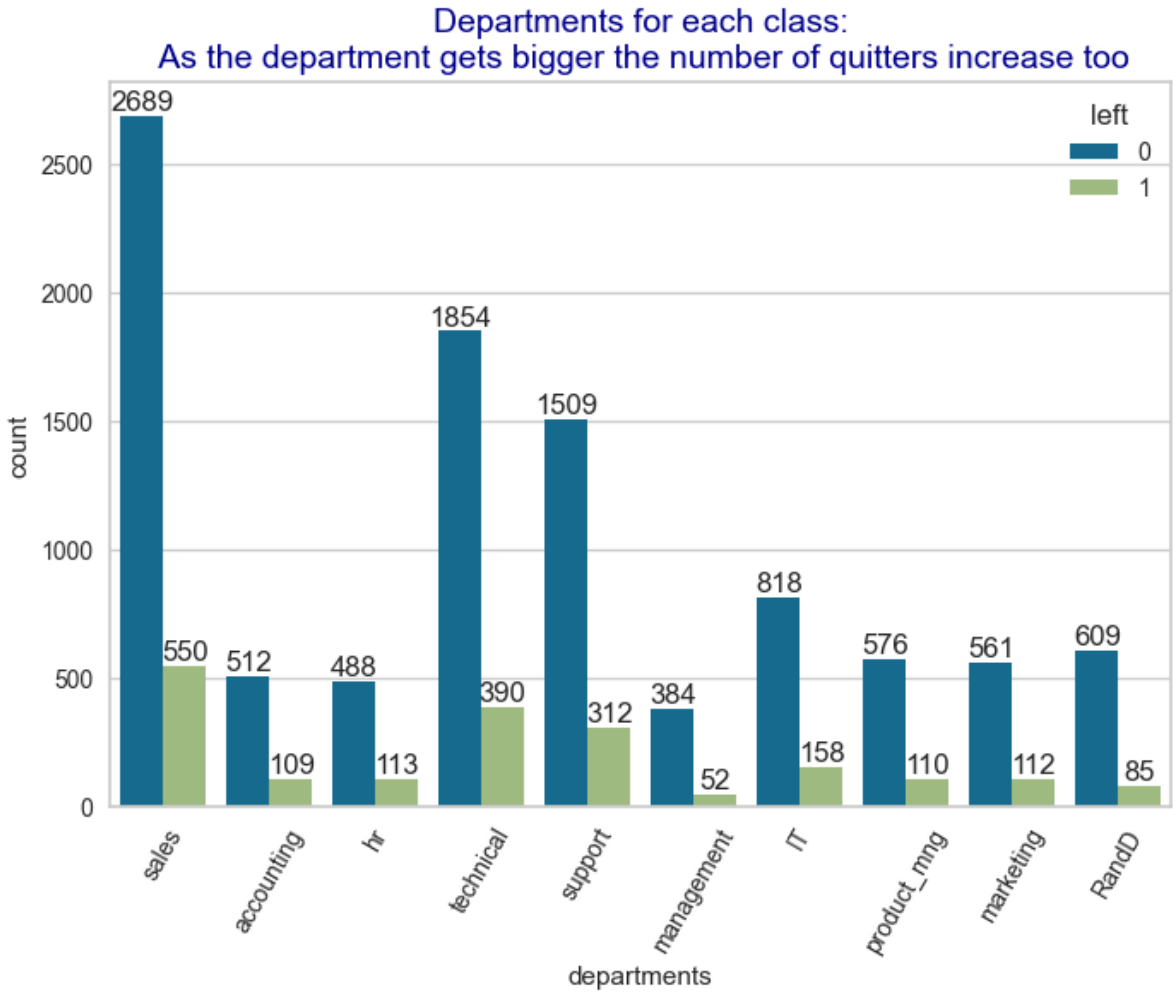



**Check the count of person in each "left" levels by departments and visualize them with countplot**

In [ ]:
```
# Checking "departments" feature by left in detail

df.groupby("departments").left.value_counts()
```

Out[ ]:
```
departments  left
IT           0        818
             1        158
RandD        0        609
             1         85
accounting   0        512
             1        109
hr           0        488
             1        113
management   0        384
             1         52
marketing    0        561
             1        112
product_mng  0        576
             1        110
sales        0       2689
             1        550
support      0       1509
             1        312
technical    0       1854
             1        390
Name: count, dtype: int64
```

In [ ]:
```
# Visualizing the number of people in each "departments" level by left

ax = sns.countplot(data=df, x="departments", hue="left")

plt.title("Departments for each class:\n As the department gets bigger the n
quitters increase too", fontsize=14, color="darkblue")

plt.xticks(rotation = 60)

for container in ax.containers:
    ax.bar_label(container);
```



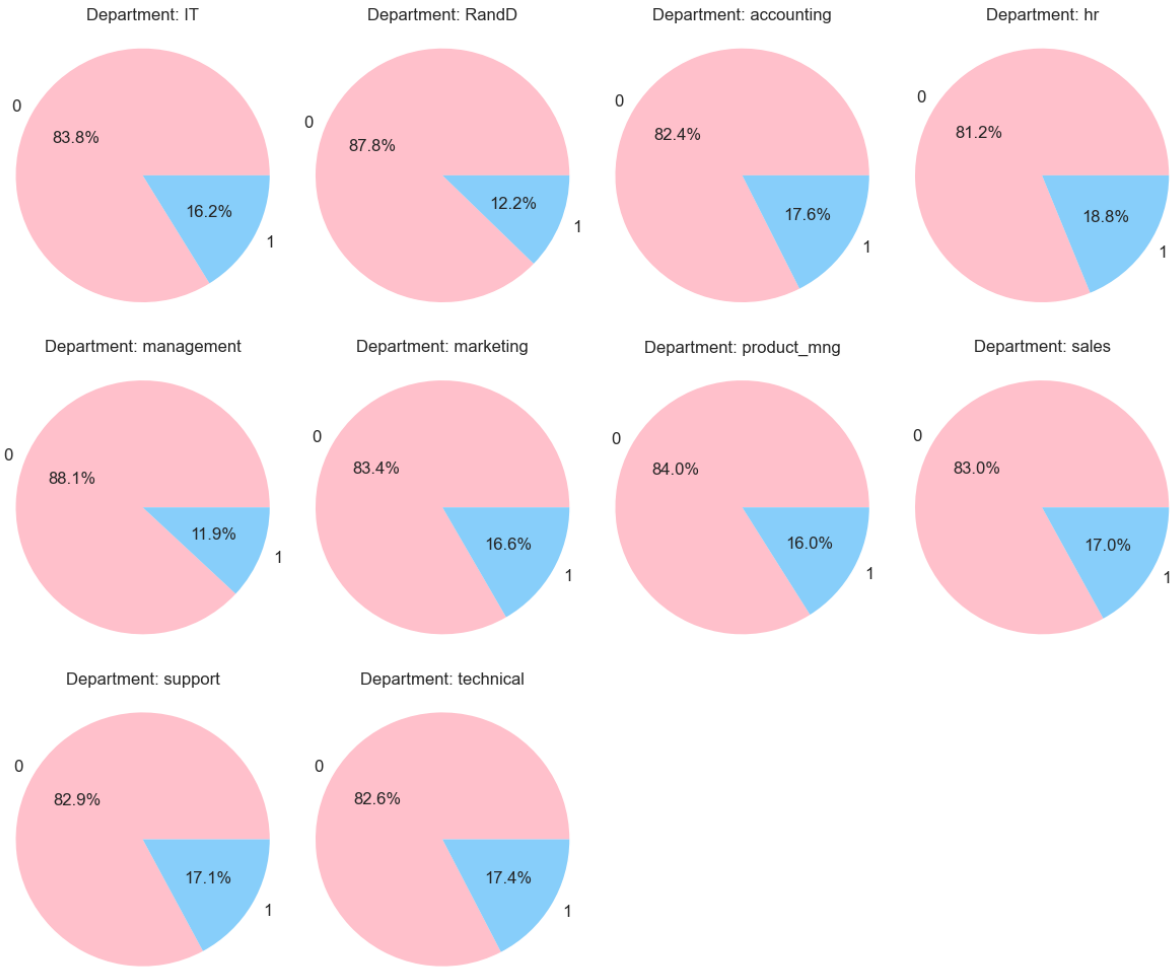

**Check the percentage distribution of person in each "left" levels by departments and visualize it with pie plot separately**

In [ ]:
```
# The Proportional Distribution of persons in each sub-categories of "depart

edu = df.groupby(["departments"]).left.value_counts(normalize=True)
edu
```

Out[ ]:
```
departments  left
IT           0       0.838115
             1       0.161885
RandD        0       0.877522
             1       0.122478
accounting   0       0.824477
             1       0.175523
hr           0       0.811980
             1       0.188020
management   0       0.880734
             1       0.119266
marketing    0       0.833581
             1       0.166419
product_mng  0       0.839650
             1       0.160350
sales        0       0.830195
             1       0.169805
support      0       0.828666
             1       0.171334
technical    0       0.826203
             1       0.173797
Name: proportion, dtype: float64
```

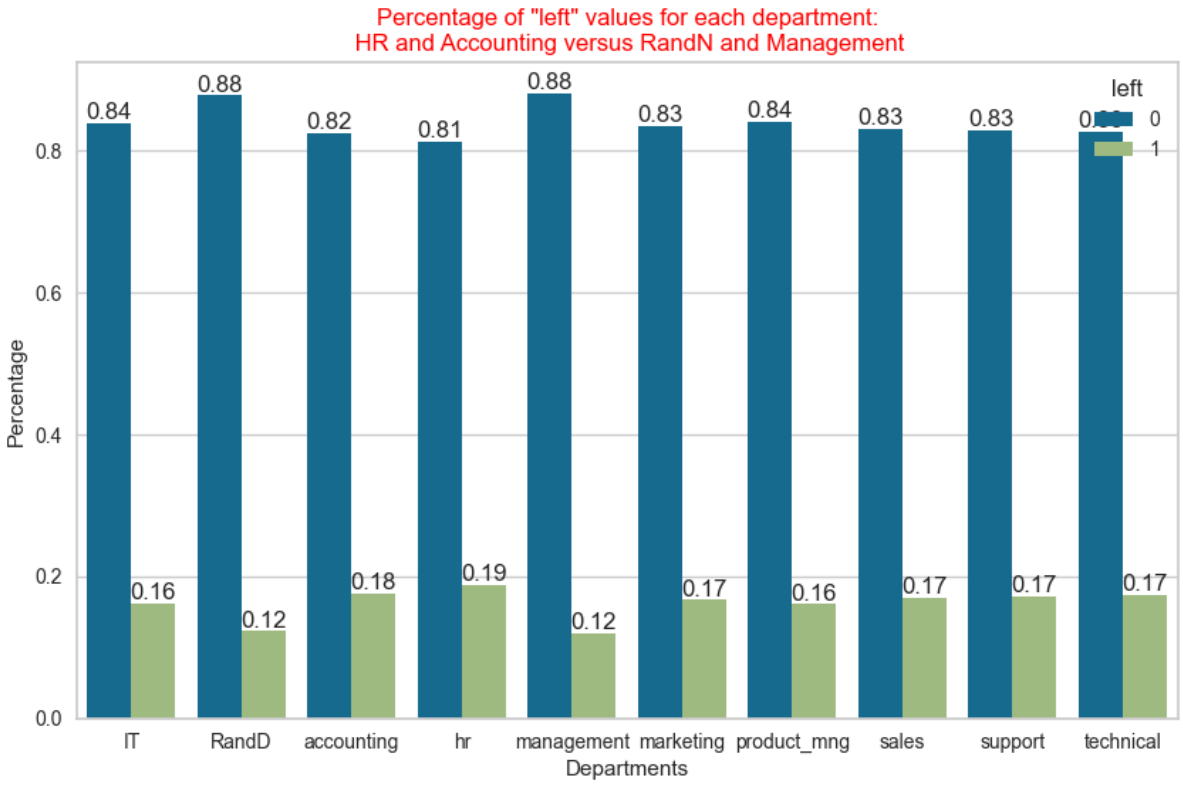
In [ ]:

```
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["departments"]).left.value_counts(normalize=True)

# Calculate the number of departments
num_departments = len(grouped_data.index.levels[0])

# Calculate the number of rows and columns in the grid
num_columns = 4
num_rows = (num_departments + num_columns - 1) // num_columns

# Create a figure with subplots
fig, axes = plt.subplots(num_rows, num_columns, figsize=(12, 10))

# Flatten the axes array to simplify iteration
axes = axes.flatten()

# Iterate over each department
for i, department in enumerate(grouped_data.index.levels[0]):
    # Filter data for the current department
    department_data = grouped_data[department]

    # Create a pie plot in the corresponding subplot
    axes[i].pie(department_data, labels=department_data.index, autopct='%1.1
    axes[i].set_title(f"Department: {department}")
    axes[i].axis('equal')  # Equal aspect ratio ensures a circular pie

# Hide any remaining empty subplots
for j in range(num_departments, num_rows * num_columns):
    axes[j].axis('off')

# Adjust the spacing between subplots
plt.tight_layout()

# Show the pie plots
plt.show()
```




In [ ]:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["departments"]).left.value_counts(normalize=True)

# Reset the index to convert the grouped series to a DataFrame
grouped_data = grouped_data.reset_index(name='percentage')

# Create a bar plot using seaborn
plt.figure(figsize=(10, 6))
ax = sns.barplot(data=grouped_data, x='departments', y='percentage', hue='le
plt.xlabel('Departments')
plt.ylabel('Percentage')
plt.title('Percentage of "left" values for each department:\n HR and Account
versus RandN and Management', fontdict={"color":"red"})
plt.bar_label(ax.containers[0], fmt='%.2f')
plt.bar_label(ax.containers[1], fmt='%.2f')

# Show the bar plot
plt.show();
```




**Check the the percentage distribution of person in each department by "left" levels and visualize it with bar plot**

In [ ]:
```
# Checking the percentage of persons at "department" by "left" levels

department = df.groupby("left").departments.value_counts(normalize=True)*100
department
```

Out[ ]:
```
left  departments
0     sales           26.890000
      technical       18.540000
      support         15.090000
      IT               8.180000
      RandD            6.090000
      product_mng      5.760000
      marketing        5.610000
      accounting       5.120000
      hr               4.880000
      management       3.840000
1     sales           27.624309
      technical       19.588147
      support         15.670517
      IT               7.935711
      hr               5.675540
      marketing        5.625314
      product_mng      5.524862
      accounting       5.474636
      RandD            4.269211
      management       2.611753
Name: proportion, dtype: float64
```

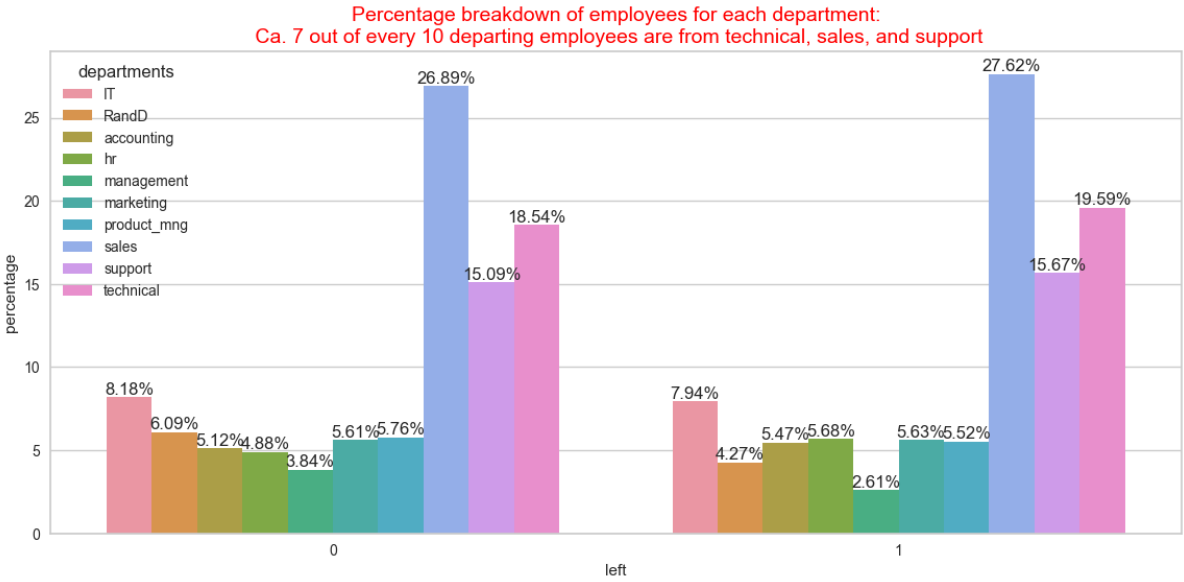
In [ ]:
```
# Creating a dataframe demonstrating the percentage of persons at "departmen

department_df = pd.DataFrame(department)
department_df.rename(columns={"proportion": "percentage"}, inplace=True)
department_df.reset_index(inplace=True)
department_df.sort_values(by=["left", "departments"], inplace=True)
department_df
```

Out[ ]:

| | left | departments | percentage |
|---|---|---|---|
| **3** | 0 | IT | 8.180000 |
| **4** | 0 | RandD | 6.090000 |
| **7** | 0 | accounting | 5.120000 |
| **8** | 0 | hr | 4.880000 |
| **9** | 0 | management | 3.840000 |
| **6** | 0 | marketing | 5.610000 |
| **5** | 0 | product_mng | 5.760000 |
| **0** | 0 | sales | 26.890000 |
| **2** | 0 | support | 15.090000 |
| **1** | 0 | technical | 18.540000 |
| **13** | 1 | IT | 7.935711 |
| **18** | 1 | RandD | 4.269211 |
| **17** | 1 | accounting | 5.474636 |
| **14** | 1 | hr | 5.675540 |
| **19** | 1 | management | 2.611753 |
| **15** | 1 | marketing | 5.625314 |
| **16** | 1 | product_mng | 5.524862 |
| **10** | 1 | sales | 27.624309 |
| **12** | 1 | support | 15.670517 |
| **11** | 1 | technical | 19.588147 |

In [ ]:

```
# Visualizing the percentage of persons at "department" by "left" levels

fig, ax = plt.subplots(figsize=(14, 6))

ax = sns.barplot(data=department_df, x="left", y="percentage", hue="departme

plt.title("Percentage breakdown of employees for each department:\n Ca. 7 ou
employees are from technical, sales, and support", fontsize=14,
          fontdict={"color":"red"});

for container in ax.containers:
    ax.bar_label(container, fmt="%.2f%%", fontsize=12);
```




**A genaral assessment for the Departments feature**:

As seen, tecnhical, sales and support departments have the highest numbers of churned employees. This means that the company should adopt some specific strategies for these departments to decrease the employee turnover rate. Accordingly, the company can consider the following department-specific strategies:

**Sales Department:**

- Provide comprehensive sales training programs to enhance skills and confidence.
- Offer competitive commission structures or incentive programs to motivate and retain top-performing sales professionals.
- Implement regular performance evaluations and feedback sessions to identify areas of improvement and provide guidance for career growth.

**Technical Department**:

- Foster a culture of innovation and continuous learning to keep technical employees engaged and challenged.
- Provide opportunities for professional development, such as attending conferences or workshops, to enhance technical skills.
- Encourage cross-functional collaboration and provide opportunities for technical employees to work on diverse projects to keep them motivated.

**Support Department**:

- Ensure adequate staffing levels to manage workload effectively and prevent burnout.
- Implement regular training programs to enhance customer service and problem-solving skills.
- Foster a supportive and collaborative environment within the department to encourage knowledge sharing and teamwork.

**Accounting Department**:

- Provide opportunities for career advancement and professional certifications in accounting and finance.
- Implement flexible work arrangements to promote work-life balance, as accounting roles often involve strict deadlines and long hours.
- Recognize and appreciate the contributions of accounting professionals through regular feedback and rewards.

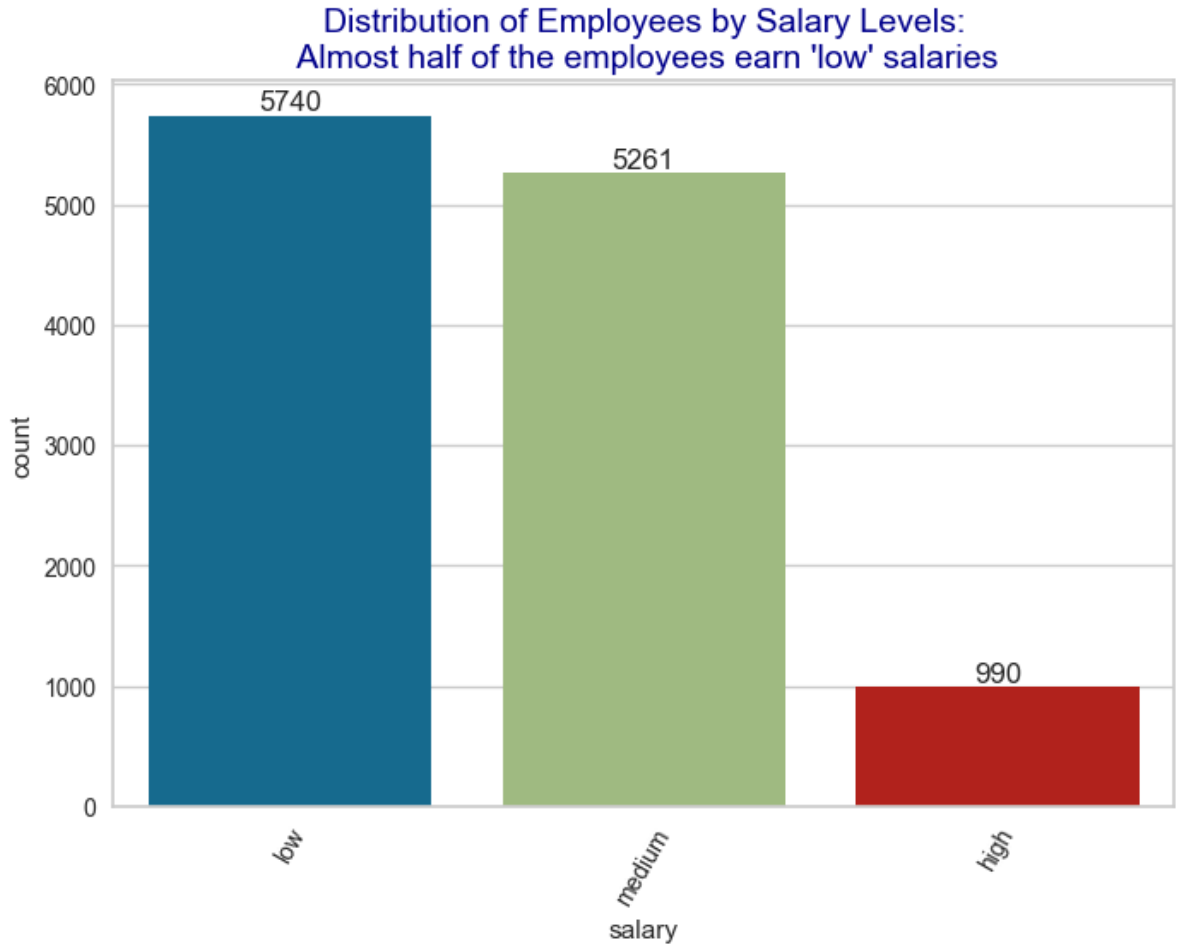
By tailoring strategies to address the specific needs and challenges of each department, the company can create a more targeted approach to decreasing turnover rates. It is essential to involve department managers and employees in the process to gain their input and support for the initiatives implemented.

### salary

In [ ]:
```
# Checking the uniques of "salary" feature and determining their numbers

df.salary.value_counts(dropna=False)
```

Out[ ]:
```
salary
low       5740
medium    5261
high       990
Name: count, dtype: int64
```

In [ ]:
```
# Visualization of "salary" feature

ax = sns.countplot(data=df, x="salary")

plt.xticks(rotation=60)

plt.title("Distribution of Employees by Salary Levels:\n Almost half of the
          fontsize=14, color="darkblue")

for container in ax.containers:
    ax.bar_label(container);
```

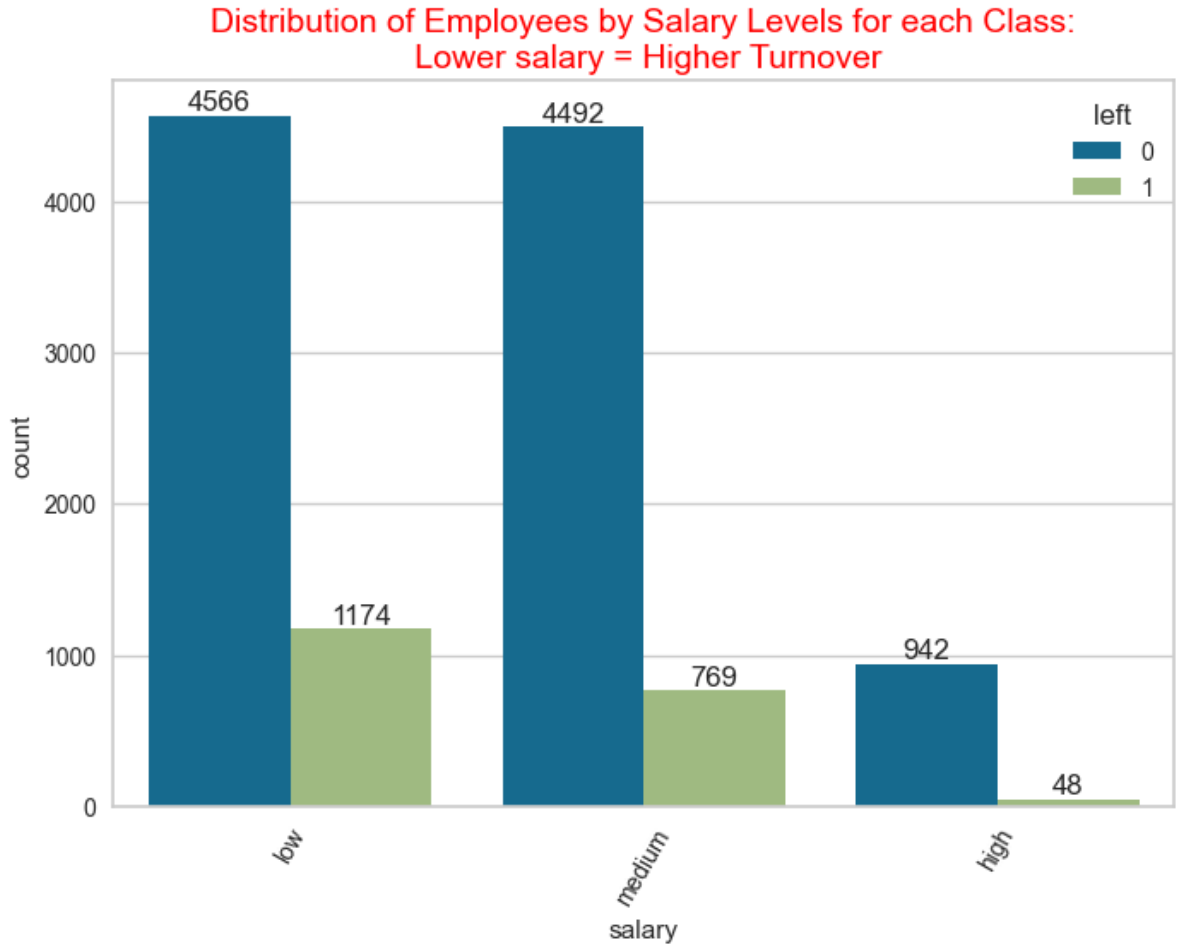



**Check the count of person in each "left" levels by salary and visualize them with countplot**

In [ ]:
```
# Checking "salary" feature by left in detail

df.groupby("salary").left.value_counts()
```

Out[ ]:
```
salary  left
high    0         942
        1          48
low     0        4566
        1        1174
medium  0        4492
        1         769
Name: count, dtype: int64
```

In [ ]:
```
# Visualizing the number of people in each "salary" level by left

ax = sns.countplot(data=df, x="salary", hue="left")

plt.title("Distribution of Employees by Salary Levels for each Class:\n Lowe
          fontsize=14, color="red")

plt.xticks(rotation = 60)

for container in ax.containers:
    ax.bar_label(container);
```



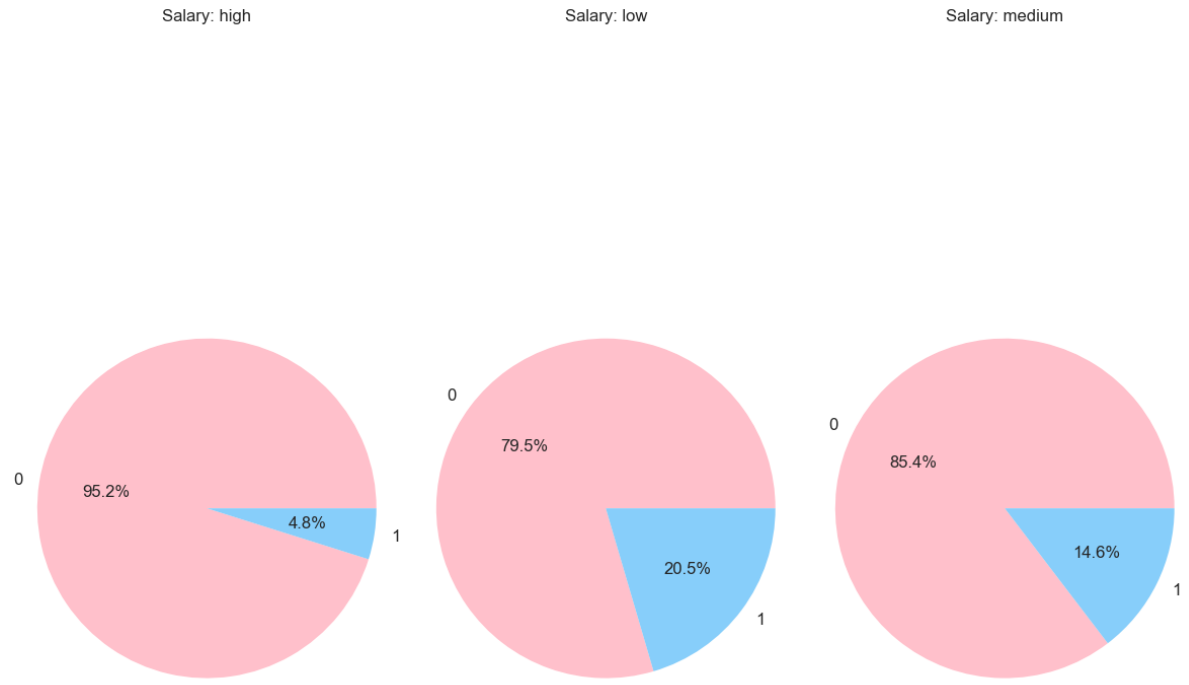

**Check the percentage distribution of person in each "left" levels by salary and visualize it with pie plot separately**

In [ ]:
```
# The Proportional Distribution of persons in each sub-categories of "salary

edu = df.groupby(["salary"]).left.value_counts(normalize=True)
edu
```

Out[ ]:
```
salary  left
high    0       0.951515
        1       0.048485
low     0       0.795470
        1       0.204530
medium  0       0.853830
        1       0.146170
Name: proportion, dtype: float64
```

In [ ]:
```
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["salary"]).left.value_counts(normalize=True)

# Calculate the number of salary
num_salary = len(grouped_data.index.levels[0])

# Calculate the number of rows and columns in the grid
num_columns = 3
num_rows = (num_salary + num_columns - 1) // num_columns

# Create a figure with subplots
fig, axes = plt.subplots(num_rows, num_columns, figsize=(12, 10))

# Flatten the axes array to simplify iteration
```

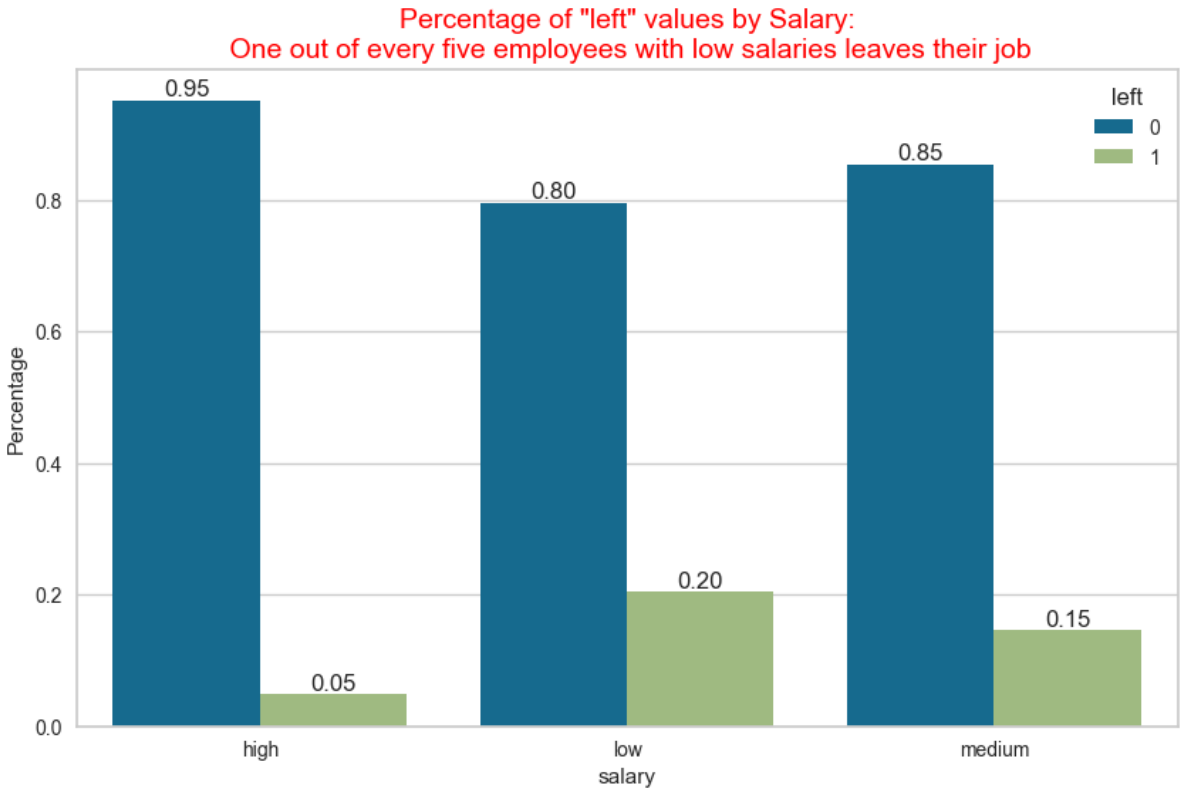
```
axes = axes.flatten()

# Iterate over each department
for i, department in enumerate(grouped_data.index.levels[0]):
    # Filter data for the current department
    department_data = grouped_data[department]

    # Create a pie plot in the corresponding subplot
    axes[i].pie(department_data, labels=department_data.index, autopct='%1.1
    axes[i].set_title(f"Salary: {department}")
    axes[i].axis('equal')  # Equal aspect ratio ensures a circular pie

# Hide any remaining empty subplots
for j in range(num_salary, num_rows * num_columns):
    axes[j].axis('off')

# Adjust the spacing between subplots
plt.tight_layout()

# Show the pie plots
plt.show()
```




In [ ]:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["salary"]).left.value_counts(normalize=True)

# Reset the index to convert the grouped series to a DataFrame
grouped_data = grouped_data.reset_index(name='percentage')

# Create a bar plot using seaborn
plt.figure(figsize=(10, 6))
```

```
ax = sns.barplot(data=grouped_data, x='salary', y='percentage', hue='left')
plt.xlabel('salary')
plt.ylabel('Percentage')
plt.title('Percentage of "left" values by Salary:\n One out of every five em
their job', fontsize=14, color="red")
plt.bar_label(ax.containers[0], fmt='%.2f')
plt.bar_label(ax.containers[1], fmt='%.2f')

# Show the bar plot
plt.show();
```




**Check the the percentage distribution of person in each salary by "left" levels and visualize it with bar plot**

In [ ]:
```
# Checking the percentage of persons at "salary" by "left" levels

salary = df.groupby("left").salary.value_counts(normalize=True)*100
salary
```

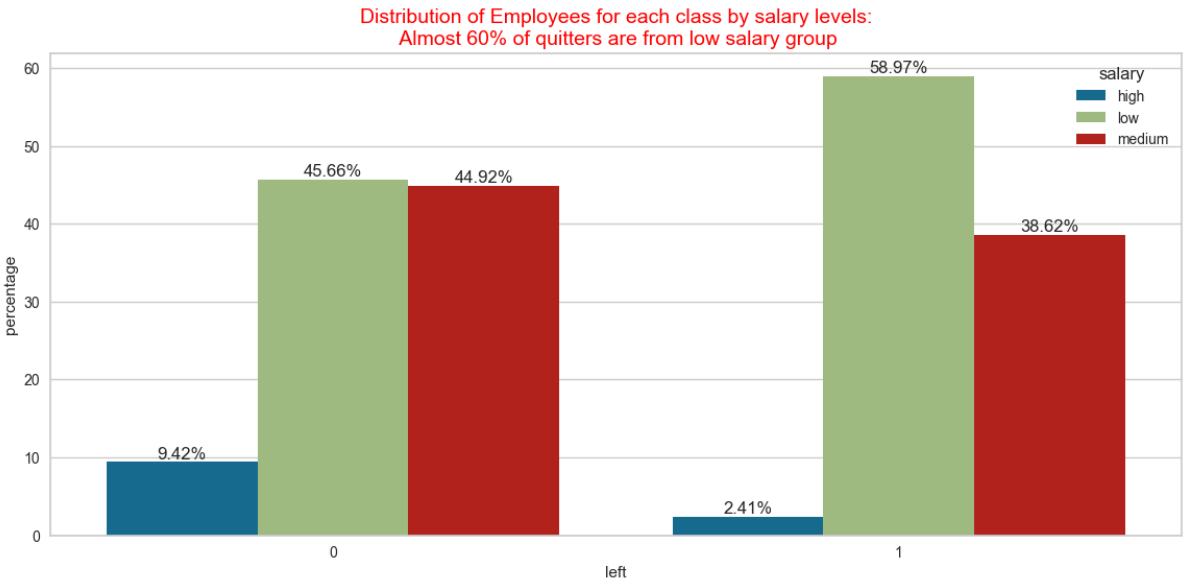
Out[ ]:
```
left  salary
0     low       45.660000
      medium    44.920000
      high       9.420000
1     low       58.965344
      medium    38.623807
      high       2.410849
Name: proportion, dtype: float64
```

In [ ]:
```
# Creating a dataframe demonstrating the percentage of persons at "salary" b

salary_df = pd.DataFrame(salary)
salary_df.rename(columns={"proportion": "percentage"}, inplace=True)
salary_df.reset_index(inplace=True)
salary_df.sort_values(by=["left", "salary"], inplace=True)
salary_df
```

Out[ ]:

| | left | salary | percentage |
|---|---|---|---|
| **2** | 0 | high | 9.420000 |
| **0** | 0 | low | 45.660000 |
| **1** | 0 | medium | 44.920000 |
| **5** | 1 | high | 2.410849 |
| **3** | 1 | low | 58.965344 |
| **4** | 1 | medium | 38.623807 |

In [ ]:
```
# Visualizing the percentage of persons at "salary" by "left" levels

fig, ax = plt.subplots(figsize=(14, 6))

ax = sns.barplot(data=salary_df, x="left", y="percentage", hue="salary")

plt.title("Distribution of Employees for each class by salary levels:\n Almo
are from low salary group", fontsize=14, color="red");

for container in ax.containers:
    ax.bar_label(container, fmt="%.2f%%", fontsize=12);
```




**General Assessment for Salary**:

Salary is an important factor in an employee's decision to leave a job. Compensation plays a significant role in job satisfaction and overall employee motivation. If an employee feels that their salary is not competitive or does not align with their skills, experience, or the market value for their role, it can lead to dissatisfaction and a higher likelihood of considering other job opportunities. Adequate and fair compensation is crucial in attracting and retaining talented employees and reducing turnover rates. As seen from the visuals and statistical measures above, most of the quitters are from low salary group. As the salary level increases, turnover rates decreases for this company.

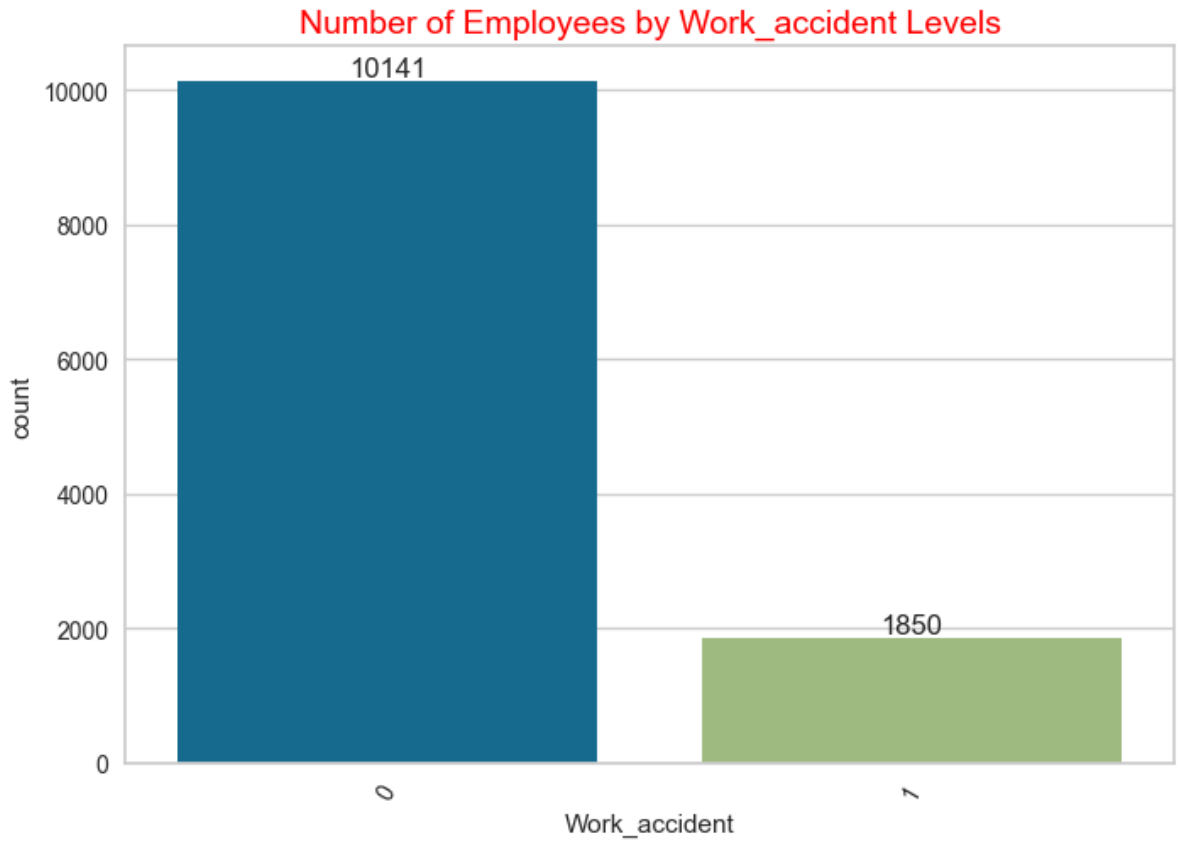
## Work_accident

In [ ]:
```
# Checking the uniques of "Work_accident" feature and determining their numb

df.Work_accident.value_counts(dropna=False)
```

Out[ ]:
```
Work_accident
0    10141
1     1850
Name: count, dtype: int64
```

**Visualize the count of person in Work_accident**

In [ ]:
```
# Visualization of "Work_accident" feature

ax = sns.countplot(data=df, x="Work_accident")

plt.xticks(rotation=60)

plt.title("Number of Employees by Work_accident Levels", fontsize=14, color=

for container in ax.containers:
    ax.bar_label(container);
```



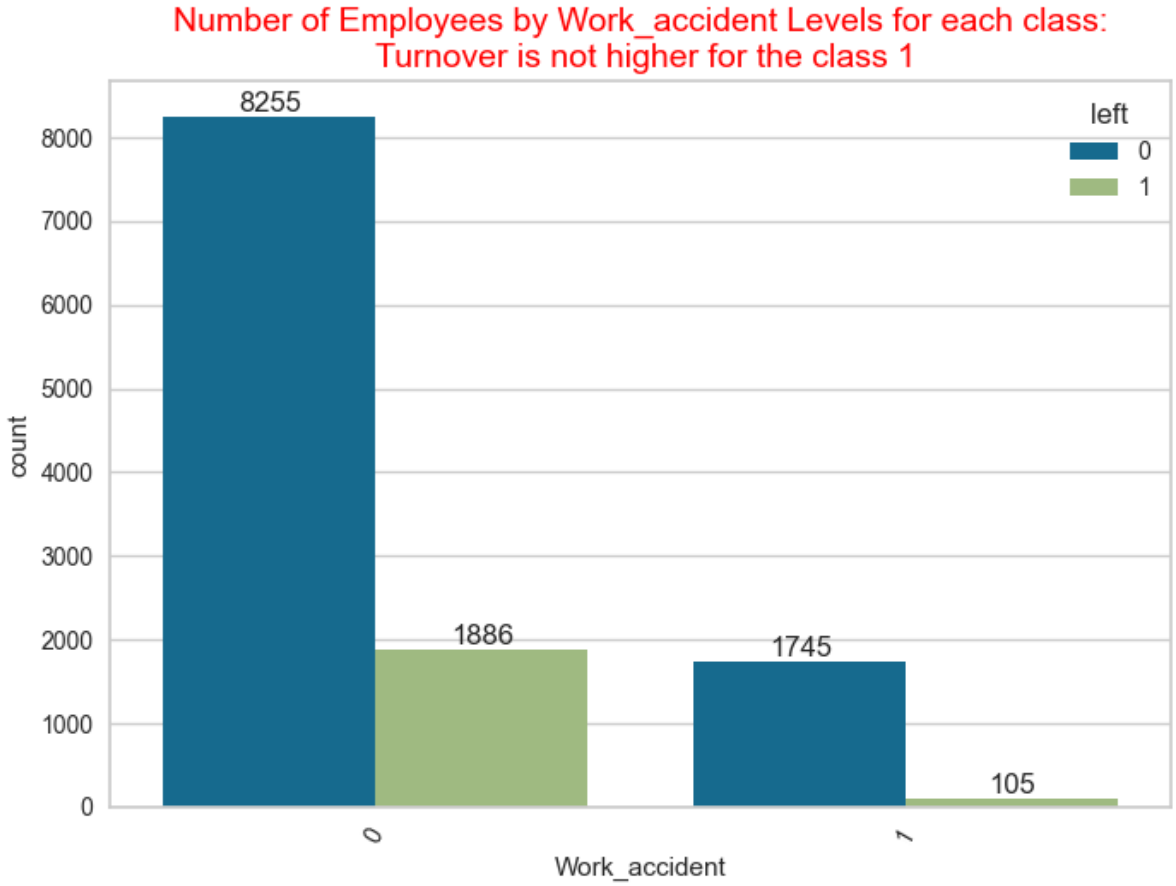

**Check the count of person in each "left" levels by Work_accident and visualize them with countplot**

In [ ]:
```
# Checking "Work_accident" feature by left in detail

df.groupby("Work_accident").left.value_counts()
```

Out[ ]:
```
Work_accident  left
0              0       8255
               1       1886
1              0       1745
               1        105
Name: count, dtype: int64
```

In [ ]:
```
# Visualizing the number of people in each "Work_accident" level by left

ax = sns.countplot(data=df, x="Work_accident", hue="left")
```

```
plt.title("Number of Employees by Work_accident Levels for each class:\n Tur

plt.xticks(rotation = 60)

for container in ax.containers:
    ax.bar_label(container);
```



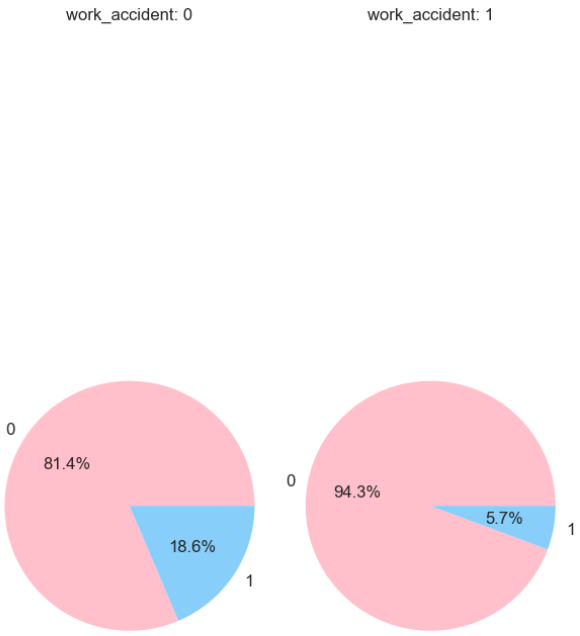

**Check the percentage distribution of person in each "left" levels by Work_accident and visualize it with pie plot separately**

In [ ]:
```
# The Proportional Distribution of persons in each sub-categories of "Work_a

edu = df.groupby(["Work_accident"]).left.value_counts(normalize=True)
edu
```

Out[ ]:
```
Work_accident  left
0              0       0.814022
               1       0.185978
1              0       0.943243
               1       0.056757
Name: proportion, dtype: float64
```

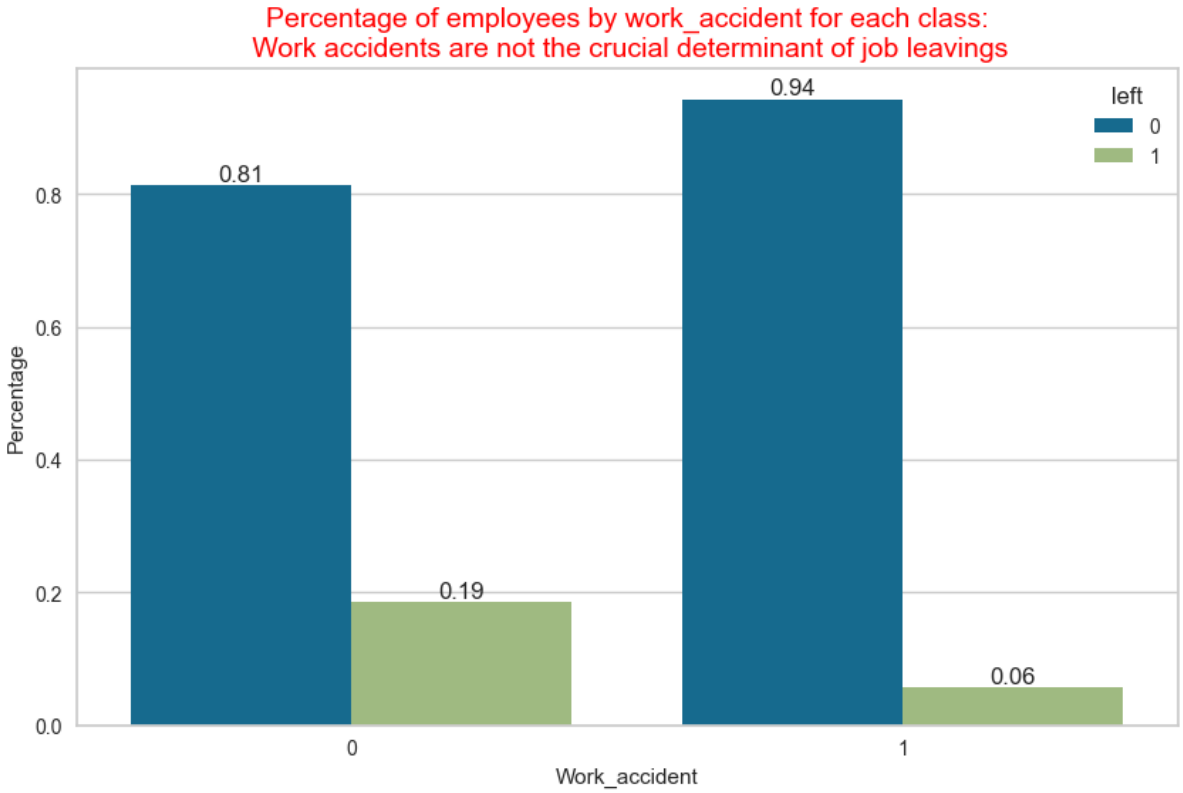
In [ ]:
```
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["Work_accident"]).left.value_counts(normalize=Tru

# Calculate the number of Work_accident
num_Work_accident = len(grouped_data.index.levels[0])

# Calculate the number of rows and columns in the grid
num_columns = 4
num_rows = (num_Work_accident + num_columns - 1) // num_columns
```

```
# Create a figure with subplots
fig, axes = plt.subplots(num_rows, num_columns, figsize=(12, 10))

# Flatten the axes array to simplify iteration
axes = axes.flatten()

# Iterate over each work_accident
for i, work_accident in enumerate(grouped_data.index.levels[0]):
    # Filter data for the current work_accident
    work_accident_data = grouped_data[work_accident]

    # Create a pie plot in the corresponding subplot
    axes[i].pie(work_accident_data, labels=work_accident_data.index, autopct
    axes[i].set_title(f"work_accident: {work_accident}")
    axes[i].axis('equal')  # Equal aspect ratio ensures a circular pie

# Hide any remaining empty subplots
for j in range(num_Work_accident, num_rows * num_columns):
    axes[j].axis('off')

# Adjust the spacing between subplots
plt.tight_layout()

# Show the pie plots
plt.show()
```




In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["Work_accident"]).left.value_counts(normalize=Tru

# Reset the index to convert the grouped series to a DataFrame
```

```
grouped_data = grouped_data.reset_index(name='percentage')

# Create a bar plot using seaborn
plt.figure(figsize=(10, 6))
ax = sns.barplot(data=grouped_data, x='Work_accident', y='percentage', hue='
plt.xlabel('Work_accident')
plt.ylabel('Percentage')
plt.title('Percentage of employees by work_accident for each class:\n Work a
the crucial determinant of job leavings', fontsize=14, color="red")
plt.bar_label(ax.containers[0], fmt='%.2f')
plt.bar_label(ax.containers[1], fmt='%.2f')

# Show the bar plot
plt.show();
```




**Check the the percentage distribution of person in each work_accident by "left" levels and visualize it with bar plot**

In [ ]:
```
# Checking the percentage of persons at "work_accident" by "left" levels

work_accident = df.groupby("left").Work_accident.value_counts(normalize=True
work_accident
```

Out[ ]:
```
left  Work_accident
0     0                82.550000
      1                17.450000
1     0                94.726268
      1                 5.273732
Name: proportion, dtype: float64
```

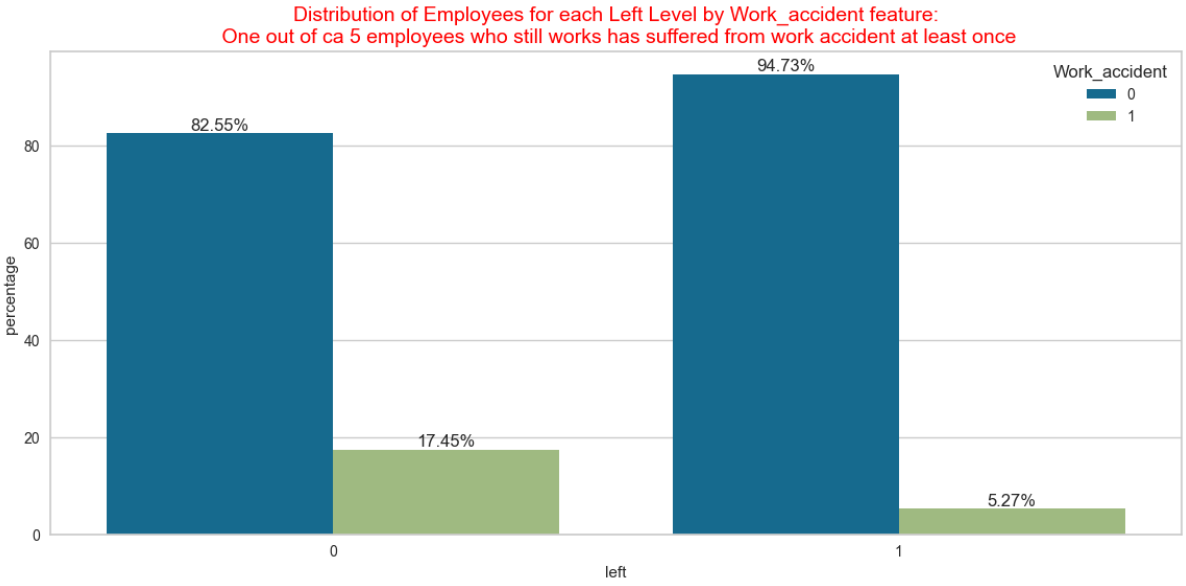
In [ ]:
```
# Creating a dataframe demonstrating the percentage of persons at "work_acci

work_accident_df = pd.DataFrame(work_accident)
work_accident_df.rename(columns={"proportion": "percentage"}, inplace=True)
work_accident_df.reset_index(inplace=True)
work_accident_df.sort_values(by=["left", "Work_accident"], inplace=True)
work_accident_df
```

Out[ ]:

| | left | Work_accident | percentage |
|---|---|---|---|
| **0** | 0 | 0 | 82.550000 |
| **1** | 0 | 1 | 17.450000 |
| **2** | 1 | 0 | 94.726268 |
| **3** | 1 | 1 | 5.273732 |

In [ ]:

```
# Visualizing the percentage of persons at "Work_accident" by "left" levels

fig, ax = plt.subplots(figsize=(14, 6))

ax = sns.barplot(data=work_accident_df, x="left", y="percentage", hue="Work_

plt.title("Distribution of Employees for each Left Level by Work_accident fe
of ca 5 employees who still works has suffered from work accident at least o

for container in ax.containers:
    ax.bar_label(container, fmt="%.2f%%", fontsize=12);
```




**General assessment for work accident feature**:

The average work accident level for white-collar jobs tends to be relatively lower compared to jobs that involve manual labor or physical tasks. White-collar jobs typically involve office-based work, administrative tasks, or professional services that do not pose the same level of physical risk as jobs in industries such as construction, manufacturing, or transportation. While there may still be potential hazards in white-collar workplaces, such as ergonomic issues, slips, trips, and falls, or exposure to certain substances, the overall risk of work accidents is generally lower. White-collar workers typically have a lower likelihood of sustaining severe injuries or accidents related to heavy machinery, hazardous materials, or physically demanding tasks.

In case of this data, it can be seen that the work accidents are pretty high (1850 out of ca 12k). Nevertheless, it is interesting that the most of the employees, both in number and ratio, who suffered from work accidents are fromamong the employees who did not leave the job and still continue working. There can be several factors underlying this fact. Here are some potential reasons:

- **Employee Loyalty**: Some employees may have a strong sense of loyalty towards their job or the company they work for. Despite experiencing a work accident, they may choose to continue working, believing that the incident was an isolated occurrence or that the company will take steps to improve safety measures.
- **Financial Considerations**: The financial implications of leaving a job after a work accident can be a significant factor. Employees may hesitate to quit their job if they rely on their income to support themselves or their families. They may be concerned about finding a new job with similar compensation or benefits, particularly if they have ongoing medical expenses resulting from the work accident.
- **Job Market Concerns**: Employees may perceive limited job opportunities or increased competition in the job market, which could deter them from leaving their current position, even after experiencing a work accident. This concern may be particularly relevant if the employees believe that the work accident was an isolated incident and that their current job still offers long-term stability.
- **Supportive Work Environment**: If the employees feel that their employer is supportive and takes appropriate actions to address the work accident, they may be more inclined to continue working. This could involve providing medical assistance, implementing safety improvements, or offering compensation for any damages or losses incurred.
- **Personal Attachment or Career Prospects**: Employees may have a personal attachment to their job or career prospects within the company that outweigh their concerns about the work accident. They may have invested significant time and effort into their position and see potential for growth or advancement in the future.

It's important to note that individual motivations and circumstances can vary, and these factors are just some potential explanations. Further analysis and investigation for this company, including employee surveys or interviews, can provide deeper insights into the reasons why employees choose to continue working despite experiencing work accidents.

### promotion_last_5years

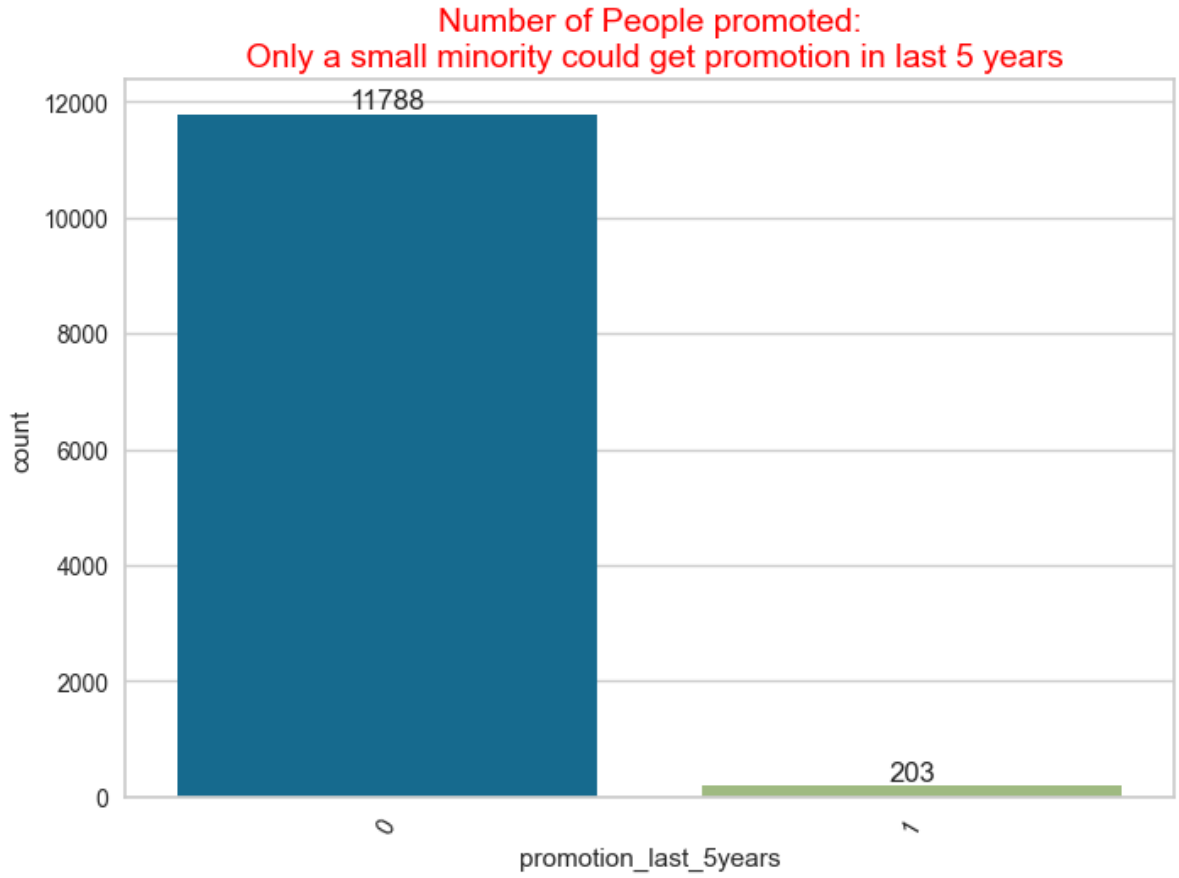
In [ ]:
```
# Checking the uniques of "promotion_last_5years" feature and determining th

df.promotion_last_5years.value_counts(dropna=False)
```

Out[ ]:
```
promotion_last_5years
0    11788
1      203
Name: count, dtype: int64
```

**Visualize the count of person in promotion_last_5years**

In [ ]:
```
# Visualization of "promotion_last_5years" feature

ax = sns.countplot(data=df, x="promotion_last_5years")

plt.xticks(rotation=60)
```

```
plt.title("Number of People promoted:\n Only a small minority could get prom
in last 5 years", fontsize=14, color="red")

for container in ax.containers:
    ax.bar_label(container);
```




**Check the count of person in each "left" levels by promotion_last_5years and visualize them with countplot**

In [ ]:
```
# Checking "promotion_last_5years" feature by left in detail

df.groupby("promotion_last_5years").left.value_counts()
```

Out[ ]:
```
promotion_last_5years  left
0                      0       9805
                       1       1983
1                      0        195
                       1          8
Name: count, dtype: int64
```

In [ ]:
```
# Visualizing the number of people in each "promotion_last_5years" level by

ax = sns.countplot(data=df, x="promotion_last_5years", hue="left")

plt.title("Number of employees for each promotion group by target classes",

plt.xticks(rotation = 60)

for container in ax.containers:
    ax.bar_label(container);
```

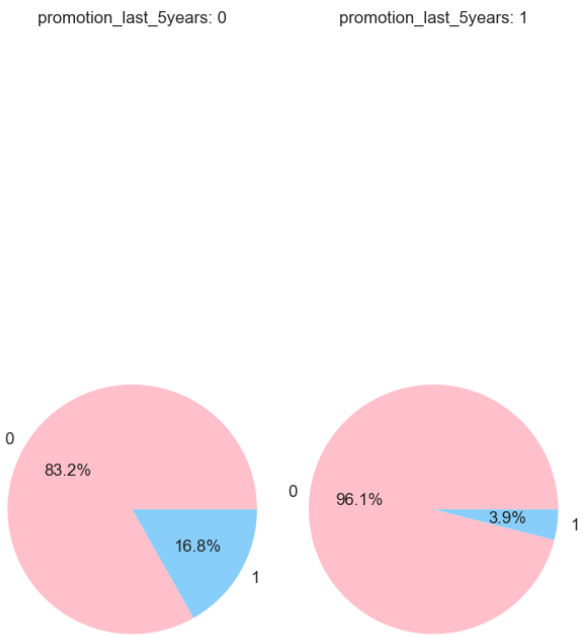

**Check the percentage distribution of person in each "left" levels by promotion_last_5years and visualize it with pie plot separately**

In [ ]:
```
# The Proportional Distribution of persons in each sub-categories of "promot

edu = df.groupby(["promotion_last_5years"]).left.value_counts(normalize=True
edu
```

Out[ ]:
```
promotion_last_5years  left
0                      0       0.831778
                       1       0.168222
1                      0       0.960591
                       1       0.039409
Name: proportion, dtype: float64
```

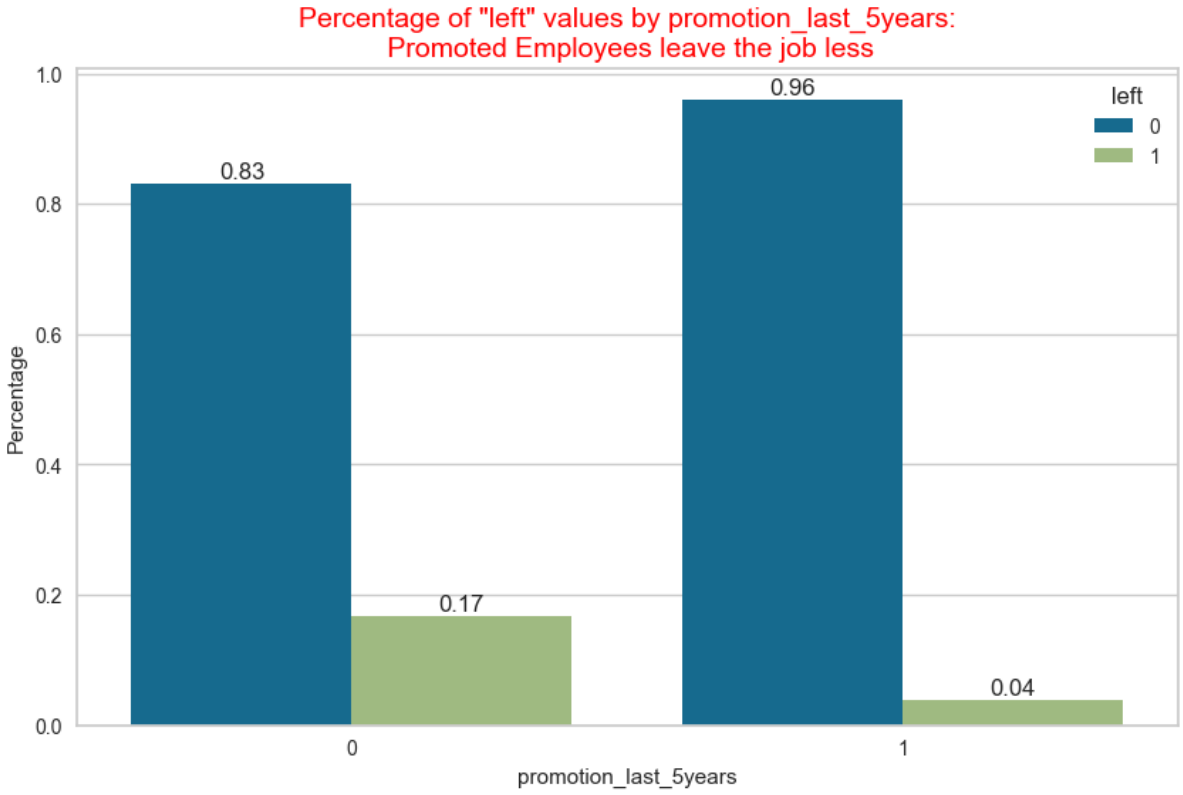
In [ ]:
```
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["promotion_last_5years"]).left.value_counts(norma

# Calculate the number of promotion_last_5years
num_promotion_last_5years = len(grouped_data.index.levels[0])

# Calculate the number of rows and columns in the grid
num_columns = 4
num_rows = (num_promotion_last_5years + num_columns - 1) // num_columns

# Create a figure with subplots
fig, axes = plt.subplots(num_rows, num_columns, figsize=(12, 10))

# Flatten the axes array to simplify iteration
axes = axes.flatten()

# Iterate over each promotion_last_5years
for i, promotion_last_5years in enumerate(grouped_data.index.levels[0]):
    # Filter data for the current promotion_last_5years
```

```
    promotion_last_5years_data = grouped_data[promotion_last_5years]

    # Create a pie plot in the corresponding subplot
    axes[i].pie(promotion_last_5years_data, labels=promotion_last_5years_dat
    axes[i].set_title(f"promotion_last_5years: {promotion_last_5years}")
    axes[i].axis('equal')  # Equal aspect ratio ensures a circular pie

# Hide any remaining empty subplots
for j in range(num_promotion_last_5years, num_rows * num_columns):
    axes[j].axis('off')

# Adjust the spacing between subplots
plt.tight_layout()

# Show the pie plots
plt.show()
```




In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Grouped data
grouped_data = df.groupby(["promotion_last_5years"]).left.value_counts(norma

# Reset the index to convert the grouped series to a DataFrame
grouped_data = grouped_data.reset_index(name='percentage')

# Create a bar plot using seaborn
plt.figure(figsize=(10, 6))
ax = sns.barplot(data=grouped_data, x='promotion_last_5years', y='percentage
plt.xlabel('promotion_last_5years')
plt.ylabel('Percentage')
plt.title('Percentage of "left" values by promotion_last_5years:\n Promoted
          fontsize=14, color="red")
```

```
plt.bar_label(ax.containers[0], fmt='%.2f')
plt.bar_label(ax.containers[1], fmt='%.2f')

# Show the bar plot
plt.show();
```




**Check the the percentage distribution of person in each promotion_last_5years by "left" levels and visualize it with bar plot**

In [ ]:
```
# Checking the percentage of persons at "promotion_last_5years" by "left" le

promotion_last_5years = df.groupby("left").promotion_last_5years.value_count
promotion_last_5years
```

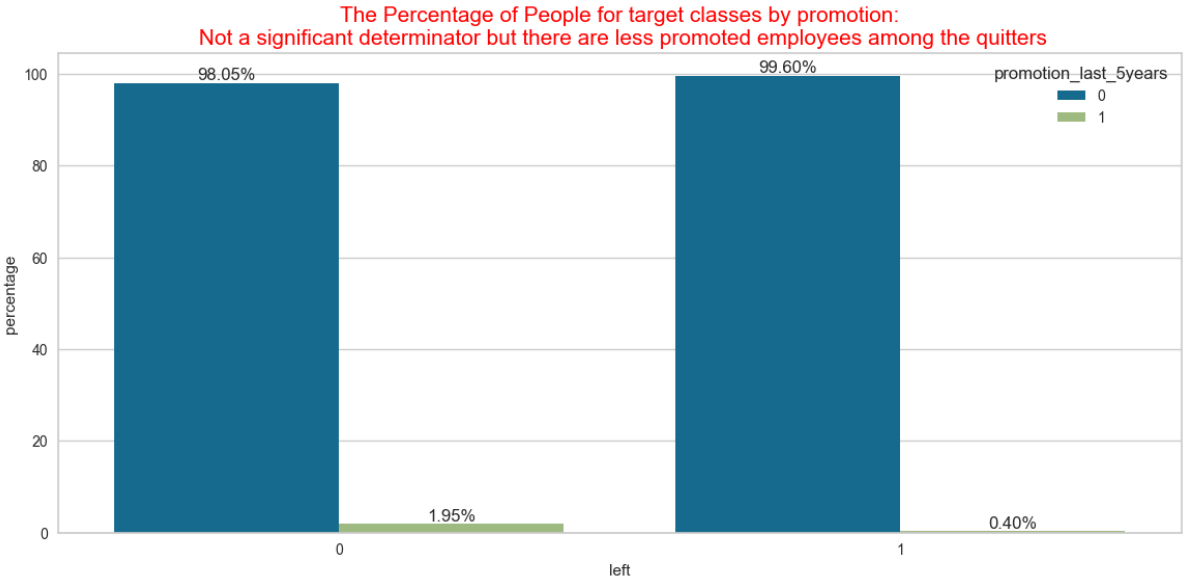
Out[ ]:
```
left  promotion_last_5years
0     0                        98.050000
      1                         1.950000
1     0                        99.598192
      1                         0.401808
Name: proportion, dtype: float64
```

In [ ]:
```
# Creating a dataframe demonstrating the percentage of persons at "promotion

promotion_last_5years_df = pd.DataFrame(promotion_last_5years)
promotion_last_5years_df.rename(columns={"proportion": "percentage"}, inplac
promotion_last_5years_df.reset_index(inplace=True)
promotion_last_5years_df.sort_values(by=["left", "promotion_last_5years"], i
promotion_last_5years_df
```

Out[ ]:

| | left | promotion_last_5years | percentage |
|---|---|---|---|
| **0** | 0 | 0 | 98.050000 |
| **1** | 0 | 1 | 1.950000 |
| **2** | 1 | 0 | 99.598192 |
| **3** | 1 | 1 | 0.401808 |

In [ ]:
```
# Visualizing the percentage of persons at "promotion_last_5years" by "left"

fig, ax = plt.subplots(figsize=(14, 6))

ax = sns.barplot(data=promotion_last_5years_df, x="left", y="percentage", hu

plt.title("The Percentage of People for target classes by promotion:\n Not a
but there are less promoted employees among the quitters", fontsize=15, colo

for container in ax.containers:
    ax.bar_label(container, fmt="%.2f%%", fontsize=12);
```




**General assessment of promotion feature**:

Promotions can be an important factor in an employee's decision to stay in or leave a company. There are several underlying reasons for this effect. For example:

- **Career Growth and Development**: Promotions provide employees with opportunities for career advancement and growth within the company. It allows them to take on new responsibilities, learn new skills, and expand their knowledge and experience. Employees who see a clear path for growth and development are more likely to be motivated and engaged in their work, leading to increased job satisfaction and a lower likelihood of leaving the company.

- **Recognition and Rewards**: Promotions often come with increased compensation, benefits, and recognition for the employee's hard work and contributions. Being promoted can signify that their efforts and achievements are valued by the company. This recognition and reward can enhance job satisfaction and increase loyalty towards the organization, reducing the likelihood of seeking employment elsewhere.

- **Increased Responsibility and Authority**: Promotions typically involve higher-level roles with greater responsibility and decision-making authority. Employees who aspire to take on more challenging and impactful roles may see promotions as stepping stones towards their desired career trajectory. Having a sense of autonomy and influence in their work can contribute to job satisfaction and a desire to stay with the company.

- **Competitive Advantage in the Job Market**: Employees who have received promotions and have a track record of career progression are often viewed more favorably in the job market. The promotion demonstrates their capabilities, skills, and potential for future success, making them more attractive to other employers. Offering opportunities for advancement and promoting employees can help companies retain top talent and prevent them from seeking opportunities elsewhere.
- **Motivation and Engagement**: Promotions can serve as motivational factors for employees to perform at their best and strive for excellence. The prospect of advancement and the associated benefits can drive employees to stay committed and engaged in their work. It creates a sense of purpose and challenge that can contribute to job satisfaction and employee retention.

Despite all these advantages of the promotions, as can be seen above, only a couple hundred of employees could get a promotion in this company. Considering that the average working years of employees in this company is only 3, this means that most employees have not worked for a sufficient amount of time to be eligible for promotion. However, on the contrary, if the employees notice that other employees with long service years are not being promoted, they can label the company as an organisation without an accurate promotion policy. This might be a reason for the employees to leave the company in search of better career opportunities in other companies. For a company with a high turnover rate of around 20%, this second interpretation appears to be more logical.

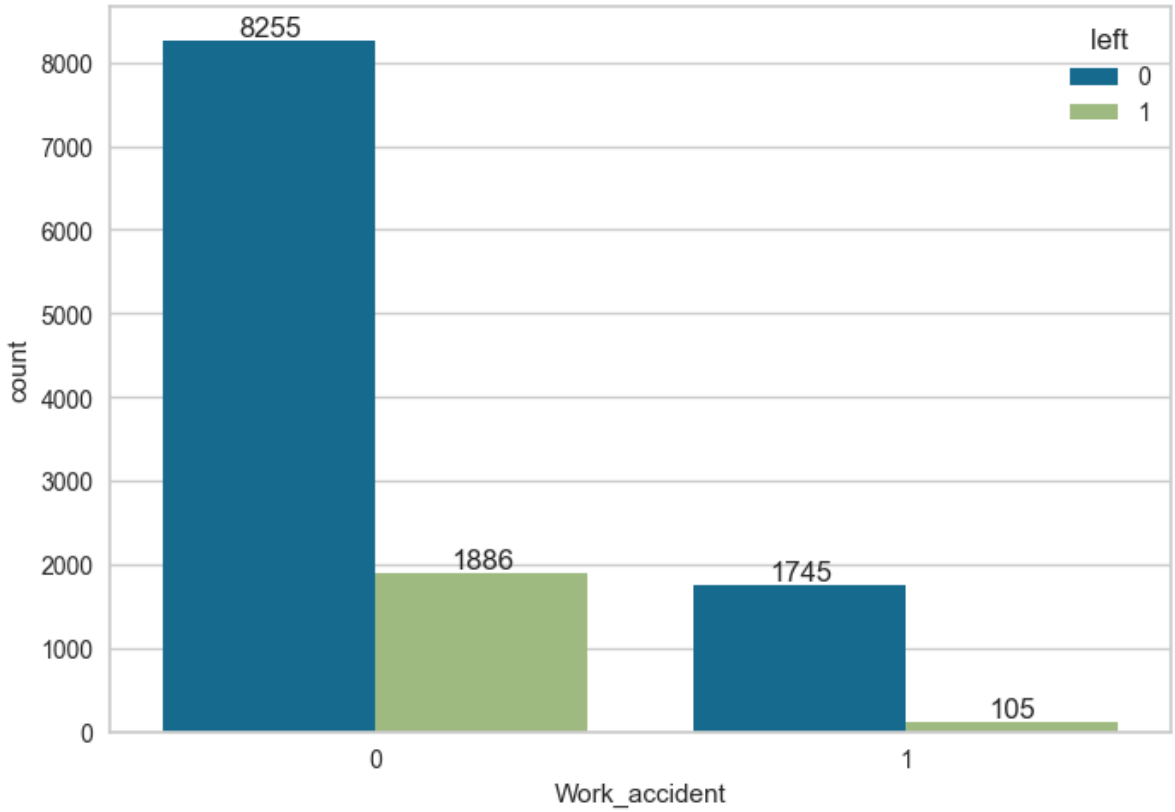
## Questions and statistical significance tests

In this section of the project, we will attempt to digest the data more by diving into details via some specific questions.The statistical tests such as t test, shapiro,fisher exact or chi-square tests will enable us to check whether the insights we have extrated from the detailed visual analysis above are statistically significant or not.

### What is the distribution of the "left" variable among different categories of the "Work_accident" column?

In [ ]:
```
ax = sns.countplot(x='Work_accident', hue='left', data=df)
plt.bar_label(ax.containers[0])
plt.bar_label(ax.containers[1])
```

Out[ ]:
```
[Text(0, 0, '1886'), Text(0, 0, '105')]
```




## Is there a significant difference in the proportion of employees who left the company based on whether they had a work accident or not?

If the data is imbalanced, it is appropriate to use Fisher's exact test instead of the chi-square test. Fisher's exact test is commonly used when the sample size is small or when the data is imbalanced.

**What is Fisher's Exact test and why is it used?**

The Fisher's exact test is a statistical test used to determine the significance of the association between two categorical variables in a 2x2 contingency table. It is commonly used when the sample size is small or when there are expected cell frequencies below 5.

The Fisher's exact test calculates the probability of obtaining a distribution of data as extreme as the observed data, assuming that the null hypothesis is true (i.e., no association between the variables). If the calculated p-value is below a predetermined significance level (e.g., 0.05), it indicates that the association between the variables is statistically significant.

The Fisher's exact test is particularly useful in situations where other statistical tests, such as the chi-square test, may not be valid due to small sample sizes or low expected cell frequencies. It provides a more accurate assessment of the association between variables in such cases.

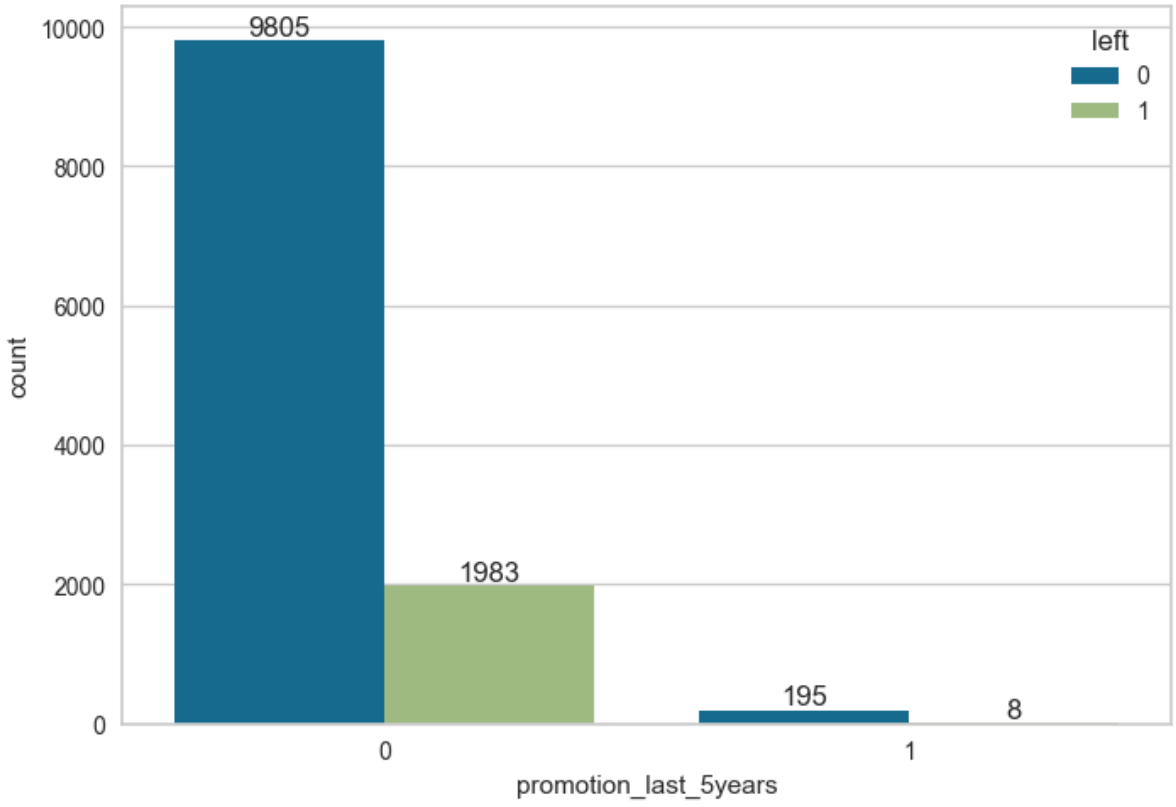
In [ ]:
```
import pandas as pd
import scipy.stats as stats

# Create a contingency table of 'Work_accident' and 'left' variables
contingency_table = pd.crosstab(df['Work_accident'], df['left'])
```

```
# Perform Fisher's exact test
odds_ratio, p_value = stats.fisher_exact(contingency_table)

# Display the test results
print("Odds ratio:", odds_ratio)
print("p-value:", p_value)

alpha = 0.05
if p_value < alpha:
    print("The results are statistically significant at a significance level
    print("There is evidence to suggest a significant difference in the prop
else:
    print("The results are not statistically significant at a significance l
    print("There is no significant difference in the proportion of employees
```

```
Odds ratio: 0.26337179093729396
p-value: 3.594805527537249e-52
The results are statistically significant at a significance level of 0.05
There is evidence to suggest a significant difference in the proportion of e
mployees who left the company based on whether they had a work accident or n
ot.
```

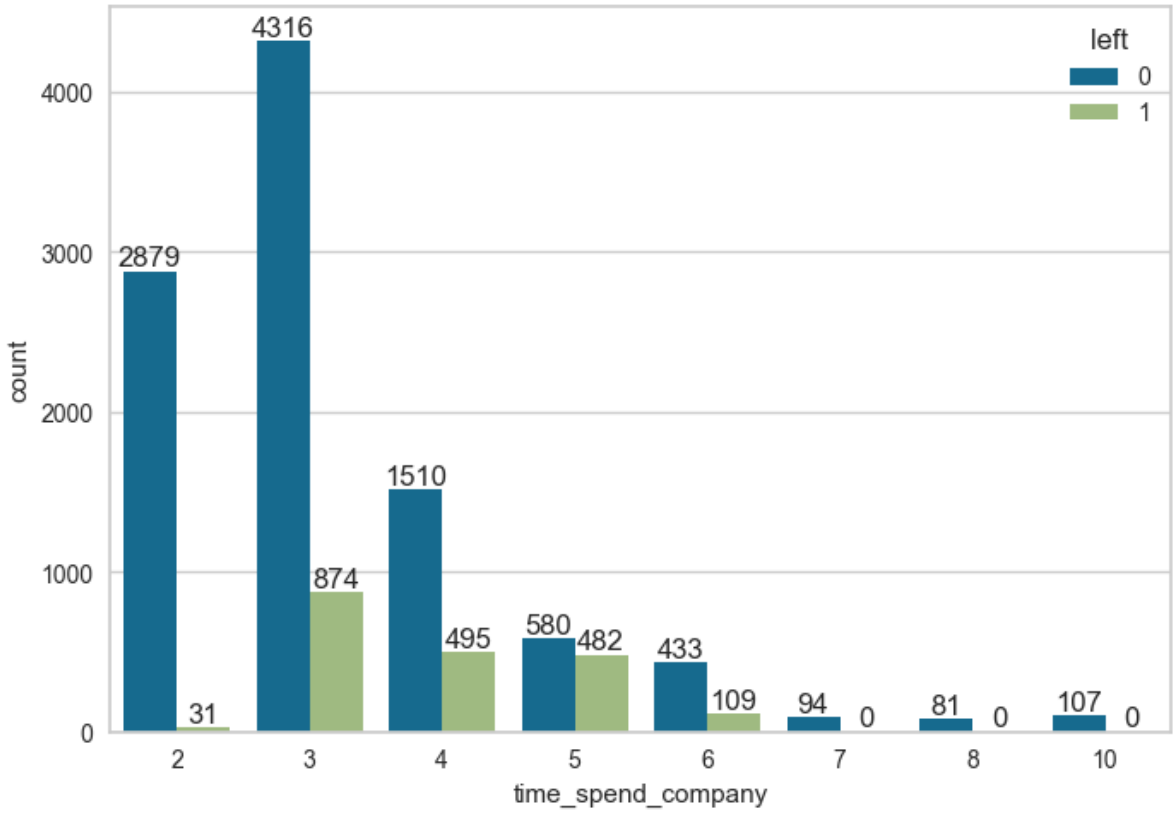
## What is the distribution of the "left" variable among different categories of the "promotion_last_5years" column?

In [ ]:
```
ax = sns.countplot(x='promotion_last_5years', hue='left', data=df)
plt.bar_label(ax.containers[0])
plt.bar_label(ax.containers[1]);
```



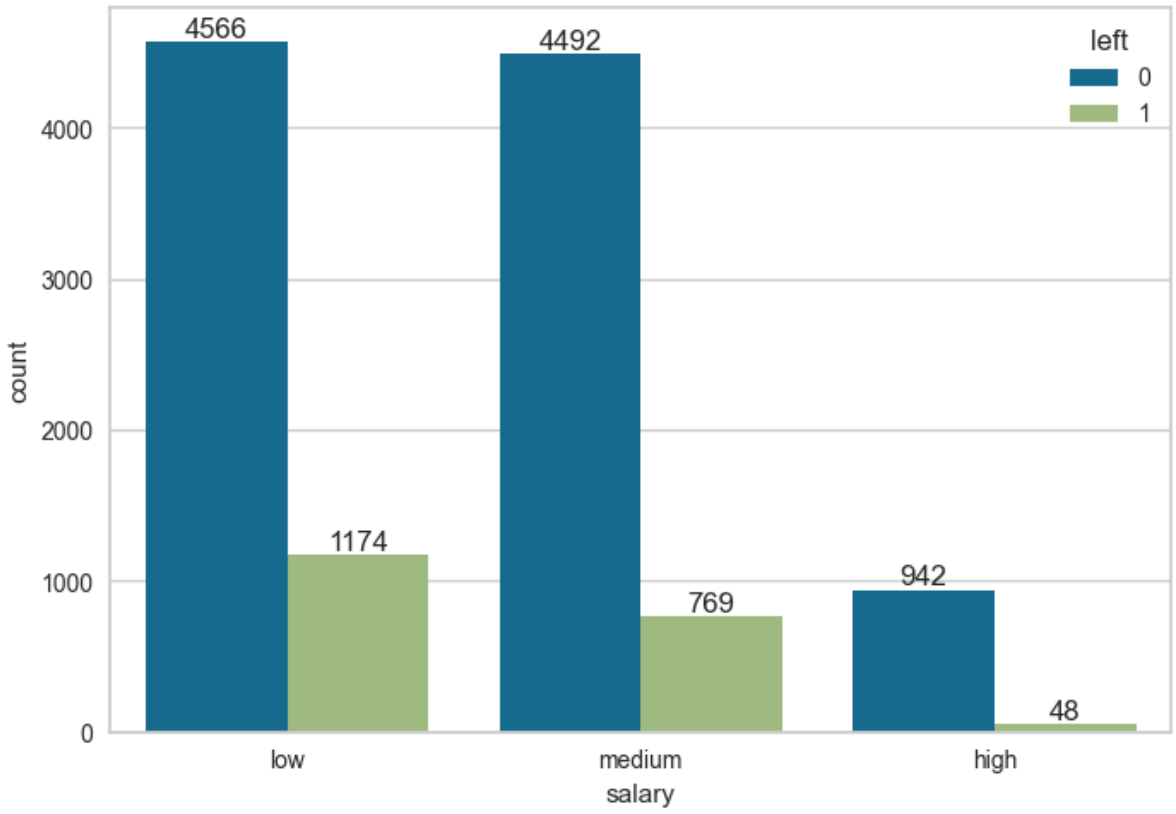

## What is the distribution of the "left" variable among different categories of the "time_spend_company" column?

In [ ]:
```
ax = sns.countplot(x='time_spend_company', hue='left', data=df)
plt.bar_label(ax.containers[0])
plt.bar_label(ax.containers[1]);
```



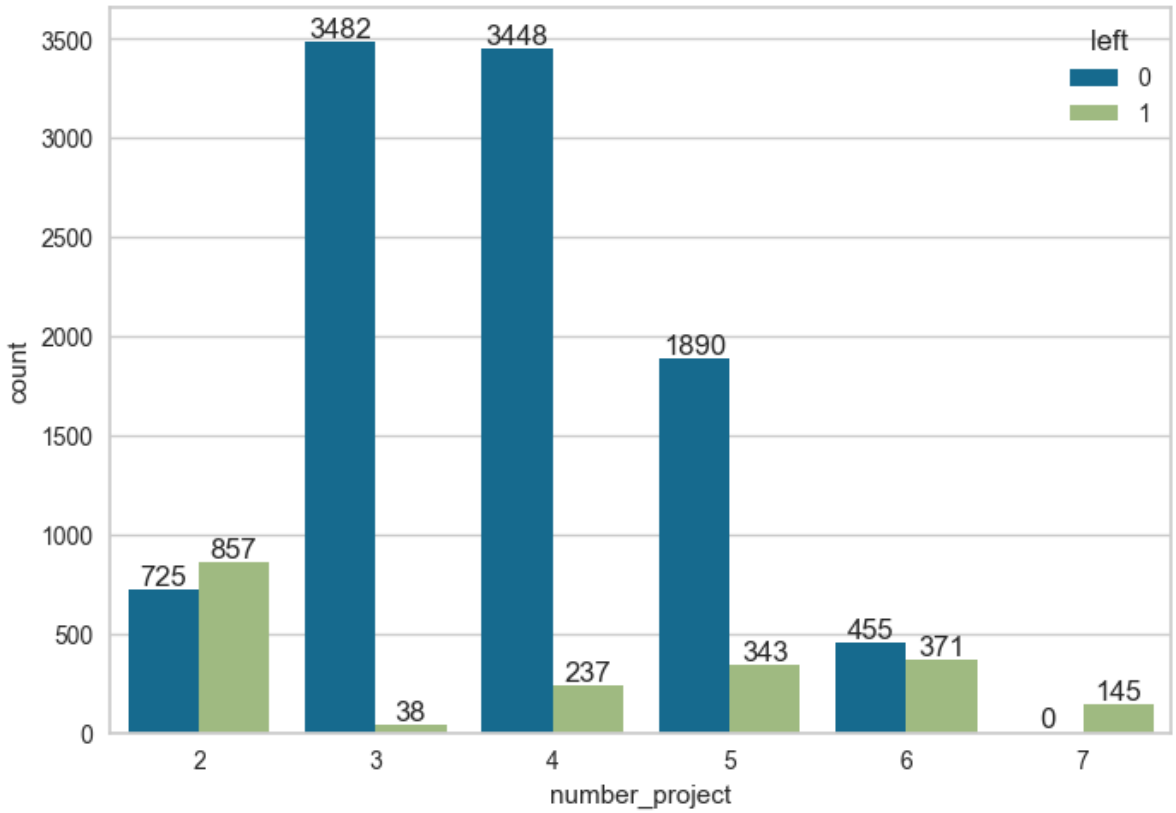

## What is the distribution of the "left" variable among different categories of the "salary" column?

In [ ]:
```
ax = sns.countplot(x='salary', hue='left', data=df)
plt.bar_label(ax.containers[0])
plt.bar_label(ax.containers[1]);
```



## What is the distribution of the "left" variable among different categories of the "number_project" column?

In [ ]:
```
ax = sns.countplot(x='number_project', hue='left', data=df)
plt.bar_label(ax.containers[0])
plt.bar_label(ax.containers[1]);
```




## How is the "left" variable related to the other columns? Are there any significant differences in average values or distributions between employees who left and those who stayed?

In [ ]:
```
df.select_dtypes('number').groupby("left").mean()
```

Out[ ]:

| left | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_ |
|---|---|---|---|---|---|
| 0 | 0.667365 | 0.715667 | 3.786800 | 198.94270 | |
| 1 | 0.440271 | 0.721783 | 3.883476 | 208.16223 | |

In [ ]:
```
import pandas as pd
import scipy.stats as stats


# Select numerical columns
numerical_columns = df.select_dtypes('number').columns

# Perform statistical tests for each numerical column
for column in numerical_columns:
    left_values = df[df['left'] == 1][column]
    stayed_values = df[df['left'] == 0][column]
```

```
    # Perform t-test or Mann-Whitney U test based on data distribution
    if stats.normaltest(df[column])[1] < 0.05:
        # Data is not normally distributed, perform Mann-Whitney U test
        statistic, p_value = stats.mannwhitneyu(left_values, stayed_values)
        test_name = "Mann-Whitney U test"
    else:
        # Data is normally distributed, perform t-test
        statistic, p_value = stats.ttest_ind(left_values, stayed_values)
        test_name = "t-test"

    # Display the results
    print(f"Statistical test for column '{column}':")
    print(f"{test_name} - statistic: {statistic:.4f}, p-value: {p_value:.4f}

    # interpretation of the results based on p-value
    alpha = 0.05
    if p_value < alpha:
        print("The p-value is less than the significance level of", alpha)
        print("There is a significant difference in average values between e
    else:
        print("The p-value is greater than or equal to the significance leve
        print("There is no significant difference in average values between

    print()
```

```
Statistical test for column 'satisfaction_level':
Mann-Whitney U test - statistic: 5037062.0000, p-value: 0.0000
The p-value is less than the significance level of 0.05
There is a significant difference in average values between employees who le
ft and those who stayed for column satisfaction_level

Statistical test for column 'last_evaluation':
Mann-Whitney U test - statistic: 10044036.0000, p-value: 0.5278
The p-value is greater than or equal to the significance level of 0.05
There is no significant difference in average values between employees who l
eft and those who stayed for column last_evaluation

Statistical test for column 'number_project':
Mann-Whitney U test - statistic: 9835415.0000, p-value: 0.3810
The p-value is greater than or equal to the significance level of 0.05
There is no significant difference in average values between employees who l
eft and those who stayed for column number_project

Statistical test for column 'average_montly_hours':
Mann-Whitney U test - statistic: 10690674.0000, p-value: 0.0000
The p-value is less than the significance level of 0.05
There is a significant difference in average values between employees who le
ft and those who stayed for column average_montly_hours

Statistical test for column 'time_spend_company':
Mann-Whitney U test - statistic: 13753466.0000, p-value: 0.0000
The p-value is less than the significance level of 0.05
There is a significant difference in average values between employees who le
ft and those who stayed for column time_spend_company

Statistical test for column 'Work_accident':
Mann-Whitney U test - statistic: 8742852.5000, p-value: 0.0000
The p-value is less than the significance level of 0.05
There is a significant difference in average values between employees who le
ft and those who stayed for column Work_accident

Statistical test for column 'promotion_last_5years':
Mann-Whitney U test - statistic: 9800877.5000, p-value: 0.0000
The p-value is less than the significance level of 0.05
There is a significant difference in average values between employees who le
ft and those who stayed for column promotion_last_5years

Statistical test for column 'left':
Mann-Whitney U test - statistic: 19910000.0000, p-value: 0.0000
The p-value is less than the significance level of 0.05
There is a significant difference in average values between employees who le
ft and those who stayed for column left
```

### Mann-Whitney U test; Definition and Usage

The Mann-Whitney U test, also known as the Wilcoxon rank-sum test, is a non-parametric statistical test used to compare the distributions of two independent samples. It is commonly used when the data does not meet the assumptions of normality required for parametric tests like the t-test.

The Mann-Whitney U test assesses whether there is a significant difference between the medians of the two groups. It works by ranking all the observations from both groups together and calculating the sum of ranks for each group. The test statistic, U, represents the probability that a randomly selected value from one group will be greater than a randomly selected value from the other group.

The Mann-Whitney U test provides a p-value that indicates the likelihood of observing the data if there is no difference between the groups. If the p-value is below a predetermined significance level (e.g., 0.05), it suggests that there is a statistically significant difference between the groups.

It is important to note that the Mann-Whitney U test does not provide information about the direction or magnitude of the difference between the groups. It only determines if there is a significant difference.

The Mann-Whitney U test is particularly useful when working with ordinal or skewed data, or when the assumptions of parametric tests are not met. It is widely used in various fields, including healthcare, social sciences, and business, to compare two groups and determine if they differ significantly.

**T-Test; Definition and Usage**

The t-test is a statistical test used to compare the means of two independent groups and determine if they are significantly different from each other. It is a parametric test that assumes the data follows a normal distribution.

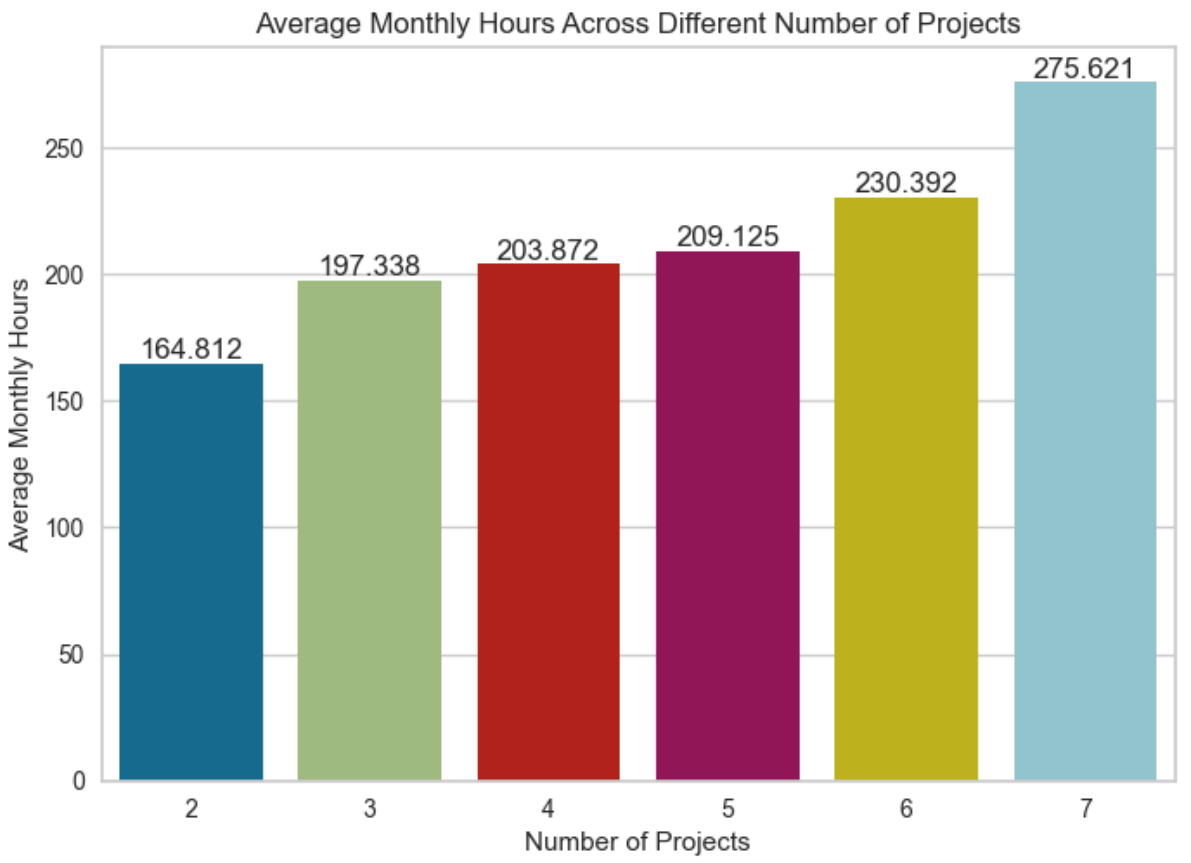
The t-test calculates a t-statistic, which measures the difference between the means of the two groups relative to the variability within each group. The test also considers the sample sizes of the groups and the standard deviations of the data.

There are two main types of t-tests: the independent samples t-test and the paired samples t-test. The independent samples t-test is used when the two groups being compared are independent and have distinct individuals in each group. The paired samples t-test, on the other hand, is used when the two groups represent measurements from the same individuals or matched pairs.

The t-test provides a p-value, which indicates the probability of obtaining the observed difference in means if there is no true difference between the groups. If the p-value is below a predetermined significance level (e.g., 0.05), it suggests that there is a statistically significant difference between the groups.

## How does the average monthly hours vary across different number of projects done?

In [ ]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


# Group the data by 'number_project' and calculate the average monthly hours
avg_hours_by_project = df.groupby('number_project')['average_montly_hours'].

# Plotting the average monthly hours for different number of projects
ax = sns.barplot(x=avg_hours_by_project.index, y=avg_hours_by_project.values
plt.xlabel('Number of Projects')
plt.ylabel('Average Monthly Hours')
plt.title('Average Monthly Hours Across Different Number of Projects')
plt.bar_label(ax.containers[0])
plt.show()
```




## Can we identify any differences in the average satisfaction level between employees who had a work accident and those who didn't?

In [ ]:

```
import pandas as pd
import scipy.stats as stats

# Selecting the relevant columns
columns_of_interest = ['satisfaction_level', 'Work_accident']
data_subset = df[columns_of_interest]

# Splitting the data based on 'Work_accident'
had_accident = data_subset[data_subset['Work_accident'] == 1]
no_accident = data_subset[data_subset['Work_accident'] == 0]

# Performing the Shapiro-Wilk test for normality
_, p_value_had_accident = stats.shapiro(had_accident['satisfaction_level'])
_, p_value_no_accident = stats.shapiro(no_accident['satisfaction_level'])

# Checking normality assumption
if p_value_had_accident > 0.05 and p_value_no_accident > 0.05:
    # Data is normally distributed, performing t-test
    t_statistic, p_value = stats.ttest_ind(had_accident['satisfaction_level'
    test_name = "t-test"
else:
    # Data is not normally distributed, performing Mann-Whitney U test
    t_statistic, p_value = stats.mannwhitneyu(had_accident['satisfaction_lev
    test_name = "Mann-Whitney U test"

# Displaying the results
print("Statistical Test for Average Satisfaction Level:")
print("Test Used:", test_name)
print("p-value:", p_value)
print()
```

```
alpha = 0.05
if p_value < alpha:
    print("The p-value is less than the significance level of", alpha)
    print("There is a significant difference in the average satisfaction lev
else:
    print("The p-value is greater than or equal to the significance level of
    print("There is no significant difference in the average satisfaction le
```

```
Statistical Test for Average Satisfaction Level:
Test Used: Mann-Whitney U test
p-value: 5.9414904057540924e-05

The p-value is less than the significance level of 0.05
There is a significant difference in the average satisfaction level between
employees who had a work accident and those who didn't.
```

**The Shapiro-Wilk test; Definition and Usage**

The Shapiro-Wilk test is a statistical test used to assess the normality of a given dataset. It tests the null hypothesis that the data is normally distributed against the alternative hypothesis that it is not.

The Shapiro-Wilk test calculates a test statistic based on the discrepancy between the observed data and the expected values under the assumption of normality. It produces a p-value, which indicates the probability of obtaining the observed data if the null hypothesis is true (i.e., if the data is normally distributed).

If the p-value from the Shapiro-Wilk test is greater than a predetermined significance level (e.g., 0.05), it suggests that there is no strong evidence to reject the null hypothesis of normality. Conversely, if the p-value is below the significance level, it indicates that the data significantly deviates from a normal distribution.

The Shapiro-Wilk test is particularly useful when determining whether a dataset follows a normal distribution, which is often an assumption for many parametric statistical tests. By assessing the normality of the data, researchers can make informed decisions about the appropriateness of parametric tests or consider alternative non-parametric methods if the data is non-normal.

## Is there a statistically significant association between the salary level of employees and the likelihood of them leaving the company?

In [ ]:
```
import pandas as pd
import scipy.stats as stats

# Creating a contingency table of salary and left columns
contingency_table = pd.crosstab(df['salary'], df['left'])

# Performing the chi-square test of independence
chi2_statistic, p_value, _, _ = stats.chi2_contingency(contingency_table)

# Displaying the results
print("Chi-Square Test of Independence:")
print("Chi-square statistic:", chi2_statistic)
print("p-value:", p_value)
print()
```

```
alpha = 0.05
if p_value < alpha:
    print("The p-value is less than the significance level of", alpha)
    print("There is a statistically significant association between the sala
else:
    print("The p-value is greater than or equal to the significance level of
    print("There is no statistically significant association between the sal
```

```
Chi-Square Test of Independence:
Chi-square statistic: 175.21071935727466
p-value: 8.984123357404531e-39

The p-value is less than the significance level of 0.05
There is a statistically significant association between the salary level of
employees and the likelihood of them leaving the company.
```

**The chi-square test; Definiton and Usage**

The chi-square test is a statistical test used to determine if there is a significant association between two categorical variables. It assesses whether the observed frequencies of the categories in the data differ significantly from the expected frequencies under the assumption of independence.

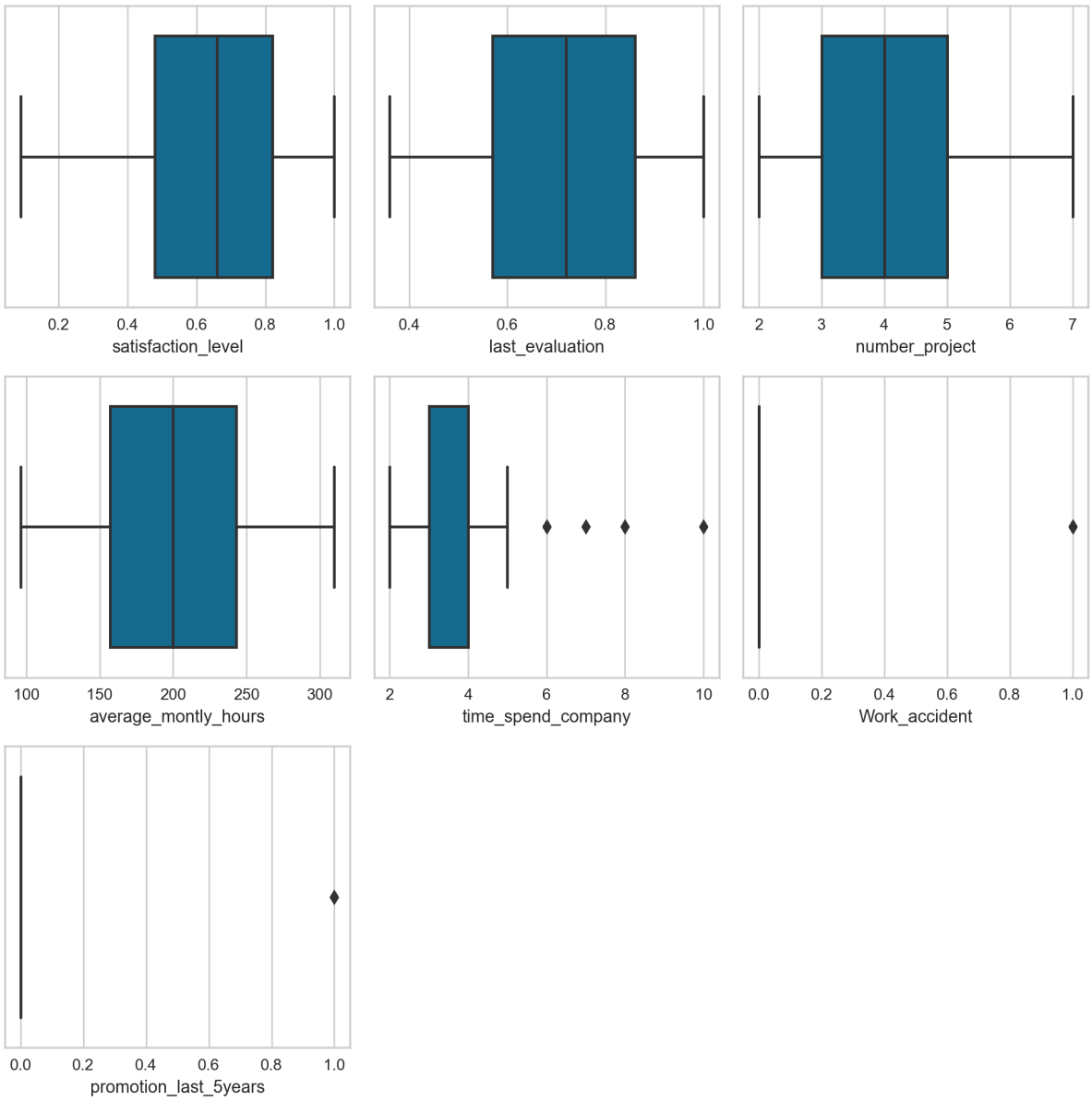
The test calculates a test statistic, called chi-square ($\chi^2$), which measures the discrepancy between the observed and expected frequencies. The expected frequencies are calculated based on the assumption of independence between the variables. The chi-square statistic follows a chi-square distribution, and the test produces a p-value that indicates the probability of obtaining the observed data if the variables are independent.

If the p-value from the chi-square test is below a predetermined significance level (e.g., 0.05), it suggests that there is evidence to reject the null hypothesis of independence. In other words, it indicates that there is a significant association or relationship between the variables. On the other hand, if the p-value is above the significance level, it implies that there is no strong evidence to reject the null hypothesis, and the variables are considered independent.

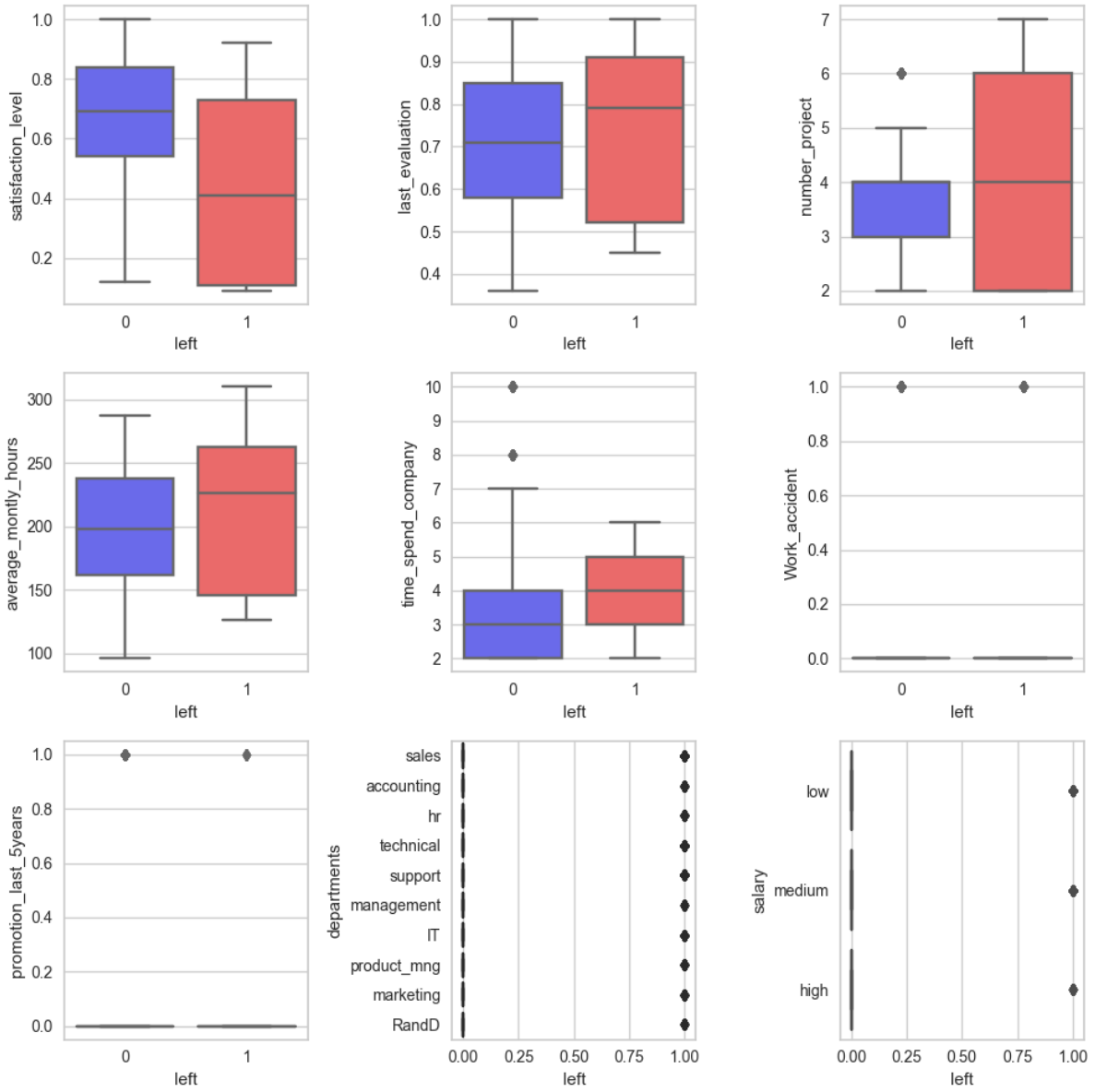
The chi-square test is commonly used in various fields, such as social sciences, epidemiology, market research, and genetics, to analyze categorical data and examine relationships between variables. It is especially useful when dealing with nominal or ordinal data, where the categories are mutually exclusive and do not have a natural ordering.

# Handling with Outliers

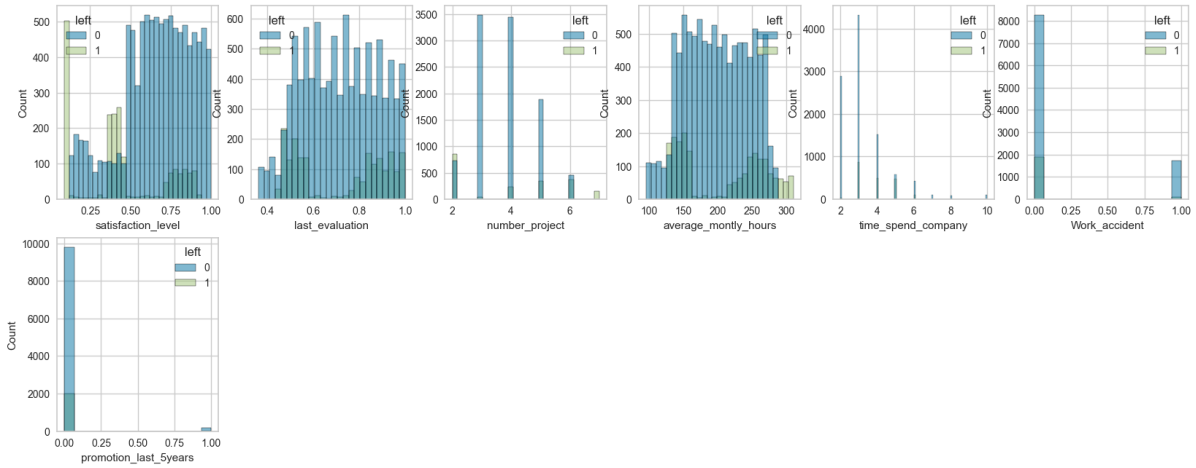
In [ ]:
```
fig = plt.figure(figsize=(10,10), dpi=200)

for i, col in enumerate(df.select_dtypes('number').columns[:-1]):
        plt.subplot(3,3,i+1)
        ax = sns.boxplot(x=df[col])

plt.tight_layout();
```




In [ ]:
```
fig = plt.figure(figsize=(10,10))

for i, col in enumerate(df.columns[:-1]):
        plt.subplot(3,3,i+1)
        ax = sns.boxplot(x="left", y=df[col], data=df, palette='seismic')

plt.tight_layout();
```




In [ ]:
```
# a general overview of all numerical features for each class
plt.figure(figsize=(20, 20))
for idx, col in enumerate(df.select_dtypes(include='number').columns[:-1]):
    plt.subplot(5, 6, idx+1)
    sns.histplot(data=df, x=col, hue='left')
```




In [ ]:
```
# Assuming outliers are values that are more than 1.5 times the IQR away frc

def count_outliers(data):
  columns = data.columns
  # Initialize an empty dictionary to store the results
```

```
  results = {}
  # Loop through each column name in the list
  for col in columns:
    # Calculate the first and third quartile of the column
    q1 = data[col].quantile(0.25)
    q3 = data[col].quantile(0.75)
    # Calculate the IQR
    iqr = q3 - q1
    # Count the number of values that are more than 1.5 times the IQR away f
    outliers = data[col].apply(lambda x: (x < q1 - 1.5 * iqr) or (x > q3 + 1
    # Count the number of values that are below the first quartile
    below_q1 = data[col].apply(lambda x: x < q1 - 1.5 * iqr).sum()
    # Count the number of values that are above the third quartile
    above_q3 = data[col].apply(lambda x: x > q3 + 1.5 * iqr).sum()
    # Store the result in the dictionary with the column name as the key and
    # The tuple contains the number of outliers, the number of values below
    results[col] = (outliers, below_q1, above_q3)
    # Print the result for each column
    print(f"Column {col} has {outliers} outliers, {below_q1} values below Q1
  # Return the dictionary
  return results
```

In [ ]:
```
count_outliers(df_numeric[df_numeric.left == 0])
```

```
Column satisfaction_level has 0 outliers, 0 values below Q1, and 0 values ab
ove Q3.
Column last_evaluation has 0 outliers, 0 values below Q1, and 0 values above
Q3.
Column number_project has 455 outliers, 0 values below Q1, and 455 values ab
ove Q3.
Column average_montly_hours has 0 outliers, 0 values below Q1, and 0 values
above Q3.
Column time_spend_company has 188 outliers, 0 values below Q1, and 188 value
s above Q3.
Column Work_accident has 1745 outliers, 0 values below Q1, and 1745 values a
bove Q3.
Column promotion_last_5years has 195 outliers, 0 values below Q1, and 195 va
lues above Q3.
Column left has 0 outliers, 0 values below Q1, and 0 values above Q3.
```

Out[ ]:
```
{'satisfaction_level': (0, 0, 0),
 'last_evaluation': (0, 0, 0),
 'number_project': (455, 0, 455),
 'average_montly_hours': (0, 0, 0),
 'time_spend_company': (188, 0, 188),
 'Work_accident': (1745, 0, 1745),
 'promotion_last_5years': (195, 0, 195),
 'left': (0, 0, 0)}
```

In [ ]:
```
count_outliers(df_numeric[df_numeric.left == 1])
```

```
Column satisfaction_level has 0 outliers, 0 values below Q1, and 0 values ab
ove Q3.
Column last_evaluation has 0 outliers, 0 values below Q1, and 0 values above
Q3.
Column number_project has 0 outliers, 0 values below Q1, and 0 values above
Q3.
Column average_montly_hours has 0 outliers, 0 values below Q1, and 0 values
above Q3.
Column time_spend_company has 0 outliers, 0 values below Q1, and 0 values ab
ove Q3.
Column Work_accident has 105 outliers, 0 values below Q1, and 105 values abo
ve Q3.
Column promotion_last_5years has 8 outliers, 0 values below Q1, and 8 values
above Q3.
Column left has 0 outliers, 0 values below Q1, and 0 values above Q3.
```

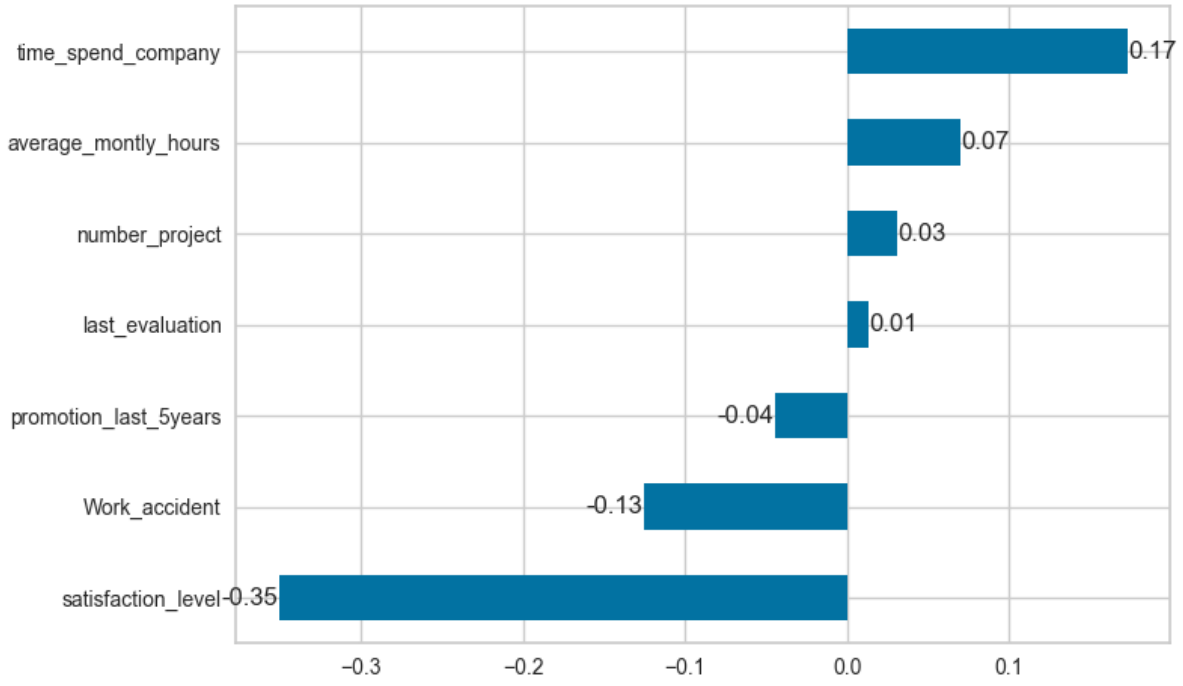
Out[ ]:
```
{'satisfaction_level': (0, 0, 0),
 'last_evaluation': (0, 0, 0),
 'number_project': (0, 0, 0),
 'average_montly_hours': (0, 0, 0),
 'time_spend_company': (0, 0, 0),
 'Work_accident': (105, 0, 105),
 'promotion_last_5years': (8, 0, 8),
 'left': (0, 0, 0)}
```

There is no serious outlier that stands out.

### Check the correlations between the features

In [ ]:
```
plt.figure(figsize=(10,8), dpi=200)
sns.heatmap(df.select_dtypes('number').corr(), annot=True, cmap = 'coolwarm'

# result: no multicolinearity between the features
```


In [ ]:
```
# display the correlation of independent features with target feature "left"
ax = df.select_dtypes('number').corr()["left"].drop("left").sort_values().pl
ax.bar_label(ax.containers[0], fmt="%.2f");

# result: except time_spend__company and satisfaction level, there are no st
```




# 3. Data Preprocessing

In [ ]:
```
df.sample(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend |
|---|---|---|---|---|---|
| **8391** | 0.83 | 0.37 | 5 | 101 | |

In [ ]:
```
X = df.drop("left", axis=1)
y = df["left"]
```

For k-means clustering we will just use numerical features.

In [ ]:
```
X = X.drop(columns=['Work_accident','promotion_last_5years','departments','s
```

In [ ]:
```
X.sample(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend |
|---|---|---|---|---|---|
| **2029** | 0.51 | 0.8 | 3 | 218 | |

In [ ]:
```
X.describe().T
```

Out[ ]:

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **satisfaction_level** | 11991.0 | 0.629658 | 0.241070 | 0.09 | 0.48 | 0.66 | 0.82 | 1.0 |
| **last_evaluation** | 11991.0 | 0.716683 | 0.168343 | 0.36 | 0.57 | 0.72 | 0.86 | 1.0 |
| **number_project** | 11991.0 | 3.802852 | 1.163238 | 2.00 | 3.00 | 4.00 | 5.00 | 7.0 |
| **average_montly_hours** | 11991.0 | 200.473522 | 48.727813 | 96.00 | 157.00 | 200.00 | 243.00 | 310.0 |
| **time_spend_company** | 11991.0 | 3.364857 | 1.330240 | 2.00 | 3.00 | 3.00 | 4.00 | 10.0 |

We should do the scaling for clustering since the features are not on the same range.

# 4. Cluster Analysis

## 4.1 K-means Clustering

In [ ]:
```
from sklearn.neighbors import BallTree


def hopkins(data_frame, sampling_size):
    """Assess the clusterability of a dataset. A score between 0 and 1, a sc
    no clusterability and a score tending to 0 express a high cluster tenden
    Parameters
    ----------
    data_frame : numpy array
        The input dataset
    sampling_size : int
        The sampling size which is used to evaluate the number of DataFrame.
    Returns
    ---------------------
    score : float
```

```
        The hopkins score of the dataset (between 0 and 1)
    """
    if type(data_frame) == np.ndarray:
        data_frame = pd.DataFrame(data_frame)
    # Sample n observations from D : P
    if sampling_size > data_frame.shape[0]:
        raise Exception(
            'The number of sample of sample is bigger than the shape of D')
    data_frame_sample = data_frame.sample(n=sampling_size)
    # Get the distance to their neirest neighbors in D : X
    tree = BallTree(data_frame, leaf_size=2)
    dist, _ = tree.query(data_frame_sample, k=2)
    data_frame_sample_distances_to_nearest_neighbours = dist[:, 1]
    # Randomly simulate n points with the same variation as in D : Q.
    max_data_frame = data_frame.max()
    min_data_frame = data_frame.min()
    uniformly_selected_values_0 = np.random.uniform(min_data_frame[0], max_d
    uniformly_selected_values_1 = np.random.uniform(min_data_frame[1], max_d
    uniformly_selected_observations = np.column_stack((uniformly_selected_va
    if len(max_data_frame) >= 2:
        for i in range(2, len(max_data_frame)):
            uniformly_selected_values_i = np.random.uniform(min_data_frame[i
            to_stack = (uniformly_selected_observations, uniformly_selected_
            uniformly_selected_observations = np.column_stack(to_stack)
    uniformly_selected_observations_df = pd.DataFrame(uniformly_selected_obs
    # Get the distance to their neirest neighbors in D : Y
    tree = BallTree(data_frame, leaf_size=2)
    dist, _ = tree.query(uniformly_selected_observations_df, k=1)
    uniformly_df_distances_to_nearest_neighbours = dist
    # return the hopkins score
    x = sum(data_frame_sample_distances_to_nearest_neighbours)
    y = sum(uniformly_df_distances_to_nearest_neighbours)
    if x + y == 0:
        raise Exception('The denominator of the hopkins statistics is null')
    return x / (x + y)[0]
```

In [ ]:
```
# import the scalers and try both of them to find the one with higher hopkin
from sklearn.preprocessing import StandardScaler, MinMaxScaler, PowerTransfo
```

In [ ]:
```
# hopkins without scaler
hopkins(X, X.shape[0])
```

Out[ ]:
```
0.17313565474327108
```

In [ ]:
```
# hopkins with Standard scaler
hopkins(StandardScaler().fit_transform(X), X.shape[0])
```

Out[ ]:
```
0.22312775537300156
```

In [ ]:
```
hopkins(PowerTransformer().fit_transform(X), X.shape[0])
```

Out[ ]:
```
0.2657030015179773
```

In [ ]:
```
hopkins(MinMaxScaler().fit_transform(X), X.shape[0])
# Compared to standard scaler, MinMaxScaler has a better (lower) score and t
```

Out[ ]:
```
0.23920100263385694
```

In [ ]:
```
# check the skewness of the features and perform lof transformation on an un
# feature to see  if it improves the hopkins score or not
X.skew()
```

Out[ ]:
```
satisfaction_level      -0.538702
last_evaluation         -0.031686
number_project           0.332152
average_montly_hours     0.027208
time_spend_company       1.816577
dtype: float64
```

In [ ]:
```
X2 = X.copy()
```

In [ ]:
```
# promotion_last_years or departments features are categorical features. Tak
X2['time_spend_company'] = X2.time_spend_company.apply(np.log)
```

In [ ]:
```
hopkins(MinMaxScaler().fit_transform(X2), X2.shape[0])
```

Out[ ]:
```
0.25839914733800173
```

In [ ]:
```
hopkins(PowerTransformer().fit_transform(X2), X2.shape[0])
```

Out[ ]:
```
0.26277773411636823
```

In [ ]:
```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

There is no need to do log transformation. MinMaxScaler is enough in this case.

In [ ]:
```
# transform into a dataframe
X_scaled = pd.DataFrame(X_scaled, columns = X.columns)
X_scaled.head(3)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_co |
|---|---|---|---|---|---|
| **0** | 0.318681 | 0.265625 | 0.0 | 0.285047 | |
| **1** | 0.780220 | 0.781250 | 0.6 | 0.775701 | |
| **2** | 0.021978 | 0.812500 | 1.0 | 0.822430 | |

In [ ]:
```
X_scaled.agg(["mean", "std"]).round()
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend |
|---|---|---|---|---|---|
| **mean** | 1.0 | 1.0 | 0.0 | 0.0 | |
| **std** | 0.0 | 0.0 | 0.0 | 0.0 | |

In [ ]:
```
```

## Choosing The Optimal Number of Clusters

## UMAP Embedding

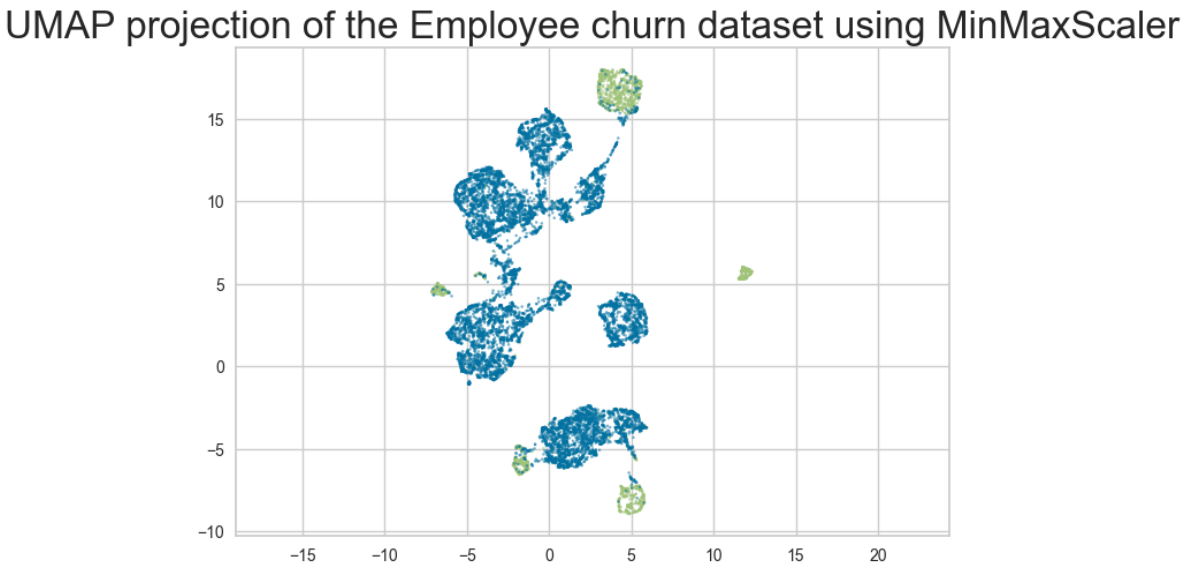
In [ ]:
```
import umap.umap_ as umap

reducer = umap.UMAP()
```

In [ ]:
```
X_scaled_minmax = MinMaxScaler().fit_transform(X)
```

In [ ]:
```
embedding = reducer.fit_transform(X_scaled_minmax)
embedding.shape
```

Out[ ]:
```
(11991, 2)
```

In [ ]:
```
plt.scatter(
    embedding[:, 0],
    embedding[:, 1],
    c=[sns.color_palette()[x] for x in y],
    alpha =0.5,
    s=2)
plt.gca().set_aspect('equal', 'datalim')
plt.title('UMAP projection of the Employee churn dataset using MinMaxScaler'
```




In [ ]:
```
# Preprocessing numerical using PowerTransformer
numerical = X.copy()

for c in numerical.columns:
    pt = PowerTransformer()
    numerical.loc[:, c] = pt.fit_transform(np.array(numerical[c]).reshape(-1
```

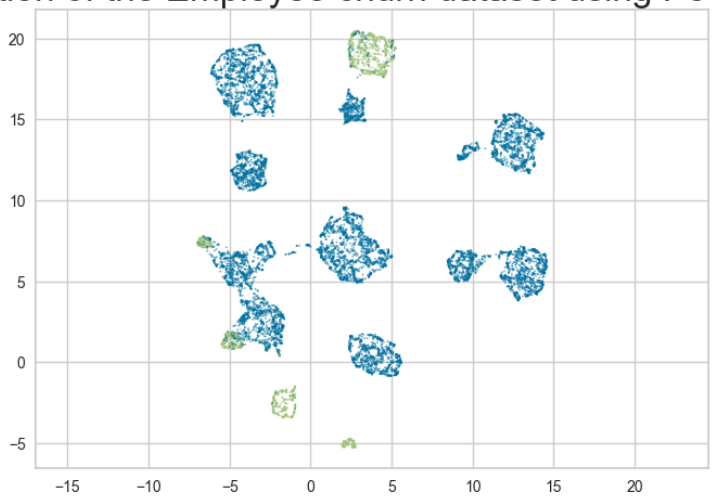
In [ ]:
```
reducer = umap.UMAP()
```

In [ ]:
```
embedding = reducer.fit_transform(numerical)
embedding.shape
```

Out[ ]:
```
(11991, 2)
```

In [ ]:
```
plt.scatter(
    embedding[:, 0],
    embedding[:, 1],
    c=[sns.color_palette()[x] for x in y], alpha=0.5, s=1)
plt.gca().set_aspect('equal', 'datalim')
plt.title('UMAP projection of the Employee churn dataset using PowerTransfor
```

UMAP projection of the Employee churn dataset using PowerTransformer



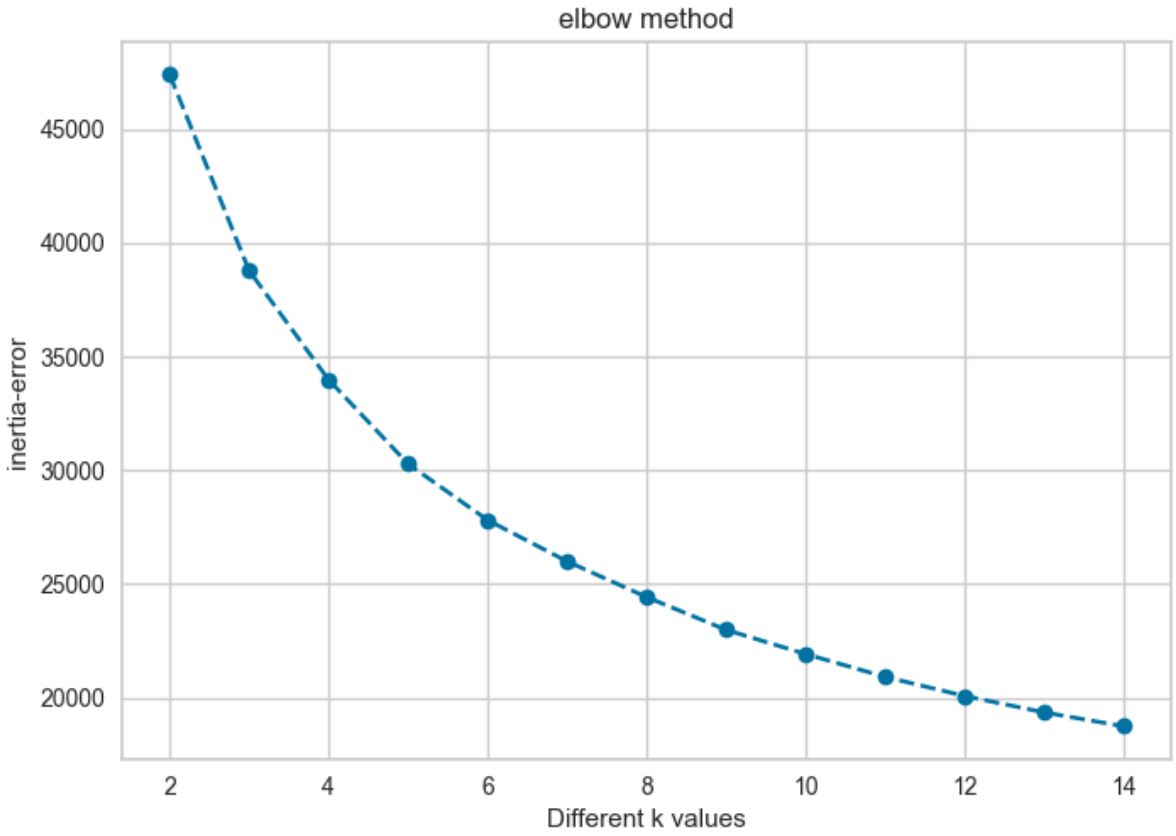

## Elbow metod

In [ ]:
```
from sklearn.cluster import KMeans
```

In [ ]:
```
ssd = []

K = range(2,15)

for k in K:
    model = KMeans(n_clusters =k,
                   random_state=42)
    model.fit(numerical)
    ssd.append(model.inertia_)
```

In [ ]:
```
plt.plot(K, ssd, "bo--")
plt.xlabel("Different k values")
plt.ylabel("inertia-error")
plt.title("elbow method")
```

Out[ ]:
```
Text(0.5, 1.0, 'elbow method')
```



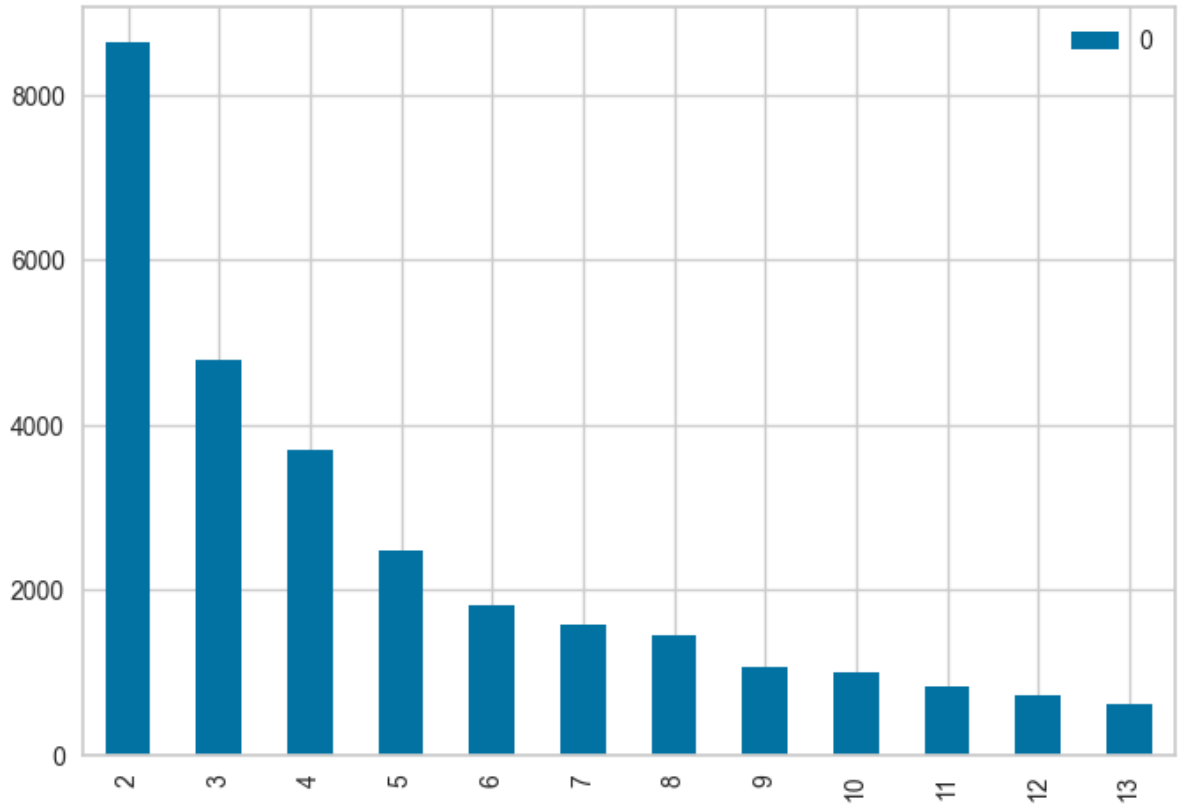

In [ ]:
```
df_diff =pd.DataFrame(-pd.Series(ssd).diff()).rename(index = lambda x : x+1)
df_diff
```

Out[ ]:

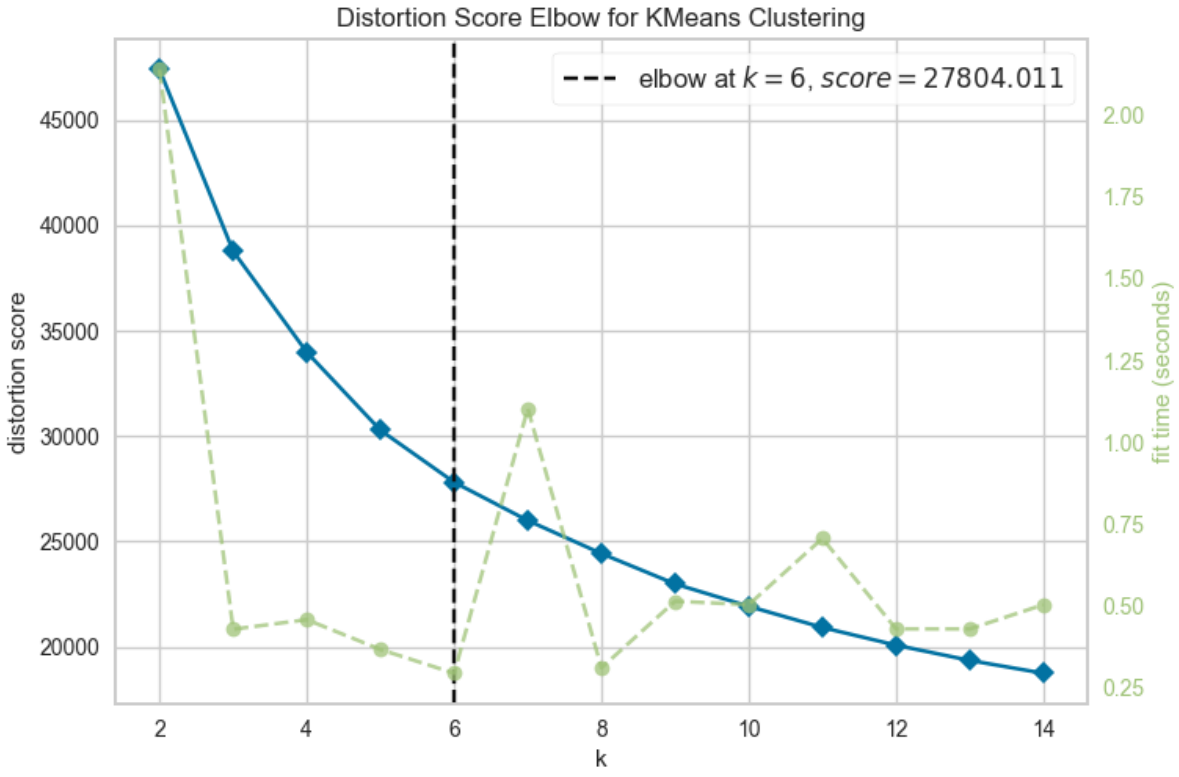
| | 0 |
|---|---|
| 1 | NaN |
| 2 | 8622.068401 |
| 3 | 4793.551963 |
| 4 | 3701.374950 |
| 5 | 2467.413891 |
| 6 | 1816.585323 |
| 7 | 1568.943355 |
| 8 | 1440.998081 |
| 9 | 1062.422479 |
| 10 | 1005.266122 |
| 11 | 839.560118 |
| 12 | 721.888640 |
| 13 | 610.754807 |

In [ ]:
```
df_diff[1:].plot(kind='bar')
```

Out[ ]:
```
<Axes: >
```




In [ ]:
```
from yellowbrick.cluster import KElbowVisualizer

model_ = KMeans(random_state=42)
visualizer = KElbowVisualizer(model_, k=(2,15))

visualizer.fit(numerical)        # Fit the data to the visualizer
visualizer.poof();
```



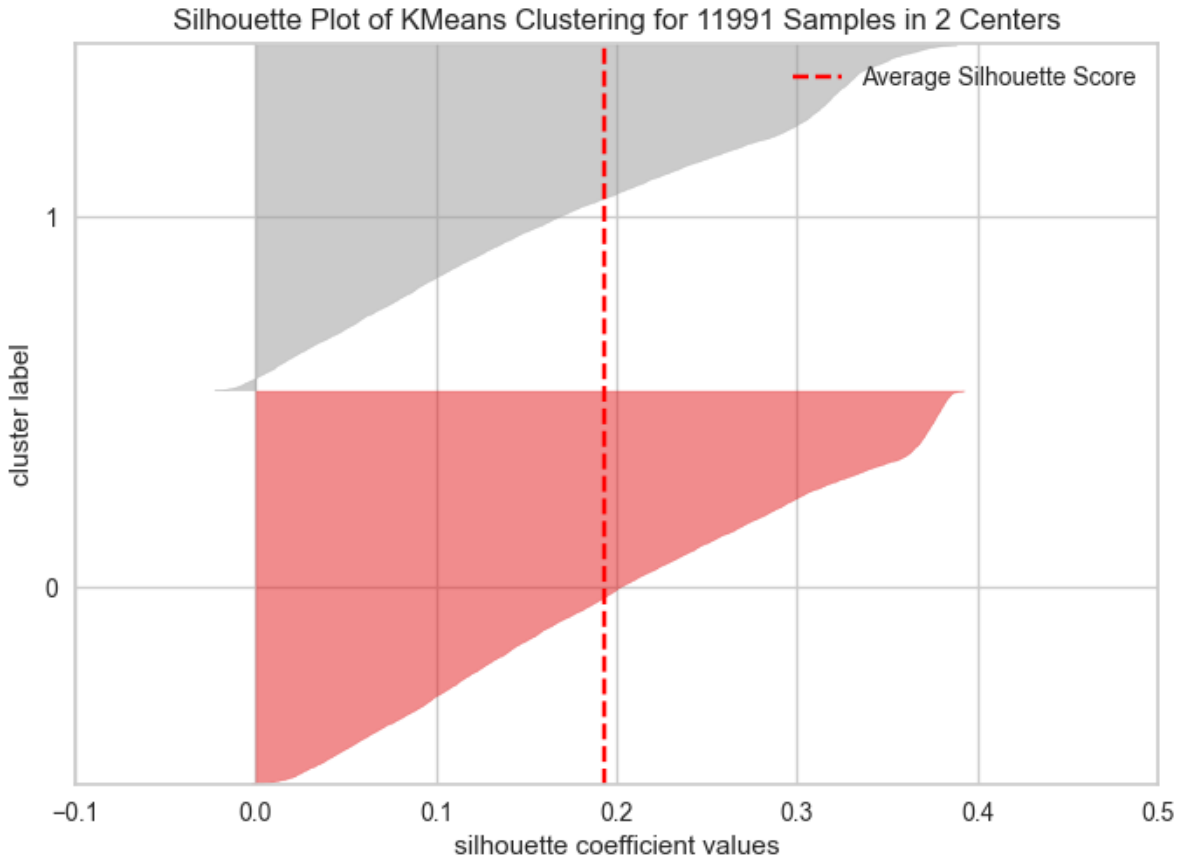

## Silhouette Score

In [ ]:
```
# The distance between clusters
```

In [ ]:
```
from sklearn.metrics import silhouette_score
range_n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12,13,14,15]

for num_clusters in range_n_clusters:


    kmeans = KMeans(n_clusters=num_clusters, max_iter=50)
    kmeans.fit(numerical)

    cluster_labels = kmeans.labels_


    silhouette_avg = silhouette_score(numerical, cluster_labels)
    print("For n_clusters={0}, the silhouette score is {1}".format(num_clust
```

```
For n_clusters=2, the silhouette score is 0.1930900947906741
For n_clusters=3, the silhouette score is 0.20981070095155172
For n_clusters=4, the silhouette score is 0.22076850920671193
For n_clusters=5, the silhouette score is 0.22550516485068975
For n_clusters=6, the silhouette score is 0.2252396142826553
For n_clusters=7, the silhouette score is 0.22606033578347426
For n_clusters=8, the silhouette score is 0.2307998702959853
For n_clusters=9, the silhouette score is 0.23443904640703062
For n_clusters=10, the silhouette score is 0.23405190419148086
For n_clusters=11, the silhouette score is 0.23382672217489153
For n_clusters=12, the silhouette score is 0.23498142980765527
For n_clusters=13, the silhouette score is 0.2337966354691385
For n_clusters=14, the silhouette score is 0.2332403983599534
For n_clusters=15, the silhouette score is 0.23521371392607054
```

In [ ]:
```
from sklearn.cluster import KMeans

from yellowbrick.cluster import SilhouetteVisualizer

model2 = KMeans(n_clusters=2,
                random_state=42)          # n_clusters=2'e karar verdik !
visualizer = SilhouetteVisualizer(model2)

visualizer.fit(numerical)    # Fit the data to the visualizer
visualizer.poof();
```




In [ ]:
```
for i in range(2):
    label = (model2.labels_== i)
    print(f"mean silhouette score for label {i:<4} : {visualizer.silhouette_
print(f"mean silhouette score for all labels : {visualizer.silhouette_score_
```

```
mean silhouette score for label 0    : 0.2086933606122504
mean silhouette score for label 1    : 0.17635528864549813
mean silhouette score for all labels : 0.1935693586449794
```

In [ ]:
```
# since we know the number of actual clusters (2) in this data, we will buil

kmeans = KMeans(n_clusters = 2).fit(numerical)
```

In [ ]:
```
# get the labels
labels = kmeans.labels_
```

In [ ]:
```
pd.DataFrame(labels).value_counts()
```

Out[ ]:
```
0    6226
1    5765
Name: count, dtype: int64
```

In [ ]:
```
y.value_counts()
```

Out[ ]:
```
left
0    10000
1     1991
Name: count, dtype: int64
```

In [ ]:
```
# display actual and predicted clusters in the same dataframe
df_compare = pd.concat([y, pd.Series(labels,name='kmeans_label')], axis=1)
```

In [ ]:
```
df_compare
```

Out[ ]:

| | left | kmeans_label |
|---|---|---|
| **0** | 1 | 0 |
| **1** | 1 | 1 |
| **2** | 1 | 1 |
| **3** | 1 | 1 |
| **4** | 1 | 0 |
| **...** | ... | ... |
| **11986** | 0 | 1 |
| **11987** | 0 | 1 |
| **11988** | 0 | 0 |
| **11989** | 0 | 1 |
| **11990** | 0 | 0 |

11991 rows × 2 columns

In [ ]:
```
# 6445 out of 11991 are correctly predicted
df_compare[df_compare.left == df_compare.kmeans_label]
```

Out[ ]:

| | left | kmeans_label |
|---|---|---|
| **1** | 1 | 1 |
| **2** | 1 | 1 |
| **3** | 1 | 1 |
| **6** | 1 | 1 |
| **7** | 1 | 1 |
| **...** | ... | ... |
| **11977** | 0 | 0 |
| **11979** | 0 | 0 |
| **11982** | 0 | 0 |
| **11988** | 0 | 0 |
| **11990** | 0 | 0 |

6445 rows × 2 columns

In [ ]:
```
# 5546 out of 11991 are incorrectly predicted
df_compare[df_compare.left != df_compare.kmeans_label]
```

Out[ ]:

| | left | kmeans_label |
|---|---|---|
| **0** | 1 | 0 |
| **4** | 1 | 0 |
| **5** | 1 | 0 |
| **9** | 1 | 0 |
| **10** | 1 | 0 |
| **...** | ... | ... |
| **11984** | 0 | 1 |
| **11985** | 0 | 1 |
| **11986** | 0 | 1 |
| **11987** | 0 | 1 |
| **11989** | 0 | 1 |

5546 rows × 2 columns

In [ ]:
```
confusion_matrix(y,labels)
```

Out[ ]:
```
array([[5340, 4660],
       [ 886, 1105]])
```

In [ ]:
```
df_cluster = pd.concat([numerical,pd.Series(labels,name='predicted_cluster')
```

In [ ]:
```
df_cluster
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spen |
|---|---|---|---|---|---|
| **0** | -1.078828 | -1.109354 | -1.711465 | -0.882014 | |
| **1** | 0.683566 | 0.851849 | 1.030694 | 1.242418 | |
| **2** | -1.852951 | 0.970190 | 2.363373 | 1.431579 | |
| **3** | 0.291431 | 0.911026 | 1.030694 | 0.485518 | |
| **4** | -1.112227 | -1.169056 | -1.711465 | -0.838591 | |
| **...** | ... | ... | ... | ... | |
| **11986** | 1.213244 | -0.990000 | -0.647892 | 1.185301 | |
| **11987** | 0.386874 | 1.383953 | 1.030694 | 1.318306 | |
| **11988** | 0.942845 | -1.049669 | -0.647892 | -0.286355 | |
| **11989** | -1.242072 | -0.394230 | -0.647892 | -0.559731 | |
| **11990** | -0.648111 | 0.081242 | 0.248445 | -0.390883 | |

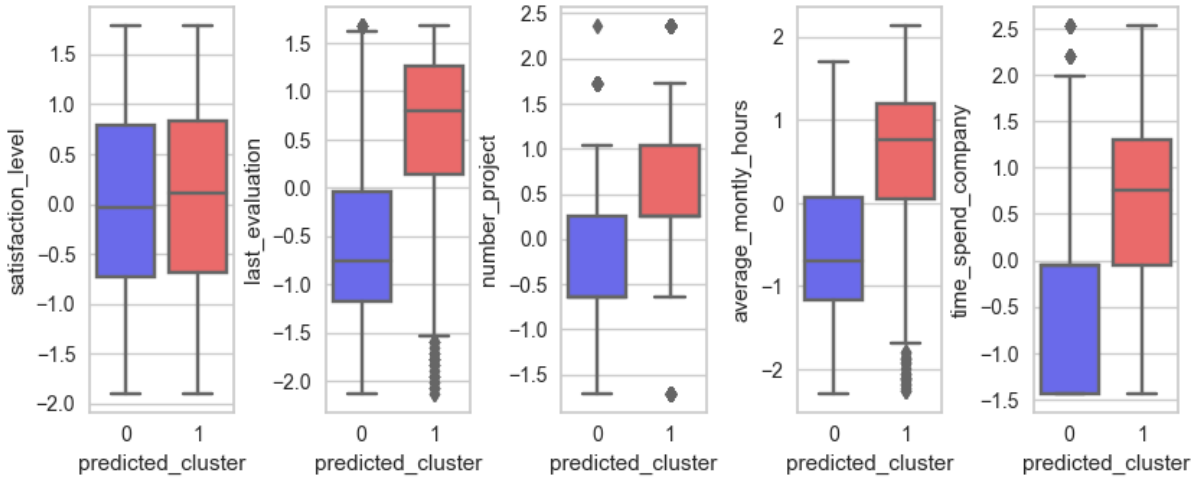
11991 rows × 6 columns

In [ ]:
```
df_cluster.groupby('predicted_cluster').agg({np.mean,np.std})
```

Out[ ]:

| | satisfaction_level | | last_evaluation | | number_project | | average_ |
|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | mean | std | mea |
| predicted_cluster | | | | | | | |
| 0 | 0.030426 | 0.912191 | -0.567414 | 0.817997 | -0.490054 | 0.883079 | -0.52537 |
| 1 | -0.032859 | 1.086096 | 0.612788 | 0.796463 | 0.529242 | 0.835812 | 0.56738 |

In [ ]:
```
fig = plt.figure(figsize=(10,10))

for i, col in enumerate(df_cluster.columns[:-1]):
        plt.subplot(3,6,i+1)
        ax = sns.boxplot(x="predicted_cluster", y=df_cluster[col], data=df_c

plt.tight_layout();
```




## ARI Score

- The Adjusted Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.
- The value of ARI indicates no good clustering if it is close to zero or negative, and a good cluster if it is close to 1.

In [ ]:
```
from sklearn.metrics.cluster import adjusted_rand_score
```

In [ ]:
```
adjusted_rand_score(y, labels)
```

Out[ ]:
```
0.004918896518542001
```

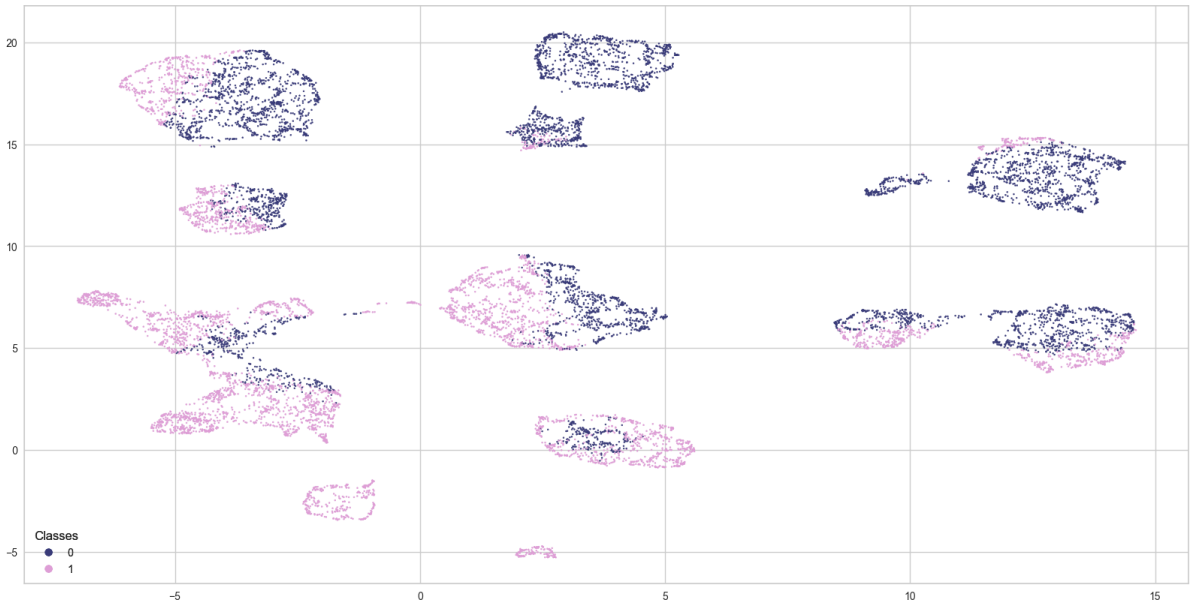
A Adjusted Rand Score (ARS) of 0 indicates a very low similarity between the clustering results and the ground truth labels (if available). This score suggests that the clustering algorithm, in this case, k-means, did not capture the underlying structure of the data well or did not align with the true cluster assignments.

A score close to zero suggests that the clustering is essentially random or provides results that are not significantly better than random chance. It implies that the clusters formed by the algorithm do not correspond well to the actual clusters or groups in the data.

In practical terms, such a low ARS score may indicate that the chosen number of clusters, the clustering algorithm itself, or the data itself may not be suitable for the clustering task.

In [ ]:
```
# plt.scatter(
#     embedding[:, 0],
#     embedding[:, 1],
#     c=[sns.color_palette()[x] for x in labels], alpha=0.5, s=1)
# plt.gca().set_aspect('equal', 'datalim')
# plt.title('UMAP projection of the Employee churn dataset using PowerTransf

fig, ax = plt.subplots()
fig.set_size_inches((20, 10))
scatter = ax.scatter(embedding[:, 0], embedding[:, 1], s=2, c=labels, cmap='

# produce a legend with the unique colors from the scatter
legend1 = ax.legend(*scatter.legend_elements(num=1),
                    loc="lower left", title="Classes")
ax.add_artist(legend1)
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fa04d1c8b20>
```




In [ ]:
```
# lgbm data
lgbm_data = numerical.copy()
```

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

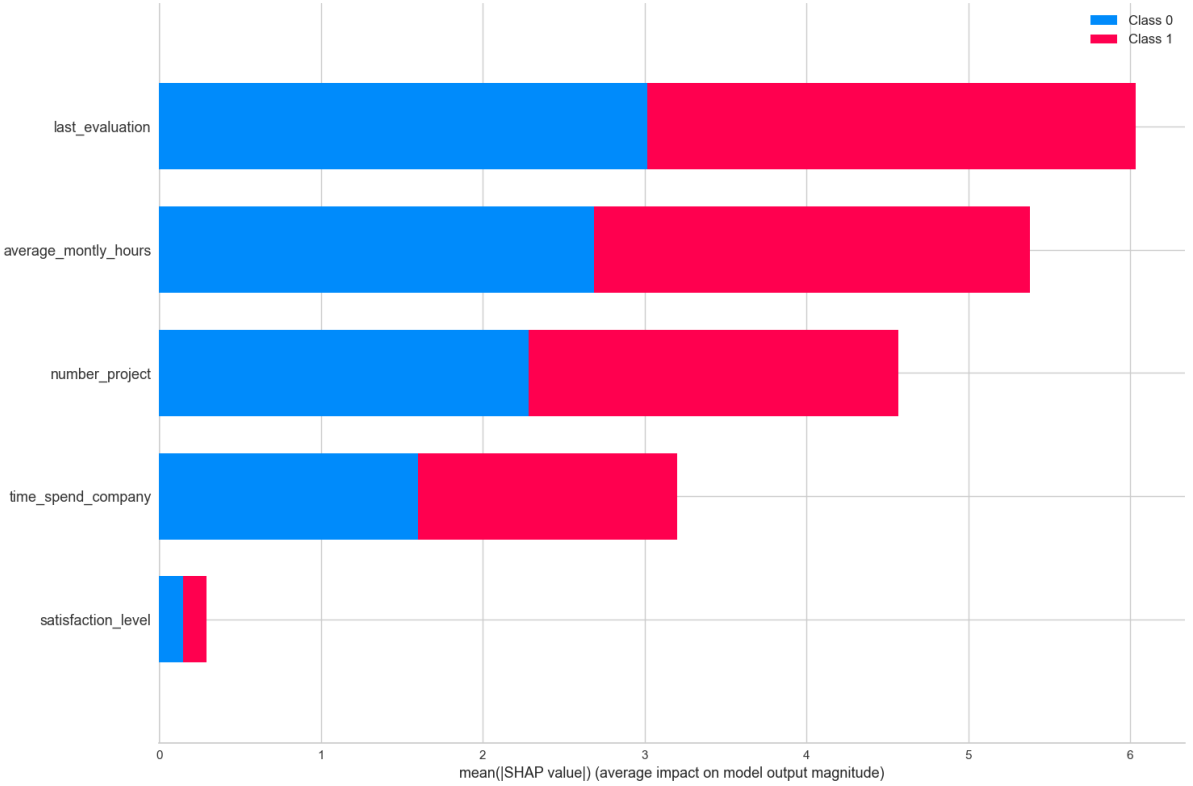
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11991 entries, 0 to 11990
Data columns (total 5 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   satisfaction_level    11991 non-null  float64
 1   last_evaluation       11991 non-null  float64
 2   number_project        11991 non-null  float64
 3   average_montly_hours  11991 non-null  float64
 4   time_spend_company    11991 non-null  float64
dtypes: float64(5)
memory usage: 468.5 KB
```

In [ ]:
```
from lightgbm import LGBMClassifier
from sklearn.model_selection import cross_val_score
```

```
clf_kp = LGBMClassifier(colsample_bytree=0.8)
cv_scores_kp = cross_val_score(clf_kp, lgbm_data, labels, scoring='f1_weight
print(f'CV F1 score for K-Prototypes clusters is {np.mean(cv_scores_kp)}')
```

```
CV F1 score for K-Prototypes clusters is 0.9802304459447093
```

In [ ]:
```
import shap
clf_kp.fit(lgbm_data, labels)
explainer_kp = shap.TreeExplainer(clf_kp)
shap_values_kp = explainer_kp.shap_values(lgbm_data)
shap.summary_plot(shap_values_kp, lgbm_data, plot_type="bar", plot_size=(15,
```




## 4.2 K-Prototypes Clustering

K-prototypes extends K-means clustering to handle datasets with both numeric and categorical features. It does this by defining a distance measure for numeric attributes based on Euclidean distance, and another measure for categorical attributes based on simple matching. It then alternates between assigning data points to clusters and updating numeric prototypes (cluster means for numeric attributes) and categorical prototypes (most frequent category for categorical attributes). This allows K-prototypes to cluster mixed-type datasets, making it useful for real-world problems.

In [ ]:
```
X = df.drop("left", axis=1)
y = df["left"]
```

In [ ]:
```
X.sample(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spen |
|---|---|---|---|---|---|
| **11558** | 0.61 | 0.82 | 4 | 184 | |

### UMAP Embedding

https://antonsruberts.github.io/kproto-audience/

UMAP (Uniform Manifold Approximation and Projection) is a dimensionality reduction technique that works by projecting high-dimensional data into a lower-dimensional space while preserving the geometric structure and relationships between data points. UMAP uses a graph-based approach to learn the underlying structure of the data and can handle non-linear and high-dimensional data.

Compared to PCA, UMAP is better suited for non-linearly correlated data with complex relationships between variables and can capture more complex and non-linear relationships between data points. UMAP is also more robust to outliers and can handle missing values, while PCA is sensitive to outliers and requires complete data.

Overall, both techniques have their strengths and weaknesses, and the choice between them depends on the characteristics of the data and the goals of the analysis.

We embed the data into 2 dimensions to see the groups of customers in plot. By doing by:

- Yeo-Johnson transform the numerical columns & One-Hot-Encode the categorical data
- Embed numerical and categorical columns seperately
- Combine the two embeddings (https://github.com/lmcinnes/umap/issues/58#issuecomment-419682509)

In [ ]:
```
from sklearn.preprocessing import PowerTransformer

# Preprocessing numerical
numerical = X.select_dtypes(exclude='object').drop(columns = ['Work_accident

for c in numerical.columns:
    pt = PowerTransformer()
    numerical.loc[:, c] = pt.fit_transform(np.array(numerical[c]).reshape(-1
```

In [ ]:
```
num_cols = numerical.columns
num_cols
```

Out[ ]:
```
Index(['satisfaction_level', 'last_evaluation', 'number_project',
       'average_montly_hours', 'time_spend_company'],
      dtype='object')
```

In [ ]:
```
# preprocessing categorical

cat_cols = list(set(X.columns) - set(numerical.columns))
categorical = X[cat_cols]
categorical['Work_accident'] = categorical['Work_accident'].map({0:'false',1
categorical['promotion_last_5years'] = categorical['promotion_last_5years'].
categorical = pd.get_dummies(categorical)
categorical.head()
```

Out[ ]:

| | promotion_last_5years_false | promotion_last_5years_true | Work_accident_false | Work_acc |
|---|---|---|---|---|
| **0** | True | False | True | |
| **1** | True | False | True | |
| **2** | True | False | True | |
| **3** | True | False | True | |
| **4** | True | False | True | |

In [ ]:
```
#Percentage of columns which are categorical is used as weight parameter in
num_categorical = len(cat_cols)
categorical_weight = num_categorical / X.shape[1]
```

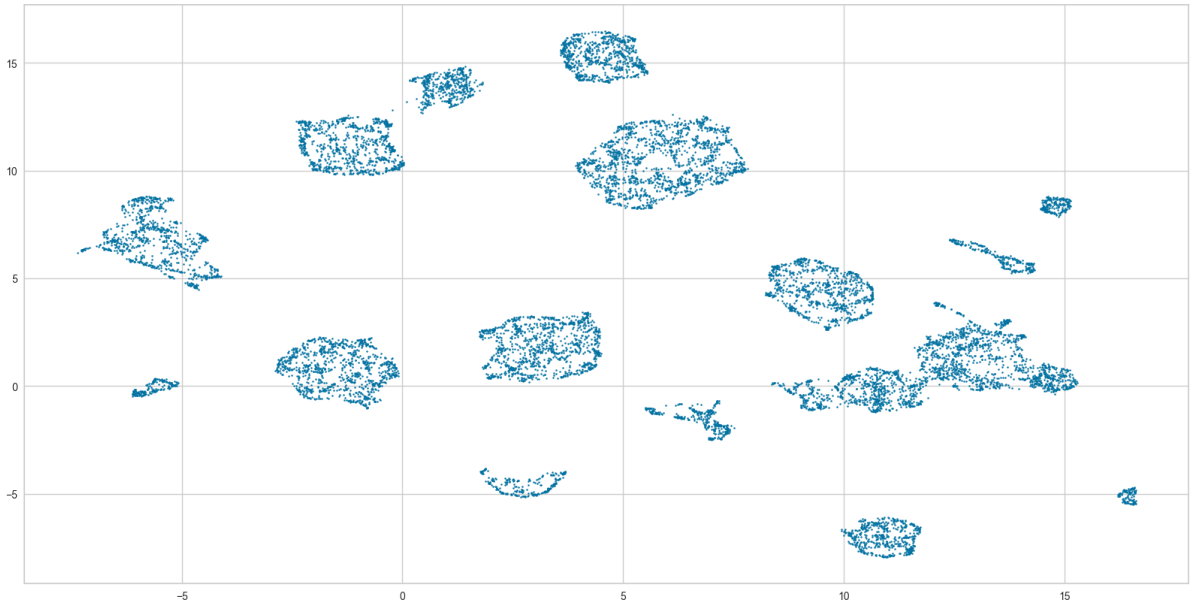
In [ ]:
```
# !pip install umap-learn
```

In [ ]:
```
import umap.umap_ as umap

#Embedding numerical & categorical
fit1 = umap.UMAP(metric='l2').fit(numerical)
fit2 = umap.UMAP(metric='dice').fit(categorical)
```

In [ ]:
```
densmap_kwds = {} # or define actual keyword arguments
output_dens = False
#Augmenting the numerical embedding with categorical
intersection = umap.general_simplicial_set_intersection(fit1.graph_, fit2.gr
intersection = umap.reset_local_connectivity(intersection)
embedding = umap.simplicial_set_embedding(fit1._raw_data, intersection, fit1
                                          fit1._initial_alpha, fit1._a
                                          fit1.repulsion_strength, fit
                                          200, 'random', np.random, fi
                                          fit1._metric_kwds, False,den
```

In [ ]:
```
embedding = embedding[0]
```

In [ ]:
```
# Visullizing the data which has both numerical and categorical values using
# When we count clusters, it seems like there are 16 clusters
# However, we first implement two clusters, then we will try 16 clusters.

plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], s=2, cmap='Spectral', alpha=1.
plt.show()
#
```




In [ ]:
```
X.sample(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_ |
|---|---|---|---|---|---|
| **966** | 0.42 | 0.54 | 2 | 130 | |

In [ ]:
```
cat_cols
```

Out[ ]:
```
['promotion_last_5years', 'Work_accident', 'salary', 'departments']
```

In [ ]:
```
# pip install kmodes
from kmodes.kprototypes import KPrototypes
from tqdm import tqdm
import plotly.graph_objects as go
```

In [ ]:
```
kprot_data = X.copy()
```

In [ ]:
```
#Pre-processing numerical columns
for c in num_cols:
    pt = PowerTransformer()
    kprot_data[c] =  pt.fit_transform(np.array(kprot_data[c]).reshape(-1, 1)

#Pre-processing categorical columns which is numeric
kprot_data['Work_accident'] = kprot_data['Work_accident'].map({0:'false',1:'
kprot_data['promotion_last_5years'] = kprot_data['promotion_last_5years'].ma
```

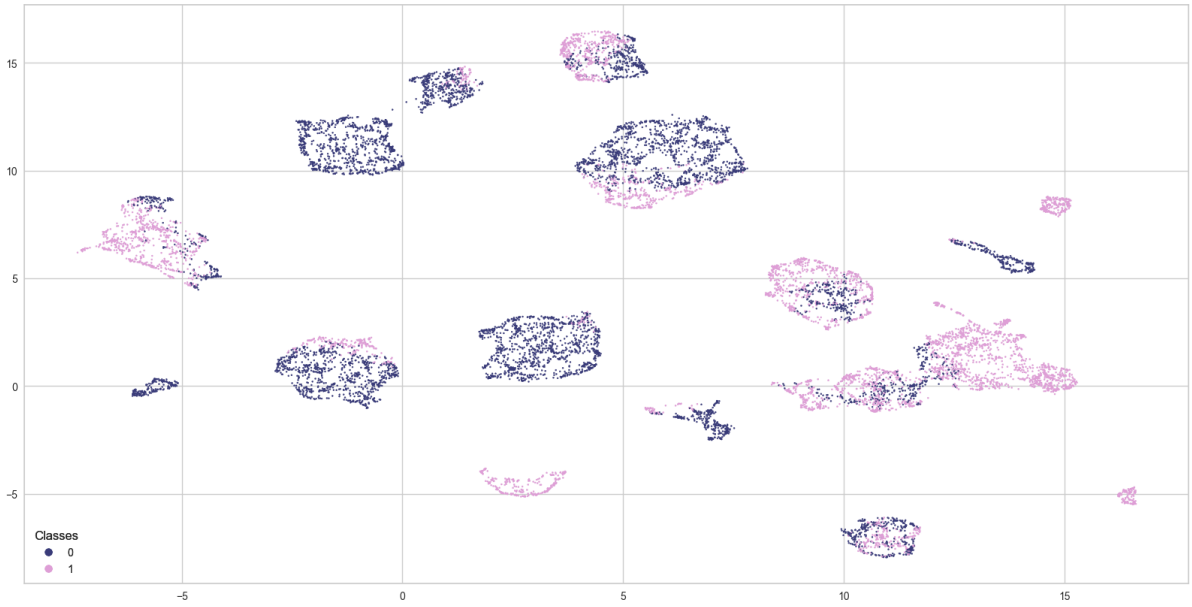
In [ ]:
```
# Kprops - two clusters

categorical_columns = [5,6,7,8] #make sure to specify correct indices

#Actual clustering
kproto = KPrototypes(n_clusters= 2, init='Cao', n_jobs = -1)
clusters = kproto.fit_predict(kprot_data, categorical=categorical_columns)

#Prints the count of each cluster group
pd.Series(clusters).value_counts()

#OPTIONAL: Elbow plot with cost (will take a LONG time)
# costs = []
# n_clusters = []
# clusters_assigned = []
```

```
# for i in tqdm(range(2, 25)):
#     try:
#         kproto = KPrototypes(n_clusters= i, init='Cao', verbose=2)
#         clusters = kproto.fit_predict(kprot_data, categorical=categorical_
#         costs.append(kproto.cost_)
#         n_clusters.append(i)
#         clusters_assigned.append(clusters)
#     except:
#         print(f"Can't cluster with {i} clusters")

# fig = go.Figure(data=go.Scatter(x=n_clusters, y=costs ))
# fig.show()
```

Out[ ]:
```
0    6729
1    5262
Name: count, dtype: int64
```

In [ ]:
```
# Visualizing the data with Kprops predicted clusters(2) using UMAP embeddin

fig, ax = plt.subplots()
fig.set_size_inches((20, 10))
scatter = ax.scatter(embedding[:, 0], embedding[:, 1], s=2, c=clusters, cmap

# produce a legend with the unique colors from the scatter
legend1 = ax.legend(*scatter.legend_elements(num=1),
                    loc="lower left", title="Classes")
ax.add_artist(legend1)
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fa054b5ee00>
```



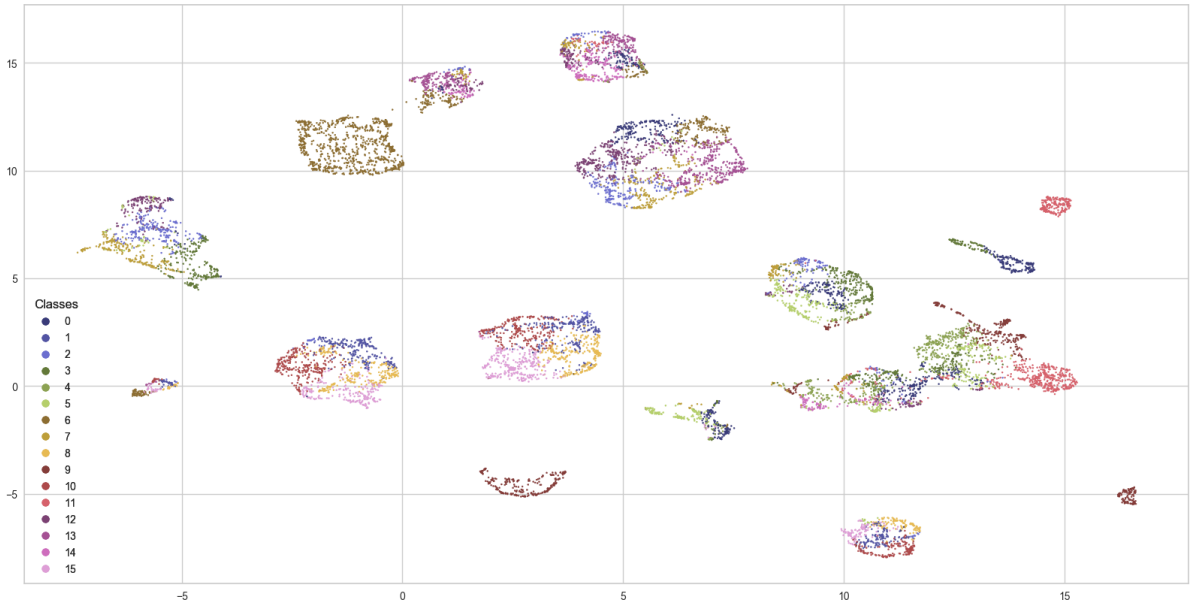

In [ ]:
```
# Kprops 16 clusters

categorical_columns = [5,6,7,8] #make sure to specify correct indices

#Actual clustering
kproto16 = KPrototypes(n_clusters= 16, init='Cao', n_jobs = -1)
clusters16 = kproto16.fit_predict(kprot_data, categorical=categorical_column

#Prints the count of each cluster group
pd.Series(clusters16).value_counts()

#OPTIONAL: Elbow plot with cost (will take a LONG time)
# costs = []
# n_clusters = []
```

```
# clusters_assigned = []

# for i in tqdm(range(2, 25)):
#     try:
#         kproto = KPrototypes(n_clusters= i, init='Cao', verbose=2)
#         clusters = kproto.fit_predict(kprot_data, categorical=categorical_
#         costs.append(kproto.cost_)
#         n_clusters.append(i)
#         clusters_assigned.append(clusters)
#     except:
#         print(f"Can't cluster with {i} clusters")

# fig = go.Figure(data=go.Scatter(x=n_clusters, y=costs ))
# fig.show()
```

Out[ ]:
```
6     1330
0      942
3      838
12     780
11     777
10     753
9      749
7      747
13     740
2      710
15     695
8      671
1      651
5      641
4      556
14     411
Name: count, dtype: int64
```

In [ ]:
```
# Visualizing the data with Kprops predicted clusters(2) using UMAP embeddin

fig, ax = plt.subplots()
fig.set_size_inches((20, 10))
scatter = ax.scatter(embedding[:, 0], embedding[:, 1], s=2, c=clusters16, cm

# produce a legend with the unique colors from the scatter
legend1 = ax.legend(*scatter.legend_elements(num=16),
                    loc="lower left", title="Classes")
ax.add_artist(legend1)
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fa046ed0610>
```



We compare the clusters by treating them as labels and developing a classification model on top of them. A high-quality set of clusters would result in a classification model that can accurately predict them. We then look at the shap explainer whether or not classifier uses the various features or not, that shows the cluster's informativeness .Overall, We evaluate the following aspects:

The distinctiveness of clusters, measured by cross-validated F1 score. The informativeness of clusters, assessed through SHAP feature importances.

In [ ]:
```
#Setting the objects to category
lgbm_data = X.copy()
lgbm_data['Work_accident'] = lgbm_data['Work_accident'].map({0:'false',1:'tr
lgbm_data['promotion_last_5years'] = lgbm_data['promotion_last_5years'].map(
for c in lgbm_data.select_dtypes(include='object'):
    lgbm_data[c] = lgbm_data[c].astype('category')
```

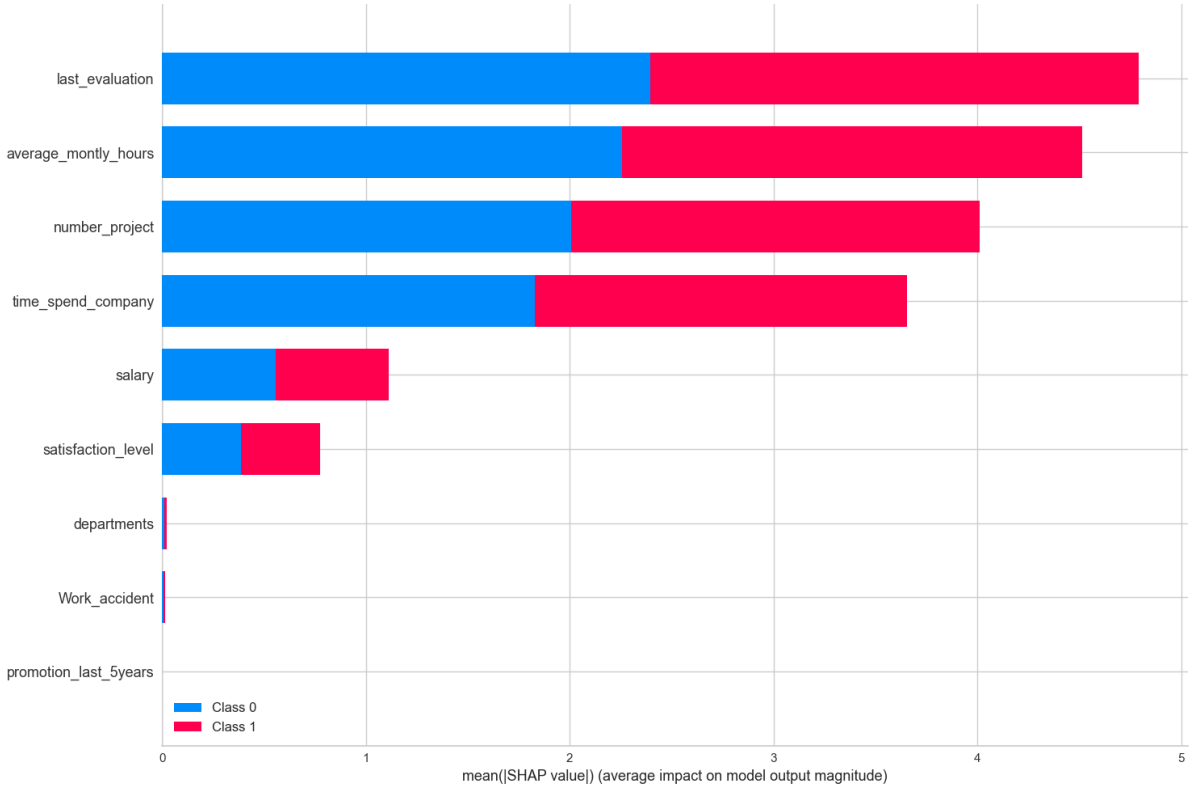
In [ ]:
```
lgbm_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11991 entries, 0 to 11990
Data columns (total 9 columns):
 #   Column                 Non-Null Count  Dtype   
---  ------                 --------------  -----   
 0   satisfaction_level     11991 non-null  float64 
 1   last_evaluation        11991 non-null  float64 
 2   number_project         11991 non-null  int64   
 3   average_montly_hours   11991 non-null  int64   
 4   time_spend_company     11991 non-null  int64   
 5   Work_accident          11991 non-null  category
 6   promotion_last_5years  11991 non-null  category
 7   departments            11991 non-null  category
 8   salary                 11991 non-null  category
dtypes: category(4), float64(2), int64(3)
memory usage: 516.1 KB
```

In [ ]:
```
from lightgbm import LGBMClassifier
from sklearn.model_selection import cross_val_score

clf_kp = LGBMClassifier(colsample_bytree=0.8)
cv_scores_kp = cross_val_score(clf_kp, lgbm_data, clusters, scoring='f1_weig
print(f'CV F1 score for K-Prototypes clusters is {np.mean(cv_scores_kp)}')
```

```
CV F1 score for K-Prototypes clusters is 0.9751321740676036
```

In [ ]:
```
import shap
clf_kp.fit(lgbm_data, clusters)
explainer_kp = shap.TreeExplainer(clf_kp)
shap_values_kp = explainer_kp.shap_values(lgbm_data)
shap.summary_plot(shap_values_kp, lgbm_data, plot_type="bar", plot_size=(15,
```




In [ ]:
```
# Same for 16 clusters
lgbm_data = X.copy()
lgbm_data['Work_accident'] = lgbm_data['Work_accident'].map({0:'false',1:'tr
lgbm_data['promotion_last_5years'] = lgbm_data['promotion_last_5years'].map(
for c in lgbm_data.select_dtypes(include='object'):
    lgbm_data[c] = lgbm_data[c].astype('category')
```

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

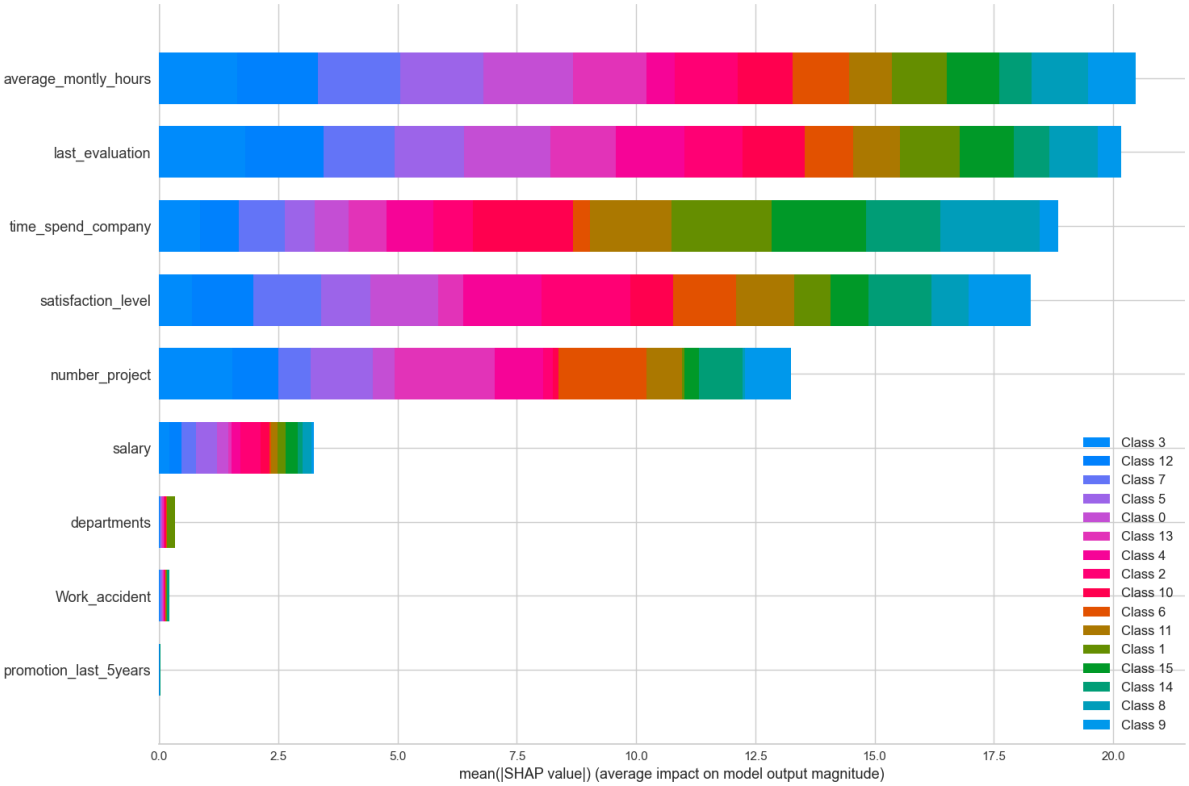
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11991 entries, 0 to 11990
Data columns (total 9 columns):
 #   Column                 Non-Null Count  Dtype   
---  ------                 --------------  -----   
 0   satisfaction_level     11991 non-null  float64 
 1   last_evaluation        11991 non-null  float64 
 2   number_project         11991 non-null  int64   
 3   average_montly_hours   11991 non-null  int64   
 4   time_spend_company     11991 non-null  int64   
 5   Work_accident          11991 non-null  category
 6   promotion_last_5years  11991 non-null  category
 7   departments            11991 non-null  category
 8   salary                 11991 non-null  category
dtypes: category(4), float64(2), int64(3)
memory usage: 516.1 KB
```

In [ ]:
```
from lightgbm import LGBMClassifier
from sklearn.model_selection import cross_val_score

clf_kp = LGBMClassifier(colsample_bytree=0.8)
cv_scores_kp = cross_val_score(clf_kp, lgbm_data, clusters16, scoring='f1_we
print(f'CV F1 score for K-Prototypes clusters is {np.mean(cv_scores_kp)}')
```

```
CV F1 score for K-Prototypes clusters is 0.9379357174549675
```

In [ ]:
```
import shap
clf_kp.fit(lgbm_data, clusters16)
explainer_kp = shap.TreeExplainer(clf_kp)
```

```
shap_values_kp = explainer_kp.shap_values(lgbm_data)
shap.summary_plot(shap_values_kp, lgbm_data, plot_type="bar", plot_size=(15,
```




### ARI Score

In [ ]:
```
adjusted_rand_score(y, clusters)
# Such a low ARS score may indicate that the chosen number of clusters, the
```

Out[ ]:
```
0.018216969522612866
```

Conclusion of the cluster analysis

When we look at the CV F1 score of LGBM model for K-Prototypes clusters, both models' clusters seems distinct according to the high scores (0.93 and 0.97). Those scores are high enough to conclude that both K-Prototypes clusters and Kmeans clusters are meaningful and distinguishable.

Compared to kmeans, kprops clusters are more informative. The classifier of kmeans used 4 features whereas, the classifier of kprops used 6 features including one categorical feature which is salary according to the shap explainers of the classifier models.

However, when we look at the Umap embedding scatter plots colored by predicted clusters and ARI scores, both kmeans and kprops did not achieve good results.

# 5. Predictive Model Building

## 5.1. Preprocessing

In [ ]:
```
from sklearn.model_selection import train_test_split, GridSearchCV, cross_va
from sklearn.metrics import confusion_matrix, accuracy_score, mean_squared_e
```

```
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassif
from sklearn.neighbors import KNeighborsClassifier
```

In [ ]:
```
# check the df
df.head(1)
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_co |
|---|---|---|---|---|---|
| **0** | 0.38 | 0.53 | 2 | 157 | |

In [ ]:
```
# split the data as features and target
X = df.drop("left", axis=1)
y = df["left"]
```

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, s
```

## 5.2 Classification Algorithms

In [ ]:
```
def eval_metric(model, X_train, y_train, X_test, y_test):
    y_train_pred = model.predict(X_train)
    y_pred = model.predict(X_test)

    print("Test_Set")
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    print()
    print("Train_Set")
    print(confusion_matrix(y_train, y_train_pred))
    print(classification_report(y_train, y_train_pred))
```

### 5.2.1 Logistic Regression

Logistic regression is a popular supervised machine learning algorithm used for binary classification tasks. It is a statistical model that predicts the probability of an instance belonging to a particular class.

The main idea behind logistic regression is to model the relationship between the input features (independent variables) and the binary target variable (dependent variable) using a logistic function. The logistic function, also known as the sigmoid function, maps any real-valued number to a value between 0 and 1, representing the probability of the instance belonging to the positive class.

The logistic regression model assumes that the log-odds of the target variable being in the positive class is a linear combination of the input features. The model calculates the weighted sum of the input features, applies the logistic function to it, and produces the probability of the positive class.

During the training phase, logistic regression optimizes the model parameters using optimization techniques like maximum likelihood estimation or gradient descent. The goal is to find the parameter values that maximize the likelihood of the observed data given the model.

Once the model is trained, it can be used to predict the class labels of new instances by comparing the predicted probabilities to a predefined threshold. If the predicted probability is above the threshold, the instance is classified as the positive class; otherwise, it is classified as the negative class.

Logistic regression has several advantages, including simplicity, interpretability, and efficiency. It works well with both linearly separable and non-linearly separable data, and it can handle categorical and continuous input features. However, logistic regression assumes a linear relationship between the features and the log-odds, and it may not perform well in cases where the relationship is highly non-linear.

The main parameters of logistic regression are as follows:

1) **Penalty**: Logistic regression can apply regularization to the model to prevent overfitting. The penalty parameter determines the type and strength of regularization. There are two commonly used penalties: L1 regularization (Lasso) and L2 regularization (Ridge). The choice of penalty depends on the problem and the nature of the data.

2) **C**: The inverse of the regularization strength is represented by the parameter C. A smaller value of C indicates stronger regularization, while a larger value of C indicates weaker regularization. It controls the trade-off between fitting the training data well and preventing overfitting.

3) **Solver**: Logistic regression models can be solved using different algorithms or solvers. The solver parameter determines the optimization algorithm to use for finding the optimal parameter values. Some popular solver options include 'liblinear', 'lbfgs', 'newton-cg', and 'sag'. The choice of solver depends on the size of the dataset and the computational efficiency required.

4) **Maximum Iterations**: The maximum number of iterations is the maximum number of times the solver iterates to converge on the optimal parameter values. It is important to set a sufficiently large value for this parameter to ensure convergence of the optimization algorithm.

5) **Class Weight**: In cases where the classes are imbalanced, you can assign different weights to each class to give more importance to the minority class. The class_weight parameter allows you to specify the weights for each class. This can help improve the performance of the model on the minority class.

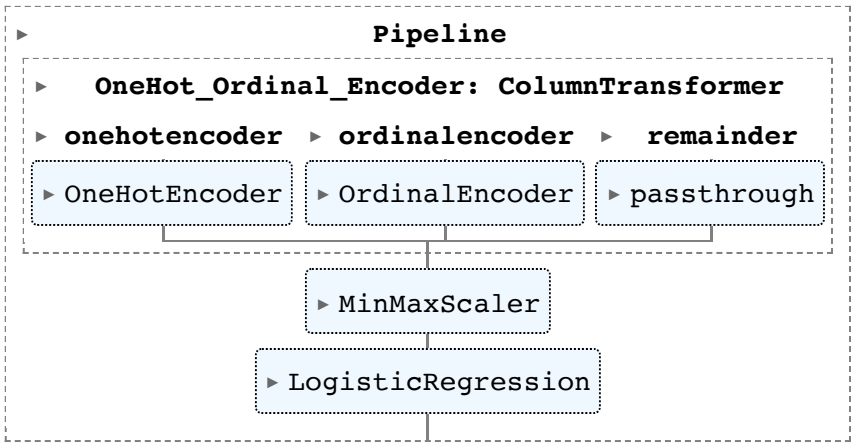
In [ ]:
```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, OrdinalEncoder, OneHotEncode

# prepare the features for encoding
cat_onehot = ['departments'] # onehot encoding for departments feature (no h
cat_ordinal = ['salary'] # ordinal encoding for salary bec of hierarchical c

cat_for_salary = ["low", "medium", "high"] # define the order among differen
```

In [ ]:
```
# encode the abovementioned features
enc_onehot = OneHotEncoder(handle_unknown="ignore", sparse=False)
enc_ordinal = OrdinalEncoder(categories= [cat_for_salary])
```

```
column_trans = make_column_transformer((enc_onehot, cat_onehot),
                                       (enc_ordinal, cat_ordinal),
                                        remainder='passthrough',
                                        verbose_feature_names_out=False)
```

In [ ]:
```
# operation order for pipeline
operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca

log_pipe = Pipeline(steps=operations).set_output(transform="pandas")

# Fit the pipeline on the training data
log_pipe.fit(X_train, y_train)
```

Out[ ]:




In [ ]:
```
print("LOG MODEL")
eval_metric(log_pipe, X_train, y_train, X_test, y_test)
```

```
LOG MODEL
Test_Set
[[1544  457]
 [  62  336]]
              precision    recall  f1-score   support

           0       0.96      0.77      0.86      2001
           1       0.42      0.84      0.56       398

    accuracy                           0.78      2399
   macro avg       0.69      0.81      0.71      2399
weighted avg       0.87      0.78      0.81      2399


Train_Set
[[6073 1926]
 [ 279 1314]]
              precision    recall  f1-score   support

           0       0.96      0.76      0.85      7999
           1       0.41      0.82      0.54      1593

    accuracy                           0.77      9592
   macro avg       0.68      0.79      0.70      9592
weighted avg       0.86      0.77      0.80      9592
```

In [ ]:
```
# check overfitting with cv
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer
from sklearn.metrics import f1_score, accuracy_score, precision_score, recal
```

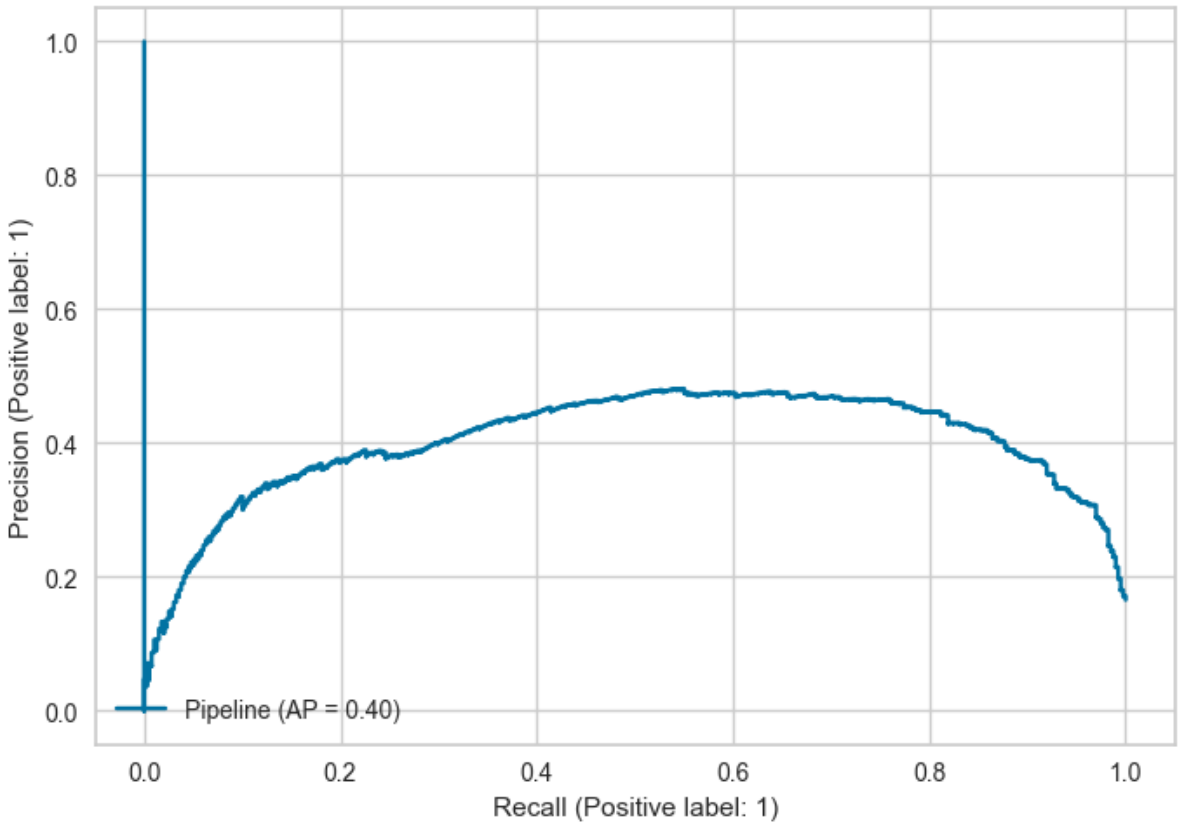
```
scoring= ["accuracy", "f1", "recall", "precision"]

operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca

pipe_model = Pipeline(steps=operations).set_output(transform="pandas")

scores = cross_validate(pipe_model,
                        X_train,
                        y_train,
                        scoring = scoring,
                        cv = 5,
                        return_train_score=True)
df_scores = pd.DataFrame(scores,
                         index = range(1, 6))
df_scores.mean()[2:]

# no overfitting but poor scores
```

Out[ ]:

```
test_accuracy      0.768558
train_accuracy     0.769495
test_f1            0.540430
train_f1           0.542281
test_recall        0.819207
train_recall       0.822191
test_precision     0.403328
train_precision    0.404558
dtype: float64
```

In [ ]:

```
from sklearn.metrics import PrecisionRecallDisplay, average_precision_score

PrecisionRecallDisplay.from_estimator(log_pipe,
                                      X_test,
                                      y_test);
# overall performance of the log model is quite low, it can separate the cla
```




In [ ]:

```
# save the scores to compare in the end with the scores of other models
y_pred = log_pipe.predict(X_test)
```

```
y_pred_proba= log_pipe.predict_proba(X_test)[:,1]

log_AP = average_precision_score(y_test, y_pred_proba)
log_rec = recall_score(y_test, y_pred)
log_f1 = f1_score(y_test,y_pred)
log_matthews =  matthews_corrcoef(y_test, y_pred)
log_time = measure_prediction_time(log_pipe,X_test)

log_AP,log_rec,log_f1,log_matthews,log_time
```

Out[ ]:
```
(0.396247497509286,
 0.8442211055276382,
 0.5642317380352645,
 0.4869902814700124,
 0.018590044975280762)
```

**What is Matthews Correlation Coefficient and why we use it for this data?**

The Matthews Correlation Coefficient (MCC) is **a measure of the quality of binary classifications, particularly in imbalanced datasets**. It takes into account true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) to calculate a single score that represents the overall performance of a binary classifier.

MCC ranges from -1 to 1, where a value of 1 indicates a perfect classifier, 0 indicates a random classifier, and -1 indicates a completely incorrect classifier. MCC considers all four classification results, unlike some other metrics that focus only on a specific aspect, such as accuracy or precision.

**MCC is particularly useful in imbalanced datasets because it balances the effect of true negatives against false negatives.** In imbalanced datasets, where the negative class (majority class) significantly outweighs the positive class (minority class), traditional metrics like accuracy can be misleading. For example, a classifier that always predicts the negative class will have a high accuracy but fails to capture the minority class correctly. MCC, on the other hand, takes into account the true negatives and false negatives and **provides a more reliable evaluation of classifier performance**.

In summary, MCC is a useful metric for evaluating the performance of binary classifiers, especially in imbalanced datasets, as it provides a balanced assessment of the classifier's ability to handle both positive and negative classes.

**Result: LOGISTIC REG model**:

What are the main measures in Confusion Matrix for this data:

- **True Negative (TN)**: The model correctly predicted that an employee continues to work (0) when they actually continue to work.
- **False Positive (FP)**: The model incorrectly predicted that an employee has churned (1) when they actually continue to work.
- **False Negative (FN)**: The model incorrectly predicted that an employee continues to work when they have actually churned.
- **True Positive (TP)**: The model correctly predicted that an employee has churned.

Based on the results otained from the vanilla model above, the logistic regression model shows lower precision, recall, and F1-score. This indicates that the model has more

difficulty correctly identifying churned employees (class 1). The precision for class 1 is particularly low, meaning there is a higher rate of false positive predictions.

The accuracy of the model is moderate, suggesting that it provides reasonably accurate predictions overall. In addition, the lower recall for class 0 indicates a higher rate of false negatives, meaning some employees who have churned are not correctly identified by the model.

### 5.2.2 KNN MODEL

K-Nearest Neighbors (KNN) is a simple yet powerful supervised learning algorithm used for both classification and regression tasks.

**Key Parameters of KNN**:

- **n_neighbors**: The number of neighbors to consider when making predictions. It is an integer value and should be carefully chosen to balance between overfitting and underfitting. A smaller value tends to increase model complexity and may lead to overfitting, while a larger value may increase bias and result in underfitting.
- **weights**: The weight function used in prediction. It can be set to 'uniform' (equal weights) or 'distance' (weights inversely proportional to distance). The choice of weights can have an impact on the performance of the algorithm.
- **metric** : The distance metric used to measure the similarity between instances. Common options include 'euclidean' (Euclidean distance), 'manhattan' (Manhattan distance), and 'cosine' (cosine similarity). The choice of metric depends on the nature of the data and the problem at hand.

It's important to note that **KNN is sensitive to the scale** and distribution of the features, so it's often beneficial to perform feature scaling before applying KNN.

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
```

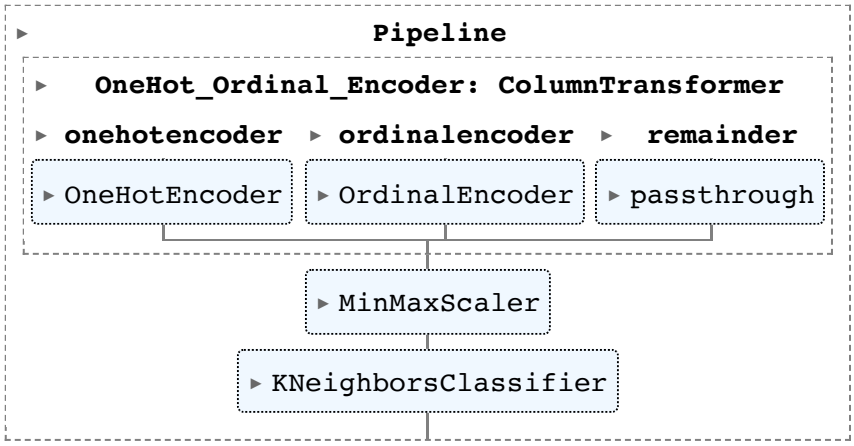
In [ ]:
```
# prepare the features for encoding
cat_onehot = ['departments'] # onehot encoding for departments feature (no h
cat_ordinal = ['salary'] # ordinal encoding for salary bec of hierarchical c

cat_for_salary = ["low", "medium", "high"] # define the order among differen
```

In [ ]:
```
# encode the abovementioned features
enc_onehot = OneHotEncoder(handle_unknown="ignore", sparse=False)
enc_ordinal = OrdinalEncoder(categories= [cat_for_salary])

column_trans = make_column_transformer((enc_onehot, cat_onehot),
                                       (enc_ordinal, cat_ordinal),
                                        remainder='passthrough',
                                        verbose_feature_names_out=False)
```

In [ ]:
```
# operation order for pipeline
operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca
              ("knn", KNeighborsClassifier(n_neighbors=5))]

knn_pipe = Pipeline(steps=operations).set_output(transform="pandas")
```

```
# Fit the pipeline on the training data
knn_pipe.fit(X_train, y_train)
```

Out[ ]:




In [ ]:
```
y_pred = knn_pipe.predict(X_test)
y_pred
```

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

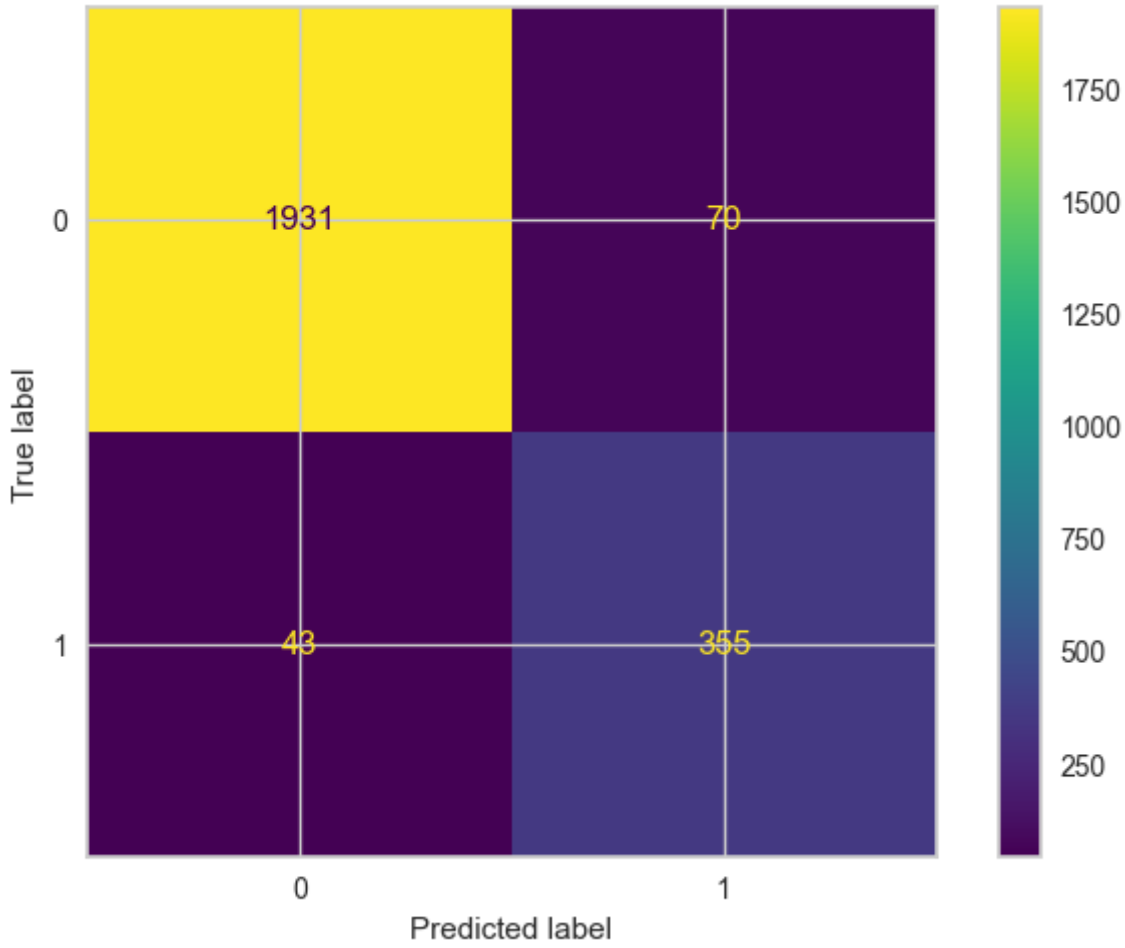
In [ ]:
```
# probabilities for being 0 or 1
y_pred_proba = knn_pipe.predict_proba(X_test)
```

In [ ]:
```
confusion_matrix(y_test, y_pred)
```

Out[ ]:
```
array([[1931,   70],
       [  43,  355]])
```

In [ ]:
```
# visualize confusion matrix
ConfusionMatrixDisplay.from_estimator(knn_pipe,X_test, y_test)
```

Out[ ]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fa04208
10f0>
```




In [ ]:
```
# check the scores
eval_metric(knn_pipe, X_train, y_train, X_test, y_test)

# KNN algorith gave higher scores even with vanilla model
```

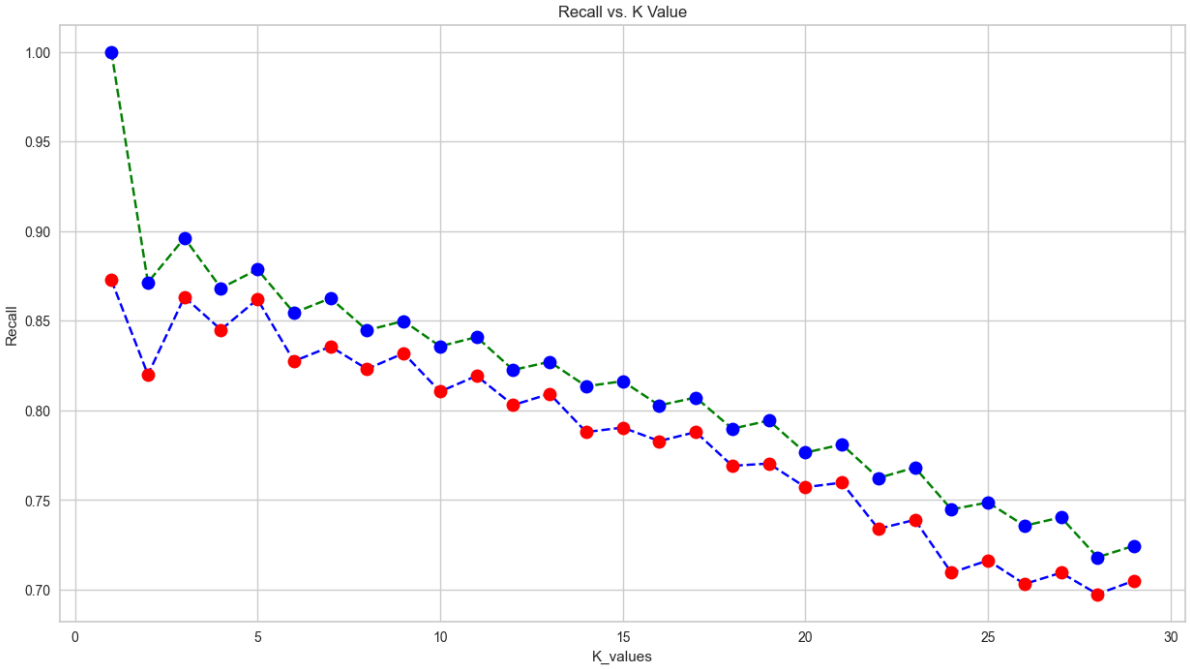
```
Test_Set
[[1931   70]
 [  43  355]]
              precision    recall  f1-score   support

           0       0.98      0.97      0.97      2001
           1       0.84      0.89      0.86       398

    accuracy                           0.95      2399
   macro avg       0.91      0.93      0.92      2399
weighted avg       0.95      0.95      0.95      2399


Train_Set
[[7803  196]
 [ 189 1404]]
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      7999
           1       0.88      0.88      0.88      1593

    accuracy                           0.96      9592
   macro avg       0.93      0.93      0.93      9592
weighted avg       0.96      0.96      0.96      9592
```

**Elbow Method for Choosing Reasonable K Values and Overfiting and underfiting control for k values**

In [ ]:
```
# cross val and gridsearch for best k values and overfitting control
test_recall_rates = []
train_recall_rates = []


for k in range(1,30):

    operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMa
                  ("knn", KNeighborsClassifier(n_neighbors=k))]

    knn_pipe_model = Pipeline(steps=operations)

    knn_pipe_model.fit(X_train,y_train)

    scores = cross_validate(knn_pipe_model, X_train, y_train, scoring = ['re

    recall_test_mean = scores["test_recall"].mean()
    recall_train_mean = scores["train_recall"].mean()


    test_recall = recall_test_mean
    train_recall = recall_train_mean
    test_recall_rates.append(test_recall)
    train_recall_rates.append(train_recall)
```

In [ ]:
```
plt.figure(figsize=(15,8))
plt.plot(range(1,30), test_recall_rates, color='blue', linestyle='--', marke
         markerfacecolor='red', markersize=10)

plt.plot(range(1,30), train_recall_rates, color='green', linestyle='--', mar
         markerfacecolor='blue', markersize=10)

plt.title('Recall vs. K Value')
plt.xlabel('K_values')
plt.ylabel('Recall')
#plt.hlines(y=0.074, xmin = 0, xmax = 30, colors= 'r', linestyles="--")
#plt.hlines(y=0.069, xmin = 0, xmax = 30, colors= 'r', linestyles="--")

# Red: validation - blues : train

# result: 5 k is the best value for highest recall and no overfitting
```

Out[ ]:
```
Text(0, 0.5, 'Recall')
```

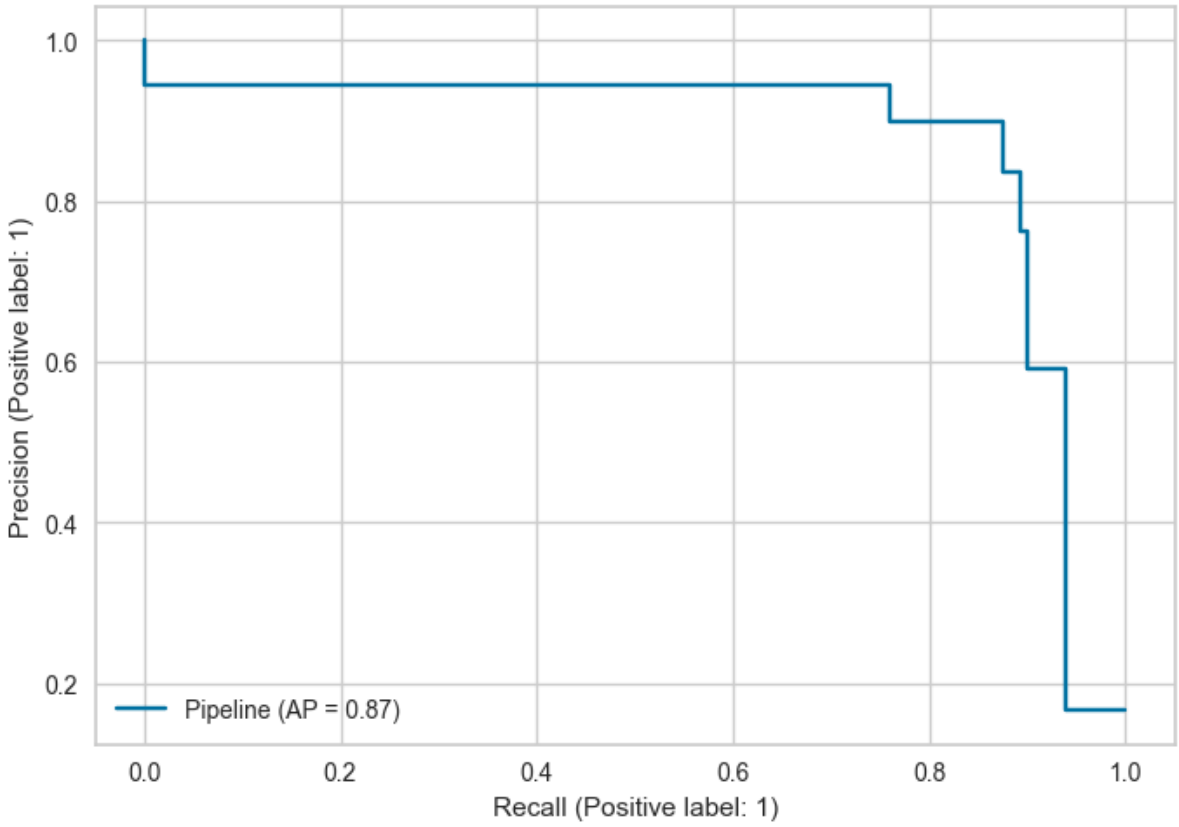



### Cross validate for optimal k-value

In [ ]:
```
operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca
                  ("knn", KNeighborsClassifier(n_neighbors=5))] # with 5 k

knn_model = Pipeline(steps=operations)

scores = cross_validate(knn_model, X_train, y_train, scoring = ['accuracy',
                                                                 'f1'], cv
df_scores = pd.DataFrame(scores, index = range(1, 6))
df_scores.mean()[2:]

# no overfitting
```

Out[ ]:
```
test_accuracy      0.947352
train_accuracy     0.957986
test_precision     0.828579
train_precision    0.869825
test_recall        0.861907
train_recall       0.878531
test_f1            0.844664
train_f1           0.874148
dtype: float64
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay , average_precision_score
PrecisionRecallDisplay.from_estimator(knn_pipe, X_test, y_test);

# general performance of model is 87%
```




In [ ]:
```
# save the scores to compare in the end with the scores of other models
y_pred = knn_pipe.predict(X_test)
y_pred_proba= knn_pipe.predict_proba(X_test)[:,1]

knn_AP = average_precision_score(y_test, y_pred_proba)
knn_rec = recall_score(y_test, y_pred)
knn_f1 = f1_score(y_test,y_pred)
knn_matthews =  matthews_corrcoef(y_test, y_pred)
knn_time = measure_prediction_time(knn_pipe,X_test)

knn_AP,knn_rec,knn_f1,knn_matthews,knn_time
```

Out[ ]:
```
(0.8742811357834483,
 0.8919597989949749,
 0.8626974483596598,
 0.8349613151335263,
 0.3146416902542114)
```

**RESULT for KNN**

Based on the scores above, the KNN algorithm consistently performed well and yielded relatively higher scores for each metric. When considering the recall score for this imbalanced data, it reached 89, while the precision and F1 scores were also higher compared to logistic regression. However, as demonstrated below, tree-based algorithms outperformed KNN and achieved higher scores for this particular dataset. Therefore, we will keep the KNN model with k=5 (the optimal number of neighbors) as is and will not conduct a grid search for it.

### 5.2.3 Support Vector Machines (SVC)

Support Vector Machines (SVM) is a popular supervised machine learning algorithm used for both classification and regression tasks. SVM is a binary classifier that aims to

find an optimal hyperplane that separates data points belonging to different classes in the feature space.

The basic idea behind SVM is to maximize the margin, which is the distance between the hyperplane and the nearest data points from each class. The hyperplane is chosen in such a way that it maximally separates the classes and minimizes the generalization error. The data points that lie closest to the hyperplane are called support vectors, and they play a crucial role in defining the decision boundary.

SVM can handle both linearly separable and non-linearly separable data by using various kernel functions. The kernel function transforms the input features into a higher-dimensional space, where the data may become linearly separable. Commonly used kernel functions include linear, polynomial, radial basis function (RBF), and sigmoid.

The training process of SVM involves solving an optimization problem to find the optimal hyperplane. This problem aims to minimize the classification error while maximizing the margin. SVM also introduces the concept of regularization to handle overfitting by adding a penalty term that balances the margin and the classification error.

Once trained, SVM can be used to predict the class labels of unseen data points. It assigns new data points to the class on the basis of which side of the decision boundary they fall.

SVM has several advantages, such as effective in high-dimensional spaces, robust against overfitting, and ability to handle both linear and non-linear data. However, SVM may suffer from scalability issues with large datasets and requires careful selection of hyperparameters for optimal performance.

The main parameters of Support Vector Machines (SVM) are as follows:

1) **Kernel**: SVM can use different types of kernels to transform the data into a higher-dimensional space. The kernel parameter determines the type of kernel to be used. Some commonly used kernels are linear, polynomial, radial basis function (RBF), and sigmoid. The choice of kernel depends on the nature of the data and the problem being solved.

2) **C**: The parameter C in SVM controls the trade-off between achieving a low training error and a low-margin decision boundary. A smaller value of C allows for a larger margin but may result in more training errors, while a larger value of C aims to minimize the training errors at the expense of a narrower margin. It determines the level of misclassification that is acceptable.

3) **Gamma**: Gamma is a parameter used in non-linear kernels (such as RBF) and controls the influence of each training example. A low value of gamma indicates a larger influence radius, resulting in a smoother decision boundary, while a high value of gamma indicates a smaller influence radius, resulting in a more complex and wiggly decision boundary.

4) **Class Weight**: Similar to logistic regression, SVM can also handle imbalanced datasets by assigning different weights to each class. The class_weight parameter

allows you to specify the weights for each class. It can be useful when the classes are imbalanced, and you want to give more importance to the minority class.

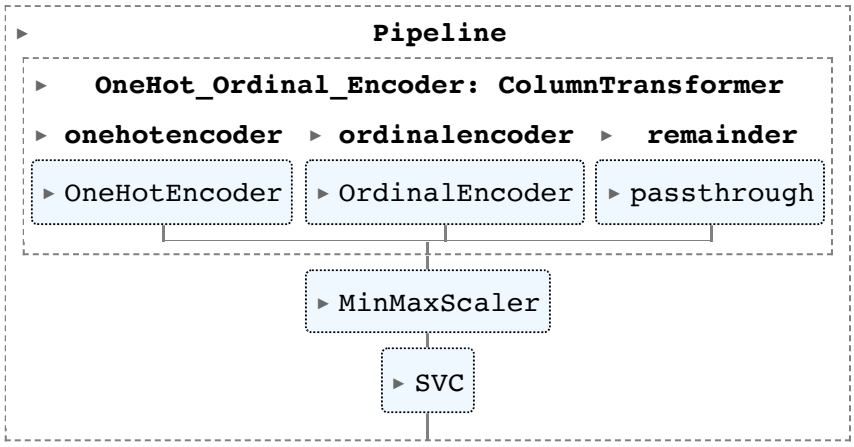
5) **Kernel Coefficient (Degree)**: This parameter is used in polynomial kernels and determines the degree of the polynomial function used to map the data into higher-dimensional space. It controls the flexibility of the model to capture non-linear relationships in the data.

In [ ]:
```
# prepare the features for encoding
cat_onehot = ['departments'] # onehot encoding for departments feature (no h
cat_ordinal = ['salary'] # ordinal encoding for salary bec of hierarchical c

cat_for_salary = ["low", "medium", "high"] # define the order among differen
```

In [ ]:
```
# encode the abovementioned features
enc_onehot = OneHotEncoder(handle_unknown="ignore", sparse=False)
enc_ordinal = OrdinalEncoder(categories= [cat_for_salary])

column_trans = make_column_transformer((enc_onehot, cat_onehot),
                                       (enc_ordinal, cat_ordinal),
                                        remainder='passthrough',
                                        verbose_feature_names_out=False)
```

In [ ]:
```
# operation order for pipeline
operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca
              ("SVC", SVC(max_iter=1000, class_weight="balanced", random_sta

SVC_pipe = Pipeline(steps=operations).set_output(transform="pandas")

# Fit the pipeline on the training data
SVC_pipe.fit(X_train, y_train)
```

Out[ ]:




In [ ]:
```
# check the scores
eval_metric(SVC_pipe, X_train, y_train, X_test, y_test)

# poor recall for 0 class and precision for  1 class
```

```
Test_Set
[[ 479 1522]
 [   5  393]]
              precision    recall  f1-score   support

           0       0.99      0.24      0.39      2001
           1       0.21      0.99      0.34       398

    accuracy                           0.36      2399
   macro avg       0.60      0.61      0.36      2399
weighted avg       0.86      0.36      0.38      2399


Train_Set
[[1759 6240]
 [  10 1583]]
              precision    recall  f1-score   support

           0       0.99      0.22      0.36      7999
           1       0.20      0.99      0.34      1593

    accuracy                           0.35      9592
   macro avg       0.60      0.61      0.35      9592
weighted avg       0.86      0.35      0.36      9592
```

In [ ]:
```
# check overfitting
operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca
              ("SVC", SVC(max_iter=1000, class_weight="balanced", random_sta
SVC_pipe = Pipeline(steps=operations)

scores = cross_validate(SVC_pipe,
                        X_train,
                        y_train,
                        scoring=['accuracy', 'precision','recall','f1'],
                        cv = 5,
                        return_train_score=True)
df_scores = pd.DataFrame(scores, index = range(1, 6))
df_scores.mean()[2:]

# too low precision scores
```

Out[ ]:
```
test_accuracy      0.358938
train_accuracy     0.355272
test_precision     0.205487
train_precision    0.204763
test_recall        0.994349
train_recall       0.997018
test_f1            0.340491
train_f1           0.339678
dtype: float64
```

**SVC with Best Parameters (GridsearchCV)**

**What is Gridsearch and why is it important?**

Grid search is a hyperparameter optimization technique used in machine learning to find the best combination of hyperparameter values for a given model. Hyperparameters are parameters that are not learned from the data but are set before training the model, such as the learning rate, regularization parameter, or number of hidden units in a neural network.

Grid search involves defining a grid of possible values for each hyperparameter and exhaustively searching through all possible combinations. For each combination, the model is trained and evaluated using a specified evaluation metric, such as accuracy or mean squared error. The combination of hyperparameter values that achieves the best performance on the evaluation metric is selected as the optimal set of hyperparameters.

In [ ]:
```
# import gridsearch
from sklearn.model_selection import GridSearchCV
```

In [ ]:
```
# define the param-grid
param_grid = {'SVC__C': [0.001, 0.05, 0.01, 0.1],
              'SVC__gamma': ["scale", "auto", 0.2, 0.3], # float values can
              'SVC__kernel': ['rbf', 'linear'], # try rbf and linear
              'SVC__class_weight': ["balanced", None]}
```

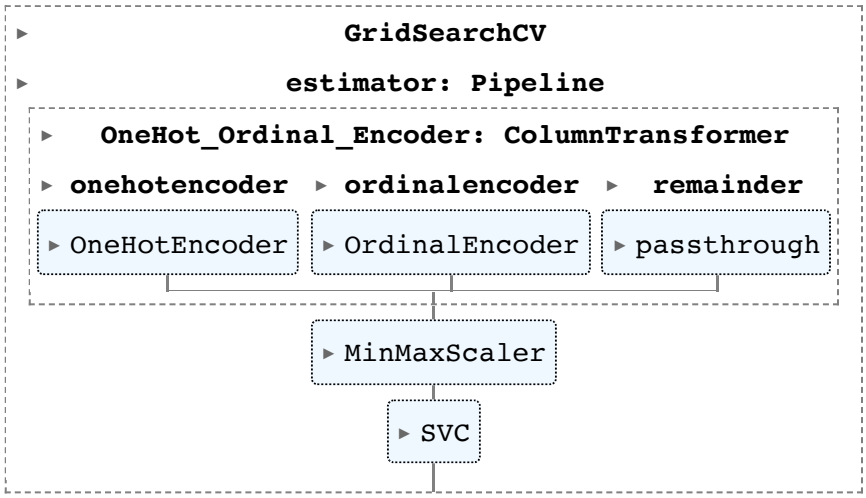
In [ ]:
```
%%time
# build the model
operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca
              ("SVC", SVC(max_iter=1000, class_weight="balanced", random_sta

SVC_pipe_model = Pipeline(steps=operations)

svm_model_grid = GridSearchCV(SVC_pipe_model,
                              param_grid,
                              scoring="recall", # recall in an imbalanced da
                              cv=5,
                              return_train_score=True)
```

```
CPU times: user 82 µs, sys: 91 µs, total: 173 µs
Wall time: 184 µs
```

In [ ]:
```
%%time
svm_model_grid.fit(X_train, y_train)
```

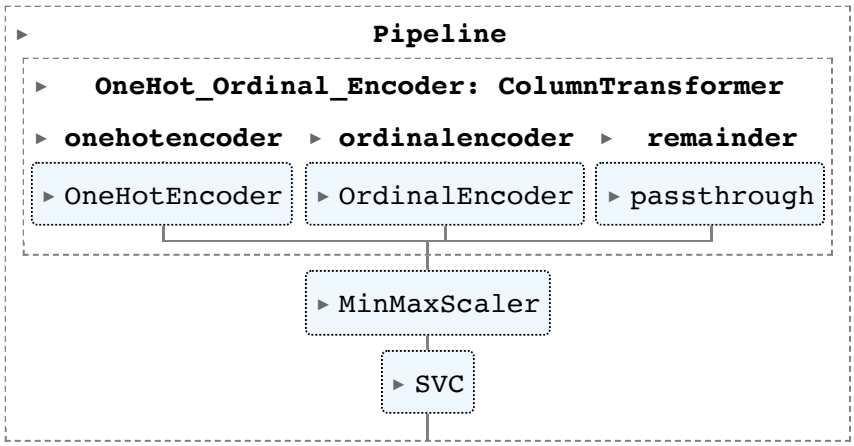
```
CPU times: user 10min 6s, sys: 10.8 s, total: 10min 17s
Wall time: 10min 35s
```

Out[ ]:




In [ ]:
```
svm_model_grid.best_estimator_
# C_0.001, kernel default rbf
```

Out[ ]:




In [ ]:
```
pd.DataFrame(svm_model_grid.cv_results_).loc[svm_model_grid.best_index_, ["m
```

Out[ ]:
```
mean_test_score     1.0
mean_train_score    1.0
Name: 0, dtype: object
```

In [ ]:
```
eval_metric(svm_model_grid, X_train, y_train, X_test, y_test)

#  model predicted all samples as belonging to the 1 class, resulting in a r
# 1 class and a recall score of 0 for the 0 class.
```

```
Test_Set
[[   0 2001]
 [   0  398]]
              precision    recall  f1-score   support

           0       0.00      0.00      0.00      2001
           1       0.17      1.00      0.28       398

    accuracy                           0.17      2399
   macro avg       0.08      0.50      0.14      2399
weighted avg       0.03      0.17      0.05      2399


Train_Set
[[   0 7999]
 [   0 1593]]
              precision    recall  f1-score   support

           0       0.00      0.00      0.00      7999
           1       0.17      1.00      0.28      1593

    accuracy                           0.17      9592
   macro avg       0.08      0.50      0.14      9592
weighted avg       0.03      0.17      0.05      9592
```

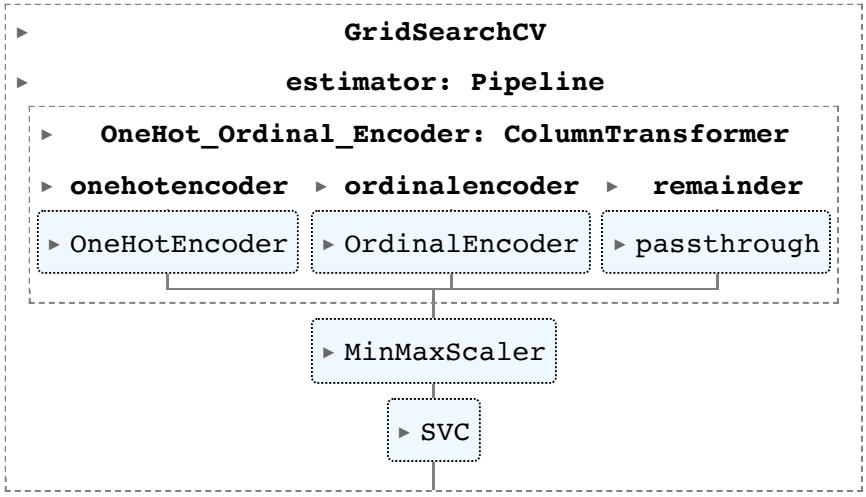
In [ ]:
```
%%time
# build the model by changing recall with f1-score this time. We aim to impr
operations = [("OneHot_Ordinal_Encoder", column_trans), ("scaler", MinMaxSca
              ("SVC", SVC(max_iter=1000, class_weight="balanced", random_sta

SVC_pipe_model_f1 = Pipeline(steps=operations)

svm_model_grid_f1 = GridSearchCV(SVC_pipe_model_f1,
                                 param_grid,
                                 scoring="f1",
                                 cv=5,
                                 return_train_score=True)
```

```
CPU times: user 39 µs, sys: 1e+03 ns, total: 40 µs
Wall time: 44.8 µs
```

In [ ]:
```
%%time
svm_model_grid_f1.fit(X_train, y_train)
```

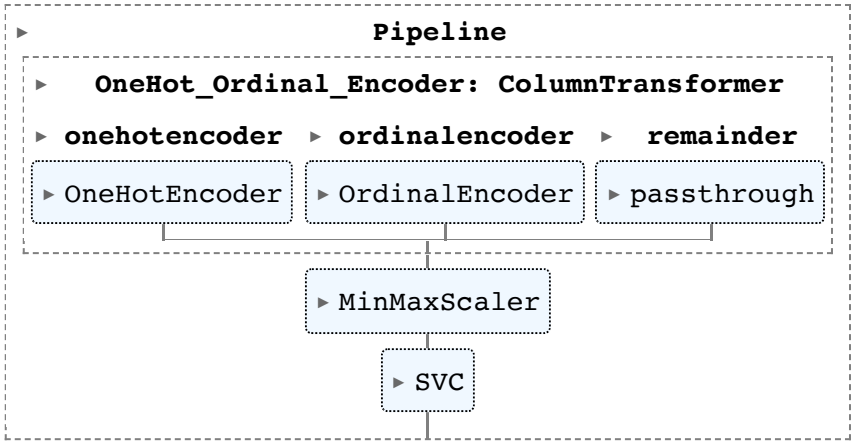
```
CPU times: user 10min 10s, sys: 10.5 s, total: 10min 21s
Wall time: 10min 41s
```

Out[ ]:




In [ ]:
```
svm_model_grid_f1.best_estimator_
```

Out[ ]:



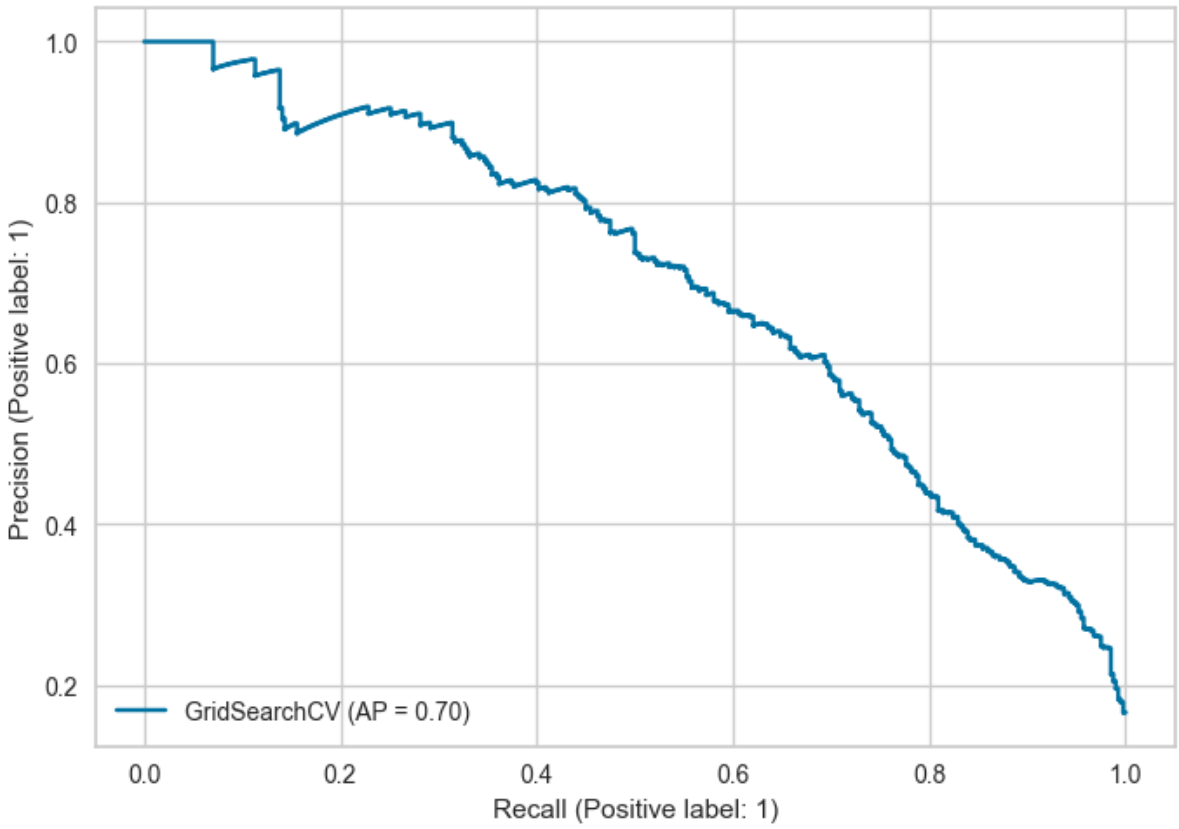

In [ ]:
```
pd.DataFrame(svm_model_grid_f1.cv_results_).loc[svm_model_grid_f1.best_index

#overfiting kontrolü
```

Out[ ]:
```
mean_test_score     0.634027
mean_train_score    0.654503
Name: 56, dtype: object
```

In [ ]:
```
eval_metric(svm_model_grid_f1, X_train, y_train, X_test, y_test)
```

```
Test_Set
[[1608  393]
 [  82  316]]
              precision    recall  f1-score   support

           0       0.95      0.80      0.87      2001
           1       0.45      0.79      0.57       398

    accuracy                           0.80      2399
   macro avg       0.70      0.80      0.72      2399
weighted avg       0.87      0.80      0.82      2399


Train_Set
[[6369 1630]
 [ 306 1287]]
              precision    recall  f1-score   support

           0       0.95      0.80      0.87      7999
           1       0.44      0.81      0.57      1593

    accuracy                           0.80      9592
   macro avg       0.70      0.80      0.72      9592
weighted avg       0.87      0.80      0.82      9592
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay , average_precision_score
PrecisionRecallDisplay.from_estimator(svm_model_grid_f1, X_test, y_test);
```




In [ ]:
```
# save the metric scores to comare with other models in the end
y_pred = svm_model_grid_f1.predict(X_test)
decision_function= svm_model_grid_f1.decision_function(X_test)

SVM_AP = average_precision_score(y_test, decision_function)
SVM_rec = recall_score(y_test, y_pred)
SVM_f1 = f1_score(y_test,y_pred)
```

```
SVM_matthews =  matthews_corrcoef(y_test, y_pred)
SVM_time =  measure_prediction_time(svm_model_grid_f1,X_test)
```

In [ ]:
```
SVM_AP, SVM_rec, SVM_f1, SVM_matthews, SVM_time
```

Out[ ]:
```
(0.699034621657618,
 0.7939698492462312,
 0.5709123757904245,
 0.48717610590661103,
 0.8221486806869507)
```

**Result for SVM**:

The scores of the SVM are lower than the other algorithms. In particular, precision scores were low for both classes. After tuning the hyperparameters via Gridsearch, the precison and f1 scores increased significantly, however as it can be seen below, other algorithms such as cat boost or xgboost perform far better than log reg or svm.

### 5.2.4 RANDOM FOREST

Random Forest is an ensemble learning method that combines multiple decision trees to make predictions. How does Random Forest work:

- **Random Subsampling**: Random Forest randomly selects a subset of the dataset (with replacement) to create a new training set for each decision tree in the ensemble. This process is known as bootstrapping.
- **Decision Tree Training**: Each decision tree in the Random Forest is trained on the bootstrapped training set using a random subset of features. This randomness helps introduce diversity among the trees.
- **Voting**: Once all the decision trees are trained, they individually make predictions on unseen data. In classification problems, the final prediction is determined by majority voting, where each tree's prediction is considered, and the class with the most votes is selected. For regression problems, the final prediction is usually the average of the individual tree predictions.

**Main advantages of Random Forests**:

- **Reduced Overfitting**: The ensemble of decision trees reduces the risk of overfitting compared to a single decision tree, as individual trees are prone to overfitting the training data.
- **Feature Importance**: Random Forests provide a measure of feature importance based on how much each feature contributes to the overall performance of the ensemble.
- **Robustness**: Random Forests are generally robust to outliers and noisy data due to the aggregation of multiple decision trees.
- **Efficiency**: The training process of Random Forests can be parallelized, making it efficient for large datasets.
- **Non-linear Relationships**: Random Forests can capture non-linear relationships between features and target variables.

It's worth noting that Random Forests have some limitations as well, such as decreased interpretability compared to individual decision trees and increased computational

complexity with a large number of trees.

**Key parameters of RF**

- **n_estimators**: This parameter determines the number of decision trees in the forest. Increasing the number of trees generally improves performance, but it also increases computation time.
- **max_depth**: It controls the maximum depth of each decision tree in the forest. Limiting the depth helps to prevent overfitting. Setting it to a lower value reduces complexity and can help improve generalization.
- **min_samples_split**: This parameter sets the minimum number of samples required to split an internal node. Increasing this value can prevent the model from creating small leaf nodes and can be used to control overfitting.
- **min_samples_leaf**: It sets the minimum number of samples required to be at a leaf node. Similar to min_samples_split, increasing this value can help prevent overfitting by avoiding the creation of small leaf nodes.
- **max_features**: This parameter controls the number of features to consider when looking for the best split. The value can be an integer, float, or string. The "sqrt" option uses the square root of the total number of features, while "log2" uses the logarithm base 2 of the total number of features. Choosing a smaller value reduces the correlation between trees and increases randomness.
- **bootstrap** : It determines whether bootstrap samples are used when building decision trees. By default, it is set to True, which means that bootstrap samples are used. Setting it to False would train each decision tree on the entire dataset.
- **class_weight**: It allows you to assign different weights to different classes. This can be useful if the dataset is imbalanced, as it helps the model to pay more attention to the minority class.

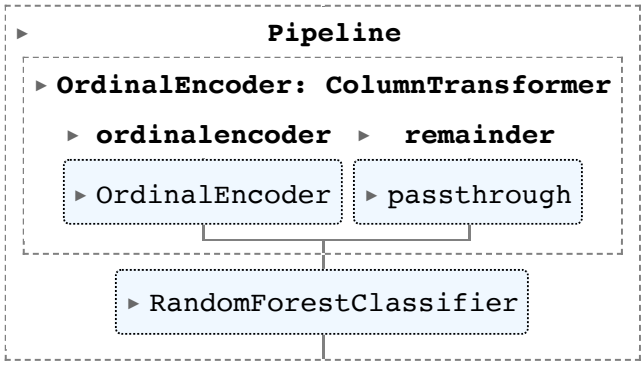
In [ ]:
```
# categorical features for encoding - ordinal encoder with better scores for
cat_ordinal = ['salary','departments']

enc_ordinal = OrdinalEncoder(handle_unknown='use_encoded_value'
                             ,unknown_value=-1)

column_trans = make_column_transformer((enc_ordinal, cat_ordinal),
                                       remainder='passthrough',
                                       verbose_feature_names_out=False).set
```

In [ ]:
```
# set the class weights
from sklearn.utils import class_weight
classes_weights = class_weight.compute_sample_weight(class_weight='balanced'
```

In [ ]:
```
operations_rf = [("OrdinalEncoder", column_trans),
                 ("RF_model", RandomForestClassifier(random_state=100 ))]

rf_pipe = Pipeline(steps=operations_rf)


params_pipe_rf = {
    'RF_model__sample_weight': classes_weights,
    # 'XGB_model__early_stopping_rounds': 10,   # Number of rounds with no i
    # 'XGB_model__eval_metric':'logloss',      # Evaluation metric for early
    # 'XGB_model__eval_set':[(X_test, y_test)]
}
```

```
# Fit the pipeline on the training data
rf_pipe.fit(X_train, y_train,**params_pipe_rf)
```

Out[ ]:




In [ ]:
```
print("RF MODEL")
eval_metric(rf_pipe, X_train, y_train, X_test, y_test)
```

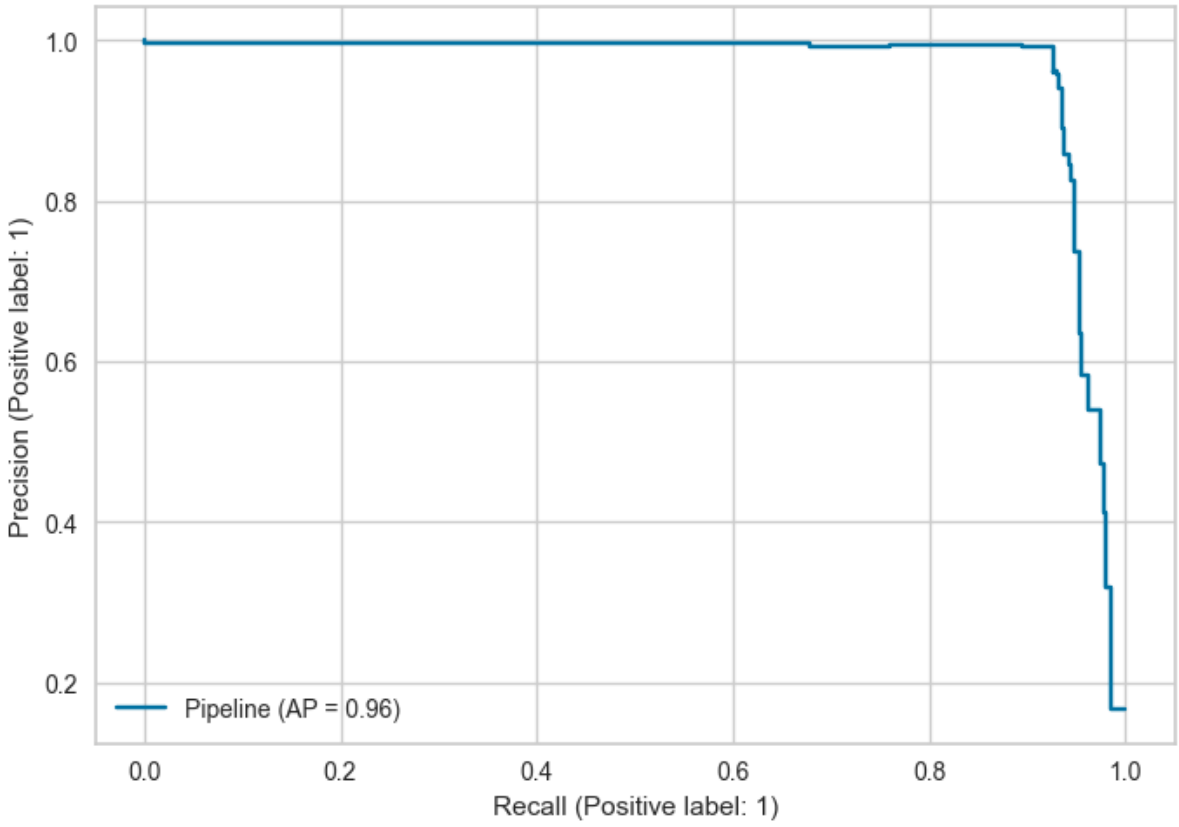
```
RF MODEL
Test_Set
[[1998    3]
 [  30  368]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      2001
           1       0.99      0.92      0.96       398

    accuracy                           0.99      2399
   macro avg       0.99      0.96      0.97      2399
weighted avg       0.99      0.99      0.99      2399


Train_Set
[[7999    0]
 [   1 1592]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      7999
           1       1.00      1.00      1.00      1593

    accuracy                           1.00      9592
   macro avg       1.00      1.00      1.00      9592
weighted avg       1.00      1.00      1.00      9592
```

In [ ]:
```
# Define the operations in the pipeline
operations_rf = [
    ("OrdinalEncoder", OrdinalEncoder()),
    ("RF_model", RandomForestClassifier (random_state=101))
]

# Create the pipeline
pipe_model = Pipeline(steps=operations_rf)

# Defining fit params for cross-validation
rf_pipe_params = {'RF_model__sample_weight':classes_weights}

# Define the scoring metric
scoring= ["accuracy", "f1", "recall", "precision"]

# Perform cross-validation
scores = cross_validate(pipe_model, X_train, y_train, scoring=scoring, cv=5,
```

```
# Create a DataFrame to store the scores
df_scores = pd.DataFrame(scores)

# Calculate the mean scores
mean_scores = df_scores.mean()[2:]

# Print the mean scores
print("Mean cross-validation scores:")
print(mean_scores)

# overfitting
```

```
Mean cross-validation scores:
test_accuracy      0.982694
train_accuracy     1.000000
test_f1            0.945546
train_f1           1.000000
test_recall        0.905838
train_recall       1.000000
test_precision     0.989049
train_precision    1.000000
dtype: float64
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay, average_precision_score

PrecisionRecallDisplay.from_estimator(rf_pipe,
                                      X_test,
                                      y_test);
```



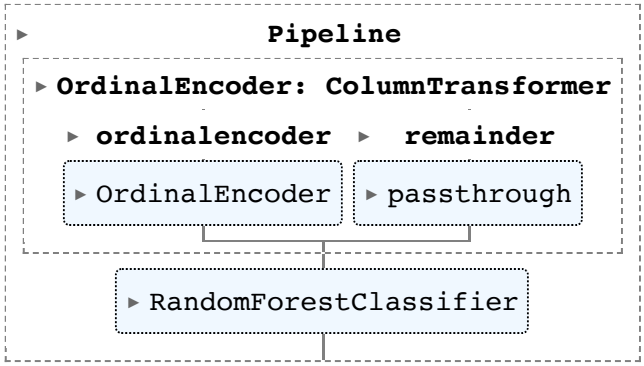

**How can we deal with overfitting in tree based models, RF in particular**

As seen above, scores for train data are 1.00 and for test data 0.92-0.93 which indicates a clear overfitting. Therefore we should tüne some of the hyperparameters to avoid overfitting. In case of tree based models, the most important hyperparameter to deal with overfitting is **max_depth** which cuts the split of the trees at a given point (3 in our

case). Another hyperparameter to avoid overfitting is **n_estimators**. We will increase its from 100 (default value) to 150 to enable the model learn better.

In [ ]:
```
# rebuild the model
operations_rf = [("OrdinalEncoder", column_trans),
                 ("RF_model", RandomForestClassifier(random_state=100, max_

rf_pipe = Pipeline(steps=operations_rf)


params_pipe_rf = {
    'RF_model__sample_weight': classes_weights,
}

# Fit the pipeline on the training data
rf_pipe.fit(X_train, y_train,**params_pipe_rf)
```

Out[ ]:




In [ ]:
```
print("RF MODEL")
eval_metric(rf_pipe, X_train, y_train, X_test, y_test)
```

```
RF MODEL
Test_Set
[[1882  119]
 [  25  373]]
              precision    recall  f1-score   support

           0       0.99      0.94      0.96      2001
           1       0.76      0.94      0.84       398

    accuracy                           0.94      2399
   macro avg       0.87      0.94      0.90      2399
weighted avg       0.95      0.94      0.94      2399


Train_Set
[[7520  479]
 [ 106 1487]]
              precision    recall  f1-score   support

           0       0.99      0.94      0.96      7999
           1       0.76      0.93      0.84      1593

    accuracy                           0.94      9592
   macro avg       0.87      0.94      0.90      9592
weighted avg       0.95      0.94      0.94      9592
```

In [ ]:
```
# check overfitting with cross val
operations_rf = [
    ("OrdinalEncoder", OrdinalEncoder()),
    ("RF_model", RandomForestClassifier (random_state=101, max_depth=3, n_es
```

```
]

# Create the pipeline
pipe_model = Pipeline(steps=operations_rf)

# Defining fit params for cross-validation
rf_pipe_params = {'RF_model__sample_weight':classes_weights}

# Define the scoring metric
scoring= ["accuracy", "f1", "recall", "precision"]

# Perform cross-validation
scores = cross_validate(pipe_model, X_train, y_train, scoring=scoring, cv=5,

# Create a DataFrame to store the scores
df_scores = pd.DataFrame(scores)

# Calculate the mean scores
mean_scores = df_scores.mean()[2:]

# Print the mean scores
print("Mean cross-validation scores:")
print(mean_scores)

# no overfitting
```

```
Mean cross-validation scores:
test_accuracy      0.939533
train_accuracy     0.941592
test_f1            0.835867
train_f1           0.841218
test_recall        0.926549
train_recall       0.931105
test_precision     0.761551
train_precision    0.767319
dtype: float64
```

In [ ]:
```
# save the metrcis to compare in the end
y_pred = rf_pipe.predict(X_test)
y_pred_proba= rf_pipe.predict_proba(X_test)[:,1]

rf_AP = average_precision_score(y_test, y_pred_proba)
rf_rec = recall_score(y_test, y_pred)
rf_f1 = f1_score(y_test,y_pred)
rf_matthews =  matthews_corrcoef(y_test, y_pred)
rf_time =  measure_prediction_time(rf_pipe,X_test,10)
```

In [ ]:
```
rf_AP, rf_rec, rf_f1, rf_matthews, rf_time
```

Out[ ]:
```
(0.9471948196677695,
 0.9371859296482412,
 0.8382022471910113,
 0.8086508210185859,
 0.0582319974899292)
```

***Feature Importance with RF**

In [ ]:
```
rf_pipe["RF_model"].feature_importances_
```

Out[ ]:
```
array([4.74444270e-03, 1.30364565e-04, 3.27861873e-01, 1.14836534e-01,
       1.65087304e-01, 1.30952928e-01, 2.43701705e-01, 1.26734298e-02,
       1.14191422e-05])
```

In [ ]:
```
features = rf_pipe["OrdinalEncoder"].get_feature_names_out()
features
```

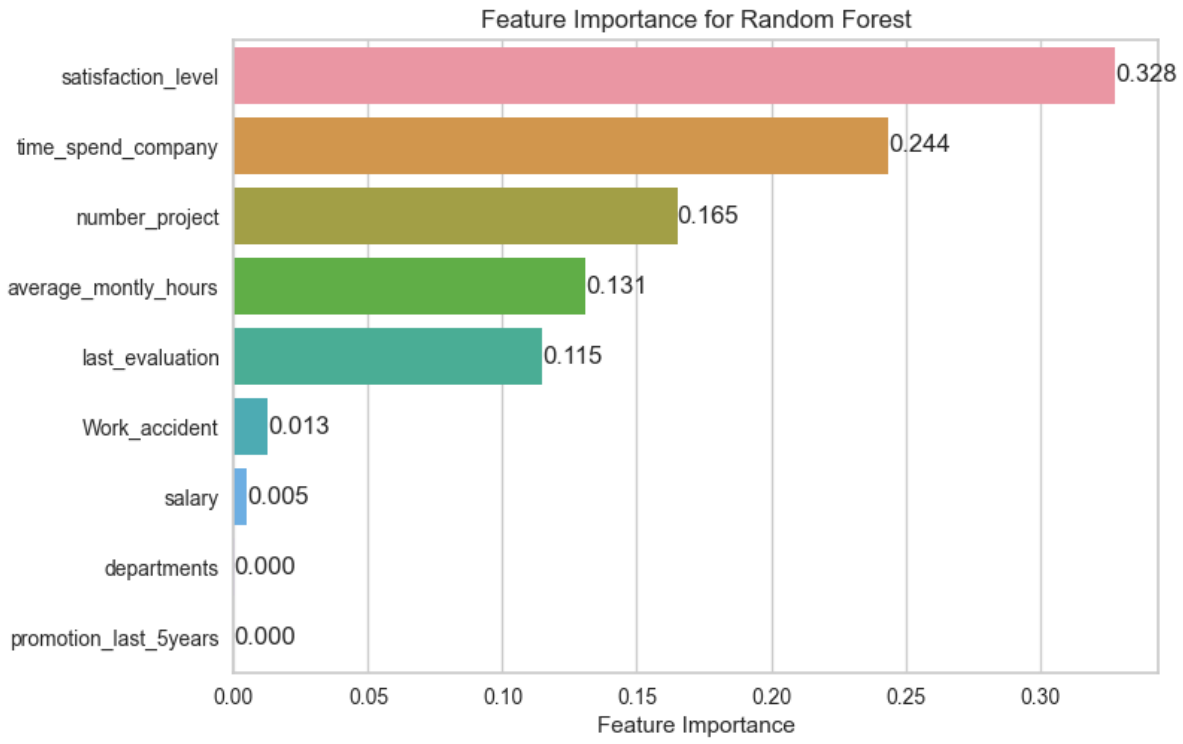
Out[ ]:
```
array(['salary', 'departments', 'satisfaction_level', 'last_evaluation',
       'number_project', 'average_montly_hours', 'time_spend_company',
       'Work_accident', 'promotion_last_5years'], dtype=object)
```

In [ ]:
```
rf_feature_imp = pd.DataFrame(data=rf_pipe["RF_model"].feature_importances_,
                              index = features, #index=X.columns
                              columns=["Feature Importance"])

rf_feature_imp = rf_feature_imp.sort_values("Feature Importance", ascending=
rf_feature_imp
```

Out[ ]:

| | Feature Importance |
|---|---|
| **satisfaction_level** | 0.327862 |
| **time_spend_company** | 0.243702 |
| **number_project** | 0.165087 |
| **average_montly_hours** | 0.130953 |
| **last_evaluation** | 0.114837 |
| **Work_accident** | 0.012673 |
| **salary** | 0.004744 |
| **departments** | 0.000130 |
| **promotion_last_5years** | 0.000011 |

In [ ]:
```
ax = sns.barplot(x=rf_feature_imp["Feature Importance"],
                 y=rf_feature_imp.index)

ax.bar_label(ax.containers[0],fmt="%.3f")
plt.title("Feature Importance for Random Forest")
plt.show()
```



**Why we will not rebuild the model with more important features:**

Normally we can rebuild the RF model with the more important features, however, since we have already limited number of features, we do not need to rebuild the model by decreasing the number of features. In addition, we willl get higher scores with other tree based models below.

### 5.2.5 Catboost

CatBoost stands for Categorical Boosting, and it is based on a boosting technique that combines multiple weak predictive models (decision trees) to create a strong predictive model. It is known for its ability to handle categorical features naturally, without requiring explicit encoding.

The algorithm works by iteratively building decision trees that predict the target variable (e.g., whether an employee will churn or not) based on the input features (e.g., employee characteristics). It uses a technique called gradient boosting, which focuses on improving the model's performance by minimizing prediction errors in each iteration.

One of the key advantages of CatBoost is its ability to automatically handle categorical features, which are often found in real-world datasets. It uses an innovative approach called ordered boosting, which transforms categorical features into numerical representations during the training process. This helps capture the underlying patterns and relationships between the categorical variables and the target variable.

Additionally, CatBoost incorporates advanced techniques to avoid overfitting and handle imbalanced datasets. It employs a combination of random permutations and ordered statistics to prevent the model from relying too heavily on individual data points or features.

In sum, CatBoost is a powerful and user-friendly algorithm that can deliver accurate predictions even with complex datasets containing categorical features. It simplifies the preprocessing steps and provides good performance out of the box, making it a popular choice for classification tasks.

Some of the main parameters of Catboost algorithm are:

- **learning_rate**: This parameter controls the step size at each iteration of the boosting process. A higher learning rate allows for faster convergence but may lead to overfitting, while a lower learning rate can help prevent overfitting but may require more iterations for convergence.
- **depth**: This parameter determines the depth of each decision tree in the ensemble. Increasing the depth allows the model to capture more complex interactions in the data but also increases the risk of overfitting. It is important to find an optimal depth that balances model complexity and generalization.
- **iterations**: This parameter specifies the number of boosting iterations to perform. Each iteration adds a new decision tree to the ensemble. Increasing the number of

iterations can improve the model's performance, but it also increases the computational time.

- **l2_leaf_reg**: This parameter controls the L2 regularization applied to the leaf weights of the decision trees. It helps to prevent overfitting by adding a penalty term to the loss function based on the magnitudes of the leaf weights.
- **random_strength**: This parameter introduces randomness into the feature selection process during tree construction. It controls the probability of selecting a random feature at each split. Adding randomness can help improve generalization and reduce overfitting.
- **border_count**: This parameter determines the number of discrete values used to approximate numerical features. A higher value allows for more precise representation of numerical features but also increases memory usage.

In [ ]:
```
from catboost import CatBoostClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

cat_features = X_train.select_dtypes(include=['object']).columns.tolist()


# Create a CatBoostClassifier model
cat_model = CatBoostClassifier(random_state = 42, verbose=500, auto_class_we
                               early_stopping_rounds=10, depth=3, use_best_m

# Fit the model on the training data
cat_model.fit(X_train, y_train, eval_set=(X_test, y_test))
```

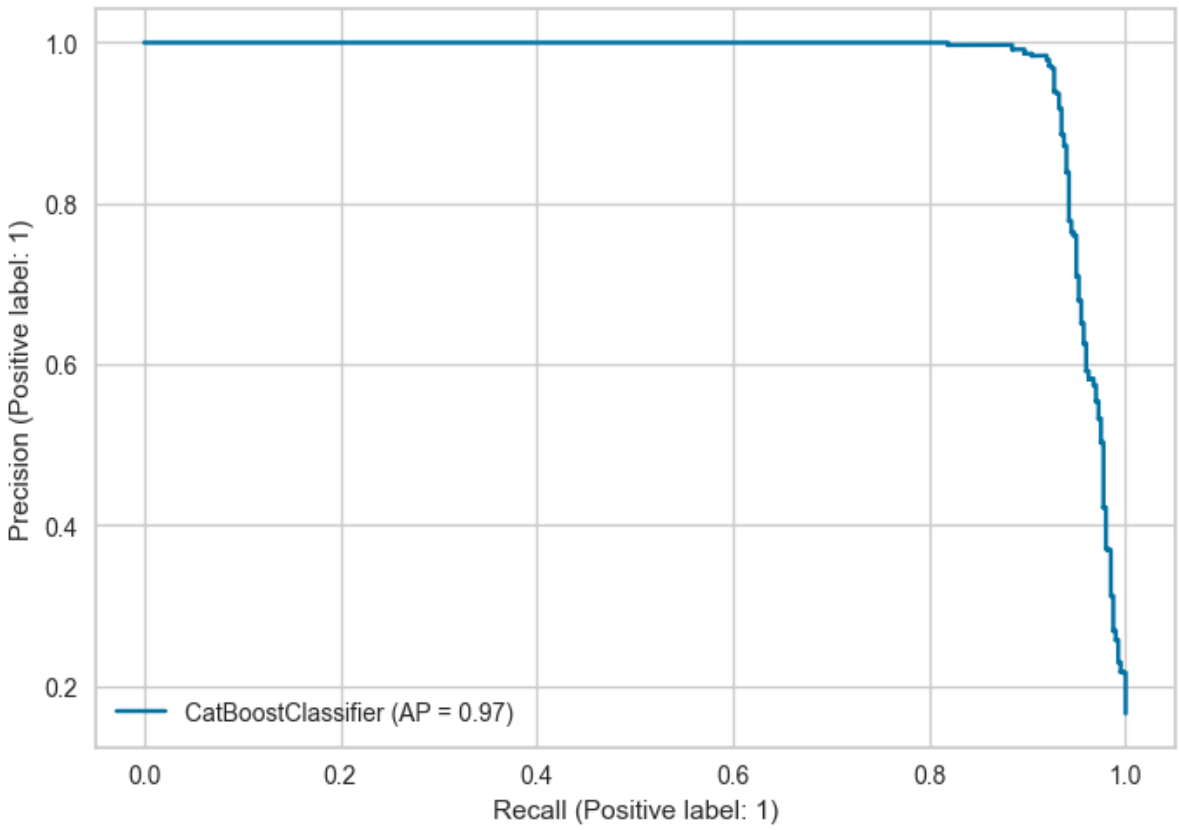
```
Learning rate set to 0.055391
0:      learn: 0.6423499        test: 0.6410303 best: 0.6410303 (0)     tota
l: 75.7ms       remaining: 1m 15s
Stopped by overfitting detector  (10 iterations wait)

bestTest = 0.1359220334
bestIteration = 239

Shrink model to first 240 iterations.
```

Out[ ]:
```
<catboost.core.CatBoostClassifier at 0x7fa199972c20>
```

In [ ]:
```
print("CATBOOST MODEL")
eval_metric(cat_model, X_train, y_train, X_test, y_test)
```

```
CATBOOST MODEL
Test_Set
[[1946   55]
 [  25  373]]
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      2001
           1       0.87      0.94      0.90       398

    accuracy                           0.97      2399
   macro avg       0.93      0.95      0.94      2399
weighted avg       0.97      0.97      0.97      2399


Train_Set
[[7788  211]
 [  93 1500]]
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      7999
           1       0.88      0.94      0.91      1593

    accuracy                           0.97      9592
   macro avg       0.93      0.96      0.94      9592
weighted avg       0.97      0.97      0.97      9592
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay

y_pred_proba = cat_model.predict_proba(X_test)

PrecisionRecallDisplay.from_estimator(cat_model,X_test, y_test)
plt.show();
```




In [ ]:
```
y_pred = cat_model.predict(X_test)
y_pred_proba= cat_model.predict_proba(X_test)[:,1]
```

```
cat_AP = average_precision_score(y_test, y_pred_proba)
cat_rec = recall_score(y_test, y_pred)
cat_f1 = f1_score(y_test,y_pred)
cat_matthews =  matthews_corrcoef(y_test, y_pred)
cat_time =  measure_prediction_time(cat_model,X_test)

cat_AP,cat_rec,cat_f1,cat_matthews,cat_time
```

Out[ ]:
```
(0.9688734453740755,
 0.9371859296482412,
 0.9031476997578693,
 0.8838893929306114,
 0.01158144474029541)
```

### Catboost Hyperparameter Tuning

In [ ]:
```
from flaml import AutoML

cat_automl = AutoML()
settings = {
    "time_budget": 120,  # total running time in seconds
    "metric": 'accuracy',  # primary metrics for regression can be chosen fr
    "estimator_list": ['catboost'],  # list of ML learners; we tune XGBoost
    "task": 'classification',  # task type
    "seed": 42,  # random seed
}
cat_automl.fit(X_train=X_train, y_train=y_train, **settings)
```

```
[flaml.automl.automl: 06-18 21:19:27] {2726} INFO - task = classification
[flaml.automl.automl: 06-18 21:19:27] {2728} INFO - Data split method: strat
ified
[flaml.automl.automl: 06-18 21:19:27] {2731} INFO - Evaluation method: cv
[flaml.automl.automl: 06-18 21:19:27] {2858} INFO - Minimizing error metric:
1-accuracy
[flaml.automl.automl: 06-18 21:19:27] {3004} INFO - List of ML learners in A
utoML Run: ['catboost']
[flaml.automl.automl: 06-18 21:19:27] {3334} INFO - iteration 0, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:32] {3472} INFO - Estimated sufficient tim
e budget=47242s. Estimated necessary time budget=47s.
[flaml.automl.automl: 06-18 21:19:32] {3519} INFO -  at 4.9s,   estimator ca
tboost's best error=0.0185,     best estimator catboost's best error=0.0185
[flaml.automl.automl: 06-18 21:19:32] {3334} INFO - iteration 1, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:37] {3519} INFO -  at 9.5s,   estimator ca
tboost's best error=0.0185,     best estimator catboost's best error=0.0185
[flaml.automl.automl: 06-18 21:19:37] {3334} INFO - iteration 2, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:39] {3519} INFO -  at 11.7s,  estimator ca
tboost's best error=0.0185,     best estimator catboost's best error=0.0185
[flaml.automl.automl: 06-18 21:19:39] {3334} INFO - iteration 3, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:42] {3519} INFO -  at 15.0s,  estimator ca
tboost's best error=0.0185,     best estimator catboost's best error=0.0185
[flaml.automl.automl: 06-18 21:19:42] {3334} INFO - iteration 4, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:45] {3519} INFO -  at 17.4s,  estimator ca
tboost's best error=0.0185,     best estimator catboost's best error=0.0185
[flaml.automl.automl: 06-18 21:19:45] {3334} INFO - iteration 5, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:49] {3519} INFO -  at 21.7s,  estimator ca
tboost's best error=0.0185,     best estimator catboost's best error=0.0185
[flaml.automl.automl: 06-18 21:19:49] {3334} INFO - iteration 6, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:51] {3519} INFO -  at 24.1s,  estimator ca
tboost's best error=0.0185,     best estimator catboost's best error=0.0185
[flaml.automl.automl: 06-18 21:19:51] {3334} INFO - iteration 7, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:55] {3519} INFO -  at 27.7s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:19:55] {3334} INFO - iteration 8, current lea
rner catboost
[flaml.automl.automl: 06-18 21:19:58] {3519} INFO -  at 30.8s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:19:58] {3334} INFO - iteration 9, current lea
rner catboost
[flaml.automl.automl: 06-18 21:20:02] {3519} INFO -  at 34.4s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:02] {3334} INFO - iteration 10, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:04] {3519} INFO -  at 37.3s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:04] {3334} INFO - iteration 11, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:08] {3519} INFO -  at 40.7s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:08] {3334} INFO - iteration 12, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:11] {3519} INFO -  at 43.5s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:11] {3334} INFO - iteration 13, current le
arner catboost
```

```
[flaml.automl.automl: 06-18 21:20:14] {3519} INFO -  at 46.6s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:14] {3334} INFO - iteration 14, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:17] {3519} INFO -  at 49.7s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:17] {3334} INFO - iteration 15, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:20] {3519} INFO -  at 52.5s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:20] {3334} INFO - iteration 16, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:24] {3519} INFO -  at 56.8s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:24] {3334} INFO - iteration 17, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:26] {3519} INFO -  at 59.2s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:26] {3334} INFO - iteration 18, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:31] {3519} INFO -  at 63.4s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:31] {3334} INFO - iteration 19, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:34] {3519} INFO -  at 67.1s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:34] {3334} INFO - iteration 20, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:38] {3519} INFO -  at 70.4s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:38] {3334} INFO - iteration 21, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:40] {3519} INFO -  at 73.3s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:40] {3334} INFO - iteration 22, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:43] {3519} INFO -  at 76.3s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:43] {3334} INFO - iteration 23, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:46] {3519} INFO -  at 79.2s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:46] {3334} INFO - iteration 24, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:49] {3519} INFO -  at 81.8s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:49] {3334} INFO - iteration 25, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:52] {3519} INFO -  at 85.3s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:52] {3334} INFO - iteration 26, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:55] {3519} INFO -  at 88.0s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:55] {3334} INFO - iteration 27, current le
arner catboost
[flaml.automl.automl: 06-18 21:20:59] {3519} INFO -  at 91.3s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:20:59] {3334} INFO - iteration 28, current le
arner catboost
[flaml.automl.automl: 06-18 21:21:03] {3519} INFO -  at 96.2s,  estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:21:03] {3334} INFO - iteration 29, current le
arner catboost
```

```
[flaml.automl.automl: 06-18 21:21:08] {3519} INFO -  at 100.4s, estimator ca
tboost's best error=0.0181,     best estimator catboost's best error=0.0181
[flaml.automl.automl: 06-18 21:21:08] {3334} INFO - iteration 30, current le
arner catboost
[flaml.automl.automl: 06-18 21:21:12] {3519} INFO -  at 105.2s, estimator ca
tboost's best error=0.0179,     best estimator catboost's best error=0.0179
[flaml.automl.automl: 06-18 21:21:12] {3334} INFO - iteration 31, current le
arner catboost
[flaml.automl.automl: 06-18 21:21:16] {3519} INFO -  at 108.9s, estimator ca
tboost's best error=0.0179,     best estimator catboost's best error=0.0179
[flaml.automl.automl: 06-18 21:21:16] {3334} INFO - iteration 32, current le
arner catboost
[flaml.automl.automl: 06-18 21:21:19] {3519} INFO -  at 112.1s, estimator ca
tboost's best error=0.0179,     best estimator catboost's best error=0.0179
[flaml.automl.automl: 06-18 21:21:19] {3334} INFO - iteration 33, current le
arner catboost
[flaml.automl.automl: 06-18 21:21:23] {3519} INFO -  at 116.3s, estimator ca
tboost's best error=0.0179,     best estimator catboost's best error=0.0179
[flaml.automl.automl: 06-18 21:21:24] {3783} INFO - retrain catboost for 0.7
s
[flaml.automl.automl: 06-18 21:21:24] {3790} INFO - retrained model: <catboo
st.core.CatBoostClassifier object at 0x7fa1153c4850>
[flaml.automl.automl: 06-18 21:21:24] {3034} INFO - fit succeeded
[flaml.automl.automl: 06-18 21:21:24] {3035} INFO - Time taken to find the b
est model: 105.24280595779419
```

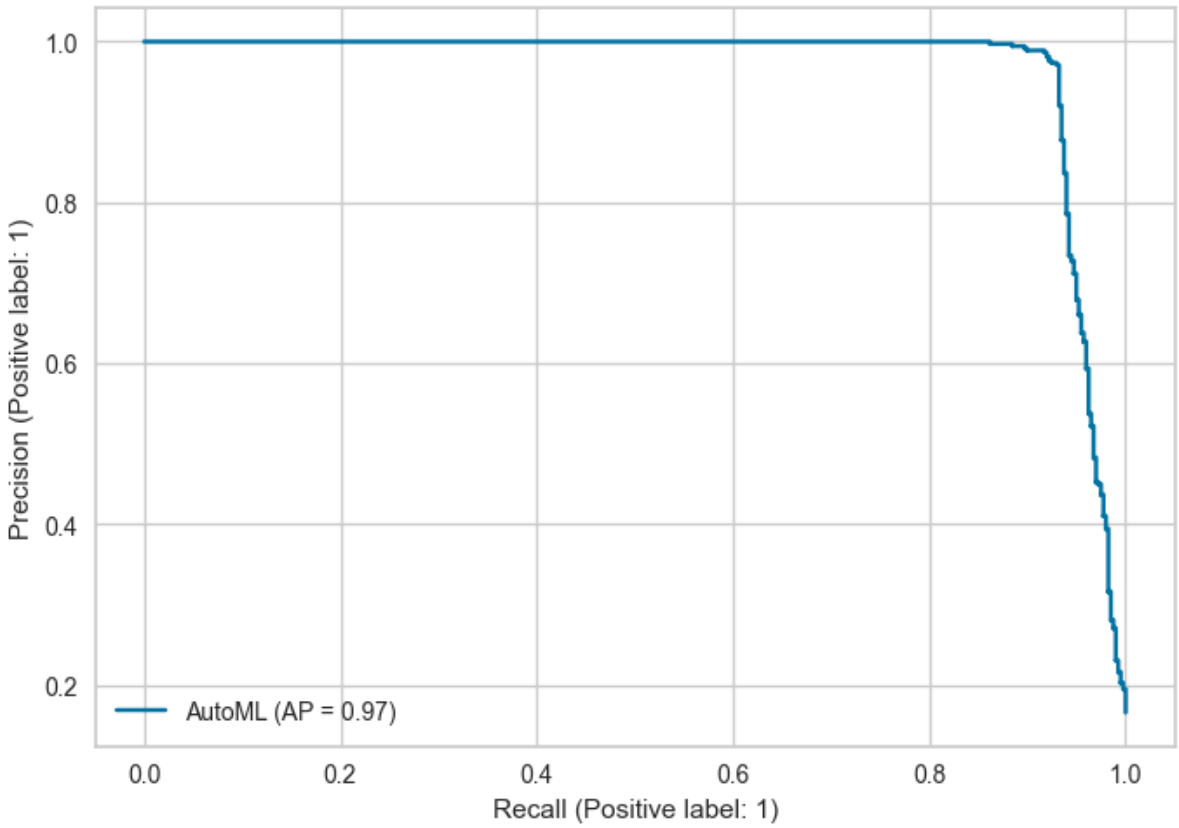
In [ ]:
```
'''retrieve best config and best learner'''
print('Best ML leaner:', cat_automl.best_estimator)
print('Best hyperparmeter config:', cat_automl.best_config)
print('Best accuracy on validation data: {0:.4g}'.format(1-cat_automl.best_l
print('Training duration of best run: {0:.4g} s'.format(cat_automl.best_conf
```

```
Best ML leaner: catboost
Best hyperparmeter config: {'early_stopping_rounds': 14, 'learning_rate': 0.
08896864159030861, 'n_estimators': 170}
Best accuracy on validation data: 0.9821
Training duration of best run: 0.6723 s
```

In [ ]:
```
print("CAT-Tuned MODEL")
eval_metric(cat_automl, X_train, y_train, X_test, y_test)
```

```
CAT-Tuned MODEL
Test_Set
[[1990   11]
 [  27  371]]
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      2001
           1       0.97      0.93      0.95       398

    accuracy                           0.98      2399
   macro avg       0.98      0.96      0.97      2399
weighted avg       0.98      0.98      0.98      2399


Train_Set
[[7967   32]
 [ 120 1473]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      7999
           1       0.98      0.92      0.95      1593

    accuracy                           0.98      9592
   macro avg       0.98      0.96      0.97      9592
weighted avg       0.98      0.98      0.98      9592
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay, average_precision_score

PrecisionRecallDisplay.from_estimator(cat_automl,
                                      X_test,
                                      y_test);
```




In [ ]:
```
y_pred = cat_automl.predict(X_test)
y_pred_proba= cat_automl.predict_proba(X_test)[:,1]

cata_AP = average_precision_score(y_test, y_pred_proba)
```

```
cata_rec = recall_score(y_test, y_pred)
cata_f1 = f1_score(y_test,y_pred)
cata_matthews =  matthews_corrcoef(y_test, y_pred)
cata_time =  measure_prediction_time(cat_automl,X_test)

cata_AP,cata_rec,cata_f1,cata_matthews,cata_time
```

Out[ ]:
```
(0.9671398050456086,
 0.9321608040201005,
 0.9512820512820513,
 0.9421119912719026,
 0.031109547615051268)
```

**Result for CATBOOST**:

Since the scores of tuned model is better than untuned model, we will use the former one. Here is a breakdown of the different metrics and what they mean in the context of this churn prediction task:

Test Set:

- **True Negative (TN)**: 1990 - The number of instances correctly predicted as non-churned employees.
- **False Positive (FP)**: 11 - The number of instances incorrectly predicted as churned employees when they are actually non-churned employees.
- **False Negative (FN)**: 28 - The number of instances incorrectly predicted as non-churned employees when they are actually churned employees.
- **True Positive (TP)**: 370 - The number of instances correctly predicted as churned employees.

**Precision**: Precision is the ratio of true positives to the sum of true positives and false positives. In the test set, the precision for churned employees is 0.97, which means that 97% of the employees predicted as churned by the model are actually churned employees.

**Recall**: Recall is the ratio of true positives to the sum of true positives and false negatives. In the test set, the recall for churned employees is 0.93, indicating that the model identifies 93% of the actual churned employees.

**F1-Score**: The F1-score is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance. In the test set, the F1-score for churned employees is 0.95, indicating a good balance between precision and recall.

**Accuracy**: Accuracy is the overall correct prediction rate of the model. In the test set, the accuracy is 0.98, which means that the model accurately predicts whether an employee will churn or not with 97% accuracy.

All in all, the CatBoost model shows strong performance in predicting employee churn, with high precision, recall, and F1-score for churned employees. The model correctly identifies a significant number of churned employees (TP), while minimizing false positives (FP).

### 5.2.6 XGBoost

XGBoost (Extreme Gradient Boosting) is a powerful and popular machine learning algorithm known for its high performance and efficiency in handling structured data. It belongs to the family of gradient boosting algorithms, which are ensemble methods that combine multiple weak predictive models (typically decision trees) to create a strong predictive model.

**How does XGBoost work?**:

- **Boosting**: XGBoost uses a boosting technique where models are trained sequentially, with each model attempting to correct the mistakes made by the previous models. It trains a new model to predict the errors (residuals) of the previous models.
- **Gradient Optimization**: XGBoost uses gradient optimization to train the models. It calculates the gradient of a loss function (such as mean squared error or log loss) with respect to the predicted values of the previous models. The new model is trained to minimize this gradient, which helps improve the overall model's performance.
- **Decision Trees**: XGBoost uses decision trees as base learners. Decision trees are simple, hierarchical models that make predictions based on a series of if-else conditions. XGBoost can create decision trees of varying depth and complexity.
- **Regularization**: XGBoost incorporates regularization techniques to prevent overfitting and improve generalization. It includes terms in the objective function that penalize complex models, such as adding a regularization term to the loss function.
- **Feature Importance**: XGBoost provides a measure of feature importance, indicating the relevance of each feature in the predictive model. It calculates the total reduction in the loss function that is achieved by splits on a particular feature, giving insights into which features are more influential.

**The main parameters of XGBoost** can be categorized into three groups: general parameters, booster parameters, and task-specific parameters. Here's a simplified explanation of each group:

1) **General Parameters**:

- **n_estimators**: The number of boosting rounds (or decision trees) to build. It controls the overall complexity and number of models in the ensemble.
- **learning_rate**: Controls the step size at each boosting iteration. A lower learning rate requires more boosting rounds but can yield better generalization.
- **max_depth**: The maximum depth of each decision tree. It determines the complexity of individual trees and affects the model's ability to capture interactions.
- **subsample**: The fraction of training samples used for training each tree. It helps combat overfitting by introducing randomness into the training process.
- **colsample_bytree**: The fraction of features (columns) used for training each tree. It helps control the diversity of the features considered in each tree.

**Booster Parameters**:

- **booster**: The type of booster to use, such as "gbtree" for tree-based models or "gblinear" for linear models.
- **gamma**: Controls the minimum loss reduction required to make a further partition on a leaf node of the tree. It helps control the tree's complexity.
- **reg_alpha and reg_lambda**: L1 and L2 regularization terms applied to the weights of the tree. They help control overfitting by adding regularization penalties.
- **min_child_weight**: Controls the minimum sum of instance weights required in a child node. It helps control the tree's complexity and can improve robustness.

In [ ]:
```
cat_ordinal = ['salary','departments']
```

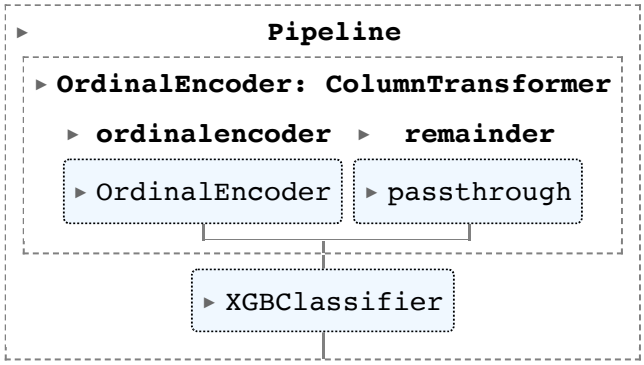
In [ ]:
```
enc_ordinal = OrdinalEncoder(handle_unknown='use_encoded_value'
                             ,unknown_value=-1)

column_trans = make_column_transformer((enc_ordinal, cat_ordinal),
                                       remainder='passthrough',
                                       verbose_feature_names_out=False).set
```

In [ ]:
```
from sklearn.utils import class_weight
classes_weights = class_weight.compute_sample_weight(class_weight='balanced'
```

In [ ]:
```
# X_train_xgb = X_train.copy()
# X_train_xgb = column_trans.fit_transform(X_train_xgb)
# X_test_xgb = column_trans.transform(X_test)
```

In [ ]:
```
operations_xgb = [("OrdinalEncoder", column_trans),
                  ("XGB_model", XGBClassifier(random_state=101, max_depth =3

xgb_pipe = Pipeline(steps=operations_xgb)

# XGboost algortimasının multiclass datalar için weight_class hyper parametr
# sample_weight hyper paremetresini kullanabiliyoruz.

params_pipe_xgb = {
    'XGB_model__sample_weight': classes_weights,
    # 'XGB_model__early_stopping_rounds': 10,   # Number of rounds with no i
    # 'XGB_model__eval_metric':'logloss',      # Evaluation metric for early
    # 'XGB_model__eval_set':[(X_test, y_test)]
}

# Fit the pipeline on the training data
xgb_pipe.fit(X_train, y_train,**params_pipe_xgb)
```

Out[ ]:




In [ ]:
```
print("XGB MODEL")
eval_metric(xgb_pipe, X_train, y_train, X_test, y_test)
```

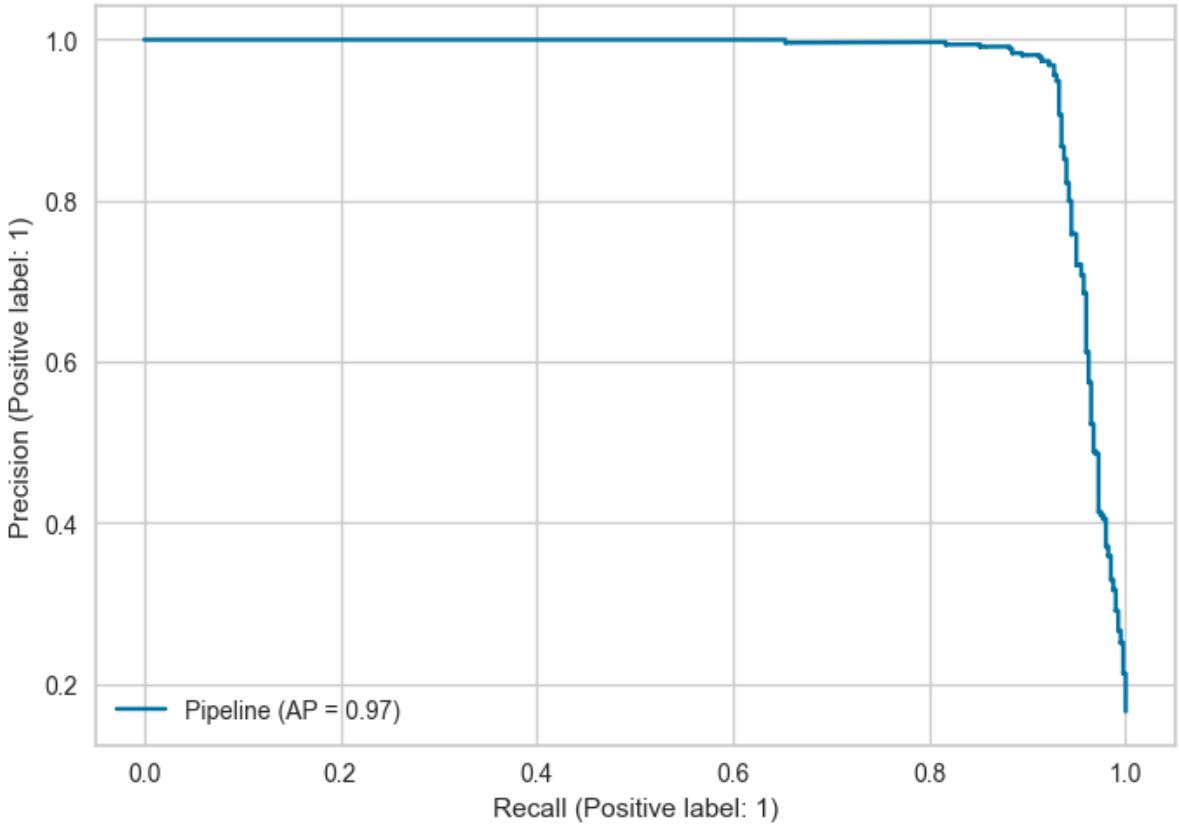
```
XGB MODEL
Test_Set
[[1949   52]
 [  26  372]]
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      2001
           1       0.88      0.93      0.91       398

    accuracy                           0.97      2399
   macro avg       0.93      0.95      0.94      2399
weighted avg       0.97      0.97      0.97      2399


Train_Set
[[7855  144]
 [  74 1519]]
              precision    recall  f1-score   support

           0       0.99      0.98      0.99      7999
           1       0.91      0.95      0.93      1593

    accuracy                           0.98      9592
   macro avg       0.95      0.97      0.96      9592
weighted avg       0.98      0.98      0.98      9592
```

In [ ]:

```python
# Define the operations in the pipeline
operations_xgb = [
    ("OrdinalEncoder", OrdinalEncoder()),
    ("XGB_model", XGBClassifier(random_state=101, max_depth=3))
]

# Create the pipeline
pipe_model = Pipeline(steps=operations_xgb)

# Defining fit params for cross-validation
xgb_pipe_params = {'XGB_model__sample_weight':classes_weights}

# Define the scoring metric
scoring= ["accuracy", "f1", "recall", "precision"]

# Perform cross-validation
scores = cross_validate(pipe_model, X_train, y_train, scoring=scoring, cv=5,

# Create a DataFrame to store the scores
df_scores = pd.DataFrame(scores)

# Calculate the mean scores
mean_scores = df_scores.mean()[2:]

# Print the mean scores
print("Mean cross-validation scores:")
print(mean_scores)
```

```
Mean cross-validation scores:
test_accuracy       0.972060
train_accuracy      0.978732
test_f1             0.917439
train_f1            0.937336
test_recall         0.934711
train_recall        0.957626
test_precision      0.900829
train_precision     0.917907
dtype: float64
```

In [ ]:
```
# check overall performance of the model

PrecisionRecallDisplay.from_estimator(xgb_pipe,
                                      X_test,
                                      y_test);
```




In [ ]:
```
y_pred = xgb_pipe.predict(X_test)
y_pred_proba= xgb_pipe.predict_proba(X_test)[:,1]

xgb_AP = average_precision_score(y_test, y_pred_proba)
xgb_rec = recall_score(y_test, y_pred)
xgb_f1 = f1_score(y_test,y_pred)
xgb_matthews =  matthews_corrcoef(y_test, y_pred)
xgb_time =  measure_prediction_time(xgb_pipe,X_test)

xgb_AP,xgb_rec,xgb_f1,xgb_matthews,xgb_time
```

Out[ ]:
```
(0.9674311493008211,
 0.9346733668341709,
 0.9051094890510949,
 0.8861609561204485,
 0.021289563179016112)
```

**XGBoost Hyper-Parameter Tuning**

In [ ]:

```
from flaml import AutoML

# minimizing 1 - matthews corrcoef
def custom_metric(X_val, y_val, estimator, labels, X_train, y_train,
                  weight_val=None, weight_train=None, config=None,
                  groups_val=None, groups_train=None):
    from sklearn.metrics import matthews_corrcoef
    import time

    start = time.time()
    y_pred = estimator.predict(X_val)
    pred_time = (time.time() - start) / len(X_val)

    val_mcc = matthews_corrcoef(y_val, y_pred)

    y_pred = estimator.predict(X_train)
    train_mcc = matthews_corrcoef(y_train, y_pred)

    alpha = 0.5
    return 1- (val_mcc * (1 + alpha) - alpha * train_mcc), {
        "val_mcc": 1 - val_mcc, "train_mcc": 1 - train_mcc, "pred_time": pre
    }

xgb_automl = AutoML()
settings = {
    "time_budget": 60,  # total running time in seconds
    "metric": custom_metric, #'ap',
    "estimator_list": ['xgboost'],  # list of ML learners; we tune XGBoost i
    "task": 'classification',  # task type
    "seed": 42,  # random seed
}
xgb_automl.fit(X_train=X_train, y_train=y_train, **settings)
```

```
[flaml.automl.automl: 06-18 23:34:38] {2726} INFO - task = classification
[flaml.automl.automl: 06-18 23:34:38] {2728} INFO - Data split method: strat
ified
[flaml.automl.automl: 06-18 23:34:38] {2731} INFO - Evaluation method: cv
[flaml.automl.automl: 06-18 23:34:38] {2858} INFO - Minimizing error metric:
customized metric
[flaml.automl.automl: 06-18 23:34:38] {3004} INFO - List of ML learners in A
utoML Run: ['xgboost']
[flaml.automl.automl: 06-18 23:34:38] {3334} INFO - iteration 0, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:40] {3472} INFO - Estimated sufficient tim
e budget=13640s. Estimated necessary time budget=14s.
[flaml.automl.automl: 06-18 23:34:40] {3519} INFO -  at 1.4s,   estimator xg
boost's best error=0.2819,      best estimator xgboost's best error=0.2819
[flaml.automl.automl: 06-18 23:34:40] {3334} INFO - iteration 1, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:40] {3519} INFO -  at 2.1s,   estimator xg
boost's best error=0.2819,      best estimator xgboost's best error=0.2819
[flaml.automl.automl: 06-18 23:34:40] {3334} INFO - iteration 2, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:41] {3519} INFO -  at 2.6s,   estimator xg
boost's best error=0.0913,      best estimator xgboost's best error=0.0913
[flaml.automl.automl: 06-18 23:34:41] {3334} INFO - iteration 3, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:41] {3519} INFO -  at 3.2s,   estimator xg
boost's best error=0.0913,      best estimator xgboost's best error=0.0913
[flaml.automl.automl: 06-18 23:34:41] {3334} INFO - iteration 4, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:42] {3519} INFO -  at 3.9s,   estimator xg
boost's best error=0.0778,      best estimator xgboost's best error=0.0778
[flaml.automl.automl: 06-18 23:34:42] {3334} INFO - iteration 5, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:43] {3519} INFO -  at 4.8s,   estimator xg
boost's best error=0.0778,      best estimator xgboost's best error=0.0778
[flaml.automl.automl: 06-18 23:34:43] {3334} INFO - iteration 6, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:45] {3519} INFO -  at 6.7s,   estimator xg
boost's best error=0.0778,      best estimator xgboost's best error=0.0778
[flaml.automl.automl: 06-18 23:34:45] {3334} INFO - iteration 7, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:47] {3519} INFO -  at 9.3s,   estimator xg
boost's best error=0.0758,      best estimator xgboost's best error=0.0758
[flaml.automl.automl: 06-18 23:34:47] {3334} INFO - iteration 8, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:48] {3519} INFO -  at 10.0s,  estimator xg
boost's best error=0.0733,      best estimator xgboost's best error=0.0733
[flaml.automl.automl: 06-18 23:34:48] {3334} INFO - iteration 9, current lea
rner xgboost
[flaml.automl.automl: 06-18 23:34:49] {3519} INFO -  at 10.6s,  estimator xg
boost's best error=0.0733,      best estimator xgboost's best error=0.0733
[flaml.automl.automl: 06-18 23:34:49] {3334} INFO - iteration 10, current le
arner xgboost
[flaml.automl.automl: 06-18 23:34:50] {3519} INFO -  at 11.4s,  estimator xg
boost's best error=0.0733,      best estimator xgboost's best error=0.0733
[flaml.automl.automl: 06-18 23:34:50] {3334} INFO - iteration 11, current le
arner xgboost
[flaml.automl.automl: 06-18 23:34:50] {3519} INFO -  at 11.9s,  estimator xg
boost's best error=0.0733,      best estimator xgboost's best error=0.0733
[flaml.automl.automl: 06-18 23:34:50] {3334} INFO - iteration 12, current le
arner xgboost
[flaml.automl.automl: 06-18 23:34:51] {3519} INFO -  at 12.8s,  estimator xg
boost's best error=0.0733,      best estimator xgboost's best error=0.0733
[flaml.automl.automl: 06-18 23:34:51] {3334} INFO - iteration 13, current le
arner xgboost
```

```
[flaml.automl.automl: 06-18 23:34:52] {3519} INFO -  at 13.3s,  estimator xg
boost's best error=0.0733,      best estimator xgboost's best error=0.0733
[flaml.automl.automl: 06-18 23:34:52] {3334} INFO - iteration 14, current le
arner xgboost
[flaml.automl.automl: 06-18 23:34:52] {3519} INFO -  at 14.3s,  estimator xg
boost's best error=0.0727,      best estimator xgboost's best error=0.0727
[flaml.automl.automl: 06-18 23:34:53] {3334} INFO - iteration 15, current le
arner xgboost
[flaml.automl.automl: 06-18 23:34:53] {3519} INFO -  at 15.1s,  estimator xg
boost's best error=0.0727,      best estimator xgboost's best error=0.0727
[flaml.automl.automl: 06-18 23:34:53] {3334} INFO - iteration 16, current le
arner xgboost
[flaml.automl.automl: 06-18 23:34:56] {3519} INFO -  at 18.2s,  estimator xg
boost's best error=0.0720,      best estimator xgboost's best error=0.0720
[flaml.automl.automl: 06-18 23:34:56] {3334} INFO - iteration 17, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:05] {3519} INFO -  at 27.0s,  estimator xg
boost's best error=0.0720,      best estimator xgboost's best error=0.0720
[flaml.automl.automl: 06-18 23:35:05] {3334} INFO - iteration 18, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:10] {3519} INFO -  at 31.3s,  estimator xg
boost's best error=0.0720,      best estimator xgboost's best error=0.0720
[flaml.automl.automl: 06-18 23:35:10] {3334} INFO - iteration 19, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:16] {3519} INFO -  at 37.4s,  estimator xg
boost's best error=0.0720,      best estimator xgboost's best error=0.0720
[flaml.automl.automl: 06-18 23:35:16] {3334} INFO - iteration 20, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:18] {3519} INFO -  at 39.4s,  estimator xg
boost's best error=0.0676,      best estimator xgboost's best error=0.0676
[flaml.automl.automl: 06-18 23:35:18] {3334} INFO - iteration 21, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:20] {3519} INFO -  at 41.7s,  estimator xg
boost's best error=0.0676,      best estimator xgboost's best error=0.0676
[flaml.automl.automl: 06-18 23:35:20] {3334} INFO - iteration 22, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:24] {3519} INFO -  at 46.1s,  estimator xg
boost's best error=0.0676,      best estimator xgboost's best error=0.0676
[flaml.automl.automl: 06-18 23:35:24] {3334} INFO - iteration 23, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:27] {3519} INFO -  at 48.8s,  estimator xg
boost's best error=0.0636,      best estimator xgboost's best error=0.0636
[flaml.automl.automl: 06-18 23:35:27] {3334} INFO - iteration 24, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:30] {3519} INFO -  at 52.0s,  estimator xg
boost's best error=0.0636,      best estimator xgboost's best error=0.0636
[flaml.automl.automl: 06-18 23:35:30] {3334} INFO - iteration 25, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:31] {3519} INFO -  at 52.9s,  estimator xg
boost's best error=0.0636,      best estimator xgboost's best error=0.0636
[flaml.automl.automl: 06-18 23:35:31] {3334} INFO - iteration 26, current le
arner xgboost
[flaml.automl.automl: 06-18 23:35:38] {3519} INFO -  at 59.6s,  estimator xg
boost's best error=0.0636,      best estimator xgboost's best error=0.0636
[flaml.automl.automl: 06-18 23:35:38] {3783} INFO - retrain xgboost for 0.6s
[flaml.automl.automl: 06-18 23:35:38] {3790} INFO - retrained model: XGBClas
sifier(base_score=None, booster=None, callbacks=[],
              colsample_bylevel=0.7336308181625548, colsample_bynode=None,
              colsample_bytree=0.9855946937981651, early_stopping_rounds=Non
e,
              enable_categorical=False, eval_metric=None, feature_types=Non
e,
              gamma=None, gpu_id=None, grow_policy='lossguide',
              importance_type=None, interaction_constraints=None,
```

```
                learning_rate=0.08807525438906838, max_bin=None,
                max_cat_threshold=None, max_cat_to_onehot=None,
                max_delta_step=None, max_depth=0, max_leaves=17,
                min_child_weight=0.05257409142517566, missing=nan,
                monotone_constraints=None, n_estimators=38, n_jobs=-1,
                num_parallel_tree=None, predictor=None, random_state=None,
...)
[flaml.automl.automl: 06-18 23:35:38] {3034} INFO - fit succeeded
[flaml.automl.automl: 06-18 23:35:38] {3035} INFO - Time taken to find the b
est model: 48.797459840774536
```

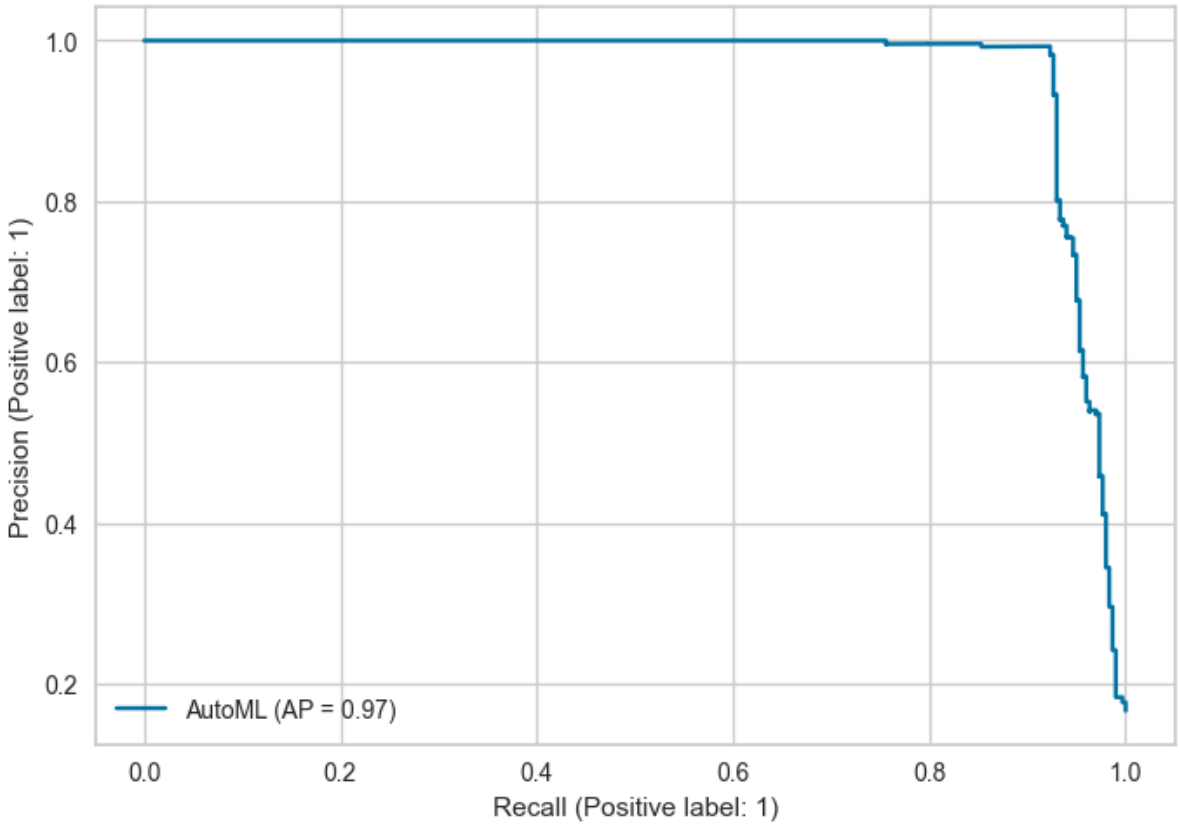
In [ ]:
```
'''retrieve best config and best learner'''
print('Best ML leaner:', xgb_automl.best_estimator)
print('Best hyperparmeter config:', xgb_automl.best_config)
print('Best score on validation data: {0:.4g}'.format(1-xgb_automl.best_loss
print('Training duration of best run: {0:.4g} s'.format(xgb_automl.best_conf
```

```
Best ML leaner: xgboost
Best hyperparmeter config: {'n_estimators': 38, 'max_leaves': 17, 'min_child
_weight': 0.05257409142517566, 'learning_rate': 0.08807525438906838, 'subsam
ple': 1.0, 'colsample_bylevel': 0.7336308181625548, 'colsample_bytree': 0.98
55946937981651, 'reg_alpha': 0.0009765625, 'reg_lambda': 0.0217946487617953
6}
Best score on validation data: 0.9364
Training duration of best run: 0.5634 s
```

In [ ]:
```
print("XGB-Tuned MODEL")
eval_metric(xgb_automl, X_train, y_train, X_test, y_test)
```

```
XGB-Tuned MODEL
Test_Set
[[1498    2]
 [  23  276]]
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      1500
           1       0.99      0.92      0.96       299

    accuracy                           0.99      1799
   macro avg       0.99      0.96      0.97      1799
weighted avg       0.99      0.99      0.99      1799


Train_Set
[[8483   17]
 [ 134 1558]]
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      8500
           1       0.99      0.92      0.95      1692

    accuracy                           0.99     10192
   macro avg       0.99      0.96      0.97     10192
weighted avg       0.99      0.99      0.98     10192
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay, average_precision_score

PrecisionRecallDisplay.from_estimator(xgb_automl,
                                      X_test,
                                      y_test);
```




In [ ]:
```
y_pred = xgb_automl.predict(X_test)
y_pred_proba= xgb_automl.predict_proba(X_test)[:,1]

xgba_AP = average_precision_score(y_test, y_pred_proba)
xgba_rec = recall_score(y_test, y_pred)
xgba_f1 = f1_score(y_test,y_pred)
xgba_matthews =  matthews_corrcoef(y_test, y_pred)
xgba_time =  measure_prediction_time(xgb_automl,X_test,10)

xgba_AP,xgba_rec,xgba_f1,xgba_matthews,xgba_time
```

Out[ ]:
```
(0.9651108440394259,
 0.9230769230769231,
 0.9566724436741768,
 0.9493018881846335,
 0.11970605850219726)
```

**Result for XGBoost**:

Based on the results above, both the XGBoost model's test and train sets show high precision, recall, and F1-score for both classes. The model performs well in correctly identifying employees who have churned (class 1), with relatively low false positive and false negative rates. The accuracy of the model is also high, indicating that it provides accurate predictions overall.

### 5.2.7 LightGBM

LightGBM is a gradient boosting framework that is designed to be efficient and scalable. It uses a tree-based learning algorithm similar to XGBoost but with some differences in implementation. Here's a simplified explanation of LightGBM:

LightGBM uses the concept of leaf-wise tree growth instead of level-wise tree growth like other gradient boosting algorithms. This allows it to grow trees more efficiently and

achieve better performance with less memory usage.

**The main parameters of LightGBM can be categorized into two groups**: general parameters, boosting parameters

1) **General Parameters**:

- **num_iterations**: The number of boosting iterations (or trees) to build.
- **learning_rate**: Controls the step size at each boosting iteration. A lower learning rate requires more boosting rounds but can yield better generalization.
- **num_leaves**: The maximum number of leaves (terminal nodes) in each tree. It determines the complexity and depth of the trees.
- **feature_fraction**: The fraction of features (columns) used for training each tree. It helps control the diversity of the features considered in each tree.
- **bagging_fraction**: The fraction of training data used for training each tree. It helps combat overfitting by introducing randomness into the training process.

2) **Boosting Parameters**:

- **boosting_type**: The type of boosting algorithm to use, such as "gbdt" for traditional gradient boosting or "dart" for a dropout-based variant.
- **max_depth**: The maximum depth of each tree. It determines the complexity of individual trees and affects the model's ability to capture interactions.
- **min_child_samples**: The minimum number of samples required to form a new leaf. It helps control the tree's complexity and can improve robustness.

In [ ]:
```
from lightgbm import LGBMClassifier
```

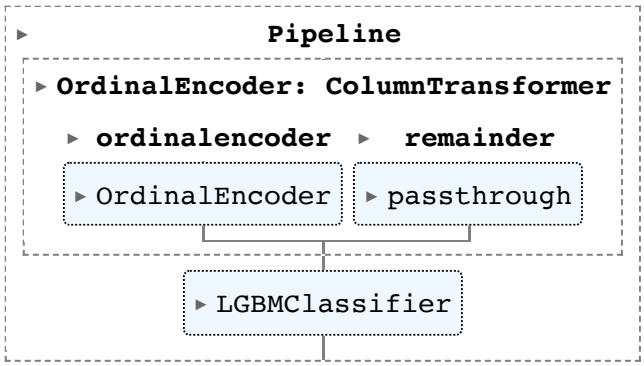
In [ ]:
```
cat_ordinal = ['salary','departments']
```

In [ ]:
```
enc_ordinal = OrdinalEncoder(handle_unknown='use_encoded_value'
                             ,unknown_value=-1)

column_trans = make_column_transformer((enc_ordinal, cat_ordinal),
                                       remainder='passthrough',
                                       verbose_feature_names_out=False).set
```

In [ ]:
```
from sklearn.utils import class_weight
classes_weights = class_weight.compute_sample_weight(class_weight='balanced'
```

In [ ]:
```
operations_lgbm = [("OrdinalEncoder", column_trans),
                   ("lgbm_model", LGBMClassifier(random_state=101, max_depth

lgbm_pipe = Pipeline(steps=operations_lgbm)

params_pipe_lgbm = {
    'lgbm_model__sample_weight': classes_weights,
    # 'lgbm_model__early_stopping_rounds': 10,   # Number of rounds with no
    # 'lgbm_model__eval_metric':'logloss',      # Evaluation metric for earl
    # 'lgbm_model__eval_set':[(X_test, y_test)]
}

# Fit the pipeline on the training data
lgbm_pipe.fit(X_train, y_train,**params_pipe_lgbm)
```

Out[ ]:




In [ ]:
```
print("LGBM MODEL")
eval_metric(lgbm_pipe, X_train, y_train, X_test, y_test)
```

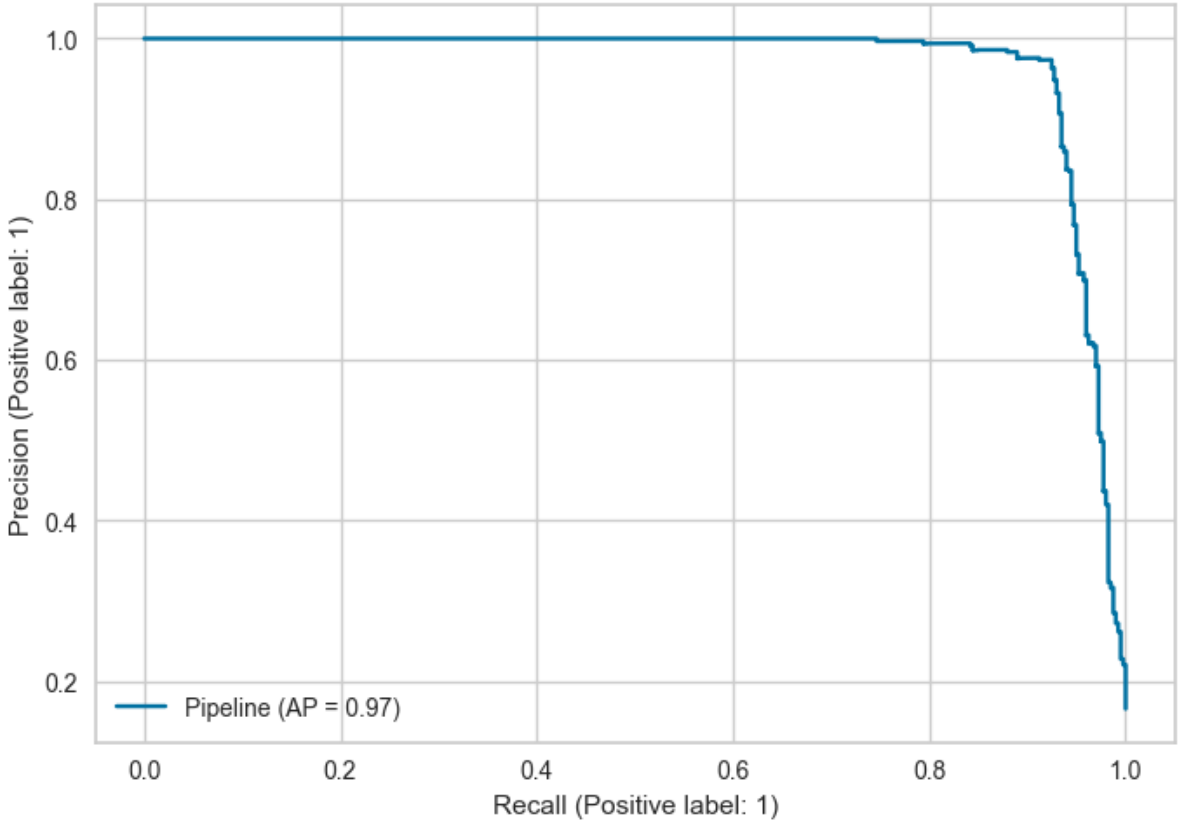
```
LGBM MODEL
Test_Set
[[1950   51]
 [  26  372]]
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      2001
           1       0.88      0.93      0.91       398

    accuracy                           0.97      2399
   macro avg       0.93      0.95      0.94      2399
weighted avg       0.97      0.97      0.97      2399


Train_Set
[[7825  174]
 [  93 1500]]
              precision    recall  f1-score   support

           0       0.99      0.98      0.98      7999
           1       0.90      0.94      0.92      1593

    accuracy                           0.97      9592
   macro avg       0.94      0.96      0.95      9592
weighted avg       0.97      0.97      0.97      9592
```

In [ ]:
```
# Define the operations in the pipeline
operations_lgbm = [
    ("OrdinalEncoder", OrdinalEncoder()),
    ("lgbm_model", LGBMClassifier(random_state=101, max_depth=3))
]

# Create the pipeline
pipe_model = Pipeline(steps=operations_lgbm)

# Defining fit params for cross-validation
lgbm_pipe_params = {'lgbm_model__sample_weight':classes_weights}

# Define the scoring metric
scoring= ["accuracy", "f1", "recall", "precision"]

# Perform cross-validation
scores = cross_validate(pipe_model, X_train, y_train, scoring=scoring, cv=5,

# Create a DataFrame to store the scores
df_scores = pd.DataFrame(scores)

# Calculate the mean scores
```

```
mean_scores = df_scores.mean()[2:]

# Print the mean scores
print("Mean cross-validation scores:")
print(mean_scores)
```

```
Mean cross-validation scores:
test_accuracy      0.971330
train_accuracy     0.972946
test_f1            0.915567
train_f1           0.920305
test_recall        0.935969
train_recall       0.940521
test_precision     0.896102
train_precision    0.900946
dtype: float64
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay, average_precision_score

PrecisionRecallDisplay.from_estimator(lgbm_pipe,
                                      X_test,
                                      y_test);
```




In [ ]:
```
y_pred = lgbm_pipe.predict(X_test)
y_pred_proba= lgbm_pipe.predict_proba(X_test)[:,1]

lgbm_AP = average_precision_score(y_test, y_pred_proba)
lgbm_rec = recall_score(y_test, y_pred)
lgbm_f1 = f1_score(y_test,y_pred)
lgbm_matthews =  matthews_corrcoef(y_test, y_pred)
lgbm_time =  measure_prediction_time(lgbm_pipe,X_test)

lgbm_AP,lgbm_rec,lgbm_f1,lgbm_matthews,lgbm_time
```

Out[ ]:
```
(0.9688742864896365,
 0.9346733668341709,
 0.9062119366626066,
 0.8874710991315558,
 0.026342129707336424)
```

**LightGBM Hyper-Parameter Tuning**

https://colab.research.google.com/github/microsoft/FLAML/blob/main/notebook/automl_clas
UPE17HU7jO

In [ ]:
```
# custom_metric in overfiti engelledigini gorduk ancak ap metrigi daha iyi s
def custom_metric(X_val, y_val, estimator, labels, X_train, y_train,
                  weight_val=None, weight_train=None, config=None,
                  groups_val=None, groups_train=None):
    from sklearn.metrics import log_loss
    import time
    start = time.time()
    y_pred = estimator.predict_proba(X_val)
    pred_time = (time.time() - start) / len(X_val)
    val_loss = log_loss(y_val, y_pred, labels=labels,
                        sample_weight=weight_val)
    y_pred = estimator.predict_proba(X_train)
    train_loss = log_loss(y_train, y_pred, labels=labels,
                          sample_weight=weight_train)
    alpha = 0.5
    return val_loss * (1 + alpha) - alpha * train_loss, {
        "val_loss": val_loss, "train_loss": train_loss, "pred_time": pred_ti
    }
    # two elements are returned:
    # the first element is the metric to minimize as a float number,
    # the second element is a dictionary of the metrics to log
```

In [ ]:
```
from flaml import AutoML

lgbm_automl = AutoML()
settings = {
    "time_budget": 60,  # total running time in seconds
    "metric": 'ap',
    "estimator_list": ['lgbm'],  # list of ML learners; we tune LightGBM in
    "task": 'classification',  # task type
    "seed": 42,  # random seed
}

# settings["custom_hp"] = {
#     "lgbm": {
#         "class_weight": {
#             "domain": "balanced",
#             "init_value": "balanced"
#         }
#     }
# }
lgbm_automl.fit(X_train=X_train, y_train=y_train, **settings)
```

```
[flaml.automl.automl: 06-18 21:22:36] {2726} INFO - task = classification
[flaml.automl.automl: 06-18 21:22:36] {2728} INFO - Data split method: strat
ified
[flaml.automl.automl: 06-18 21:22:36] {2731} INFO - Evaluation method: cv
[flaml.automl.automl: 06-18 21:22:36] {2858} INFO - Minimizing error metric:
1-ap
[flaml.automl.automl: 06-18 21:22:36] {3004} INFO - List of ML learners in A
utoML Run: ['lgbm']
[flaml.automl.automl: 06-18 21:22:36] {3334} INFO - iteration 0, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:36] {3472} INFO - Estimated sufficient tim
e budget=2237s. Estimated necessary time budget=2s.
[flaml.automl.automl: 06-18 21:22:36] {3519} INFO -  at 0.3s,   estimator lg
bm's best error=0.1605, best estimator lgbm's best error=0.1605
[flaml.automl.automl: 06-18 21:22:36] {3334} INFO - iteration 1, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:36] {3519} INFO -  at 0.6s,   estimator lg
bm's best error=0.1605, best estimator lgbm's best error=0.1605
[flaml.automl.automl: 06-18 21:22:36] {3334} INFO - iteration 2, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:37] {3519} INFO -  at 0.8s,   estimator lg
bm's best error=0.0604, best estimator lgbm's best error=0.0604
[flaml.automl.automl: 06-18 21:22:37] {3334} INFO - iteration 3, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:37] {3519} INFO -  at 1.4s,   estimator lg
bm's best error=0.0478, best estimator lgbm's best error=0.0478
[flaml.automl.automl: 06-18 21:22:37] {3334} INFO - iteration 4, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:38] {3519} INFO -  at 1.8s,   estimator lg
bm's best error=0.0478, best estimator lgbm's best error=0.0478
[flaml.automl.automl: 06-18 21:22:38] {3334} INFO - iteration 5, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:38] {3519} INFO -  at 2.2s,   estimator lg
bm's best error=0.0427, best estimator lgbm's best error=0.0427
[flaml.automl.automl: 06-18 21:22:38] {3334} INFO - iteration 6, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:38] {3519} INFO -  at 2.7s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:38] {3334} INFO - iteration 7, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:39] {3519} INFO -  at 3.1s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:39] {3334} INFO - iteration 8, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:40] {3519} INFO -  at 3.8s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:40] {3334} INFO - iteration 9, current lea
rner lgbm
[flaml.automl.automl: 06-18 21:22:40] {3519} INFO -  at 4.1s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:40] {3334} INFO - iteration 10, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:40] {3519} INFO -  at 4.7s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:40] {3334} INFO - iteration 11, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:41] {3519} INFO -  at 4.8s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:41] {3334} INFO - iteration 12, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:41] {3519} INFO -  at 5.0s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:41] {3334} INFO - iteration 13, current le
arner lgbm
```

```
[flaml.automl.automl: 06-18 21:22:41] {3519} INFO -  at 5.5s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:41] {3334} INFO - iteration 14, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:42] {3519} INFO -  at 6.4s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:42] {3334} INFO - iteration 15, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:42] {3519} INFO -  at 6.6s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:42] {3334} INFO - iteration 16, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:43] {3519} INFO -  at 6.7s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:43] {3334} INFO - iteration 17, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:43] {3519} INFO -  at 7.6s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:43] {3334} INFO - iteration 18, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:44] {3519} INFO -  at 8.4s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:44] {3334} INFO - iteration 19, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:44] {3519} INFO -  at 8.5s,   estimator lg
bm's best error=0.0323, best estimator lgbm's best error=0.0323
[flaml.automl.automl: 06-18 21:22:44] {3334} INFO - iteration 20, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:45] {3519} INFO -  at 8.8s,   estimator lg
bm's best error=0.0303, best estimator lgbm's best error=0.0303
[flaml.automl.automl: 06-18 21:22:45] {3334} INFO - iteration 21, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:45] {3519} INFO -  at 9.1s,   estimator lg
bm's best error=0.0303, best estimator lgbm's best error=0.0303
[flaml.automl.automl: 06-18 21:22:45] {3334} INFO - iteration 22, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:45] {3519} INFO -  at 9.2s,   estimator lg
bm's best error=0.0303, best estimator lgbm's best error=0.0303
[flaml.automl.automl: 06-18 21:22:45] {3334} INFO - iteration 23, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:46] {3519} INFO -  at 9.9s,   estimator lg
bm's best error=0.0303, best estimator lgbm's best error=0.0303
[flaml.automl.automl: 06-18 21:22:46] {3334} INFO - iteration 24, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:46] {3519} INFO -  at 10.1s,  estimator lg
bm's best error=0.0303, best estimator lgbm's best error=0.0303
[flaml.automl.automl: 06-18 21:22:46] {3334} INFO - iteration 25, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:46] {3519} INFO -  at 10.4s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:46] {3334} INFO - iteration 26, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:46] {3519} INFO -  at 10.6s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:46] {3334} INFO - iteration 27, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:47] {3519} INFO -  at 11.0s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:47] {3334} INFO - iteration 28, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:47] {3519} INFO -  at 11.3s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:47] {3334} INFO - iteration 29, current le
arner lgbm
```

```
[flaml.automl.automl: 06-18 21:22:47] {3519} INFO -  at 11.6s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:47] {3334} INFO - iteration 30, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:48] {3519} INFO -  at 11.9s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:48] {3334} INFO - iteration 31, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:48] {3519} INFO -  at 12.1s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:48] {3334} INFO - iteration 32, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:50] {3519} INFO -  at 14.0s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:50] {3334} INFO - iteration 33, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:50] {3519} INFO -  at 14.2s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:50] {3334} INFO - iteration 34, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:50] {3519} INFO -  at 14.4s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:50] {3334} INFO - iteration 35, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:51] {3519} INFO -  at 14.9s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:51] {3334} INFO - iteration 36, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:51] {3519} INFO -  at 15.3s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:51] {3334} INFO - iteration 37, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:51] {3519} INFO -  at 15.5s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:51] {3334} INFO - iteration 38, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:52] {3519} INFO -  at 15.8s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:52] {3334} INFO - iteration 39, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:52] {3519} INFO -  at 16.1s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:52] {3334} INFO - iteration 40, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:52] {3519} INFO -  at 16.3s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:52] {3334} INFO - iteration 41, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:52] {3519} INFO -  at 16.6s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:52] {3334} INFO - iteration 42, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:54] {3519} INFO -  at 18.4s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:54] {3334} INFO - iteration 43, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:54] {3519} INFO -  at 18.6s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:54] {3334} INFO - iteration 44, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:55] {3519} INFO -  at 18.9s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:55] {3334} INFO - iteration 45, current le
arner lgbm
```

```
[flaml.automl.automl: 06-18 21:22:55] {3519} INFO -  at 19.1s,  estimator lg
bm's best error=0.0298, best estimator lgbm's best error=0.0298
[flaml.automl.automl: 06-18 21:22:55] {3334} INFO - iteration 46, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:55] {3519} INFO -  at 19.7s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:22:55] {3334} INFO - iteration 47, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:56] {3519} INFO -  at 20.0s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:22:56] {3334} INFO - iteration 48, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:58] {3519} INFO -  at 22.2s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:22:58] {3334} INFO - iteration 49, current le
arner lgbm
[flaml.automl.automl: 06-18 21:22:58] {3519} INFO -  at 22.4s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:22:58] {3334} INFO - iteration 50, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:02] {3519} INFO -  at 26.1s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:02] {3334} INFO - iteration 51, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:02] {3519} INFO -  at 26.3s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:02] {3334} INFO - iteration 52, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:03] {3519} INFO -  at 27.0s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:03] {3334} INFO - iteration 53, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:03] {3519} INFO -  at 27.4s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:03] {3334} INFO - iteration 54, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:04] {3519} INFO -  at 27.8s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:04] {3334} INFO - iteration 55, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:06] {3519} INFO -  at 30.2s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:06] {3334} INFO - iteration 56, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:06] {3519} INFO -  at 30.5s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:06] {3334} INFO - iteration 57, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:08] {3519} INFO -  at 32.6s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:08] {3334} INFO - iteration 58, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:10] {3519} INFO -  at 34.5s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:10] {3334} INFO - iteration 59, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:11] {3519} INFO -  at 34.8s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:11] {3334} INFO - iteration 60, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:11] {3519} INFO -  at 35.5s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:11] {3334} INFO - iteration 61, current le
arner lgbm
```

```
[flaml.automl.automl: 06-18 21:23:12] {3519} INFO -  at 35.9s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:12] {3334} INFO - iteration 62, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:12] {3519} INFO -  at 36.2s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:12] {3334} INFO - iteration 63, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:13] {3519} INFO -  at 37.0s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:13] {3334} INFO - iteration 64, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:13] {3519} INFO -  at 37.3s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:13] {3334} INFO - iteration 65, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:15] {3519} INFO -  at 39.3s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:15] {3334} INFO - iteration 66, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:16] {3519} INFO -  at 40.1s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:16] {3334} INFO - iteration 67, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:16] {3519} INFO -  at 40.5s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:16] {3334} INFO - iteration 68, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:19] {3519} INFO -  at 42.9s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:19] {3334} INFO - iteration 69, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:19] {3519} INFO -  at 43.2s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:19] {3334} INFO - iteration 70, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:20] {3519} INFO -  at 44.0s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:20] {3334} INFO - iteration 71, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:20] {3519} INFO -  at 44.4s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:20] {3334} INFO - iteration 72, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:20] {3519} INFO -  at 44.7s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:20] {3334} INFO - iteration 73, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:21] {3519} INFO -  at 45.7s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:21] {3334} INFO - iteration 74, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:22] {3519} INFO -  at 46.1s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:22] {3334} INFO - iteration 75, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:25] {3519} INFO -  at 49.0s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:25] {3334} INFO - iteration 76, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:25] {3519} INFO -  at 49.6s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:25] {3334} INFO - iteration 77, current le
arner lgbm
```

```
[flaml.automl.automl: 06-18 21:23:26] {3519} INFO -  at 50.1s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:26] {3334} INFO - iteration 78, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:26] {3519} INFO -  at 50.4s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:26] {3334} INFO - iteration 79, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:28] {3519} INFO -  at 52.1s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:28] {3334} INFO - iteration 80, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:29] {3519} INFO -  at 52.7s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:29] {3334} INFO - iteration 81, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:29] {3519} INFO -  at 53.2s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:29] {3334} INFO - iteration 82, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:30] {3519} INFO -  at 53.8s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:30] {3334} INFO - iteration 83, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:30] {3519} INFO -  at 54.5s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:30] {3334} INFO - iteration 84, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:31] {3519} INFO -  at 55.0s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:31] {3334} INFO - iteration 85, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:32] {3519} INFO -  at 55.8s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:32] {3334} INFO - iteration 86, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:34] {3519} INFO -  at 57.7s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:34] {3334} INFO - iteration 87, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:34] {3519} INFO -  at 58.1s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:34] {3334} INFO - iteration 88, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:35] {3519} INFO -  at 59.0s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:35] {3334} INFO - iteration 89, current le
arner lgbm
[flaml.automl.automl: 06-18 21:23:35] {3519} INFO -  at 59.7s,  estimator lg
bm's best error=0.0288, best estimator lgbm's best error=0.0288
[flaml.automl.automl: 06-18 21:23:36] {3783} INFO - retrain lgbm for 0.1s
[flaml.automl.automl: 06-18 21:23:36] {3790} INFO - retrained model: LGBMCla
ssifier(colsample_bytree=0.9941025833950202,
               learning_rate=0.06129323571151544, max_bin=127,
               min_child_samples=4, n_estimators=108, num_leaves=20,
               reg_alpha=0.009618999870314945, reg_lambda=0.0245565364377050
15,
               verbose=-1)
[flaml.automl.automl: 06-18 21:23:36] {3034} INFO - fit succeeded
[flaml.automl.automl: 06-18 21:23:36] {3035} INFO - Time taken to find the b
est model: 19.674317836761475
```

In [ ]:
```
'''retrieve best config and best learner'''
print('Best ML leaner:', lgbm_automl.best_estimator)
```

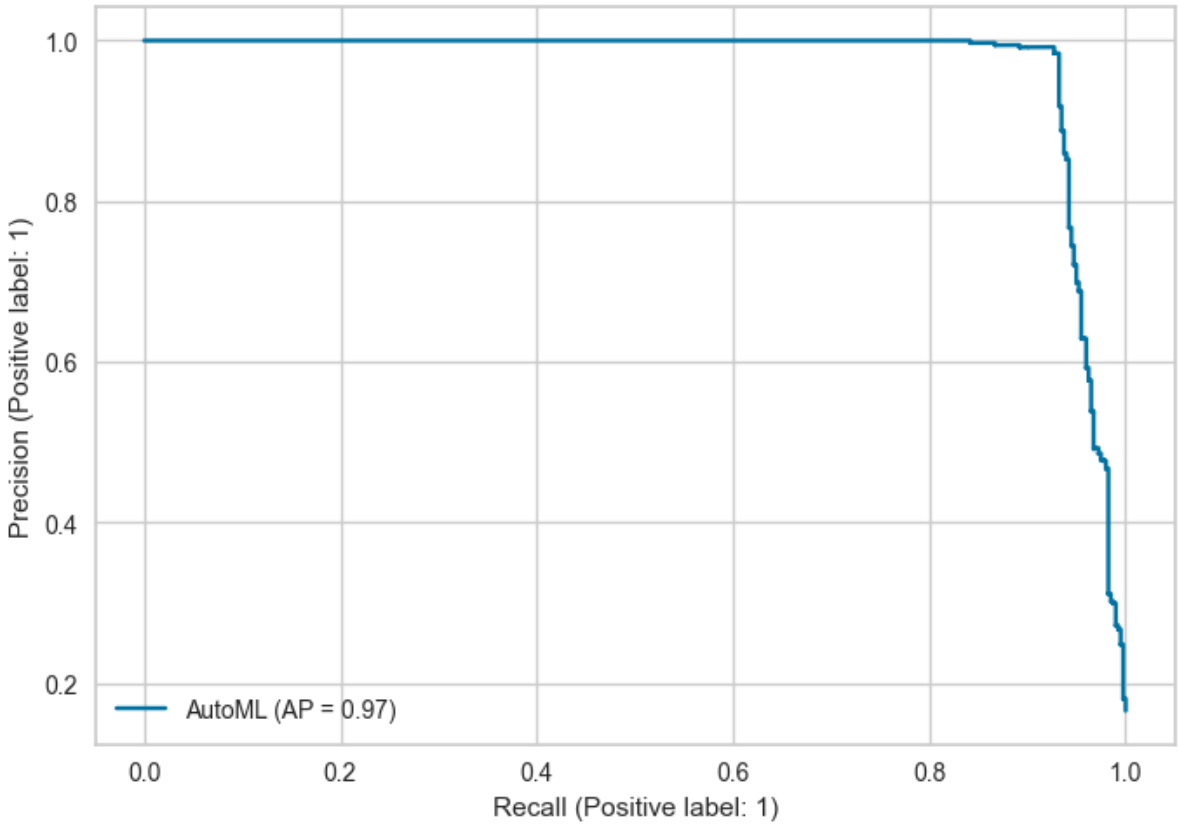
```
print('Best hyperparmeter config:', lgbm_automl.best_config)
print('Best accuracy on validation data: {0:.4g}'.format(1-lgbm_automl.best_
print('Training duration of best run: {0:.4g} s'.format(lgbm_automl.best_con
```

```
Best ML leaner: lgbm
Best hyperparmeter config: {'n_estimators': 108, 'num_leaves': 20, 'min_chil
d_samples': 4, 'learning_rate': 0.06129323571151544, 'log_max_bin': 7, 'cols
ample_bytree': 0.9941025833950202, 'reg_alpha': 0.009618999870314945, 'reg_l
ambda': 0.024556536437705015}
Best accuracy on validation data: 0.9712
Training duration of best run: 0.1143 s
```

In [ ]:
```
print("LGBM-Tuned MODEL")
eval_metric(lgbm_automl, X_train, y_train, X_test, y_test)
```

```
LGBM-Tuned MODEL
Test_Set
[[1994    7]
 [  27  371]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      2001
           1       0.98      0.93      0.96       398

    accuracy                           0.99      2399
   macro avg       0.98      0.96      0.97      2399
weighted avg       0.99      0.99      0.99      2399


Train_Set
[[7981   18]
 [ 102 1491]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      7999
           1       0.99      0.94      0.96      1593

    accuracy                           0.99      9592
   macro avg       0.99      0.97      0.98      9592
weighted avg       0.99      0.99      0.99      9592
```

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay, average_precision_score

PrecisionRecallDisplay.from_estimator(lgbm_automl,
                                      X_test,
                                      y_test);
```




In [ ]:
```
y_pred = lgbm_automl.predict(X_test)
y_pred_proba= lgbm_automl.predict_proba(X_test)[:,1]

lgbma_AP = average_precision_score(y_test, y_pred_proba)
lgbma_rec = recall_score(y_test, y_pred)
lgbma_f1 = f1_score(y_test,y_pred)
lgbma_matthews = matthews_corrcoef(y_test, y_pred)
lgbma_time =  measure_prediction_time(lgbm_automl,X_test)

lgbma_AP,lgbma_rec,lgbma_f1, lgbma_matthews,lgbma_time
```

Out[ ]:
```
(0.9689789266363271,
 0.9321608040201005,
 0.9561855670103092,
 0.9481869141004056,
 0.0615170955657959)
```

**RESULT for LightGBM**

As seen above, the general scores of the light gbm is very high, similar to catboost and xgboost models.

These scores indicate that the LGBM-tuned model is highly accurate in identifying churned employees (high precision), captures a significant portion of actual churned employees (high recall), and maintains a good balance between precision and recall (high F1-score). This suggests that the model can effectively predict which employees are likely to churn, enabling proactive measures to be taken to retain valuable employees and reduce attrition.

### 5.2.8 DEEP LEARNING - ANN

Deep learning refers to a subset of machine learning algorithms that are based on artificial neural networks (ANNs). ANNs are computational models inspired by the

structure and functioning of the human brain. Deep learning ANN, or deep neural network (DNN), is an ANN with multiple layers of interconnected neurons.

**How a deep learning ANN works**:

- **Architecture**: A deep learning ANN consists of an input layer, one or more hidden layers, and an output layer. Each layer is composed of multiple artificial neurons, also called nodes or units. Neurons in one layer are connected to neurons in the next layer through weighted connections.
- **Forward Propagation**: The input data is fed into the input layer, and the values propagate forward through the network. Each neuron in a layer receives input from the previous layer, performs a computation using the weighted connections and an activation function, and passes the output to the next layer. This process continues until the output layer produces the final prediction.
- **Activation Function**: The activation function introduces non-linearities to the network, allowing it to learn complex patterns and relationships in the data. Common activation functions include sigmoid, tanh, and ReLU (Rectified Linear Unit).
- **Training**: The deep learning ANN is trained using a process called backpropagation. Initially, the network's weights are randomly initialized. The training data, along with their known target values, are used to compute the prediction error. The error is then backpropagated through the network, adjusting the weights based on the error gradient and an optimization algorithm (e.g., gradient descent). This process is repeated iteratively, updating the weights to minimize the prediction error.
- **Learning Features**: One of the key advantages of deep learning ANN is its ability to automatically learn features from raw data. Instead of relying on manual feature engineering, deep learning models can learn and extract relevant features from the data during the training process. This feature learning capability is particularly useful in complex tasks such as image and text analysis.
- **Predictions**: Once the deep learning ANN is trained, it can be used to make predictions on new, unseen data. The input data is fed into the trained network, and the output layer produces the predicted values or classes.

In [ ]:
```
#!pip install livelossplot
```

In [ ]:
```
# Importing libraries for DL and ML
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam

from sklearn.experimental import enable_halving_search_cv
# grid search libraries
from sklearn.model_selection import GridSearchCV,\
                                    HalvingGridSearchCV,\
                                    RandomizedSearchCV
from livelossplot import PlotLossesKerasTF
```

```
2023-06-18 21:23:53.702314: I tensorflow/core/platform/cpu_feature_guard.cc:
182] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
```

In [ ]:
```
# split the features and target
X = df.drop('left', axis=1)
y = df['left'].values
```

In [ ]:
```
# split the data as train, validation and test data
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    stratify=y,
                                                    test_size=0.15,
                                                    random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train,
                                                  y_train,
                                                  stratify=y_train,
                                                  test_size=0.1,
                                                  random_state=42)
```

In [ ]:
```
# prepare the features for encoding
cat_onehot = ['departments'] # onehot encoding for departments feature (no h
cat_ordinal = ['salary'] # ordinal encoding for salary bec of hierarchical c

cat_for_salary = ["low", "medium", "high"] # define the order among differen

# encode the abovementioned features
enc_onehot = OneHotEncoder(handle_unknown="ignore", sparse=False)
enc_ordinal = OrdinalEncoder(categories= [cat_for_salary])

column_trans = make_column_transformer((enc_onehot, cat_onehot),
                                       (enc_ordinal, cat_ordinal),
                                        remainder='passthrough',
                                        verbose_feature_names_out=False)
```

In [ ]:
```
X_train = column_trans.fit_transform(X_train) # fit_transform for train, tra
X_val = column_trans.transform(X_val)
X_test = column_trans.transform(X_test)
```

In [ ]:
```
scaler = MinMaxScaler() # Minmax scaler mostly used for deep learning as a r
```

In [ ]:
```
# scale the data; fit_transform for train, transform val and test
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

In [ ]:
```
# https://medium.com/analytics-vidhya/custom-metrics-for-keras-tensorflow-ae
# this custom metric ONLY works on binary classification problems.
from keras import backend as K

def matthews_correlation_coefficient(y_true, y_pred):
    tp = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    tn = K.sum(K.round(K.clip((1 - y_true) * (1 - y_pred), 0, 1)))
    fp = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
    fn = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    num = tp * tn - fp * fn
    den = (tp + fp) * (tp + fn) * (tn + fp) * (tn + fn)
    return num / K.sqrt(den + K.epsilon())
```

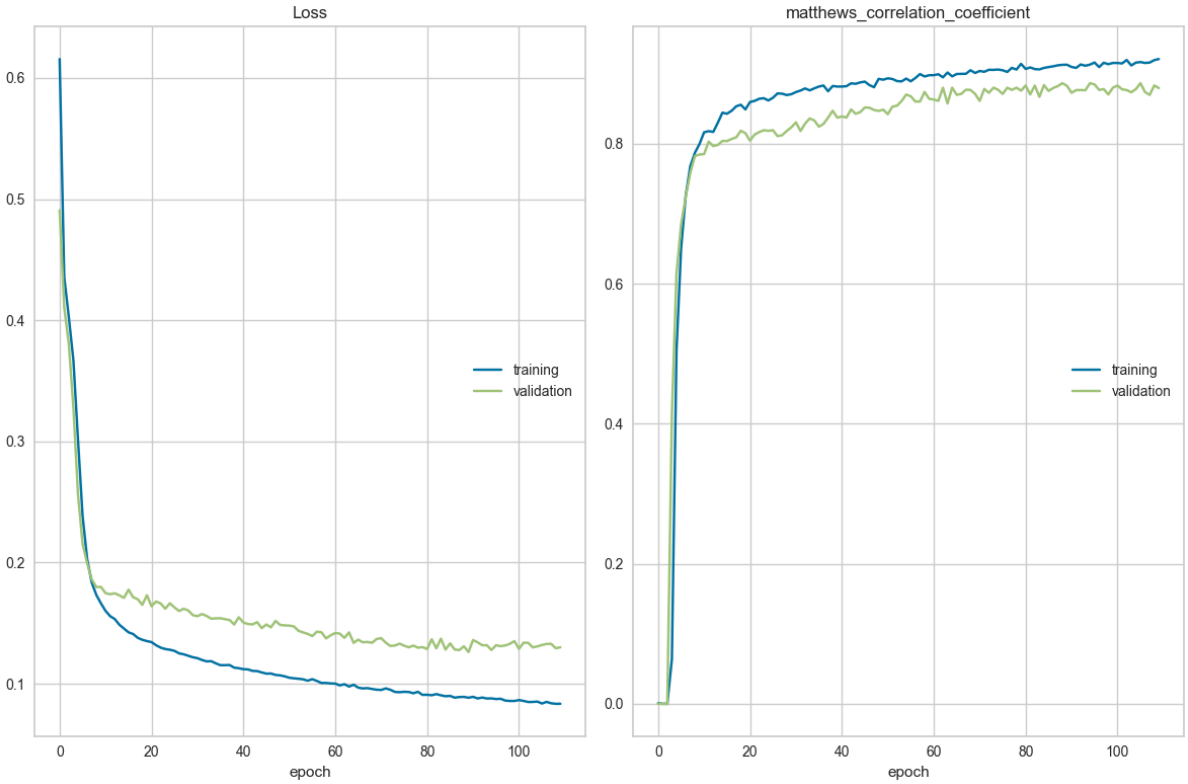
In [ ]:
```
# build the model by balancing the class weights
tf.random.set_seed(42)

model = Sequential()

model.add(Dense(28, activation="relu"))
model.add(Dense(14, activation="relu"))
model.add(Dense(8, activation="relu"))
model.add(Dense(1, activation="sigmoid"))

opt = Adam(learning_rate=0.001)
model.compile(optimizer=opt,
              loss="binary_crossentropy",
              metrics=[matthews_correlation_coefficient])
```

In [ ]:
```
early_stop = EarlyStopping(monitor="val_loss",
                           mode="auto",
                           verbose=1,
                           patience=20,
                           restore_best_weights = True)
```

In [ ]:
```
# import class_weight from sklearn to balance the classes
from sklearn.utils import class_weight

class_weights = class_weight.compute_class_weight('balanced',
                                                  classes=np.unique(y_train)
                                                  y=y_train)

class_weights = {0: class_weights[0], 1: class_weights[1]}
class_weights
```

Out[ ]:
```
{0: 0.5995554974506472, 1: 3.011162179908076}
```

In [ ]:
```
model.fit(x=X_train,
          y=y_train,
          validation_data=(X_val, y_val),
          batch_size=128,
          epochs=200,
          verbose=1,
          callbacks=[early_stop, PlotLossesKerasTF()])
          #class_weight=class_weights)
```

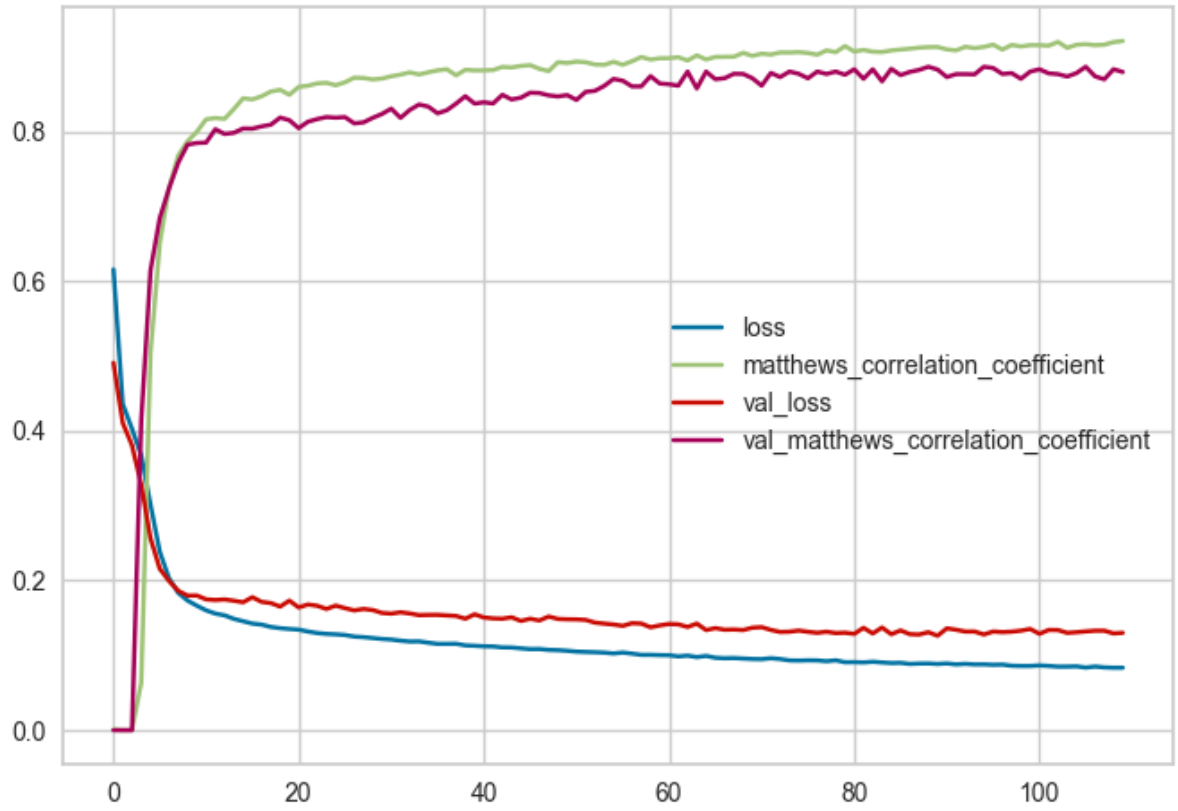



```
Loss
        training                (min:    0.083, max:    0.616, cur:    0.08
3)
        validation              (min:    0.126, max:    0.491, cur:    0.13
0)
matthews_correlation_coefficient
        training                (min:    0.000, max:    0.921, cur:    0.92
1)
        validation              (min:    0.000, max:    0.886, cur:    0.87
9)
72/72 [==============================] - 1s 11ms/step - loss: 0.0834 - matth
ews_correlation_coefficient: 0.9208 - val_loss: 0.1300 - val_matthews_correl
ation_coefficient: 0.8793
Epoch 110: early stopping
```

Out[ ]:
```
<keras.callbacks.History at 0x7fa05e6a2530>
```

In [ ]:
```
loss_df = pd.DataFrame(model.history.history)
loss_df.plot()
```

Out[ ]:
```
<Axes: >
```



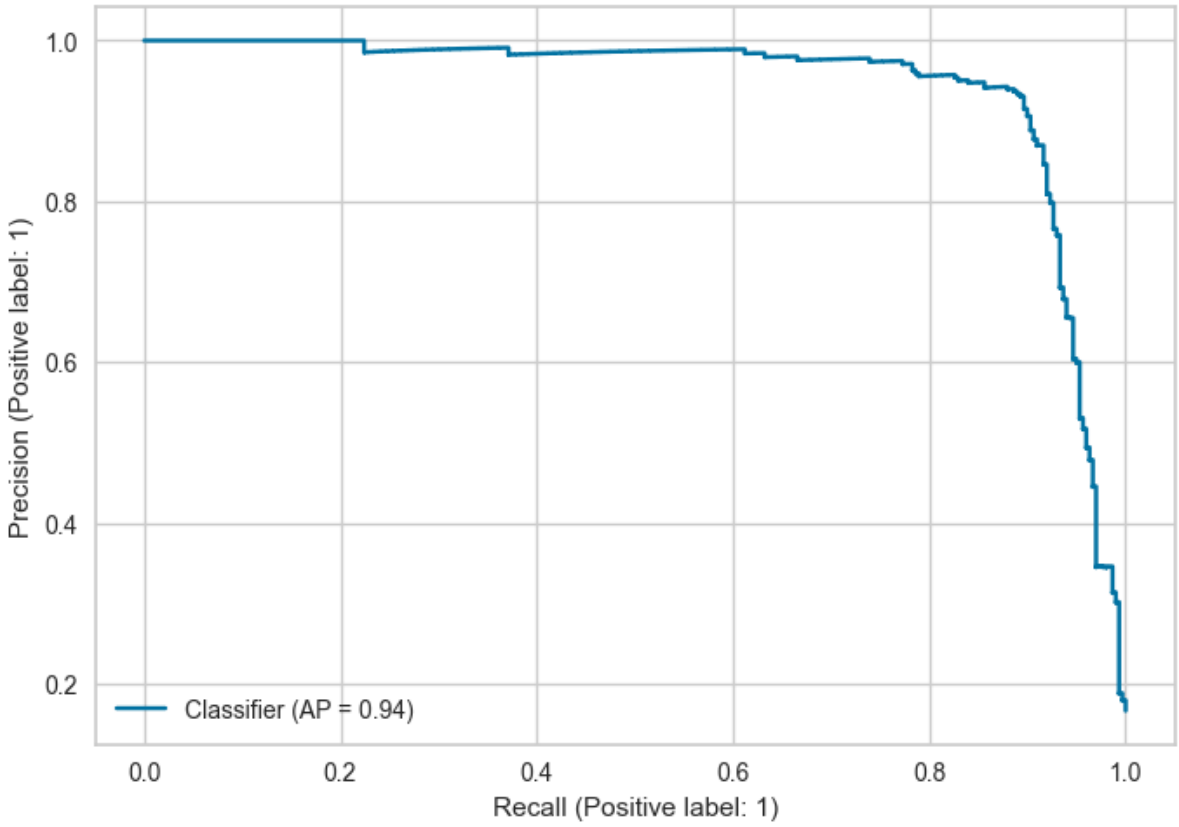

In [ ]:
```
y_pred = (model.predict(X_test) > .5).astype("int32")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
57/57 [==============================] - 1s 3ms/step
[[1484   16]
 [  41  258]]
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      1500
           1       0.94      0.86      0.90       299

    accuracy                           0.97      1799
   macro avg       0.96      0.93      0.94      1799
weighted avg       0.97      0.97      0.97      1799
```

In [ ]:
```
y_pred_proba = model.predict(X_test)
PrecisionRecallDisplay.from_predictions(y_test,y_pred_proba)
```

```
57/57 [==============================] - 0s 4ms/step
```

Out[ ]:
```
<sklearn.metrics._plot.precision_recall_curve.PrecisionRecallDisplay at 0x7f
a0650229b0>
```




In [ ]:
```
ANN_AP = average_precision_score(y_test, y_pred_proba)
ANN_f1 = f1_score(y_test, y_pred)
ANN_recall = recall_score(y_test, y_pred)
ANN_matthews =  matthews_corrcoef(y_test, y_pred)
ANN_time =  measure_prediction_time(model,X_test)
```

```
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 2ms/step
```

In [ ]:
```
ANN_AP, ANN_f1, ANN_recall, ANN_matthews, ANN_time
```

Out[ ]:
```
(0.943139598945624,
 0.900523560209424,
 0.862876254180602,
 0.8829120339475645,
 0.2628041982650757)
```

### Optuna (Hyperparameter optimization) for ANN

In [ ]:
```
import optuna
# optimizers for gradient descent to use in backpropagation

from tensorflow.keras.optimizers import Adam, Adadelta, RMSprop, Nadam, Adam
```

In [ ]:
```
# we should define functions for optuna
trial_metric = matthews_correlation_coefficient
batch_size = 128 # for a faster modelling increase the batch size
```

```
def create_model(trial):
    # Some hyperparameters we want to optimize
    n_units1 = trial.suggest_int('n_units1', 8, 128) # trial: to try between
    n_units2 = trial.suggest_int('n_units2', 8, 128)
    n_units3 = trial.suggest_int('n_units3', 8, 128)
    optimizer = trial.suggest_categorical("optimizer",
                                          [Adam, Adadelta, RMSprop, Nadam, A
    learning_rate = trial.suggest_loguniform('learning_rate', 1e-5, 1.3e-1)

    # n_units, optiizer and lr will be used below in the model layers

    tf.random.set_seed(42) # to compare, define seed
    model = Sequential()
    model.add(Dense(n_units1, input_dim=X_train.shape[1], activation='relu')
    model.add(Dense(n_units2, activation='relu'))
    model.add(Dense(n_units3, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer(learning_rate=learning_rate),
                  metrics=[trial_metric])
    return model


def objective(trial):  # we use create_model function here; class weights al
    model = create_model(trial)
    w0 = trial.suggest_loguniform("w0", 0.01, 5)
    w1 = trial.suggest_loguniform("w1", 0.01, 5)
    model.fit(X_train,
              y_train,
              validation_data=(X_val, y_val),
              batch_size=batch_size,
              epochs=100,
              callbacks=[early_stop],
              class_weight={0: w0, 1: w1},
              verbose=0)
    score = model.evaluate(X_test, y_test, verbose=0)[1]
    return score
```

In [ ]:

```
study = optuna.create_study(direction="maximize") # "direction maximize" mea
study.optimize(objective, n_trials=50) # objective function here # trial sho
study.best_params
```

```
[I 2023-06-18 21:25:58,741] A new study created in memory with name: no-name
-c064922e-a167-4a4d-aebc-c59ecc1ddfe1
Restoring model weights from the end of the best epoch: 76.
Epoch 96: early stopping
[I 2023-06-18 21:26:24,414] Trial 0 finished with value: 0.8821857571601868
and parameters: {'n_units1': 100, 'n_units2': 126, 'n_units3': 11, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.003028446
8333468192, 'w0': 0.1302163118357236, 'w1': 0.016904567329365627}. Best is t
rial 0 with value: 0.8821857571601868.
[I 2023-06-18 21:26:49,725] Trial 1 finished with value: 0.42268669605255127
and parameters: {'n_units1': 18, 'n_units2': 55, 'n_units3': 59, 'optimize
r': <class 'keras.optimizers.adadelta.Adadelta'>, 'learning_rate': 0.1291672
0971359752, 'w0': 0.015006675923280532, 'w1': 0.06096706985714558}. Best is
trial 0 with value: 0.8821857571601868.
[I 2023-06-18 21:27:15,311] Trial 2 finished with value: -0.0169100388884544
37 and parameters: {'n_units1': 26, 'n_units2': 86, 'n_units3': 127, 'optimi
zer': <class 'keras.optimizers.adadelta.Adadelta'>, 'learning_rate': 2.81079
90009164136e-05, 'w0': 0.13874862397274318, 'w1': 0.022443959266591443}. Bes
t is trial 0 with value: 0.8821857571601868.
Restoring model weights from the end of the best epoch: 33.
Epoch 53: early stopping
```

```
[I 2023-06-18 21:27:30,948] Trial 3 finished with value: 0.6261488199234009
and parameters: {'n_units1': 75, 'n_units2': 11, 'n_units3': 9, 'optimizer':
<class 'keras.optimizers.nadam.Nadam'>, 'learning_rate': 0.0014639289544581533, 'w0': 0.05008243012487752, 'w1': 2.2050316931598273}. Best is trial 0 with value: 0.8821857571601868.
Restoring model weights from the end of the best epoch: 50.
Epoch 70: early stopping
[I 2023-06-18 21:27:51,401] Trial 4 finished with value: 0.821204662322998 and parameters: {'n_units1': 34, 'n_units2': 104, 'n_units3': 58, 'optimizer': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.0003446327695696678, 'w0': 3.1689412701426285, 'w1': 0.46291048356983416}. Best is trial 0 with value: 0.8821857571601868.
Restoring model weights from the end of the best epoch: 40.
Epoch 60: early stopping
[I 2023-06-18 21:28:12,998] Trial 5 finished with value: 0.8798654079437256
and parameters: {'n_units1': 91, 'n_units2': 47, 'n_units3': 47, 'optimizer': <class 'keras.optimizers.nadam.Nadam'>, 'learning_rate': 0.10809936559348163, 'w0': 1.458184327641681, 'w1': 3.3220520119324526}. Best is trial 0 with value: 0.8821857571601868.
[I 2023-06-18 21:28:36,416] Trial 6 finished with value: 0.0 and parameters:
{'n_units1': 15, 'n_units2': 36, 'n_units3': 19, 'optimizer': <class 'keras.optimizers.adadelta.Adadelta'>, 'learning_rate': 0.0002206715223613633, 'w0': 3.327194463560365, 'w1': 0.12657628640789764}. Best is trial 0 with value: 0.8821857571601868.
[I 2023-06-18 21:28:59,899] Trial 7 finished with value: 0.8705034852027893
and parameters: {'n_units1': 44, 'n_units2': 73, 'n_units3': 57, 'optimizer': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.0001114974074059308, 'w0': 0.03949009426672143, 'w1': 0.023506754624740483}. Best is trial 0 with value: 0.8821857571601868.
Restoring model weights from the end of the best epoch: 44.
Epoch 64: early stopping
[I 2023-06-18 21:29:19,665] Trial 8 finished with value: 0.877886176109314 and parameters: {'n_units1': 21, 'n_units2': 90, 'n_units3': 110, 'optimizer': <class 'keras.optimizers.nadam.Nadam'>, 'learning_rate': 0.0030036354463899764, 'w0': 0.3206367648316107, 'w1': 0.025585375000907015}. Best is trial 0 with value: 0.8821857571601868.
Restoring model weights from the end of the best epoch: 51.
Epoch 71: early stopping
[I 2023-06-18 21:29:38,235] Trial 9 finished with value: 0.8347389698028564
and parameters: {'n_units1': 32, 'n_units2': 93, 'n_units3': 82, 'optimizer': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.005277455859990603, 'w0': 3.8481369967134613, 'w1': 0.06200074711566027}. Best is trial 0 with value: 0.8821857571601868.
Restoring model weights from the end of the best epoch: 17.
Epoch 37: early stopping
[I 2023-06-18 21:29:49,394] Trial 10 finished with value: 0.0 and parameters: {'n_units1': 125, 'n_units2': 121, 'n_units3': 32, 'optimizer': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 1.2729022566938132e-05, 'w0': 0.5973530647108966, 'w1': 0.011000234950990282}. Best is trial 0 with value: 0.8821857571601868.
Restoring model weights from the end of the best epoch: 20.
Epoch 40: early stopping
[I 2023-06-18 21:30:00,334] Trial 11 finished with value: 0.8966370224952698
and parameters: {'n_units1': 99, 'n_units2': 48, 'n_units3': 35, 'optimizer': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.03299427425997522, 'w0': 0.7975878303217644, 'w1': 3.345577376942653}. Best is trial 11 with value: 0.8966370224952698.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
```

```
[I 2023-06-18 21:30:12,177] Trial 12 finished with value: 0.9115666747093201
and parameters: {'n_units1': 111, 'n_units2': 21, 'n_units3': 30, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.014566149
822054569, 'w0': 0.7246209674795842, 'w1': 0.516219700507204}. Best is trial
12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 11.
Epoch 31: early stopping
[I 2023-06-18 21:30:20,817] Trial 13 finished with value: 0.8412349820137024
and parameters: {'n_units1': 125, 'n_units2': 20, 'n_units3': 36, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.018801647
314641734, 'w0': 0.8119031768927266, 'w1': 0.9171845732268363}. Best is tria
l 12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 13.
Epoch 33: early stopping
[I 2023-06-18 21:30:30,141] Trial 14 finished with value: 0.8819360136985779
and parameters: {'n_units1': 104, 'n_units2': 28, 'n_units3': 83, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.017372476
0605286, 'w0': 1.0708287629258557, 'w1': 4.332462010629822}. Best is trial 1
2 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 18.
Epoch 38: early stopping
[I 2023-06-18 21:30:40,602] Trial 15 finished with value: 0.8808754086494446
and parameters: {'n_units1': 66, 'n_units2': 48, 'n_units3': 32, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.024636323
672572905, 'w0': 0.4067904547731032, 'w1': 1.2544032250511965}. Best is tria
l 12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 32.
Epoch 52: early stopping
[I 2023-06-18 21:30:54,361] Trial 16 finished with value: 0.8682448267936707
and parameters: {'n_units1': 109, 'n_units2': 9, 'n_units3': 77, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0080365268784
79224, 'w0': 1.4252118691590483, 'w1': 0.3427551766602775}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 17.
Epoch 37: early stopping
[I 2023-06-18 21:31:05,270] Trial 17 finished with value: 0.8711035251617432
and parameters: {'n_units1': 78, 'n_units2': 63, 'n_units3': 26, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.043642549
80966035, 'w0': 0.24850785991738503, 'w1': 0.8730403291268632}. Best is tria
l 12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 9.
Epoch 29: early stopping
[I 2023-06-18 21:31:15,227] Trial 18 finished with value: 0.8577482104301453
and parameters: {'n_units1': 58, 'n_units2': 34, 'n_units3': 46, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.007879016
20008784, 'w0': 0.5314125814874461, 'w1': 1.8137105644387794}. Best is trial
12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 25.
Epoch 45: early stopping
[I 2023-06-18 21:31:27,172] Trial 19 finished with value: 0.8898993134498596
and parameters: {'n_units1': 90, 'n_units2': 72, 'n_units3': 43, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.058154081
551957636, 'w0': 1.6822410821135696, 'w1': 4.973017791914984}. Best is trial
12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 74.
Epoch 94: early stopping
[I 2023-06-18 21:31:49,234] Trial 20 finished with value: 0.8897220492362976
and parameters: {'n_units1': 109, 'n_units2': 23, 'n_units3': 21, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.046113929611367
274, 'w0': 0.768630640047115, 'w1': 0.600536638081748}. Best is trial 12 wit
h value: 0.9115666747093201.
```

```
Restoring model weights from the end of the best epoch: 33.
Epoch 53: early stopping
[I 2023-06-18 21:32:02,815] Trial 21 finished with value: 0.8888556957244873
and parameters: {'n_units1': 90, 'n_units2': 72, 'n_units3': 42, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.058278712
86410791, 'w0': 1.757507694908005, 'w1': 4.279715645473913}. Best is trial 1
2 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 14.
Epoch 34: early stopping
[I 2023-06-18 21:32:12,397] Trial 22 finished with value: 0.8623047471046448
and parameters: {'n_units1': 89, 'n_units2': 40, 'n_units3': 70, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.018624384
30504076, 'w0': 0.9241668011654842, 'w1': 2.3370018205601895}. Best is trial
12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 35.
Epoch 55: early stopping
[I 2023-06-18 21:32:26,385] Trial 23 finished with value: 0.9024496078491211
and parameters: {'n_units1': 116, 'n_units2': 64, 'n_units3': 47, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.072158069
8797391, 'w0': 2.2300938711360208, 'w1': 1.4631985394522788}. Best is trial
12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 25.
Epoch 45: early stopping
[I 2023-06-18 21:32:38,852] Trial 24 finished with value: 0.8903886079788208
and parameters: {'n_units1': 117, 'n_units2': 59, 'n_units3': 53, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.129210217
3072452, 'w0': 2.129909426267103, 'w1': 1.4316361549820882}. Best is trial 1
2 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
[I 2023-06-18 21:32:51,192] Trial 25 finished with value: 0.910249650478363
and parameters: {'n_units1': 112, 'n_units2': 48, 'n_units3': 25, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.01041321413122
2288, 'w0': 2.333691836922462, 'w1': 0.8553708124968765}. Best is trial 12 w
ith value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 23.
Epoch 43: early stopping
[I 2023-06-18 21:33:03,197] Trial 26 finished with value: 0.8352209329605103
and parameters: {'n_units1': 114, 'n_units2': 17, 'n_units3': 19, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.00951859639840
0201, 'w0': 2.6528161942503266, 'w1': 0.26226932264209524}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
[I 2023-06-18 21:33:15,852] Trial 27 finished with value: 0.8844936490058899
and parameters: {'n_units1': 128, 'n_units2': 30, 'n_units3': 68, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.01314961222947
67065, 'w0': 4.9849024478874515, 'w1': 0.5947317586194127}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
[I 2023-06-18 21:33:29,855] Trial 28 finished with value: 0.905492901802063
and parameters: {'n_units1': 116, 'n_units2': 81, 'n_units3': 28, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.00525147725608
69755, 'w0': 2.5786974554230744, 'w1': 0.9788786043765096}. Best is trial 1
2 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
```

```
[I 2023-06-18 21:33:44,911] Trial 29 finished with value: 0.8946544528007507
and parameters: {'n_units1': 102, 'n_units2': 79, 'n_units3': 18, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0037160244605
386277, 'w0': 4.7138167502943045, 'w1': 0.8082366211124723}. Best is trial 1
2 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
[I 2023-06-18 21:33:59,384] Trial 30 finished with value: 0.879202127456665
and parameters: {'n_units1': 80, 'n_units2': 102, 'n_units3': 11, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0021411964778
981505, 'w0': 2.6141221831306067, 'w1': 0.3832293310978749}. Best is trial 1
2 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
[I 2023-06-18 21:34:23,302] Trial 31 finished with value: 0.8973302245140076
and parameters: {'n_units1': 116, 'n_units2': 65, 'n_units3': 27, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0057002785619
415155, 'w0': 2.1618496748834635, 'w1': 1.197832301466939}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 31.
Epoch 51: early stopping
[I 2023-06-18 21:34:38,258] Trial 32 finished with value: 0.9001057744026184
and parameters: {'n_units1': 119, 'n_units2': 82, 'n_units3': 40, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0103478156404
27433, 'w0': 1.290883938042795, 'w1': 0.6275269363514682}. Best is trial 12
with value: 0.9115666747093201.
[I 2023-06-18 21:35:01,476] Trial 33 finished with value: 0.5636744499206543
and parameters: {'n_units1': 98, 'n_units2': 55, 'n_units3': 27, 'optimize
r': <class 'keras.optimizers.adadelta.Adadelta'>, 'learning_rate': 0.0251864
7022658457, 'w0': 2.200798809778809, 'w1': 1.5025287630632922}. Best is tria
l 12 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 11.
Epoch 31: early stopping
[I 2023-06-18 21:35:12,171] Trial 34 finished with value: 0.8649982213973999
and parameters: {'n_units1': 108, 'n_units2': 102, 'n_units3': 51, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0021272604733
028567, 'w0': 1.1566615777342468, 'w1': 1.0234256272261941}. Best is trial 1
2 with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 56.
Epoch 76: early stopping
[I 2023-06-18 21:35:36,523] Trial 35 finished with value: 0.8815819621086121
and parameters: {'n_units1': 122, 'n_units2': 41, 'n_units3': 11, 'optimize
r': <class 'keras.optimizers.nadam.Nadam'>, 'learning_rate': 0.0008336140099
105866, 'w0': 3.2124288562505163, 'w1': 2.274423765815468}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 7.
Epoch 27: early stopping
[I 2023-06-18 21:35:46,780] Trial 36 finished with value: 0.0 and parameter
s: {'n_units1': 112, 'n_units2': 52, 'n_units3': 100, 'optimizer': <class 'k
eras.optimizers.adadelta.Adadelta'>, 'learning_rate': 0.013274477972189351,
'w0': 1.5843867914254632, 'w1': 0.40914233848225473}. Best is trial 12 with
value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
[I 2023-06-18 21:35:59,236] Trial 37 finished with value: 0.9078329801559448
and parameters: {'n_units1': 96, 'n_units2': 78, 'n_units3': 63, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.005085677817948
603, 'w0': 2.3368336158984047, 'w1': 0.7479991027788494}. Best is trial 12 w
ith value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
```

```
[I 2023-06-18 21:36:13,411] Trial 38 finished with value: 0.8904386758804321
and parameters: {'n_units1': 84, 'n_units2': 118, 'n_units3': 117, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.004192844838805
42, 'w0': 3.430399991999606, 'w1': 0.21009427124879057}. Best is trial 12 wi
th value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 17.
Epoch 37: early stopping
[I 2023-06-18 21:36:24,252] Trial 39 finished with value: 0.8916155099868774
and parameters: {'n_units1': 96, 'n_units2': 80, 'n_units3': 96, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.005623933753130
164, 'w0': 0.5920726340890251, 'w1': 0.7229835346556306}. Best is trial 12 w
ith value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 24.
Epoch 44: early stopping
[I 2023-06-18 21:36:37,011] Trial 40 finished with value: 0.8730801343917847
and parameters: {'n_units1': 66, 'n_units2': 88, 'n_units3': 64, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.000987345694736
0301, 'w0': 1.158131693553458, 'w1': 0.5356901673031785}. Best is trial 12 w
ith value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 21.
Epoch 41: early stopping
[I 2023-06-18 21:36:48,771] Trial 41 finished with value: 0.6985325217247009
and parameters: {'n_units1': 121, 'n_units2': 94, 'n_units3': 50, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.081420793122653
32, 'w0': 2.610266460552669, 'w1': 0.9384199810833084}. Best is trial 12 wit
h value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 28.
Epoch 48: early stopping
[I 2023-06-18 21:37:03,598] Trial 42 finished with value: 0.8913087844848633
and parameters: {'n_units1': 106, 'n_units2': 61, 'n_units3': 61, 'optimize
r': <class 'keras.optimizers.nadam.Nadam'>, 'learning_rate': 0.0286901933207
5596, 'w0': 1.906144983880021, 'w1': 1.8310948714881352}. Best is trial 12 w
ith value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 32.
Epoch 52: early stopping
[I 2023-06-18 21:37:18,128] Trial 43 finished with value: 0.8223245739936829
and parameters: {'n_units1': 96, 'n_units2': 76, 'n_units3': 37, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0841050089083
3171, 'w0': 3.9464779455158174, 'w1': 1.1782263754651425}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 49.
Epoch 69: early stopping
[I 2023-06-18 21:37:36,358] Trial 44 finished with value: 0.861303985118866
and parameters: {'n_units1': 10, 'n_units2': 68, 'n_units3': 25, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.002745309107914
6356, 'w0': 2.999670689438902, 'w1': 0.6740120925037771}. Best is trial 12 w
ith value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 18.
Epoch 38: early stopping
[I 2023-06-18 21:37:47,616] Trial 45 finished with value: 0.0 and parameter
s: {'n_units1': 113, 'n_units2': 85, 'n_units3': 14, 'optimizer': <class 'ke
ras.optimizers.adadelta.Adadelta'>, 'learning_rate': 0.005863434004497959,
'w0': 1.4075303013812939, 'w1': 0.5403613643583179}. Best is trial 12 with v
alue: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 28.
Epoch 48: early stopping
[I 2023-06-18 21:38:01,200] Trial 46 finished with value: 0.8651080131530762
and parameters: {'n_units1': 104, 'n_units2': 96, 'n_units3': 56, 'optimize
r': <class 'keras.optimizers.adamw.AdamW'>, 'learning_rate': 0.0125530102315
40582, 'w0': 3.804784880192275, 'w1': 0.9326562336676337}. Best is trial 12
with value: 0.9115666747093201.
```

```
Restoring model weights from the end of the best epoch: 21.
Epoch 41: early stopping
[I 2023-06-18 21:38:12,943] Trial 47 finished with value: 0.8862410187721252
and parameters: {'n_units1': 127, 'n_units2': 56, 'n_units3': 32, 'optimize
r': <class 'keras.optimizers.rmsprop.RMSprop'>, 'learning_rate': 0.032454774
5483891, 'w0': 2.04031614668406, 'w1': 0.4509263855358557}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 17.
Epoch 37: early stopping
[I 2023-06-18 21:38:24,755] Trial 48 finished with value: 0.8975149989128113
and parameters: {'n_units1': 44, 'n_units2': 45, 'n_units3': 48, 'optimize
r': <class 'keras.optimizers.nadam.Nadam'>, 'learning_rate': 0.0177665634149
08444, 'w0': 0.9417269908136239, 'w1': 0.7674805552231632}. Best is trial 12
with value: 0.9115666747093201.
Restoring model weights from the end of the best epoch: 23.
Epoch 43: early stopping
[I 2023-06-18 21:38:36,952] Trial 49 finished with value: 0.8927183747291565
and parameters: {'n_units1': 121, 'n_units2': 16, 'n_units3': 73, 'optimize
r': <class 'keras.optimizers.adam.Adam'>, 'learning_rate': 0.007517848293722
545, 'w0': 1.4875475223477954, 'w1': 1.6313489123220972}. Best is trial 12 w
ith value: 0.9115666747093201.
```

Out[ ]:
```
{'n_units1': 111,
 'n_units2': 21,
 'n_units3': 30,
 'optimizer': keras.optimizers.rmsprop.RMSprop,
 'learning_rate': 0.014566149822054569,
 'w0': 0.7246209674795842,
 'w1': 0.516219700507204}
```

In [ ]:
```
# build model with optuna parameters
unit1, unit2, unit3, optimizer, lr, w0, w1 = (study.best_params['n_units1'],
                                study.best_params['n_units2'],
                                study.best_params['n_units3'],
                                study.best_params['optimizer'],
                                study.best_params['learning_rate'],
                                study.best_params['w0'],
                                study.best_params['w1'] )
tf.random.set_seed(42)
model = Sequential()
model.add(Dense(unit1, activation="relu"))
model.add(Dense(unit2, activation="relu"))
model.add(Dense(unit3, activation='relu'))
model.add(Dense(1, activation="sigmoid"))
opt = optimizer(learning_rate=lr)
model.compile(optimizer=opt, loss="binary_crossentropy", metrics=[matthews_c

# train model
model.fit(X_train,
          y_train,
          validation_data=(X_val, y_val),
          batch_size=128,
          epochs=100,
          callbacks=[early_stop],
          class_weight={0: w0, 1: w1},
          verbose=0)
```

```
Restoring model weights from the end of the best epoch: 8.
Epoch 28: early stopping
```

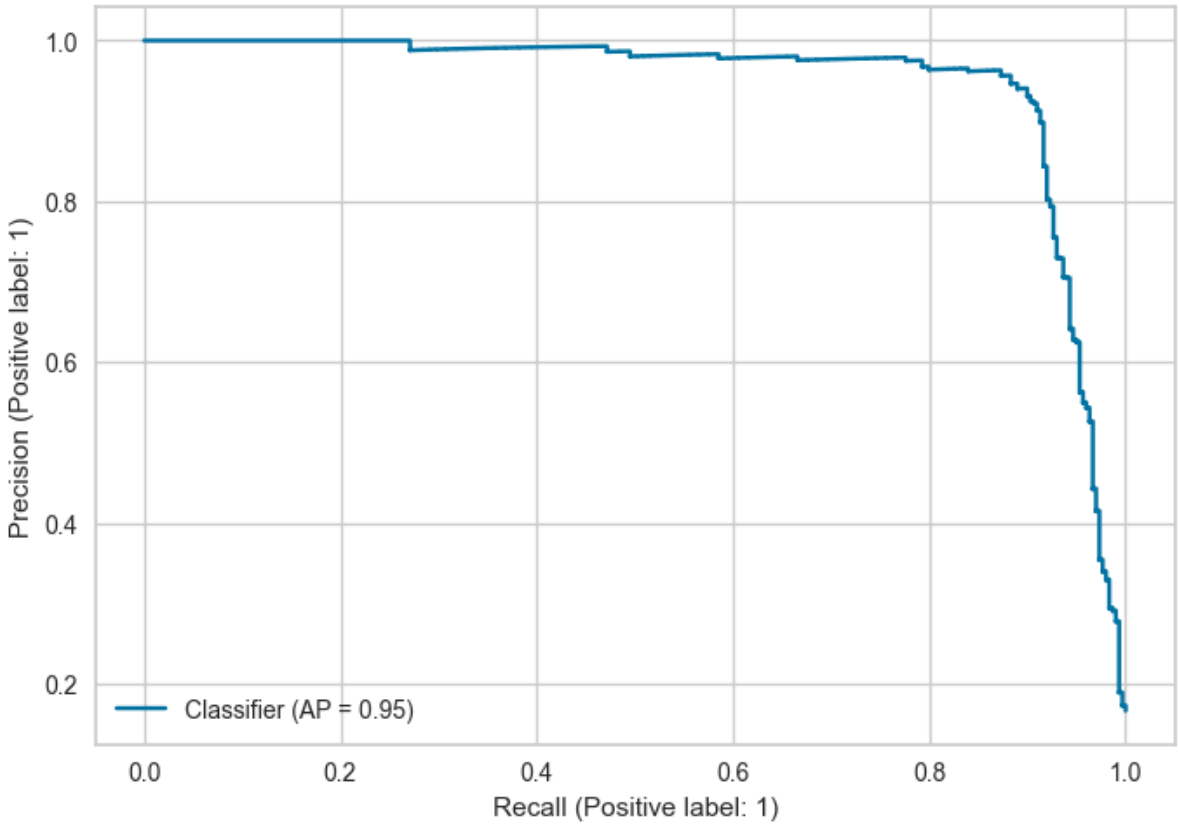
Out[ ]:
```
<keras.callbacks.History at 0x7fa061fc8790>
```

In [ ]:
```
y_pred = (model.predict(X_test) > 0.5).astype("int32")
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
23/57 [==========>.................] - ETA: 0s57/57 [==============================] - 0s 2ms/step
[[1483   17]
 [  30  269]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.98      1500
           1       0.94      0.90      0.92       299

    accuracy                           0.97      1799
   macro avg       0.96      0.94      0.95      1799
weighted avg       0.97      0.97      0.97      1799
```

In [ ]:
```
y_pred_proba = model.predict(X_test)
PrecisionRecallDisplay.from_predictions(y_test,y_pred_proba)
```

Out[ ]:
```
57/57 [==============================] - 0s 3ms/step
<sklearn.metrics._plot.precision_recall_curve.PrecisionRecallDisplay at 0x7fa064b64430>
```




In [ ]:
```
optuna_AP = average_precision_score(y_test, y_pred_proba)
optuna_AP
```

Out[ ]:
```
0.9466555320043355
```

In [ ]:
```
optuna_f1 = f1_score(y_test, y_pred)
optuna_rec = recall_score(y_test, y_pred)
optuna_matthews =  matthews_corrcoef(y_test, y_pred)
optuna_time =  measure_prediction_time(model,X_test)
```

```
57/57 [==============================] - 0s 3ms/step
57/57 [==============================] - 0s 8ms/step
57/57 [==============================] - 0s 2ms/step
16/57 [=======>......................] - ETA: 0s57/57 [==============================] - 0s 5ms/step
57/57 [==============================] - 0s 3ms/step
57/57 [==============================] - 0s 3ms/step
57/57 [==============================] - 0s 6ms/step
57/57 [==============================] - 0s 4ms/step
57/57 [==============================] - 0s 2ms/step
57/57 [==============================] - 0s 4ms/step
```

In [ ]:
```
optuna_AP, optuna_f1, optuna_rec, optuna_matthews, optuna_time
```

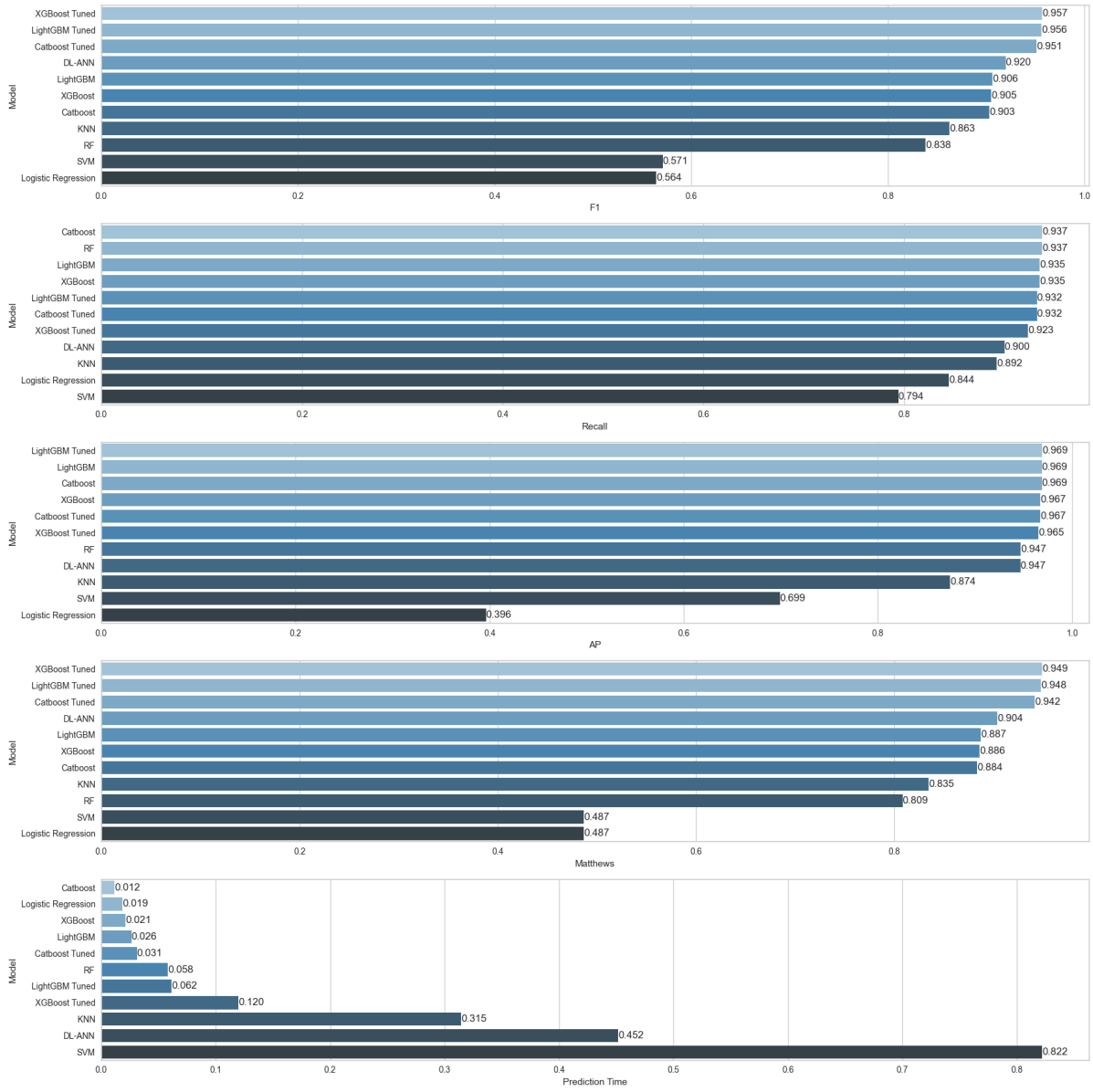
Out[ ]:
```
(0.9466555320043355,
 0.9196581196581196,
 0.8996655518394648,
 0.9043866787371029,
 0.4517411708831787)
```

# 6. Comparison of the Model Performances

- Compare model performances in terms of average-precision, recall, f1, matthews and time scores

In [ ]:
```
compare = pd.DataFrame({"Model": ["Logistic Regression", 'KNN', 'SVM','RF',
                        "F1": [log_f1, knn_f1, SVM_f1, rf_f1, cat_f1, cata_f
                        "Recall": [log_rec,knn_rec, SVM_rec, rf_rec, cat_rec
                        "AP": [log_AP, knn_AP, SVM_AP, rf_AP, cat_AP, cata_A
                        "Matthews": [log_matthews, knn_matthews, SVM_matthew
                        "Prediction Time": [log_time, knn_time, SVM_time, rf


plt.figure(figsize=(18,18))
plt.subplot(5,1,1)
compare = compare.sort_values(by="F1", ascending=False)
ax=sns.barplot(x="F1", y="Model", data=compare, palette="Blues_d")
ax.bar_label(ax.containers[0],fmt="%.3f")

plt.subplot(5,1,2)
compare = compare.sort_values(by="Recall", ascending=False)
ax=sns.barplot(x="Recall", y="Model", data=compare, palette="Blues_d")
ax.bar_label(ax.containers[0],fmt="%.3f")

plt.subplot(5,1,3)
compare = compare.sort_values(by="AP", ascending=False)
ax=sns.barplot(x="AP", y="Model", data=compare, palette="Blues_d")
ax.bar_label(ax.containers[0],fmt="%.3f")

plt.subplot(5,1,4)
compare = compare.sort_values(by="Matthews", ascending=False)
ax=sns.barplot(x="Matthews", y="Model", data=compare, palette="Blues_d")
ax.bar_label(ax.containers[0],fmt="%.3f")

plt.subplot(5,1,5)
compare = compare.sort_values(by="Prediction Time", ascending=True)
ax=sns.barplot(x="Prediction Time", y="Model", data=compare, palette="Blues_
ax.bar_label(ax.containers[0],fmt="%.3f")
```

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




# 6. Model Deployement

### MODEL CHOICE

Catboost, lightgbm, XGboost (tuned or untuned) performs well and they all have similar scores. However, sinnce we aim tour model to find all the churn cases correctly in real life, we will choose the model with the highest recall score. Therefore we will use Catboost as the final model for deployment.

In [ ]:
```
# We will not use entire data for final model since it can be overfit.
X = df.drop('left', axis=1)
y = df['left'].values
```

In [ ]:
```
# split data
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    stratify=y,
                                                    test_size=0.15,
                                                    random_state=42)
```

In [ ]:
```
cat_features = X_train.select_dtypes(include=['object']).columns.tolist()


# Create a CatBoostClassifier model
cat_model = CatBoostClassifier(random_state = 42, verbose=500, cat_features
                               use_best_model=True,early_stopping_rounds= 10

# Fit the model on the training data
cat_model.fit(X_train, y_train, eval_set=(X_test, y_test))
```

```
0:      learn: 0.6026205        test: 0.6012944 best: 0.6012944 (0)     tota
l: 11.8ms       remaining: 1.5s
Stopped by overfitting detector  (10 iterations wait)

bestTest = 0.07403529847
bestIteration = 108

Shrink model to first 109 iterations.
```

Out[ ]:
```
<catboost.core.CatBoostClassifier at 0x7fa071577df0>
```

In [ ]:
```
eval_metric(cat_model,X_train,y_train,X_test,y_test)
```

```
Test_Set
[[1494    6]
 [  22  277]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      1500
           1       0.98      0.93      0.95       299

    accuracy                           0.98      1799
   macro avg       0.98      0.96      0.97      1799
weighted avg       0.98      0.98      0.98      1799


Train_Set
[[8461   39]
 [ 126 1566]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      8500
           1       0.98      0.93      0.95      1692

    accuracy                           0.98     10192
   macro avg       0.98      0.96      0.97     10192
weighted avg       0.98      0.98      0.98     10192
```

In [ ]:
```
import pickle
pickle.dump(cat_model, open('emp_churn_final_model', 'wb'))
```

## Prediction

In [ ]:
```
loaded_model = pickle.load(open('emp_churn_final_model', 'rb'))
```

In [ ]:
```
df_sample = pd.DataFrame(X_test.iloc[0]).T
df_sample

# df = pd.DataFrame.from_dict([my_dict])
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_ |
|---|---|---|---|---|---|
| **416** | 0.11 | 0.9 | 6 | 264 | |

In [ ]:
```
churn_sample = df_sample # load_trans.transform(df_sample)
```

In [ ]:
```
churn_sample
```

Out[ ]:

| | satisfaction_level | last_evaluation | number_project | average_montly_hours | time_spend_ |
|---|---|---|---|---|---|
| **416** | 0.11 | 0.9 | 6 | 264 | |

In [ ]:
```
loaded_model.predict(churn_sample)
```

Out[ ]:
```
array([1])
```

In [ ]:
```
loaded_model.predict_proba(churn_sample)[0]
```

Out[ ]:
```
array([0.00784395, 0.99215605])
```

In [ ]:
```
# check it with the real value
df.left.loc[416]
```

Out[ ]:
```
1
```

---


---