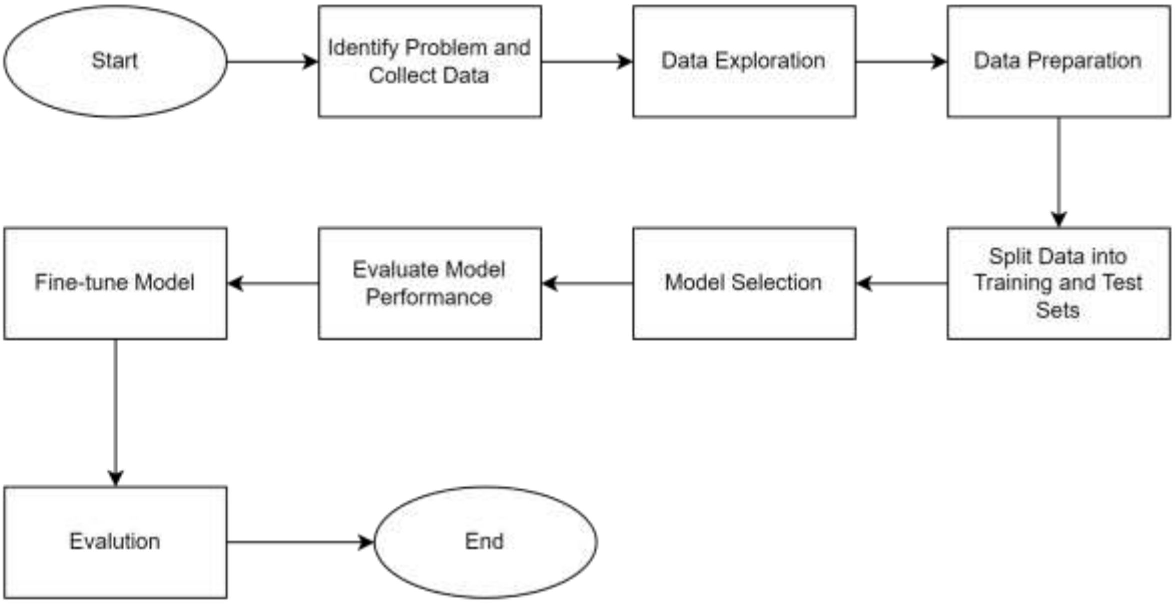
1. Introduction

In business, employee attrition is when employees leave the company for whatever reason, either they've found a new job or retired, and haven't been replaced immediately. For a company to be successful, it needs not only to attract top talent but it also needs to retain these talents. Employee is one of the most important resource in company, where a high attrition rate indicates that the company is unable to maintain their employees. In a short term, with high attrition rate, company must pay a great money to cover the cost of turnover. While in a long term, this will affect the company's performance as employees come and go the company's performance will decline.

2. Data

The data for this project was obtained from Kaggle, a popular online platform for machine learning and data science competitions., named "**IBM HR Analytics Employee Attrition & Performance**"

3. Method



The method for this project may involve using machine learning techniques to analyze the available data and build a model that can predict employee attrition. This involves a number of steps, such as:

1. Identify problem and collect the data
2. Importing necessary modules and libraries
3. Loading the dataset and exploring the data to understand its structure and characteristics
4. Preprocessing the data to prepare it for modeling, including handling missing or invalid values, encoding categorical variables, and scaling or normalizing numerical values
5. Splitting the data into training and test sets to evaluate the model's performance
6. Selecting and training a machine learning model using the training data
7. Evaluating the model's performance on the test set
8. Fine-tuning the model to improve its performance

9. Making predictions with the trained model

4. Implementation

4.1 Import module and library

```
[186] #GENERAL
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

#FEATURE EGNGG
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OrdinalEncoder, OneHotEncod

#MODEL SELECTION
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

#MODEL
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

#MODEL SCORES
from sklearn.metrics import confusion_matrix , accuracy_score ,classification_report

#FEATURE IMPORTANCE
from sklearn.inspection import permutation_importance
```

4.2 Load Dataset

```
df = pd.read_csv("/content/drive/My Drive/dataset/WA_Fn-UseC_-HR-Employee-Attrition.csv")
df.head(5)
```

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

5 rows × 35 columns

After load or read the dataset using Pandas library, we can see the top of 5 records of our dataset. Then, we'll get to see more information about our dataset:

4.3 Data Exploration

```python
print("Dataset Shape = ", df.shape)
```

```
Dataset Shape =  (1470, 35)
```

The dataset has 1470 rows for each employee and 35 attributes or columns with the detailed of information of each columns can be seen below

	Column	Missing values	Unique values	Data type
0	Age	0	43	int64
1	Attrition	0	2	object
2	BusinessTravel	0	3	object
3	DailyRate	0	886	int64
4	Department	0	3	object
5	DistanceFromHome	0	29	int64
6	Education	0	5	int64
7	EducationField	0	6	object
8	EmployeeCount	0	1	int64
9	EmployeeNumber	0	1470	int64
10	EnvironmentSatisfaction	0	4	int64
11	Gender	0	2	object
12	HourlyRate	0	71	int64
13	JobInvolvement	0	4	int64
14	JobLevel	0	5	int64
15	JobRole	0	9	object
16	JobSatisfaction	0	4	int64
17	MaritalStatus	0	3	object
18	MonthlyIncome	0	1349	int64
19	MonthlyRate	0	1427	int64
20	NumCompaniesWorked	0	10	int64
21	Over18	0	1	object
22	OverTime	0	2	object
23	PercentSalaryHike	0	15	int64
24	PerformanceRating	0	2	int64
25	RelationshipSatisfaction	0	4	int64
26	StandardHours	0	1	int64
27	StockOptionLevel	0	4	int64
28	TotalWorkingYears	0	40	int64
29	TrainingTimesLastYear	0	7	int64
30	WorkLifeBalance	0	4	int64
31	YearsAtCompany	0	37	int64
32	YearsInCurrentRole	0	19	int64
33	YearsSinceLastPromotion	0	16	int64
34	YearsWithCurrManager	0	18	int64

We can see that our columns or attributes contain 26 integers and 9 objects. We can also see the statistical value of numerical data inside our data with describe()

```python
df.describe()
```

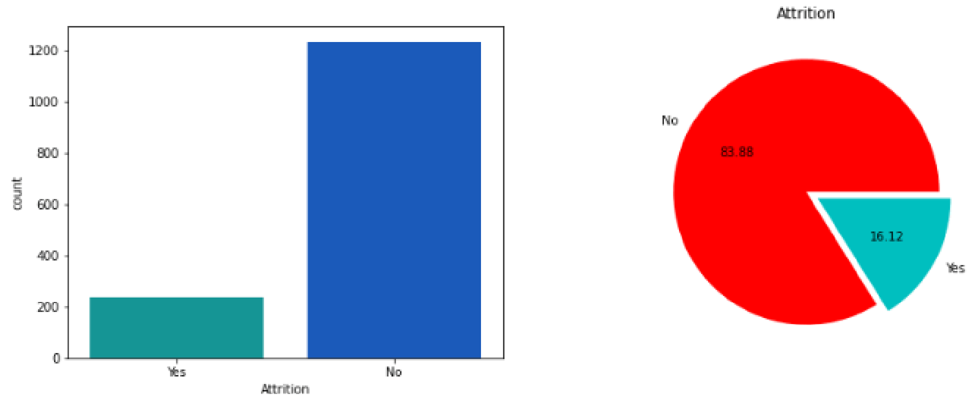
	Age	DailyRate	DistanceFromHome	Education	EmployeeCount	EmployeeNumber	EnvironmentSatisfaction	HourlyRate	JobInvolvement	JobLevel	...
count	1470.000000	1470.000000	1470.000000	1470.000000	1470.0	1470.000000	1470.000000	1470.000000	1470.000000	1470.000000	...
mean	36.923810	802.485714	9.192517	2.912925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

8 rows × 26 columns

And by using seaborn, we plot the outcome of our data to check the distribution of the target in our data.

```
plt.figure(figsize =(14,5))
plt.subplot(1,2,1)
print(df['Attrition'].value_counts())
sns.countplot(x='Attrition',data=df, palette='winter_r')
plt.subplot(1,2,2)
plt.pie(df['Attrition'].value_counts() ,colors =['r' ,'c'] ,explode =[0,0.1]  ,autopct = '%.2f' ,labels =['No' ,'Yes'])
plt.title('Attrition')
```

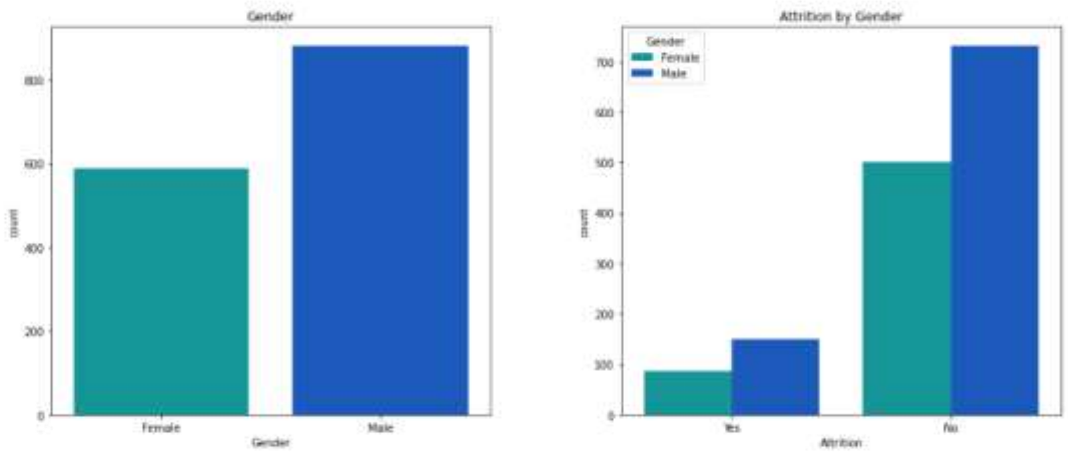
```
No      1233
Yes      237
Name: Attrition, dtype: int64
Text(0.5, 1.0, 'Attrition')
```



Almost 84% of the employees in the dataset have not left the company. We also can observe that number of “No” is far more than “Yes” which indicates that our data is imbalanced. We'll work on this imbalanced problem later.

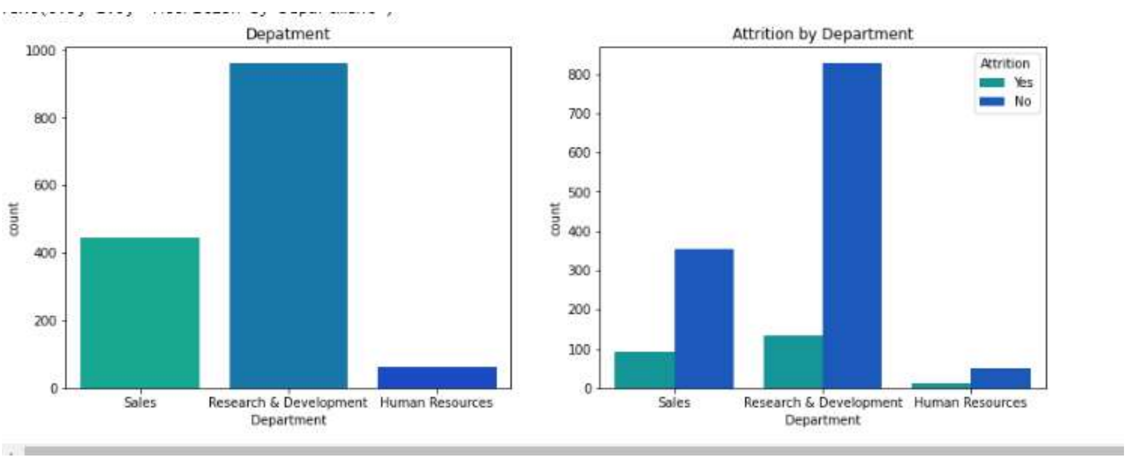
Let's Plotting the distribution of all features and how see how they affect Attrition.

- Gender



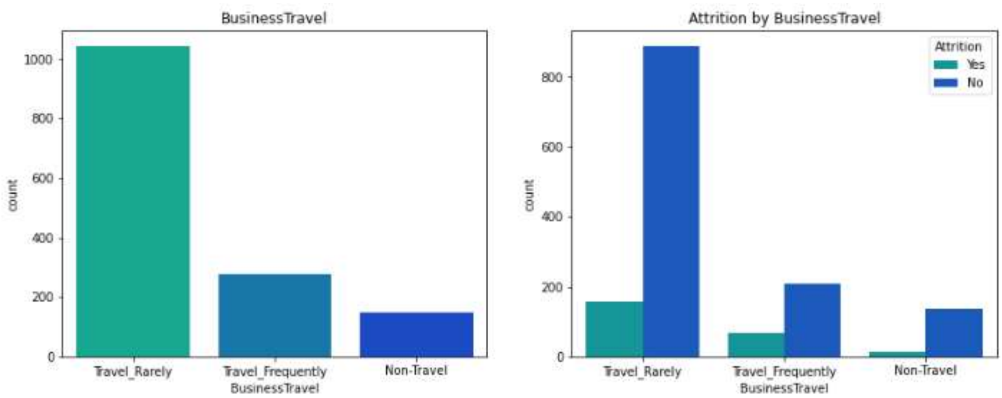
We can clearly see that the majority of the employees are males, so attritions are higher but slightly. I don't think gender is too significant a factor behind attritions.

- Department



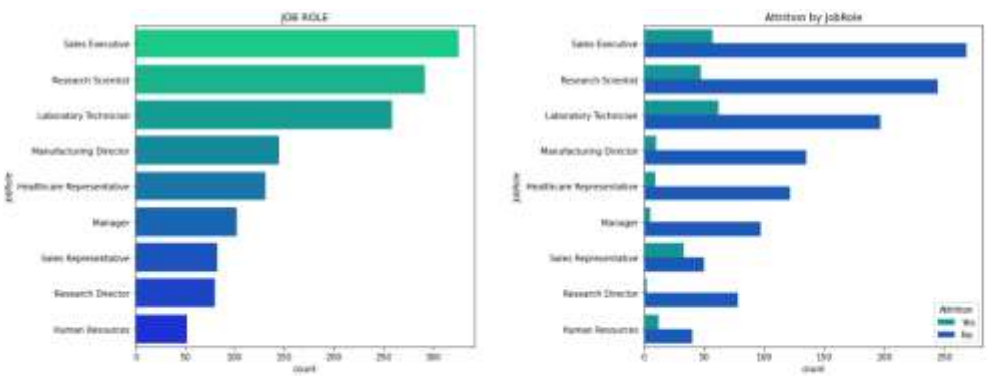
Most attritions are from the research & development department only for sales department to come second by a small margin. Human resources has the least number of attritions. But we need to keep in mind that R&D has a lot more employees than sales and HR. If we considered percentage of attritions per department, we would see that the HR department has most attritions.

- BussinessTravel

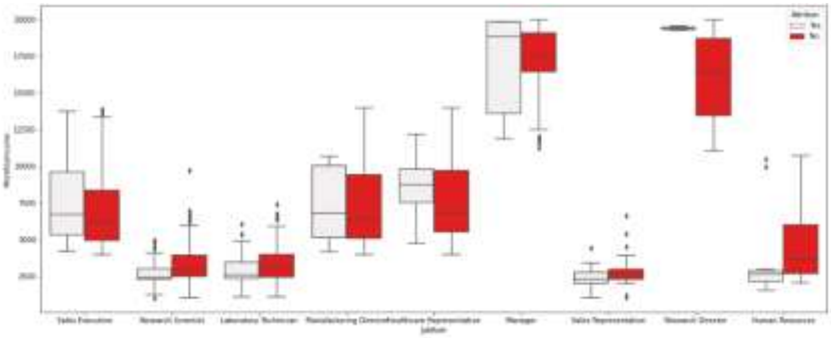


Most employees who travel rarely don't leave the company. From the plot we can tell, sending employees on business travels or not doesn't really make much of a difference and doesn't have a significant effect on attrition.

- JobRole



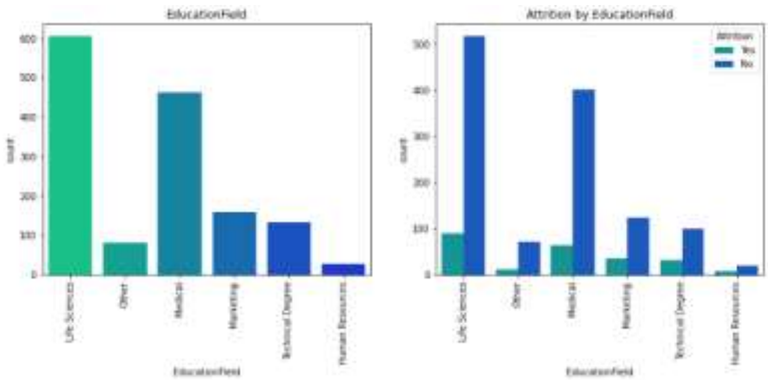
Among job roles, most laboratory technicians have departed from their jobs, only for research scientists, sales executives and sales representatives (% wise) to trail behind. We could look into salaries of each job roles and see if that may be the reason.



As doubted, laboratory technicians, research scientists and sales representatives and executives have very low salary and this could be a major factor behind attritions.

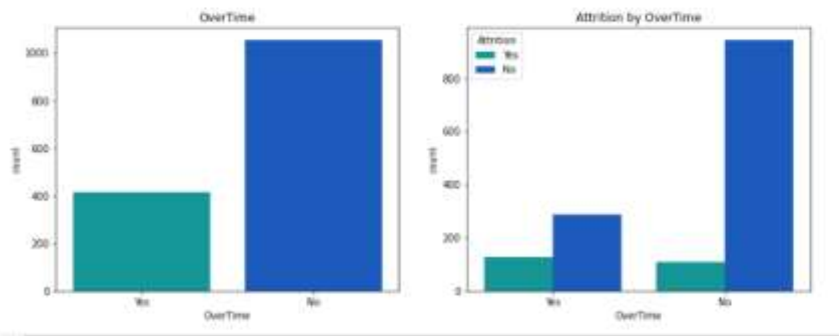
Also, as we had seen earlier, the HR department had the most attritions and we can see they have very low salaries as well so once again, this is something to think about.

- EducationField



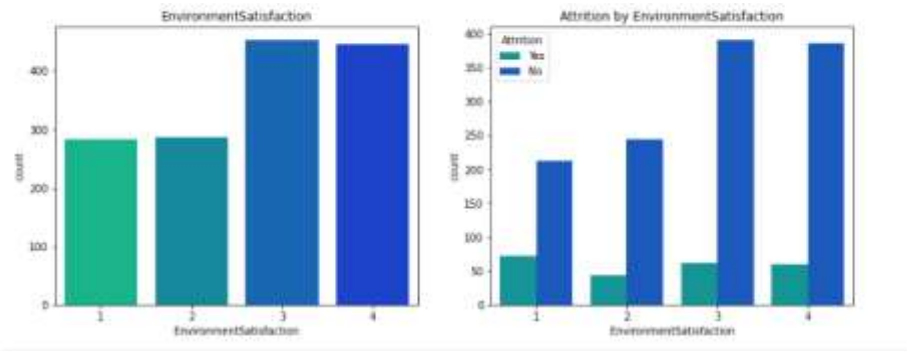
I don't think the degrees of employees really matter here as most of the number of attritions are similar.

- OverTime



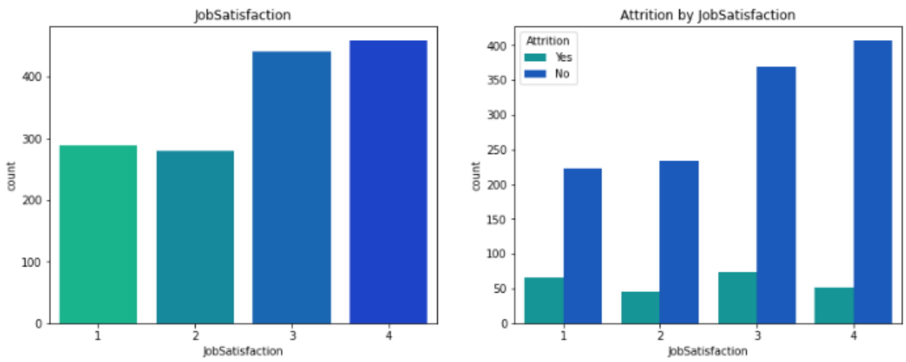
Employees who have overtimes quit job more often.

- Environtment Satisfaction



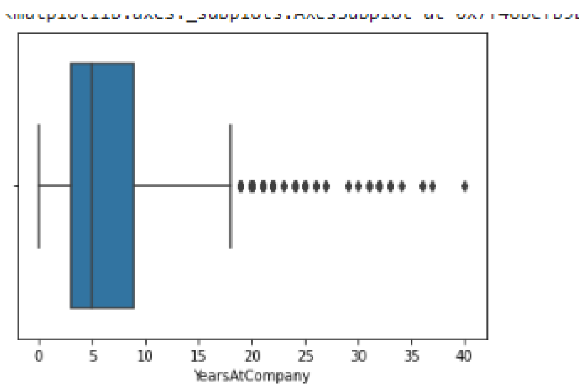
Employees with low EnviromentSatisfaction quit job more often, while employees with high EnviromentSatisfaction quit job less often.

- JobSatisfaction



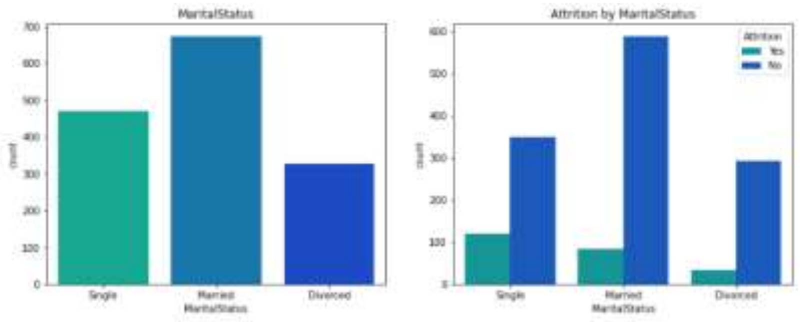
Employees with lowest JobSatisfaction quit job more often, while employees with highest JobSatisfaction quit job less often.

- YearsAtCompany



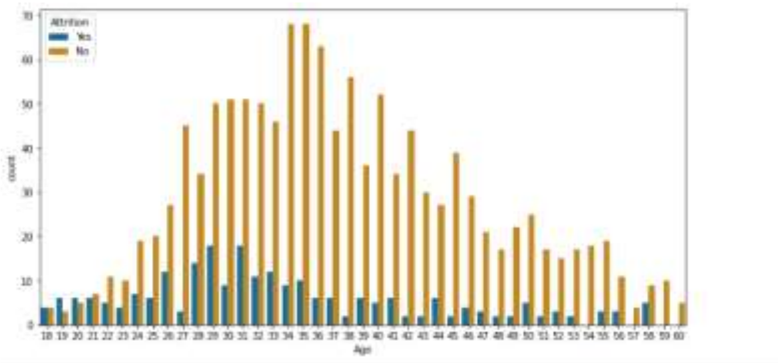
Most employees remain in the company for 3-9 years with median being 5 years.

-MaritalStatus



Single employees quit job more often. I think it's because married people have responsibilities and changes in their lives take longer to plan. Similar situation could be with divorced people, because thy could have kids from marriage that they are responsible for.

- Age



We can also see the number of employyes that left and stayed by age.

4.4 Data Preparation

We've seen that there's no missing values in the dataset. So, in the first step of preprocessing, we'll drop columns with little to no useful information.

4.4.1 Drop Unecessary Columns

```
df = df.drop(['EmployeeCount',
                 'EmployeeNumber',
                 'Over18',
                  'StandardHours'],axis = 1)
df.columns
```

```
Index(['Age', 'Attrition', 'BusinessTravel', 'DailyRate', 'Department',
       'DistanceFromHome', 'Education', 'EducationField',
       'EnvironmentSatisfaction', 'Gender', 'HourlyRate', 'JobInvolvement',
       'JobLevel', 'JobRole', 'JobSatisfaction', 'MaritalStatus',
       'MonthlyIncome', 'MonthlyRate', 'NumCompaniesWorked', 'OverTime',
       'PercentSalaryHike', 'PerformanceRating', 'RelationshipSatisfaction',
       'StockOptionLevel', 'TotalWorkingYears', 'TrainingTimesLastYear',
       'WorkLifeBalance', 'YearsAtCompany', 'YearsInCurrentRole',
       'YearsSinceLastPromotion', 'YearsWithCurrManager'],
      dtype='object')
```

Then, we have to split the datase into X dan y. This is done because we'll work with the X or features more in this preprocessing.

```python
# Splitting Dataset
X = df.drop('Attrition', axis = 1)
y = df.Attrition
```

In this dataset, as mentioned earlier, there are not only numerical features but also categorical. In ML, they can't process categorical features, that's why in this step we'll do encode to that categorical features.

4.4.2 Categorical Encoding

4.4.2.1 Binary Features Encoding

```python
#Binary Features Encoding

y_n_type = []
others =[]
for col in df.select_dtypes('object').columns:
    if(len(df[col].unique()) ==2):
        y_n_type.append(col)

y_n_type
```

```
['Attrition', 'Gender', 'OverTime']
```

```python
df['Gender'].replace({'Male':1 ,'Female':0} ,inplace = True)
df['OverTime'].replace({'Yes':1 ,'No':0} ,inplace = True)
df['Attrition'].replace({'Yes':1 ,'No':0} ,inplace = True)
```

4.4.2.2 Categorical Features Encoding

```python
#categorical features encoding

others = df.select_dtypes('object').columns
others
```

```
Index(['BusinessTravel', 'Department', 'EducationField', 'JobRole',
       'MaritalStatus'],
      dtype='object')
```

```python
le = LabelEncoder()
for col in others:
    df[col] = le.fit_transform(df[col])
```

Now, let's thee the types of each columns or variables or features in our dataset:

```
Data columns (total 30 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Age                       1470 non-null   int64
 1   BusinessTravel            1470 non-null   float64
 2   DailyRate                 1470 non-null   int64
 3   Department                1470 non-null   int8
 4   DistanceFromHome          1470 non-null   int64
 5   Education                 1470 non-null   float64
 6   EducationField            1470 non-null   int8
 7   EnvironmentSatisfaction   1470 non-null   float64
 8   Gender                    1470 non-null   int64
 9   HourlyRate                1470 non-null   int64
 10  JobInvolvement            1470 non-null   float64
 11  JobLevel                  1470 non-null   int64
 12  JobRole                   1470 non-null   int8
 13  JobSatisfaction           1470 non-null   float64
 14  MaritalStatus             1470 non-null   int8
 15  MonthlyIncome             1470 non-null   int64
 16  MonthlyRate               1470 non-null   int64
 17  NumCompaniesWorked        1470 non-null   int64
 18  OverTime                  1470 non-null   int64
 19  PercentSalaryHike         1470 non-null   int64
 20  PerformanceRating         1470 non-null   float64
 21  RelationshipSatisfaction  1470 non-null   float64
 22  StockOptionLevel          1470 non-null   int64
 23  TotalWorkingYears         1470 non-null   int64
 24  TrainingTimesLastYear     1470 non-null   int64
 25  WorkLifeBalance           1470 non-null   float64
 26  YearsAtCompany            1470 non-null   int64
 27  YearsInCurrentRole        1470 non-null   int64
 28  YearsSinceLastPromotion   1470 non-null   int64
 29  YearsWithCurrManager      1470 non-null   int64
```

There are no categorical variables any more.

4.4.3 Feature Scaling

Now, we'll try to scale the features using StandardScaler.

```
# Rescaling Data
Scaler = StandardScaler()
Scaling_Cols = ['TrainingTimesLastYear','YearsAtCompany','TotalWorkingYears',
                'YearsInCurrentRole','YearsSinceLastPromotion','YearsWithCurrManager',
                'PercentSalaryHike','Age','DailyRate','DistanceFromHome','HourlyRate',
                'MonthlyIncome','MonthlyRate','NumCompaniesWorked']
X[Scaling_Cols] = Scaler.fit_transform(X[Scaling_Cols])
```

```
X
```

	Age	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	EnvironmentSatisfaction	Gender	HourlyRate	...
0	0.446350	2	0.742527	2	-1.010909	2	1	2	0	1.383138	...
1	1.322365	1	-1.297775	1	-0.147150	1	1	3	1	-0.240677	...
2	0.008343	2	1.414363	1	-0.887515	2	4	4	1	1.284725	...
3	-0.429664	1	1.461466	1	-0.764121	4	1	4	0	-0.486709	...
4	-1.086676	2	-0.524295	1	-0.887515	1	3	1	1	-1.274014	...
...	...	...	...	...	...	...	...	...	...	...	...
1465	-0.101159	1	0.202082	1	1.703764	2	3	3	1	-1.224807	...
1466	0.227347	2	-0.469754	1	-0.393938	1	3	4	1	-1.175601	...
1467	-1.086676	2	-1.605183	1	-0.640727	3	1	2	1	1.038693	...
1468	1.322365	1	0.546677	2	-0.887515	3	3	4	1	-0.142264	...
1469	-0.320163	2	-0.432568	1	-0.147150	3	3	2	1	0.792660	...

1470 rows × 30 columns

Now, we're done working with the X.

4.5 Split Data into Training and Test Sets

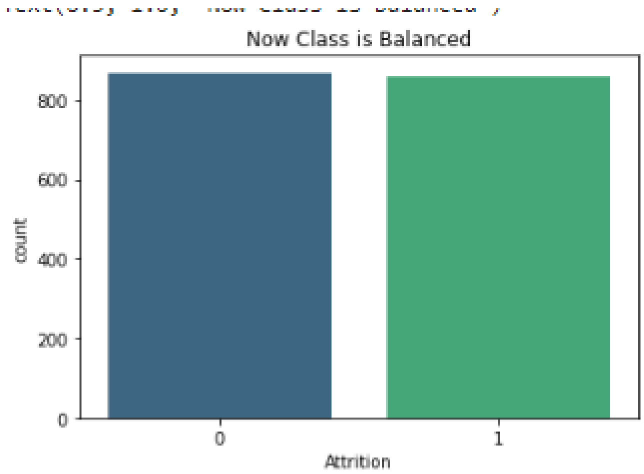
```
# Splitting data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.3)
print('X train size: ', len(X_train))
print('X test size: ', len(X_test))
print('y train size: ', len(y_train))
print('y test size: ', len(y_test))
```

```
X train size:  1726
X test size:  740
y train size:  1726
y test size:  740
```

The training set is used to fit the model, while the test set is used to evaluate the model's performance on unseen data. The purpose of splitting the data into training and test sets is to have a way to measure how well the model generalizes to new data.

4.6 Handling Imbalance Problem

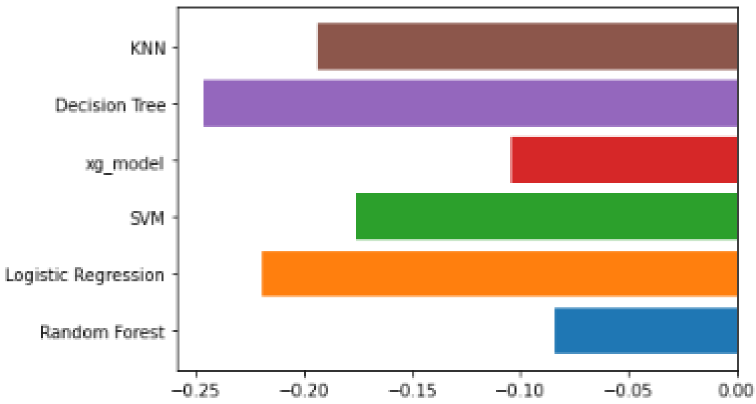
Before modeling, we have to handle the imbalanced problem as mentioned earlier. In this project, we use oversampling SMOTE **and have a 50/50 distribution** for both classes during training



4.7 Model Selection

Cross-validation is a resampling procedure used to evaluate the performance of machine learning model. Here, we'll use k-fold. In k-fold cross-validation, the dataset is divided into k folds, and the model is trained and evaluated k times, with a different fold used as the test set each time. The performance of the model is then averaged across all k iterations.

By using k-fold on some models, we could see the neg_mean_squared_error and also its standard deviation




```
Random Forest: neg_mean_squared_error = -0.076, Standard deviation = 0.007
Logistic Regression: neg_mean_squared_error = -0.191, Standard deviation = 0.017
SVM: neg_mean_squared_error = -0.149, Standard deviation = 0.017
xg_model: neg_mean_squared_error = -0.095, Standard deviation = 0.006
Decision Tree: neg_mean_squared_error = -0.210, Standard deviation = 0.011
KNN: neg_mean_squared_error = -0.176, Standard deviation = 0.015

Best model: Random Forest
```

We can see that the best model is Random Forest.

4.8 Modeling

We'll use Random Forest to train our model but I want to see the performance of other methods.

```python
models = [('Logistic Regression', LogisticRegression()),
          ("KNN" , KNeighborsClassifier()),
          ('Random Forest', RandomForestClassifier()),
          ("SVM", SVC()),
          ("XGBoost", XGBClassifier())]

# Create an empty list to store the dictionaries representing each model
data = []

# Loop through the models and calculate the accuracy score for each one
for name, model in models:
  model.fit(X_train, y_train)
  y_pred = model.predict(X_test)
  accuracy = accuracy_score(y_test, y_pred)
  f1=f1_score(y_test, y_pred)
  auc = roc_auc_score(y_test, y_pred)

  data.append({'model': name, 'accuracy': accuracy, 'f1 score':f1, "roc auc score":auc})

# Create a dataframe from the list of dictionaries
df = pd.DataFrame(data)

# View the first few rows of the dataframe
df.head()
```

Here are the predictions of each models and its metrix. Here, we also use F1 Score and AUC Score considering our data is imbalanced so the accuracy score is not right and enough in this case.

	model	accuracy	f1 score	roc auc score
0	Logistic Regression	0.831081	0.828532	0.831434
1	KNN	0.821622	0.848624	0.819361
2	Random Forest	0.924324	0.923077	0.924712
3	SVM	0.860811	0.859097	0.861132
4	XGBoost	0.901351	0.898752	0.901863

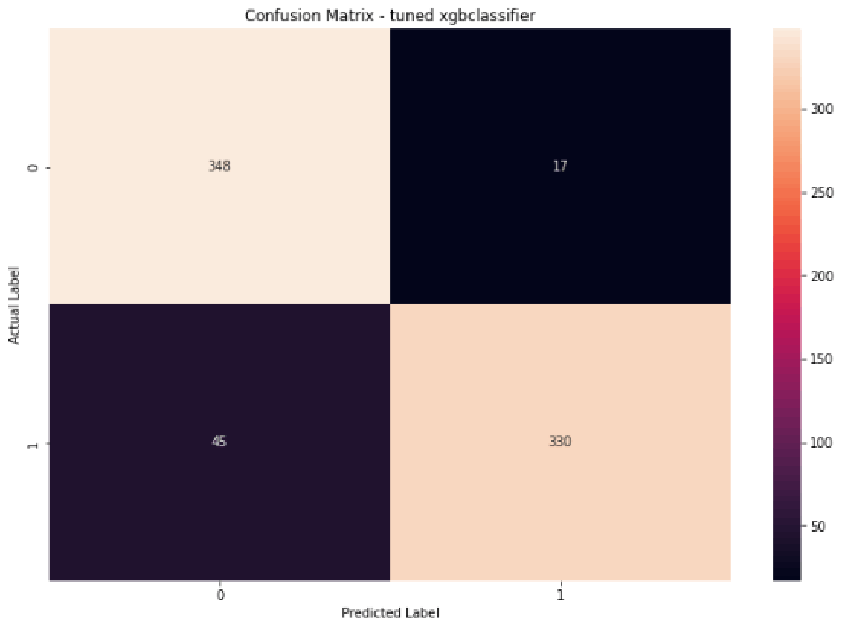
We could see that Randon Forest indeed shows best performance, following by XGBoost and SVM.

4.9 Fine-tune Model

Let's use RandomSearchCV to tune our models and see if we can improve our results.

4.9.1 Tuning XGBoost

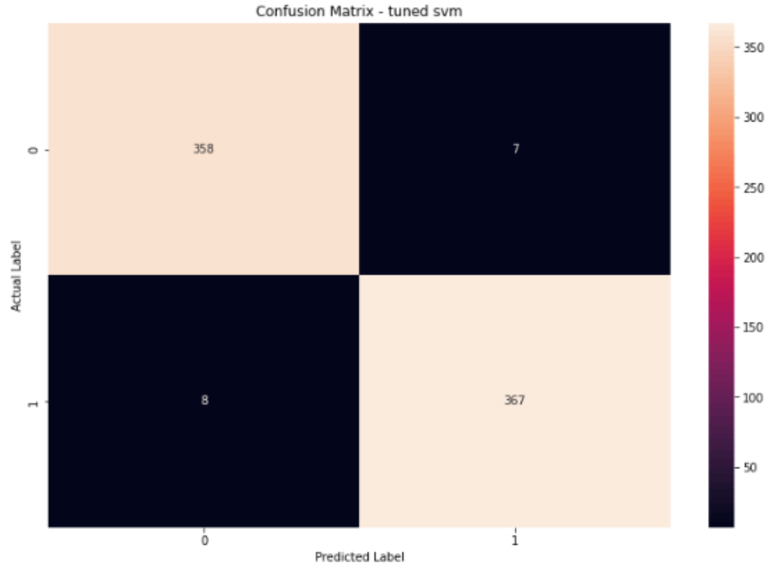
```
Accuracy: 91.62%
              precision    recall  f1-score   support

           0       0.89      0.95      0.92       365
           1       0.95      0.88      0.91       375

    accuracy                           0.92       740
   macro avg       0.92      0.92      0.92       740
weighted avg       0.92      0.92      0.92       740

[Text(0, 0.5, '0'), Text(0, 1.5, '1')]
```



4.9.2 Tuning SVM

	precision	recall	f1-score	support
0	0.98	0.98	0.98	365
1	0.98	0.98	0.98	375
accuracy			0.98	740
macro avg	0.98	0.98	0.98	740
weighted avg	0.98	0.98	0.98	740

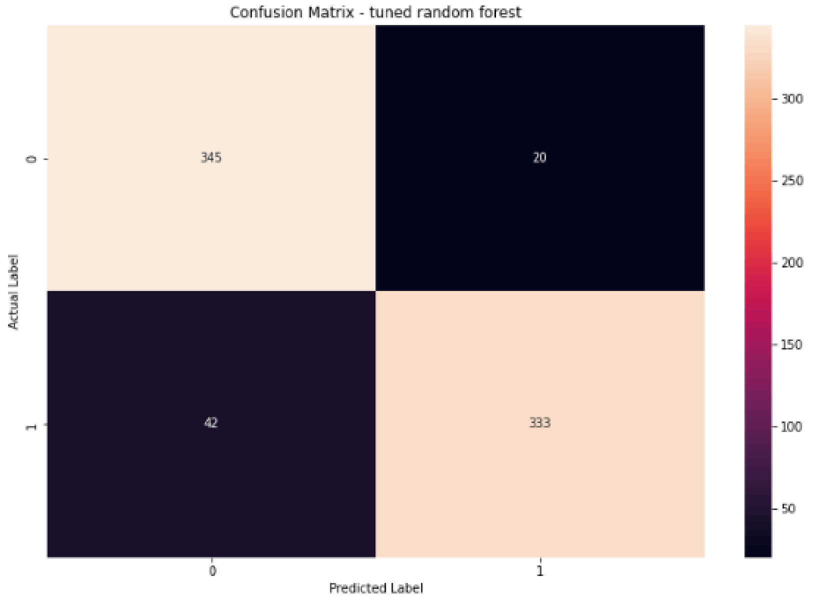
[Text(0, 0.5, '0'), Text(0, 1.5, '1')]



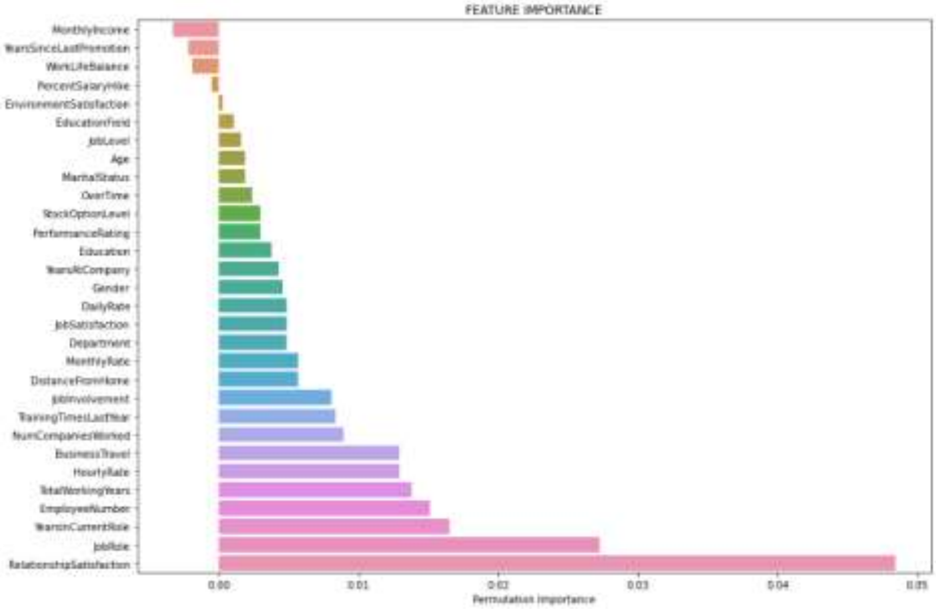
4.9.3 Tuning Random Forest

	precision	recall	f1-score	support
0	0.89	0.95	0.92	365
1	0.94	0.89	0.91	375
accuracy			0.92	740
macro avg	0.92	0.92	0.92	740
weighted avg	0.92	0.92	0.92	740

[Text(0, 0.5, '0'), Text(0, 1.5, '1')]



Feature Importance



5. Conclusion

In conclusion, the results of this study indicate that the Random Forest model performed the best among the three tested models in terms of both accuracy and f1-score when no hyperparameters were used. The XGBoost model also showed good performance, but was slightly outperformed by the Random Forest model. The SVM model performed the worst among the three models without hyperparameters. However, when hyperparameters were used, the SVM model outperformed the other two models in terms of accuracy, with the Random Forest and XGBoost models showing similar performance. These results suggest that the SVM model may be the most suitable for this problem when hyperparameters are properly tuned. Further research and evaluation will be needed to confirm these findings and to explore other potential applications.