**Colab notebook link:**

https://colab.research.google.com/drive/1BvE9EArigs6pNnnAdf1B4X7uNWQAL5DB?usp=sharing

---

# Insights:

## Analyzing the dataset:

- There are total 19104 rows and 14 columns.
- Columns are: ['Unnamed: 0', 'MMM-YY', 'Driver_ID', 'Age', 'Gender', 'City', 'Education_Level', 'Income', 'Dateofjoining', 'LastWorkingDate', 'Joining Designation', 'Grade', 'Total Business Value', 'Quarterly Rating']

---

## Basic data cleaning and exploration:

**Removing Unnecessary Column:**

- We removed the 'Unnamed: 0' column, it is an unnecessary column.

**Duplicate rows in the dataset:**

- There are no duplicates in the dataset.

**Missing values in the dataset:**

- There are 3 columns ('Age', 'Gender' and 'LastWorkingDate'), which have missing values.

**Missing values Treatment:**

- For missing values treatment, using KNN imputation technique. It is time taking but good method to fill missing values in a dataset using the K-Nearest Neighbors'. The KNN-Imputer predicts the value of a missing value by observing trends in related columns. It is use for numerical columns.
- After missing values treatment for numerical columns by KNN-Imputer, concate these columns with remaining columns and forming new data set.

**Merging of rows and aggregation of fields:**

- We can observed that the details of driver are divide as per monthly basis reporting date, so we merge these rows according to their Driver ID.
- Further, we carefully aggregate the other fields as per Driver ID and making new dataset of total numbers of rows 2381 and columns 12.
- After merging of rows and aggregation of fields, renaming columns as per understanding and for better usability.

**Column-wise unique entries:**

```
for i in data.columns:
  print(f"Unique entries for column {i:20} = {data[i].nunique()}")
```

```
Unique entries for column No_of_records        = 24
Unique entries for column Age                  = 61
Unique entries for column Gender               = 6
Unique entries for column City                 = 29
Unique entries for column Education_Level      = 3
Unique entries for column Income               = 2339
Unique entries for column Date_of_joining      = 869
Unique entries for column Last_working_date    = 493
Unique entries for column Joining_designation  = 5
Unique entries for column Grade                = 5
Unique entries for column Total_Business_value = 1629
Unique entries for column Quarterly_rating     = 4
```

- We can observe from the above chart that some of the columns are categorical columns, so we are arranging them as per their variables.

**Updating Date-time columns:**

- 'Date_of_joining' and 'Last_working_date' columns, both are represents date-time columns, so convert these columns to date-time columns.

**Conversion of Categorical Attributes to Category:**

- 'Gender', 'City', 'Education_Level', 'Joining_designation', 'Grade' and 'Quarterly_rating', these 6 columns are categorical attribute, so converting these columns into category columns.
- After this conversion, the new dataset have 6 category columns, 2 date time columns, 3 float type columns and 1 integer column.

## Feature Engineering:

### Whether the Quarterly Rating has increased for that driver –

```
data['Quarterly_rating_increased'] = df.groupby(['Driver_ID'])['Quarterly Rating'].unique().apply(check_value)
```

```
data.Quarterly_rating_increased.value_counts()
```

```
0    1789
1     592
Name: Quarterly_rating_increased, dtype: int64
```

---

- As per Driver ID, whose Quarterly Rating has increased we assign the value 1 and the remaining as 0.

### Whether the monthly Income has increased for that driver -

```
data['Income_increased'] = df.groupby(['Driver_ID'])['Income'].unique().apply(check_value)
```

```
data.Income_increased.value_counts()
```

```
0    2337
1      44
Name: Income_increased, dtype: int64
```

- As per Driver ID, whose monthly Income has increased we assign the value 1 and remaining as 0.

### Target variable creation:

```
def check_day(x):
  if x == 0:
    return 0
  else:
    return 1
```

```
data['Target'] = (data['Last_working_date'].fillna(0)).apply(check_day)
```

```
data.Target.value_counts()
```

```
1    1616
0     765
Name: Target, dtype: int64
```

- As per Driver ID, whose last working day is present will have the value 1and the remaining as 0.

### Number of Months working:

```
data['No_of_months'] = (data['Last_working_date'].dt.year - data['Date_of_joining'].dt.year) * 12 + (data['Last_working_date'].dt.month - data['Date_of_joining'].dt.month)
```

```
data['No_of_months'].fillna(0, inplace = True)
```

- For better analysis, creating a new column based on their months of working.

**Extract Year from Date:**

```
data['Joining_year'] = data['Date_of_joining'].dt.year

data['Leaving_year'] = data['Last_working_date'].dt.year
```

```
data['Joining_year'].fillna(0, inplace = True)
data['Leaving_year'].fillna(0, inplace = True)
```

- For better analysis, extract joining year and leaving year from 'Date_of_joining' and 'Last_working_date' columns respectively, and creating new columns.

**Final Dataset after removing unnecessary columns:**

```
f_data = data.drop(columns=['Date_of_joining', 'Last_working_date', 'Quarterly_rating'])
```

```
f_data.shape
```

```
(2381, 15)
```

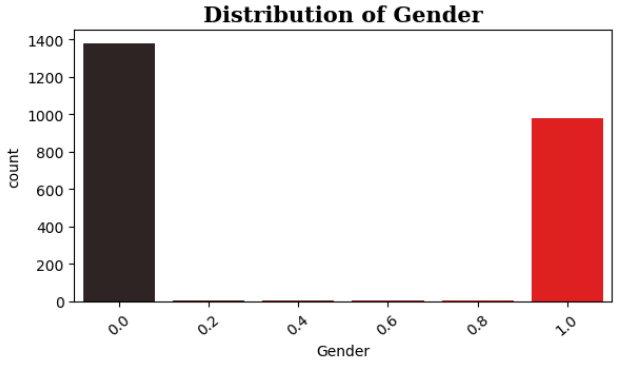
- After extracting year and rearranging the columns, removing those columns which are unnecessary for our future use.

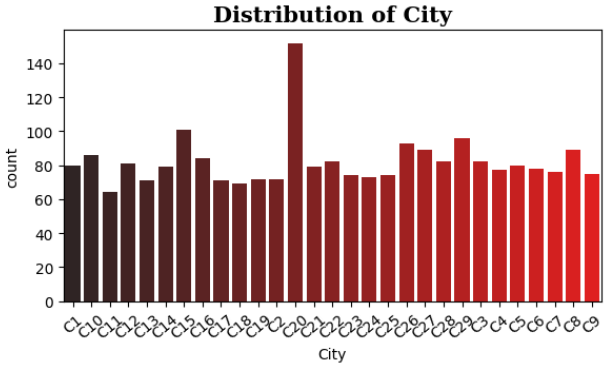
**Finally after feature engineering, rearranging columns and removing columns, we get our new dataset, which have total 2381 number of rows and 15 numbers of columns. Now our features are ready for machine learning.**

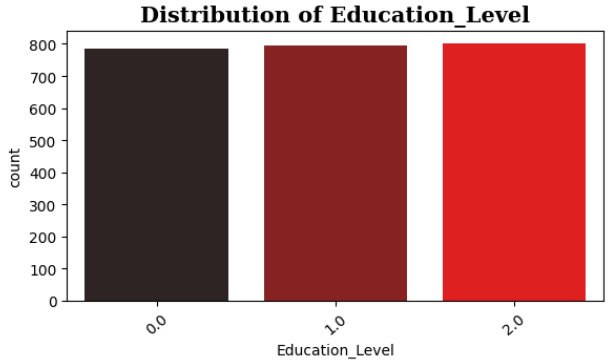
---

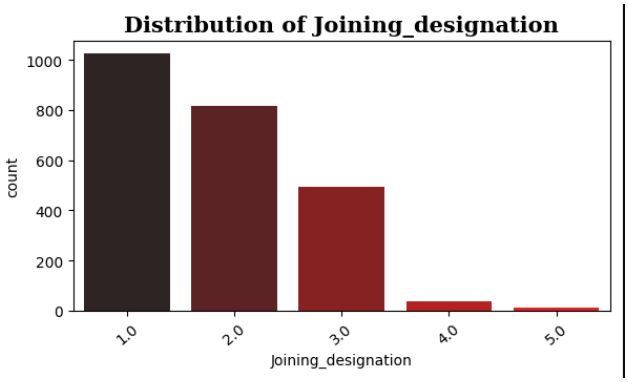
## Uni-variate analysis:

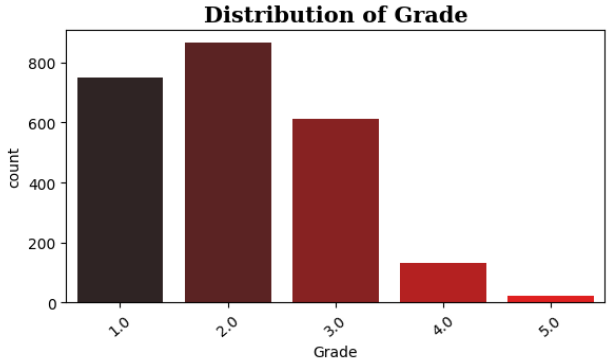
**Distribution Plot for Categorical Variables:**

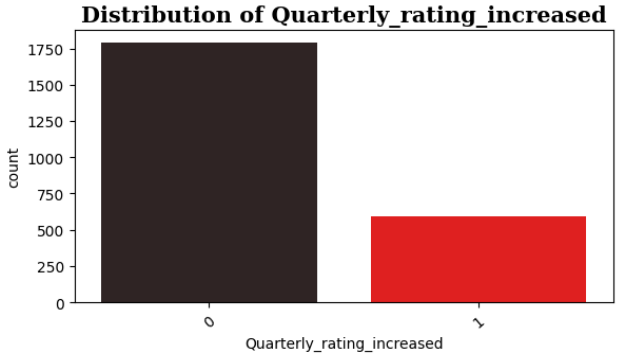


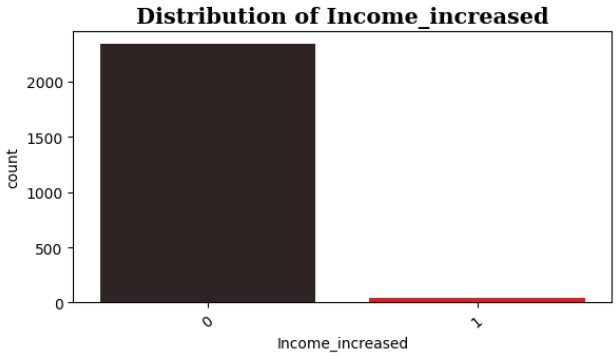



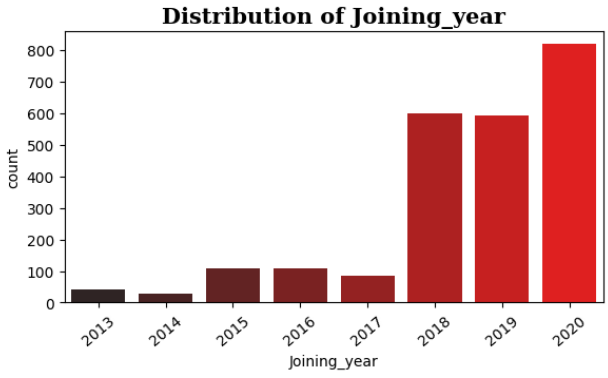


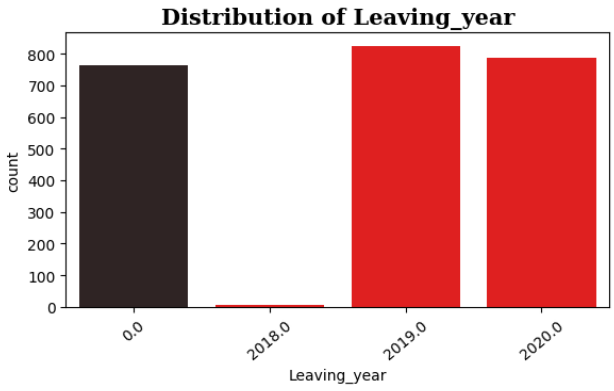



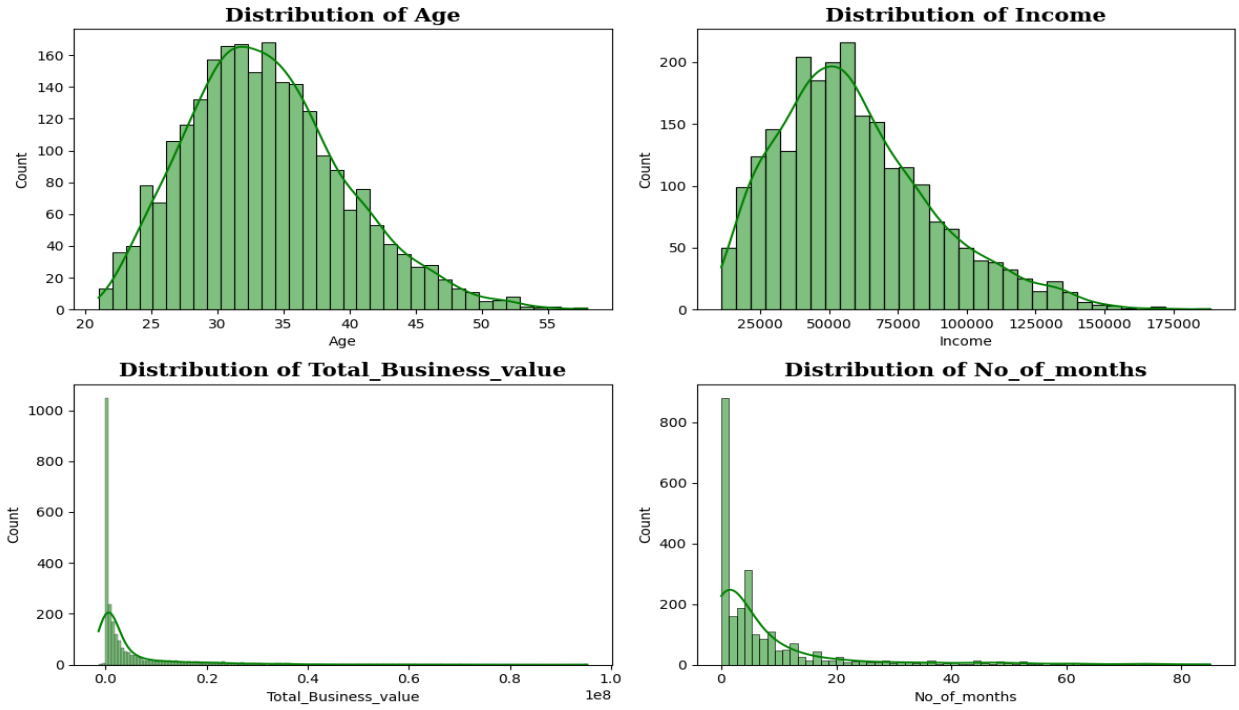




- **Gender:** Majority of drivers are male. Out of 2381 drivers, 1382 drivers are male, rest are females.

- **City:** As per dataset, total 2381 drivers from total 29 cities and most of the drivers from C20 city. Total 152 drivers are from C20 city.
- **Education Level:** Out of 2381 drivers, total 802 drivers have completed graduation and 12+ educations.
- **Grade:** Most of the drivers are from grade 2, followed by grade 1 and grade 3 respectively.
- **Joining Designation:** Most of the drivers joining designation are 1.
- **Quarterly Rating Increased:** Out of 2381 drivers, only 592 drivers quarterly rating has been increased.
- **Income Increased:** Out of 2381 drivers, only 44 drivers monthly income has been increased.
- **Joining Year:** Most of the drivers have joined in 2020, followed by 2018 and 2019 respectively.
- **Leaving Year:** Most of the drivers leaving in 2019, followed by 2020 and nearly 750 drivers are still working.

**Distribution Plot for Continuous Variables:**



- **Age:** Age of drivers range from 21 to 58 and most of the driver's age range from 30 to 35.
- **Income:** Most of the drivers income less than or equal to 75835.
- **Total Business Value:** 75% drivers acquired 4173650 as total business values.
- **No of Months:** In the terms of no of months working is ranges from 0 to 85 and 75% drivers no of working months is less than or equal to 9.

---

## Bi-variate Analysis:

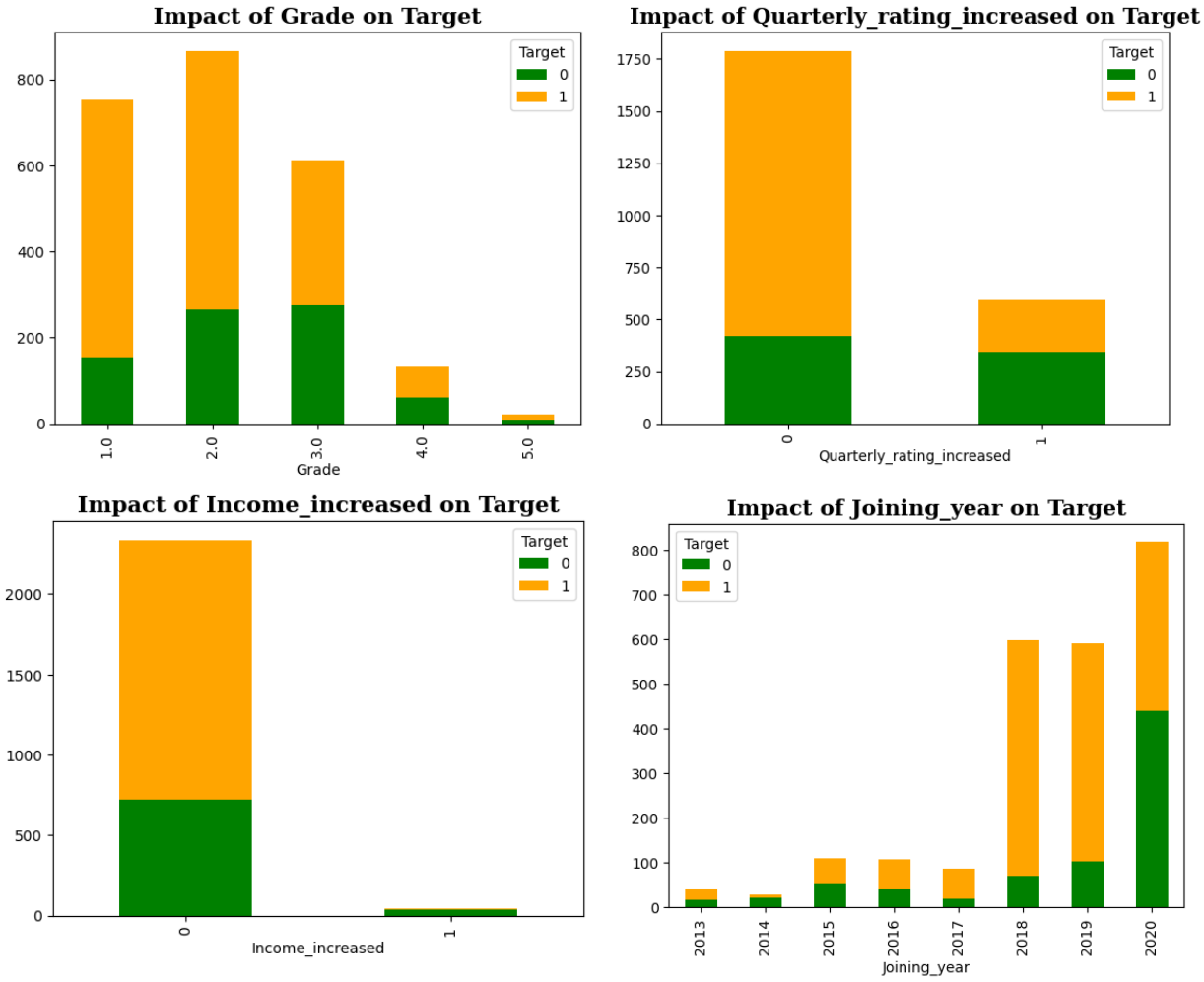
### Impact of Categorical variables on Target variable:

- **Gender :** Impact of gender is nearly same for both of the employees, who left the organization and who did not leave.

- **City:** Most of the drivers from city C20 left the organization as respective to proportion of working.

- **Education Level:** Impact of education level is nearly same for all 10+, 12+ and graduate employess, who left the organization and who did not leave.

- **Joining Designation:** Most of the drivers who left the organization are from joining designation 1.



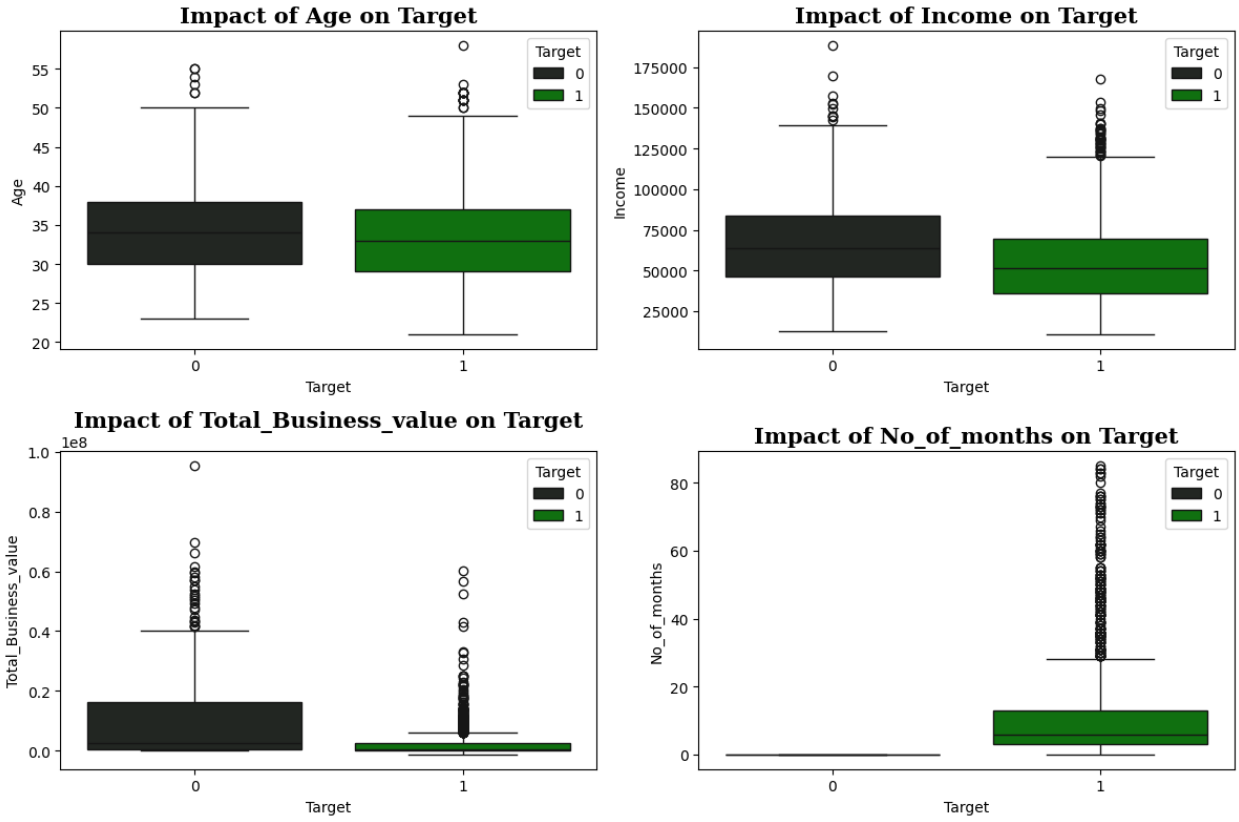

- **Grade:** Employees who have grade as 3 and 4 at the time of joining are less likely to leave the organization.

- **Quarterly Rating Increased:** Employees, whose quarter rating has increased are less likely to leave the organization.

- **Income Increased:** Employees, whose monthly income has not increased are more likely to leave the organization.

- **Joining Year:** Employees, who joined the organization in the year 2020 are less likely to leave the organization.

**Impact of Contineous variables on Target variable:**

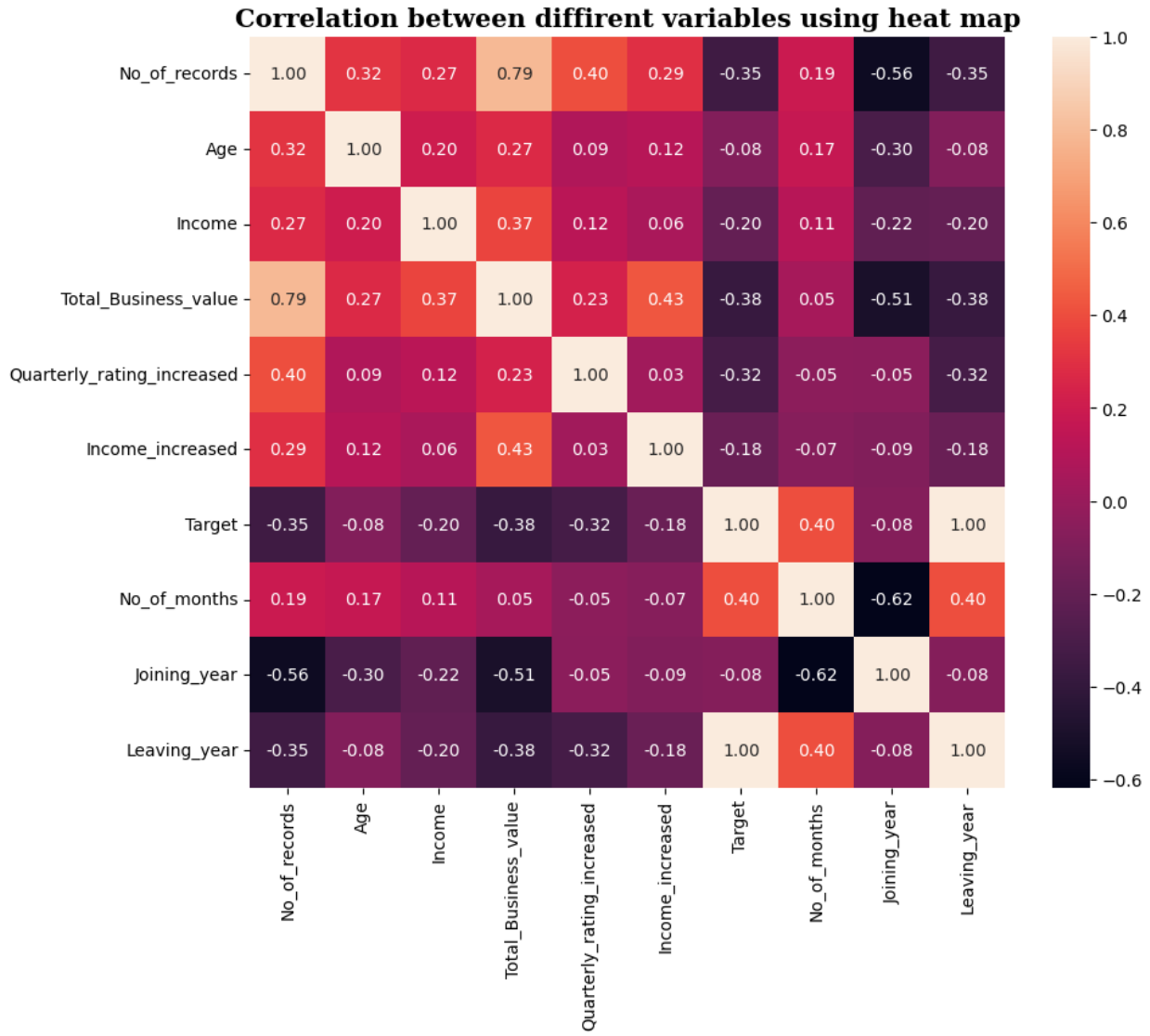



- **Impact of Age:** Impact of age is nearly same for both of the employees, who left the organization and who did not leave.

- **Impact of Income:** The mean income of drivers who left the organization is less than the drivers, who did not leave.

- **Impact of Toat Business Value:** The mean of total business values of drivers who did not leave the organization is more than the drivers who leave the organization.

## Multi-variate Analysis:




From the above heat map, we can observed that

- Total Business value strongly correlated with no of records.
- Income increased and total business values are correlated to each other.
- Joining year is negatively correlated with all other features (i.e. No of records, age, income, total business value, quarterly rating increased, income increased, target (churn), no of months and leaving year).
- No of months positively correlated with leaving year.

## Model Building:

### Data Preparations for Model Building:

```
#pre-processing of Data
x = f_data.drop('Target', axis = 1)
y = f_data['Target']
```

```
print("Shape of x: ", x.shape)
print("Shape of y: ", y.shape)
```

```
Shape of x:  (2381, 14)
Shape of y:  (2381,)
```

- Now, we differentiate our features columns and target column by x and y respectively

### Transforming Categorical Columns by OneHotEncoding:

```
x = pd.get_dummies(x, columns = ['City'])

x.head()
```

- Other than city columns, all of the columns have numerical value, so only city column needs encoding for further use.

### Split the data:

```
#further split the validation-train data to train and test data
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state = 1)
```

```
print("Shape of x_train, y_train: ", x_train.shape, y_train.shape)
print("Shape of x_test, y_test: ", x_test.shape, y_test.shape)
```

```
Shape of x_train, y_train:  (1904, 42) (1904,)
Shape of x_test, y_test:  (477, 42) (477,)
```

- Size of dataset is small, so we split our data into training data and test data by 80% and 20%.
- If we split our dataset into train, test and validation then there is a chance of overfiting, already I checked it personally and I have faced overfiting problem, so I split the data into train and test data for ignoring this overfiting problem.
- Finally we get, 80% training data and 20% validation data, which will help us in model building.

## Class Imbalance Treatment:

```
#checking train data Imbalance
y_train.value_counts()
```

```
1    1287
0     617
Name: Target, dtype: int64
```

- We can see that, the data is imbalanced, so for balancing the data, we are doing oversample technique (SMOTE).

**Oversampling by SMOTE:**

```
#oversampling by SMOTE technique
smt = SMOTE()
```

```
#fit SMOTE to training data
X_sm, y_sm = smt.fit_resample(x_train, y_train)
```

- SMOTE means Synthetically Minority Oversampling Technique. It creates synthetically new samples for minority class.
This is best approach to balancing the data by oversampling.

```
#After oversampling imbalance check
y_sm.value_counts()
```

```
1    1287
0    1287
Name: Target, dtype: int64
```

- We can see, after using SMOTE oversampling, the data is balanced.

- **Note:** But note that, further for machine learning algorithm, I did not use this dataset, because after using SMOTE technique, it produces some missing values in train dataset. May be this was happening due to small dataset. And if again we use KNN Imputation for missing values then over fitting problem has been occurred, so I ignore this dataset for further use.

**Standardization of training data:**

```
scaler = StandardScaler()

#standardization
X_train = pd.DataFrame(scaler.fit_transform(x_train), columns = x.columns)

X_test = pd.DataFrame(scaler.transform(x_test), columns = x.columns)
```

- Standardizing the StandardScaler technique. It scales the values to a specific value range without changing the shape of the original distribution.

## Ensemble Learning - Bagging Algorithm:

- Ensemble Learning is a machine learning technique; it is a learning methodology that combines smartly several baseline models to build a bigger single yet more powerful model that its constituents. It helps to increase the accuracy of classification models or to reduce the mean absolute error for regression models.
- Bagging (Random Forest) and Boosting (Gradient Boosted Decision Tree) are the types of Ensemble Learning.

**Bagging:**

- Bagging simply means Bootstrapped Aggregating, is an ensemble learning technique that helps to improve the performance and accuracy of machine learning algorithms. It is used to deal with bias-variance trade-offs and reduces the variance of a prediction model.

***Implementation of Random Forest:***

```
rf_clf = RandomForestClassifier(class_weight='balanced', random_state=7, max_depth=4, n_estimators=100)
```

```
rf_clf.fit(X_train, y_train)
```

```
RandomForestClassifier(class_weight='balanced', max_depth=4, random_state=7)
```

- Random Forest is an ensemble method based on Bagging and Decision trees. It combines the output of multiple decision trees to reach a single result.
- Note that it is an imbalanced data, so here we are using random forest classifier with balanced class_weight for better output.

```
print("Training Score:", rf_clf.score(X_train, y_train)*100)
print("Testing Score:", rf_clf.score(X_test, y_test)*100)
```

```
Training Score: 100.0
Testing Score: 100.0
```

- We can see, both the training score and testing score for RF classifier is 100%. It looks like over fitting our model.

```
f1 Score for Train data: 1.0
f1 Score for Test data: 1.0
Prescision Score for Train data: 1.0
Precision Score for Test data: 1.0
Recall Score for Train data: 1.0
Recall Score for Test data: 1.0
```

- Also, we can observe that, our f1 score, precision score and recall score for both of the train and test data is 100%.

- It looks like overfiting model.

Using KFold and Cross validation for better result:

```
kfold = KFold(n_splits=10)
cv_acc_results = cross_validate(rf_clf, X_train, y_train, cv=kfold, scoring='accuracy', return_train_score=True)

print(f"K-Fold Accuracy Mean: \n Train: {cv_acc_results['train_score'].mean()*100:.2f} \n Test: {cv_acc_results['test_score'].mean()*100:.2f}")
print(f"K-Fold Accuracy Std: \n Train: {cv_acc_results['train_score'].std()*100:.2f}, \n Test: {cv_acc_results['test_score'].std()*100:.2f}")
```

```
K-Fold Accuracy Mean:
 Train: 100.00
 Test: 100.00
K-Fold Accuracy Std:
 Train: 0.00,
 Test: 0.00
```

- KFold is a class, which we used for split our data to k folds and cross validation score is a function which evalutes a data and return the accuracy score.
- After using KFold and cross validation, still the score for both of test and train data is 100%, which is doubtable score. Now tuning the hyper parameter for overcoming this problem and making the better model.

**Hyperparameter Tuning:**

- ***Grid Search:*** GridSearchCV is a technique for finding the optimal parameter values from a given set of parameters in a grid. It's essentially a cross-validation technique. The model as well as the parameters must be entered.

```
# Defining parameters -

params = {
          'n_estimators' : [100,200,300,400],
          'max_depth' : [3,5,10],
          'criterion' : ['gini', 'entropy'],
          'bootstrap' : [True, False],
          'max_features' : [8,9,10]
         }
```

- We are using 'n_estimators', 'max_depth', 'criterion', 'bootstrap' and 'max_features' parameters in grid search cv.

```
grid = GridSearchCV(estimator = RandomForestClassifier(),
                    param_grid = params,
                    scoring = 'accuracy',
                    cv = 3,
                    n_jobs=-1
                    )
```

- Estimator we are using in grid search cv is Random forest classifier.

```
grid.fit(X_train, y_train)

print("Best params: ", grid.best_params_)
print("Best score: ", grid.best_score_)
```

```
Best params:  {'bootstrap': True, 'criterion': 'gini', 'max_depth': 3, 'max_features': 8, 'n_estimators': 100}
Best score:  1.0
```

- After fitting the grid search cv in train data, we get the best parameters for our model.

- Now taking from above results, we take our parameters ‘bootstrap’ is ‘True’, ‘criterion’ is ‘gini’ and ‘n_estimators’ is 100, but I am taking ‘max_depth’ is ‘2’ and ‘max_feature’ is 4 for eliminating overfiting.

```
rf_clf2 = RandomForestClassifier(class_weight='balanced', random_state=7, bootstrap=True, criterion='gini',
                                 max_depth=2, max_features=4, n_estimators=100)

kfold = KFold(n_splits=10)
cv_acc_results = cross_validate(rf_clf2, X_train, y_train, cv=kfold, scoring='accuracy', return_train_score=True)

print(f"K-Fold Accuracy Mean: \n Train: {cv_acc_results['train_score'].mean()*100:.3f} \n Test: {cv_acc_results['test_score'].mean()*100:.3f}")
print(f"K-Fold Accuracy Std: \n Train: {cv_acc_results['train_score'].std()*100:.3f}, \n Test: {cv_acc_results['test_score'].std()*100:.3f}")
```

```
K-Fold Accuracy Mean:
 Train: 99.568
 Test: 99.633
K-Fold Accuracy Std:
 Train: 0.098,
 Test: 0.409
```

- After using best parameters, we get our best training score 99.568% and test score 99.633%.

- ***Randomized Search:*** Random Search CV is a technique that explores a predefined search space of hyperparameters by randomly selecting combinations to evaluate model performance. When limited computational resource then we used this technique. Here for finding the optimal value of ccp_alpha of our model we used randomized search.

```
# Defining parameters -
params = {'ccp_alpha': uniform(loc=0, scale=0.4)}
```

```
random = RandomizedSearchCV(estimator = RandomForestClassifier(class_weight='balanced', random_state=7, bootstrap=True, criterion='gini',
                                                               max_depth=2, max_features=4, n_estimators=100),
                            param_distributions = params,
                            scoring = 'accuracy',
                            cv = 3,
                            n_iter=15,
                            n_jobs=-1
                            )
```

```
random.fit(X_train, y_train)

print("Best param: ", random.best_params_)
print("Best score: ", random.best_score_)
```

```
Best param:  {'ccp_alpha': 0.2861641672327031}
Best score:  1.0
```

- After using randomized search cv, we get the best ccp_alpha 0.286 for best score 100%.

```
rf_clf3 = RandomForestClassifier(class_weight='balanced', random_state=7, bootstrap=True, criterion='gini',
                                 max_depth=2, max_features=4, ccp_alpha = 0.28, n_estimators=100)

kfold = KFold(n_splits=10)
cv_acc_results = cross_validate(rf_clf3, X_train, y_train, cv=kfold, scoring='accuracy', return_train_score=True)

print(f"K-Fold Accuracy Mean: \n Train: {cv_acc_results['train_score'].mean()*100:.3f} \n Test: {cv_acc_results['test_score'].mean()*100:.3f}")
print(f"K-Fold Accuracy Std: \n Train: {cv_acc_results['train_score'].std()*100:.3f}, \n Test: {cv_acc_results['test_score'].std()*100:.3f}")
```

```
K-Fold Accuracy Mean:
 Train: 99.796
 Test: 99.843
K-Fold Accuracy Std:
 Train: 0.409,
 Test: 0.335
```

- After hyperparameter tuning, we get the best random forest model, whose train score is 99.796% and test score is 99.843%.

---

## Result Evaluation:

```
rf_clf3.fit(X_train, y_train)
```

```
RandomForestClassifier(ccp_alpha=0.04, class_weight='balanced', max_depth=2,
                       max_features=4, random_state=7)
```

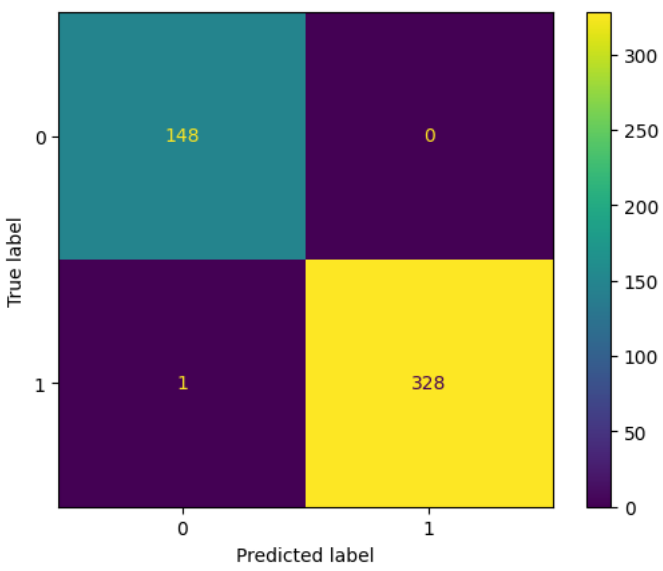
- The above parameters, we will be used for our model's result evaluation.

**Classification Report & Confusion Matrix:**

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 1.00 | 1.00 | 148 |
| 1 | 1.00 | 1.00 | 1.00 | 329 |
| accuracy | | | 1.00 | 477 |
| macro avg | 1.00 | 1.00 | 1.00 | 477 |
| weighted avg | 1.00 | 1.00 | 1.00 | 477 |

- From the above chart, we can observe that
  - For 0: precision score 99%, recall 100% and f1 score 100%.
  - For 1: precision score 100%, recall 100% and f1 score 100%.

- This result is impressive, we can say our model is best model.




**True Negative (TN) = 148**

**False Positive (FP) = 0**

**False Negative (FN) = 1**

**True Positive (TP) = 328**

**FP = Type-1 error**

**FN = Type-2 error**

- We can see, only 0 false positive and 1 false negative, that's impressive score. Although, after hyperparameter tuning, I used low value of parameters which is below best parameters value for eliminating overfiting, still I get the best score for our model.

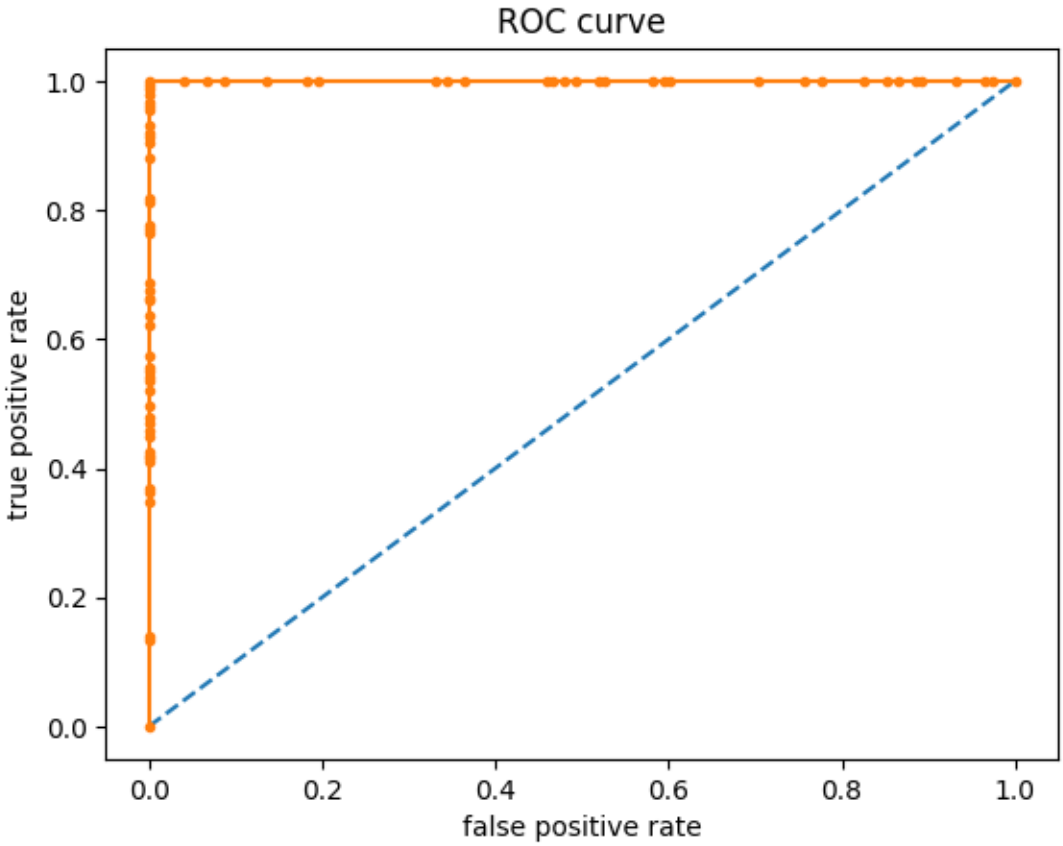
**ROC-AUC Curve*:***

- The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a binary classification model. It helps to improve the performance of model by finding the correct threshold.

- The ROC curve is created by plotting the TPR on the y-axis against the FPR on the x-axis for different thresholds and our aim is to pick that threshold which has high TPR and low FPR.
  - TPR: True Positive Rate, also called Sensitivity is same as recall. A high sensitivity model has high true positive (TP) and low false negative (FN).
  - FPR: False Positive rate, it determines how many negative out of all the negative values have been incorrectly predicted.

```
y_pred = rf_clf3.predict(X_test)
prob = rf_clf3.predict_proba(X_test)
probs = prob[:,1]
```

```
fpr, tpr, thresholds = roc_curve(y_test, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
plt.title("ROC curve")
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
# show the plot
plt.show()
```




- AUC: The area under the ROC curve is commonly used metric that evaluates how well a logistic regression model classifies positive and negative outcomes at all possible cutoffs. Higher AUC means model's prediction is higher to correct.

- ROC-AUC of 0.1 signifies that the model is discriminate 100% between the positive and negative class.
    - ROC-AUC score 0.1, it means our model performing well.

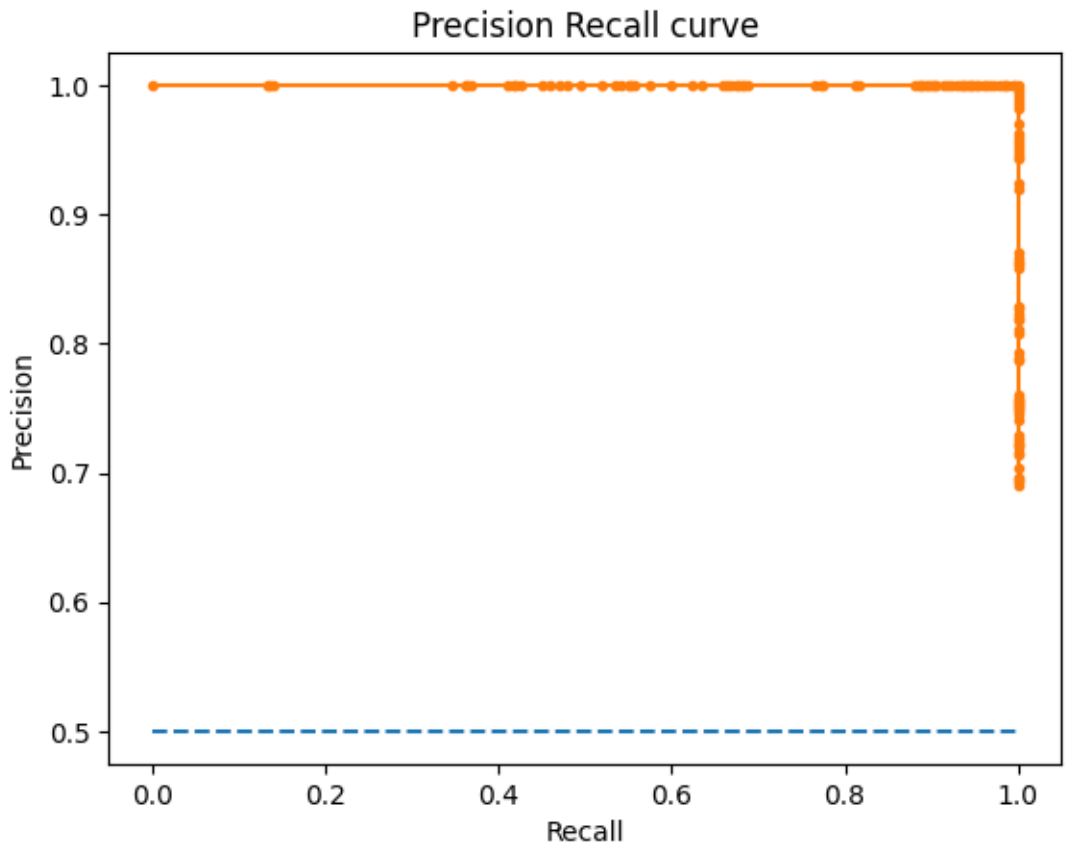
**Precision Recall Curve:**

- The Precision Recall Curve is another graphical representation, which shows the tradeoff between precision and recall for different threshold. Precision-recall is very useful for imbalanced data.

- Similar to ROC curve, the PR curve is created by plotting precision on the y-axis against recall on the x-axis for different threshold values.
    - Precision is defined as the number of true positives (TP) over the number of true positives (TP) plus the number of false positives (FP).

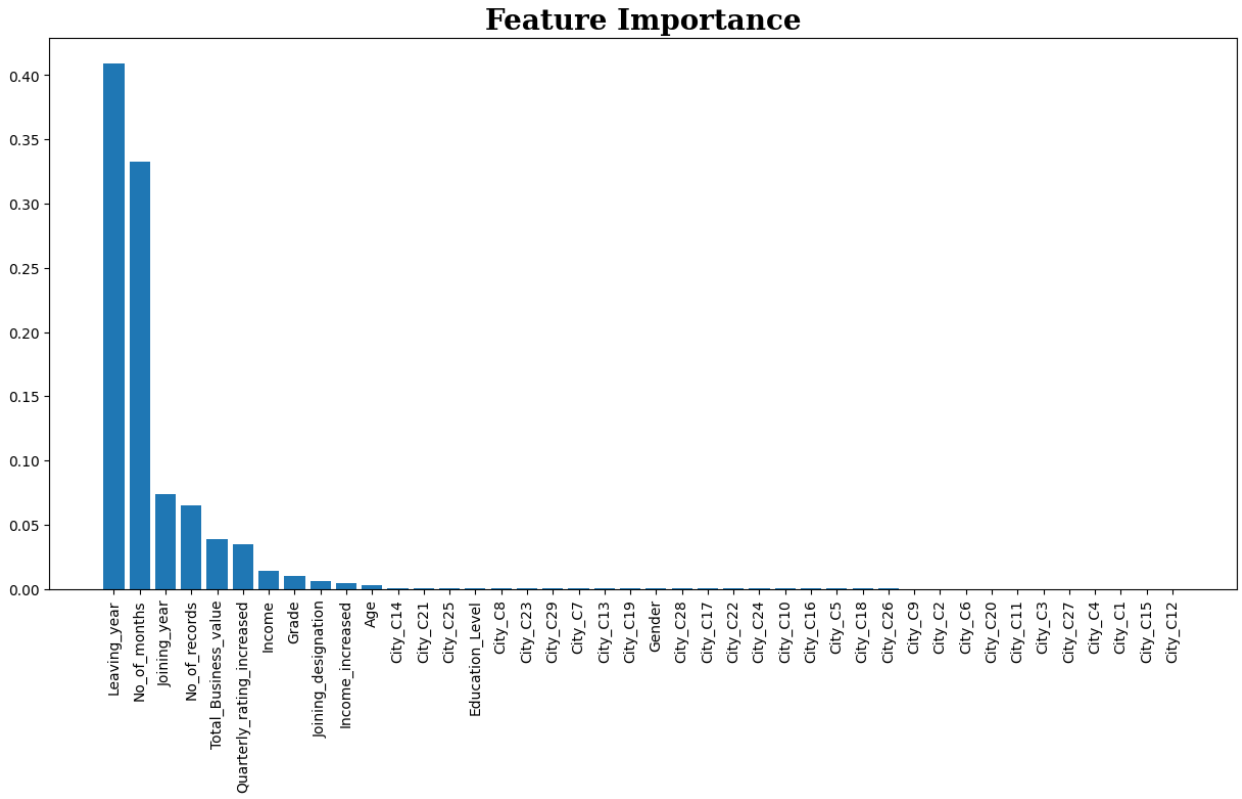
- Recall is defined as the number of true positives (TP) over the all actual positives (TP + FN)

```
precision, recall, thresholds = precision_recall_curve(y_test, probs)
plt.plot([0, 1], [0.5, 0.5], linestyle='--')
# plot the precision-recall curve for the model
plt.plot(recall, precision, marker='.')
plt.title("Precision Recall curve")
plt.xlabel('Recall')
plt.ylabel('Precision')
# show the plot
plt.show()
```




- Precision-recall AUC curve score is 0.1, it's showing that our PRC worked best, no need to further improvement in performance.

## Computing Feature Importance:




- From the above graph, we can observe that leaving year is most important feature, followed by no of month, joining year respectively.

---

## Ensemble Learning- Boosting Algorithm:

**Boosting:**

- Boosting is an ensemble learning method that combines a set of weak learners into a strong learner to minimize training errors. Boosting algorithms can improve the predictive power of your data mining initiatives.

  - **Gradient Boosting Classifier:** The gradient bossting algorithm is to build models sequentially and these subsequent models try to reduce the errors of the previous model.

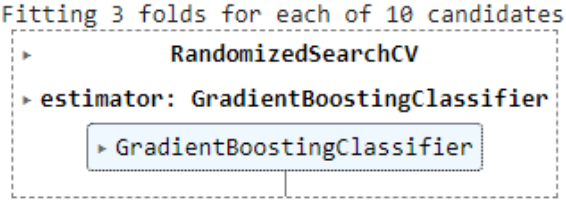
```
parameters = {
    "n_estimators": [50,80,100],
    "max_depth" : [2, 3, 4],
    "max_leaf_nodes" : [5, 10, 20],
    "learning_rate": [0.1, 0.2]
}
```

```
gbc = GradientBoostingClassifier()

clf = RandomizedSearchCV(gbc, parameters, scoring = "accuracy", cv=3, n_jobs = -1, verbose = 1)
```

- For Gradient boosting classifier, we used randomized search cv to find the best parameters for our model.

```
clf.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
```




- Now, fit the randomized search cv to train data.

```
res = clf.cv_results_

for i in range(len(res["params"])):
  print(f"Parameters:{res['params'][i]} Mean_score: {res['mean_test_score'][i]} Rank: {res['rank_test_score'][i]}")
```

```
Parameters:{'n_estimators': 100, 'max_leaf_nodes': 10, 'max_depth': 4, 'learning_rate': 0.2} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 80, 'max_leaf_nodes': 10, 'max_depth': 4, 'learning_rate': 0.1} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 50, 'max_leaf_nodes': 5, 'max_depth': 3, 'learning_rate': 0.2} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 100, 'max_leaf_nodes': 5, 'max_depth': 3, 'learning_rate': 0.2} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 80, 'max_leaf_nodes': 20, 'max_depth': 3, 'learning_rate': 0.2} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 80, 'max_leaf_nodes': 20, 'max_depth': 4, 'learning_rate': 0.1} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 50, 'max_leaf_nodes': 10, 'max_depth': 3, 'learning_rate': 0.1} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 50, 'max_leaf_nodes': 5, 'max_depth': 4, 'learning_rate': 0.1} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 100, 'max_leaf_nodes': 5, 'max_depth': 2, 'learning_rate': 0.1} Mean_score: 1.0 Rank: 1
Parameters:{'n_estimators': 100, 'max_leaf_nodes': 10, 'max_depth': 4, 'learning_rate': 0.1} Mean_score: 1.0 Rank: 1
```

```
print(clf.best_estimator_)
```

```
GradientBoostingClassifier(learning_rate=0.2, max_depth=4, max_leaf_nodes=10)
```

- We get the best parameters for our gradient boosting classifier. Although we can observe that all of the hyperparameter combination gives us the best score value 100%.

```
gbc = clf.best_estimator_

gbc.fit(X_train, y_train)
```

```
GradientBoostingClassifier
GradientBoostingClassifier(learning_rate=0.2, max_depth=4, max_leaf_nodes=10)
```

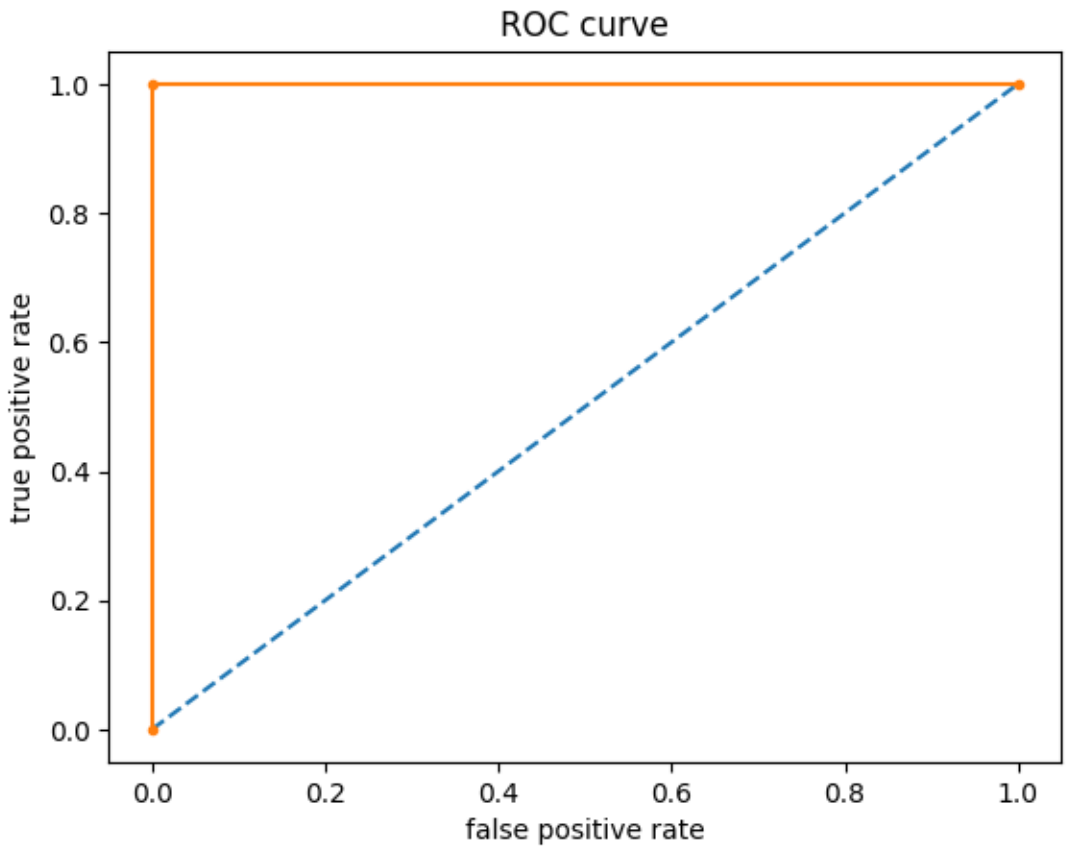
**Result Evaluation:**

***Classification Report and Confusion Matrix:***

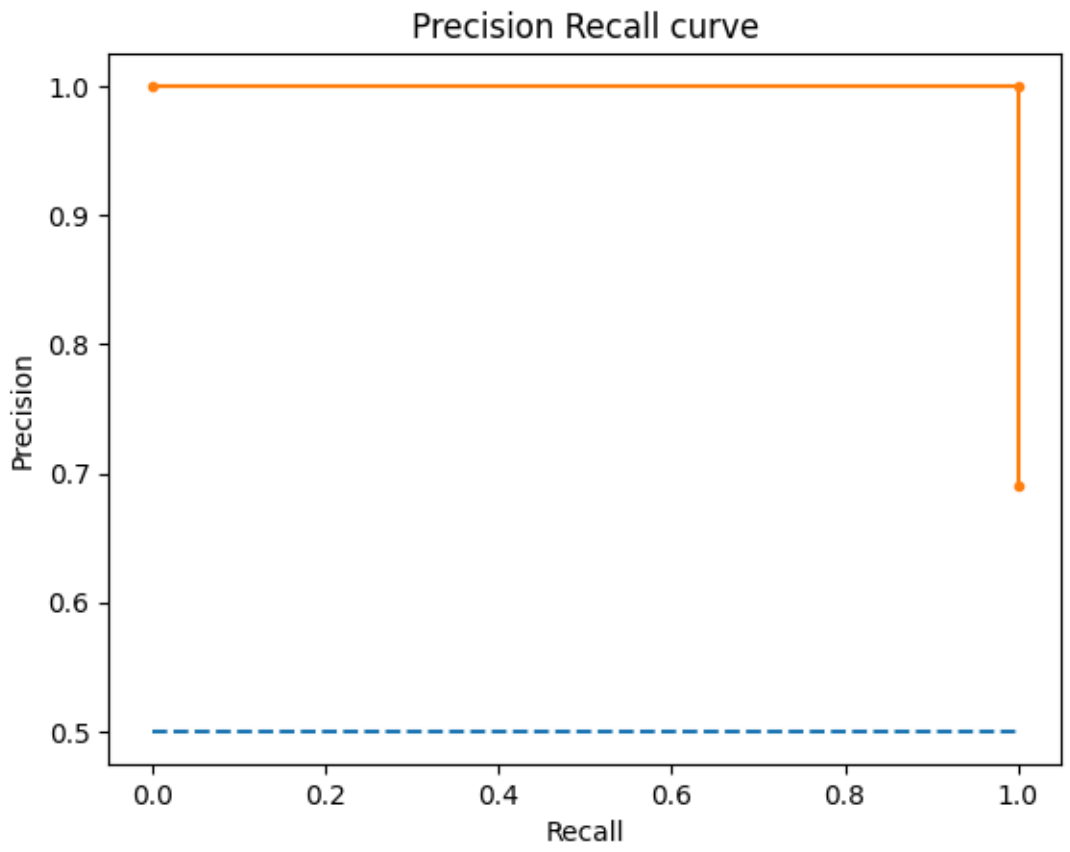
```
Train Score :  1.0

Test Score :  1.0

Accuracy Score :  1.0

[[148   0]
 [  0 329]] ---> confusion Matrix

ROC-AUC score  test dataset:   1.0

precision score  test dataset:   1.0

Recall score  test dataset:   1.0

f1 score  test dataset :   1.0
```

- We can observe that, the train score, test score and accuracy score of our model for gradient boosting classifier is 100%.
- Also, we can see the ROC-AUC score, precision score and recall score is 100%. These are impressive.
- From the confusion matrix, we can observe that there is no false negative as well as no false positive.

**ROC-AUC Curve:**



**Precision Recall Curve:**




**Finally we can say, our model is best model. It's impressive that our random forest classification as well as gradient boosting classification score accuracy is 100%. Although, 100% accuracy means higher chance of overfiting. Since I am taking max_depth for RF is 2, max_features is 4 for RF and low parameters value for GBC, still getting 100% score. It makes me sure that my model is best model and it is not overfitted. And I think my missing values treatment, scaling and labeling makes this dataset perfect.**

---

**Colab notebook link:**

**https://colab.research.google.com/drive/1BvE9EArigs6pNnnAdf1B4X7uNWQAL5DB?usp=sharing**