In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier, NearestNeighbors
from sklearn.svm import SVC
from sklearn.metrics import roc_curve, accuracy_score, precision_score, recall_scor
from sklearn import preprocessing

%matplotlib inline
```

In [2]:
```
data_path = "EmployeeAttritionDataset.csv"
df=pd.read_csv(data_path)
df.shape
```

Out[2]:
```
(1470, 35)
```

Number of rows / Number of samples: 1470

Number of columns / Number of features: 35

In [3]:
```
df.columns
```

Out[3]:
```
Index(['Age', 'Attrition', 'BusinessTravel', 'DailyRate', 'Department',
       'DistanceFromHome', 'Education', 'EducationField', 'EmployeeCount',
       'EmployeeNumber', 'EnvironmentSatisfaction', 'Gender', 'HourlyRate',
       'JobInvolvement', 'JobLevel', 'JobRole', 'JobSatisfaction',
       'MaritalStatus', 'MonthlyIncome', 'MonthlyRate', 'NumCompaniesWorked',
       'Over18', 'OverTime', 'PercentSalaryHike', 'PerformanceRating',
       'RelationshipSatisfaction', 'StandardHours', 'StockOptionLevel',
       'TotalWorkingYears', 'TrainingTimesLastYear', 'WorkLifeBalance',
       'YearsAtCompany', 'YearsInCurrentRole', 'YearsSinceLastPromotion',
       'YearsWithCurrManager'],
      dtype='object')
```

These are all the columns (Attributes) in the dataset.

Attrition is our target variable

In [4]:
```
pd.set_option('display.max_columns', None)
df.head()
```

Out[4]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Educa
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life
4	27	No	Travel_Rarely	591	Research & Development	2	1	

This is what the dataset initially looks like

In [5]:
```
df=df.drop(['EmployeeCount', 'EmployeeNumber', 'Over18', 'StandardHours', 'HourlyRa
df.head()
```

Out[5]:

	Age	Attrition	BusinessTravel	Department	DistanceFromHome	Education	EducationField	E
0	41	Yes	Travel_Rarely	Sales	1	2	Life Sciences	
1	49	No	Travel_Frequently	Research & Development	8	1	Life Sciences	
2	37	Yes	Travel_Rarely	Research & Development	2	2	Other	
3	33	No	Travel_Frequently	Research & Development	3	4	Life Sciences	
4	27	No	Travel_Rarely	Research & Development	2	1	Medical	

We dropped EmployeeCount because it is always 1

EmployeeNumber because it is just the id for each employee

Over18 because every employee is above 18

StandardHours is 80hrs for all

HourlyRate, MonthlyRate, DailyRate do not make sense in the data and are in unreasonable ranges, and also since we already have monthly income factor, we can use that for analysis purpose.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1470 entries, 0 to 1469
Data columns (total 28 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Age                       1470 non-null   int64
 1   Attrition                 1470 non-null   object
 2   BusinessTravel            1470 non-null   object
 3   Department                1470 non-null   object
 4   DistanceFromHome          1470 non-null   int64
 5   Education                 1470 non-null   int64
 6   EducationField            1470 non-null   object
 7   EnvironmentSatisfaction   1470 non-null   int64
 8   Gender                    1470 non-null   object
 9   JobInvolvement            1470 non-null   int64
 10  JobLevel                  1470 non-null   int64
 11  JobRole                   1470 non-null   object
 12  JobSatisfaction           1470 non-null   int64
 13  MaritalStatus             1470 non-null   object
 14  MonthlyIncome             1470 non-null   int64
 15  NumCompaniesWorked        1470 non-null   int64
 16  OverTime                  1470 non-null   object
 17  PercentSalaryHike         1470 non-null   int64
 18  PerformanceRating         1470 non-null   int64
 19  RelationshipSatisfaction  1470 non-null   int64
 20  StockOptionLevel          1470 non-null   int64
 21  TotalWorkingYears         1470 non-null   int64
 22  TrainingTimesLastYear     1470 non-null   int64
 23  WorkLifeBalance           1470 non-null   int64
 24  YearsAtCompany            1470 non-null   int64
 25  YearsInCurrentRole        1470 non-null   int64
 26  YearsSinceLastPromotion   1470 non-null   int64
 27  YearsWithCurrManager      1470 non-null   int64
dtypes: int64(20), object(8)
memory usage: 321.7+ KB
```

In [7]:
```
tp = df['NumCompaniesWorked'] == 0

num_samples_with_zero_companies = tp.sum()

print("Number of samples where NumCompaniesWorked is 0:", num_samples_with_zero_com
```

```
Number of samples where NumCompaniesWorked is 0: 197
```

Since there are samples where number of companies worked has a value of 0, to get the number os years in one company, we will divide the total number of working years by number of companies worked plus one.

In [8]:
```
df['YearsInOneCompany']= df['TotalWorkingYears'] / (df['NumCompaniesWorked']+1)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1470 entries, 0 to 1469
Data columns (total 29 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Age                       1470 non-null   int64
 1   Attrition                 1470 non-null   object
 2   BusinessTravel            1470 non-null   object
 3   Department                1470 non-null   object
 4   DistanceFromHome          1470 non-null   int64
 5   Education                 1470 non-null   int64
 6   EducationField            1470 non-null   object
 7   EnvironmentSatisfaction   1470 non-null   int64
 8   Gender                    1470 non-null   object
 9   JobInvolvement            1470 non-null   int64
 10  JobLevel                  1470 non-null   int64
 11  JobRole                   1470 non-null   object
 12  JobSatisfaction           1470 non-null   int64
 13  MaritalStatus             1470 non-null   object
 14  MonthlyIncome             1470 non-null   int64
 15  NumCompaniesWorked        1470 non-null   int64
 16  OverTime                  1470 non-null   object
 17  PercentSalaryHike         1470 non-null   int64
 18  PerformanceRating         1470 non-null   int64
 19  RelationshipSatisfaction  1470 non-null   int64
 20  StockOptionLevel          1470 non-null   int64
 21  TotalWorkingYears         1470 non-null   int64
 22  TrainingTimesLastYear     1470 non-null   int64
 23  WorkLifeBalance           1470 non-null   int64
 24  YearsAtCompany            1470 non-null   int64
 25  YearsInCurrentRole        1470 non-null   int64
 26  YearsSinceLastPromotion   1470 non-null   int64
 27  YearsWithCurrManager      1470 non-null   int64
 28  YearsInOneCompany         1470 non-null   float64
dtypes: float64(1), int64(20), object(8)
memory usage: 333.2+ KB
```

We added the column named YearsInOneCompany, since it shows on an average how many years did the employee stay in each company he has worked in.

In [9]:
```
df.head()
```

Out[9]:

	Age	Attrition	BusinessTravel	Department	DistanceFromHome	Education	EducationField	E
0	41	Yes	Travel_Rarely	Sales	1	2	Life Sciences	
1	49	No	Travel_Frequently	Research & Development	8	1	Life Sciences	
2	37	Yes	Travel_Rarely	Research & Development	2	2	Other	
3	33	No	Travel_Frequently	Research & Development	3	4	Life Sciences	
4	27	No	Travel_Rarely	Research & Development	2	1	Medical	

Now, after removing the non required columns, and adding one column, the number of columns is 29

In [10]:
```
df.isnull().sum()
```

Out[10]:
```
Age                         0
Attrition                   0
BusinessTravel              0
Department                  0
DistanceFromHome            0
Education                   0
EducationField              0
EnvironmentSatisfaction     0
Gender                      0
JobInvolvement              0
JobLevel                    0
JobRole                     0
JobSatisfaction             0
MaritalStatus               0
MonthlyIncome               0
NumCompaniesWorked          0
OverTime                    0
PercentSalaryHike           0
PerformanceRating           0
RelationshipSatisfaction    0
StockOptionLevel            0
TotalWorkingYears           0
TrainingTimesLastYear       0
WorkLifeBalance             0
YearsAtCompany              0
YearsInCurrentRole          0
YearsSinceLastPromotion     0
YearsWithCurrManager        0
YearsInOneCompany           0
dtype: int64
```

There are no null values in any of the column

Here we can see that all the columns are within the expected range and there aren't any weird values.

In [11]:
```
df['isMale'] = df['Gender'].replace({'Male': 1, 'Female': 0})
df = df.drop(['Gender'],axis=1)
```

In [12]:
```
df['Attrition'] = df['Attrition'].replace({'Yes': 1, 'No': 0})
```

In [13]:
```
df['OverTime'] = df['OverTime'].replace({'Yes': 1, 'No': 0})
```

We have converted OverTime, Gender and Attrition to binary variables because they are only two categories in them.

In [14]:
```
df.head()
```

Out[14]:

	Age	Attrition	BusinessTravel	Department	DistanceFromHome	Education	EducationField	E
0	41	1	Travel_Rarely	Sales	1	2	Life Sciences	
1	49	0	Travel_Frequently	Research & Development	8	1	Life Sciences	
2	37	1	Travel_Rarely	Research & Development	2	2	Other	
3	33	0	Travel_Frequently	Research & Development	3	4	Life Sciences	
4	27	0	Travel_Rarely	Research & Development	2	1	Medical	

In [15]:
```
df.describe().round(3)
```

Out[15]:

	Age	Attrition	DistanceFromHome	Education	EnvironmentSatisfaction	JobInvolvemen
count	1470.000	1470.000	1470.000	1470.000	1470.000	1470.00
mean	36.924	0.161	9.193	2.913	2.722	2.73
std	9.135	0.368	8.107	1.024	1.093	0.71
min	18.000	0.000	1.000	1.000	1.000	1.00
25%	30.000	0.000	2.000	2.000	2.000	2.00
50%	36.000	0.000	7.000	3.000	3.000	3.00
75%	43.000	0.000	14.000	4.000	4.000	3.00
max	60.000	1.000	29.000	5.000	4.000	4.00

In [16]:
```
list_categories=df['YearsAtCompany'].value_counts()
print(list_categories)
```

```
YearsAtCompany
5     196
1     171
3     128
2     127
10    120
4     110
7      90
9      82
8      80
6      76
0      44
11     32
20     27
13     24
15     20
14     18
22     15
12     14
21     14
18     13
16     12
19     11
17      9
24      6
33      5
25      4
26      4
31      3
32      3
27      2
36      2
29      2
23      2
37      1
40      1
34      1
30      1
Name: count, dtype: int64
```

In [17]:

```
attrition_counts = df.groupby('BusinessTravel')['Attrition'].value_counts().unstack

# Calculate the percentage of 1s in each category
attrition_percent = (attrition_counts[1] / attrition_counts.sum(axis=1)) * 100

# Display the counts and percentage
print("Counts of 1s and 0s in Attrition column for each category:")
print(attrition_counts)
print("\nPercentage of 1s in each category of BusinessTravel:")
print(attrition_percent.round(3))
```

```
Counts of 1s and 0s in Attrition column for each category:
Attrition            0    1
BusinessTravel
Non-Travel         138   12
Travel_Frequently  208   69
Travel_Rarely      887  156

Percentage of 1s in each category of BusinessTravel:
BusinessTravel
Non-Travel           8.000
Travel_Frequently   24.910
Travel_Rarely       14.957
dtype: float64
```

Here we can see that maximum attrition happens in cases where business travel is frequent, and attrition is low where there in no business travel.

This means that business travel as an attribute is important.

In [18]:
```
attrition_counts = df.groupby('Department')['Attrition'].value_counts().unstack()

# Calculate the percentage of 1s in each category
attrition_percent = (attrition_counts[1] / attrition_counts.sum(axis=1)) * 100

# Display the counts and percentage
print("Counts of 1s and 0s in Attrition column for each category:")
print(attrition_counts)
print("\nPercentage of 1s in each category of BusinessTravel:")
print(attrition_percent.round(3))
```

```
Counts of 1s and 0s in Attrition column for each category:
Attrition                 0    1
Department
Human Resources          51   12
Research & Development  828  133
Sales                   354   92

Percentage of 1s in each category of BusinessTravel:
Department
Human Resources           19.048
Research & Development    13.840
Sales                     20.628
dtype: float64
```

Attrition rate in sales and HR department is almost the same whereas it a bit lower in R&D department.

In [19]:
```
attrition_counts = df.groupby('EducationField')['Attrition'].value_counts().unstack

# Calculate the percentage of 1s in each category
attrition_percent = (attrition_counts[1] / attrition_counts.sum(axis=1)) * 100

# Display the counts and percentage
print("Counts of 1s and 0s in Attrition column for each category:")
print(attrition_counts)
print("\nPercentage of 1s in each category of BusinessTravel:")
print(attrition_percent.round(3))
```

```
Counts of 1s and 0s in Attrition column for each category:
Attrition           0   1
EducationField
Human Resources    20   7
Life Sciences     517  89
Marketing         124  35
Medical           401  63
Other              71  11
Technical Degree  100  32

Percentage of 1s in each category of BusinessTravel:
EducationField
Human Resources     25.926
Life Sciences       14.686
Marketing           22.013
Medical             13.578
Other               13.415
Technical Degree    24.242
dtype: float64
```

This observation goes in line with what we saw above. Maximum attrition happens when education field is human resources. Education in a technical field and marketing also seems to be having high attrition, almost same as HR. Attrition rate is lowest in Medical, life sciences and other fields.

In [20]:
```
attrition_counts = df.groupby('JobRole')['Attrition'].value_counts().unstack()

# Calculate the percentage of 1s in each category
attrition_percent = (attrition_counts[1] / attrition_counts.sum(axis=1)) * 100

# Display the counts and percentage
print("Counts of 1s and 0s in Attrition column for each category:")
print(attrition_counts)
print("\nPercentage of 1s in each category of BusinessTravel:")
print(attrition_percent.round(3))
```

```
Counts of 1s and 0s in Attrition column for each category:
Attrition                   0   1
JobRole
Healthcare Representative  122   9
Human Resources             40  12
Laboratory Technician      197  62
Manager                     97   5
Manufacturing Director     135  10
Research Director           78   2
Research Scientist         245  47
Sales Executive            269  57
Sales Representative        50  33

Percentage of 1s in each category of BusinessTravel:
JobRole
Healthcare Representative     6.870
Human Resources              23.077
Laboratory Technician        23.938
Manager                       4.902
Manufacturing Director        6.897
Research Director             2.500
Research Scientist           16.096
Sales Executive              17.485
Sales Representative         39.759
dtype: float64
```

From the above data we can see that the attrition rate is very high for sales representatives. Following that, it is HR and laboratory technician. Managers, Research directors, manufacturing directors and healthcare representatives have very low attrition.

This means that Job Role as an attribute is important.

All the above observations go in line with each other. Sales representatives would be the ones who travel frequently and have a much hectic work schedule compared to the ones in fields like medical, research and managers.

In [21]:
```
attrition_counts = df.groupby('MaritalStatus')['Attrition'].value_counts().unstack(

# Calculate the percentage of 1s in each category
attrition_percent = (attrition_counts[1] / attrition_counts.sum(axis=1)) * 100

# Display the counts and percentage
print("Counts of 1s and 0s in Attrition column for each category:")
print(attrition_counts)
print("\nPercentage of 1s in each category of BusinessTravel:")
print(attrition_percent.round(3))
```

```
Counts of 1s and 0s in Attrition column for each category:
Attrition        0    1
MaritalStatus
Divorced       294   33
Married        589   84
Single         350  120

Percentage of 1s in each category of BusinessTravel:
MaritalStatus
Divorced    10.092
Married     12.481
Single      25.532
dtype: float64
```

We see that there is a lot more attrition in single employees compared to married employees. We can assume that the divorced employees would be of much higher age than single employees, and would be having more family responsibilities which means they would not leave their jobs that easily.

This means that marital status as an attribute is important.

In [22]:
```
attrition_counts = df.groupby('StockOptionLevel')['Attrition'].value_counts().unsta

# Calculate the percentage of 1s in each category
attrition_percent = (attrition_counts[1] / attrition_counts.sum(axis=1)) * 100

# Display the counts and percentage
print("Counts of 1s and 0s in Attrition column for each category:")
print(attrition_counts)
print("\nPercentage of 1s in each category of BusinessTravel:")
print(attrition_percent.round(3))
```

```
Counts of 1s and 0s in Attrition column for each category:
Attrition           0    1
StockOptionLevel
0                 477  154
1                 540   56
2                 146   12
3                  70   15

Percentage of 1s in each category of BusinessTravel:
StockOptionLevel
0    24.406
1     9.396
2     7.595
3    17.647
dtype: float64
```

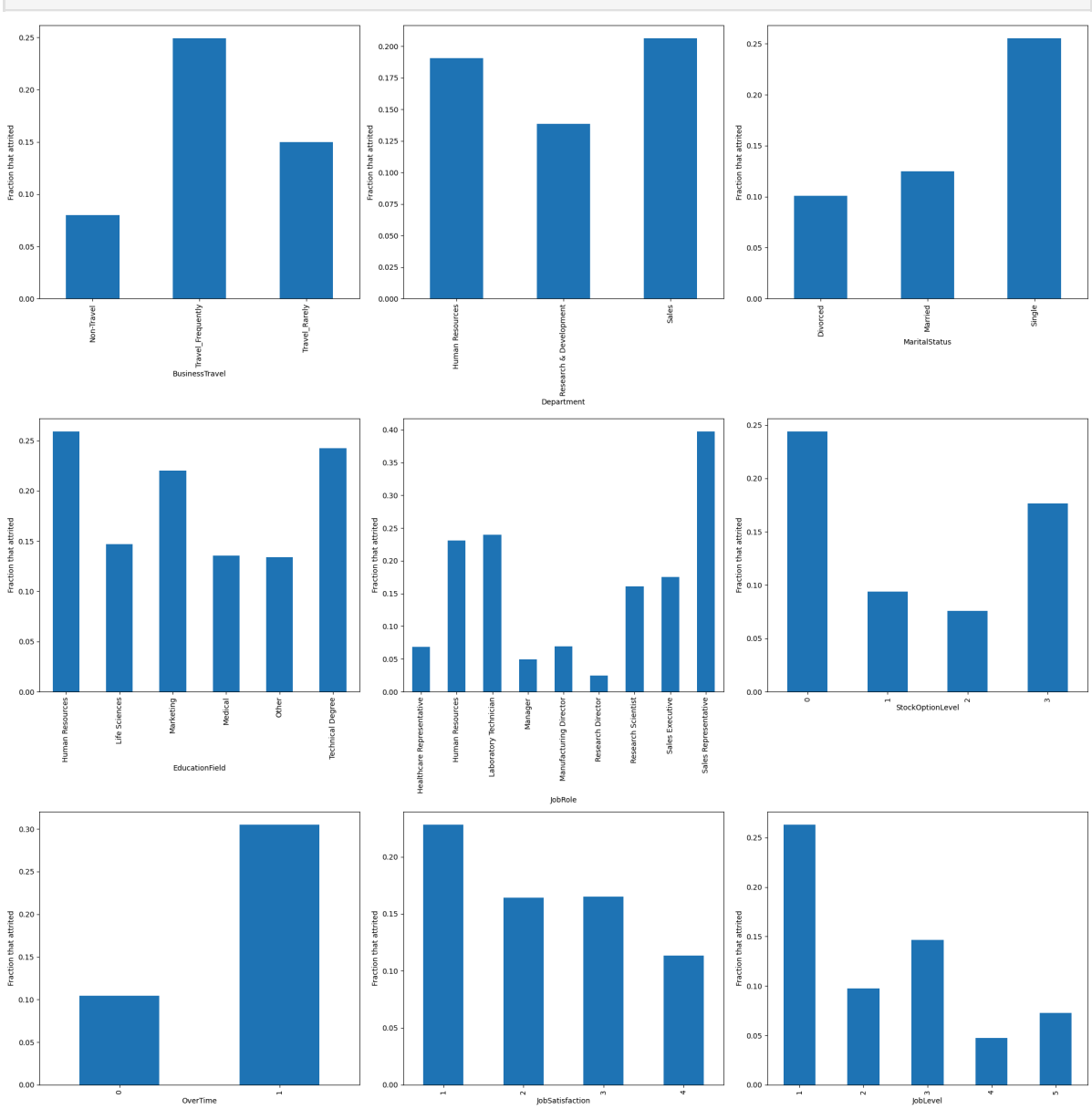
Employees with stock option level of 0 have a high attrition rate. This might mean that employees having low stock option are the ones with lower job levels which are more likely to leave the company.

In [23]:
```
attrition_counts = df.groupby('OverTime')['Attrition'].value_counts().unstack()

# Calculate the percentage of 1s in each category
attrition_percent = (attrition_counts[1] / attrition_counts.sum(axis=1)) * 100

# Display the counts and percentage
print("Counts of 1s and 0s in Attrition column for each category:")
print(attrition_counts)
print("\nPercentage of 1s in each category of BusinessTravel:")
print(attrition_percent.round(3))
```

```
Counts of 1s and 0s in Attrition column for each category:
Attrition    0    1
OverTime
0          944  110
1          289  127

Percentage of 1s in each category of BusinessTravel:
OverTime
0    10.436
1    30.529
dtype: float64
```

Representing the above categorical data in charts.

In [24]:
```
# Names of different columns
categorical_cols = ["BusinessTravel", "Department", "MaritalStatus", "EducationFiel
target_col = "Attrition"
predictor_cols = categorical_cols

# This is to plot everything in a 2x2 space
rows, cols = 3, 3
fig, axs = plt.subplots(ncols=cols, nrows=rows, figsize=(7*cols, 7*rows))
axs = axs.flatten()
posn = 0

# Plot categorical features
for col in categorical_cols:
    df.groupby(col)[target_col].mean().plot(kind="bar", ax=axs[posn])
    axs[posn].set_ylabel("Fraction that attrited")
    posn += 1

plt.tight_layout()
```



In [25]:
```
dummies = pd.get_dummies(df['Department'], prefix='Department', drop_first=True)
dummies = dummies.astype(int)
df = pd.concat([df, dummies], axis=1)

df.drop(columns=['Department'], inplace=True)
```

In [26]:
```
dummies = pd.get_dummies(df['BusinessTravel'], prefix='BusinessTravel', drop_first=
dummies = dummies.astype(int)
df = pd.concat([df, dummies], axis=1)

df.drop(columns=['BusinessTravel'], inplace=True)
```

In [27]:
```
dummies = pd.get_dummies(df['MaritalStatus'], prefix='MaritalStatus', drop_first=Tr
dummies = dummies.astype(int)
df = pd.concat([df, dummies], axis=1)

df.drop(columns=['MaritalStatus'], inplace=True)
```

In [28]:
```
dummies = pd.get_dummies(df['EducationField'], prefix='EducationField', drop_first=
dummies = dummies.astype(int)
df = pd.concat([df, dummies], axis=1)

df.drop('EducationField', axis=1, inplace=True)
```

In [29]:
```
dummies = pd.get_dummies(df['JobRole'], prefix='JobRole', drop_first=True)
dummies = dummies.astype(int)
df = pd.concat([df, dummies], axis=1)

df.drop(columns=['JobRole'], inplace=True)
```

Did one-hot encoding for the categorical columns

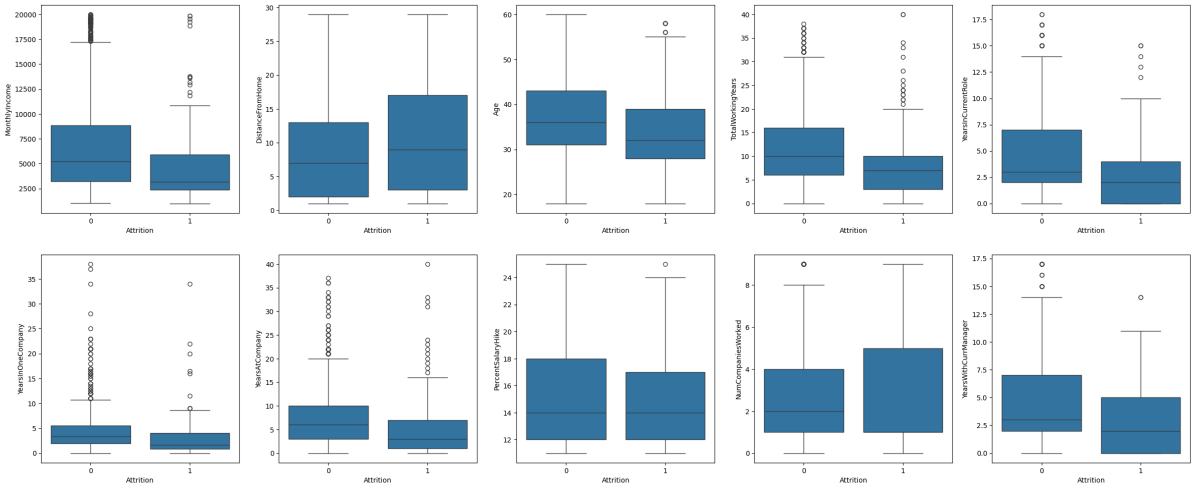
In [30]:
```
df.head()
```

Out[30]:

	Age	Attrition	DistanceFromHome	Education	EnvironmentSatisfaction	JobInvolvement	JobLe
0	41	1	1	2	2	3	
1	49	0	8	1	3	2	
2	37	1	2	2	4	2	
3	33	0	3	4	4	3	
4	27	0	2	1	1	3	

As we can see in the above data samples, the columns BusinessTravel,Department, MaritalStatus, EducationField, JobRole, have been converted to one-hot encoded columns

In [31]:
```
## side-by-side boxplots
import seaborn as sns
import matplotlib.pyplot as plt

fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(30, 12))


sns.boxplot(x='Attrition', y='MonthlyIncome', data=df, ax=axes[0,0])
sns.boxplot(x='Attrition', y='DistanceFromHome', data=df, ax=axes[0,1])
sns.boxplot(x='Attrition', y='Age', data=df, ax=axes[0,2])
sns.boxplot(x='Attrition', y='TotalWorkingYears', data=df, ax=axes[0,3])
sns.boxplot(x='Attrition', y='YearsInCurrentRole', data=df, ax=axes[0,4])
sns.boxplot(x='Attrition', y='YearsInOneCompany', data=df, ax=axes[1,0])
sns.boxplot(x='Attrition', y='YearsAtCompany', data=df, ax=axes[1,1])
sns.boxplot(x='Attrition', y='PercentSalaryHike', data=df, ax=axes[1,2])
sns.boxplot(x='Attrition', y='NumCompaniesWorked', data=df, ax=axes[1,3])
sns.boxplot(x='Attrition', y='YearsWithCurrManager', data=df, ax=axes[1,4])

plt.suptitle("")

plt.show()
```



Here, we have plotted all the continuous variables againt attrition.

Employees who have lower monthly income are more likely to attrite than employees with higher income. We can only see few cases where high income employees left the company. This goes in line with the above observation of job role.

Also, employees leaving the company are found to be living far from the office. (Not much of a difference in distance but might be one of the factors affecting the employee).

Majority of the employees attriting are observed to be around 30. The employees not attriting are around the age of late 30s. Having higher attrition chance in early 30s might be due to salary jumps offered by other companies, and still comparatively lesser family responsibilities.

Similar observation can be made by total working years. Employees attriting have generally worked lesser number of years than non attriting ones.

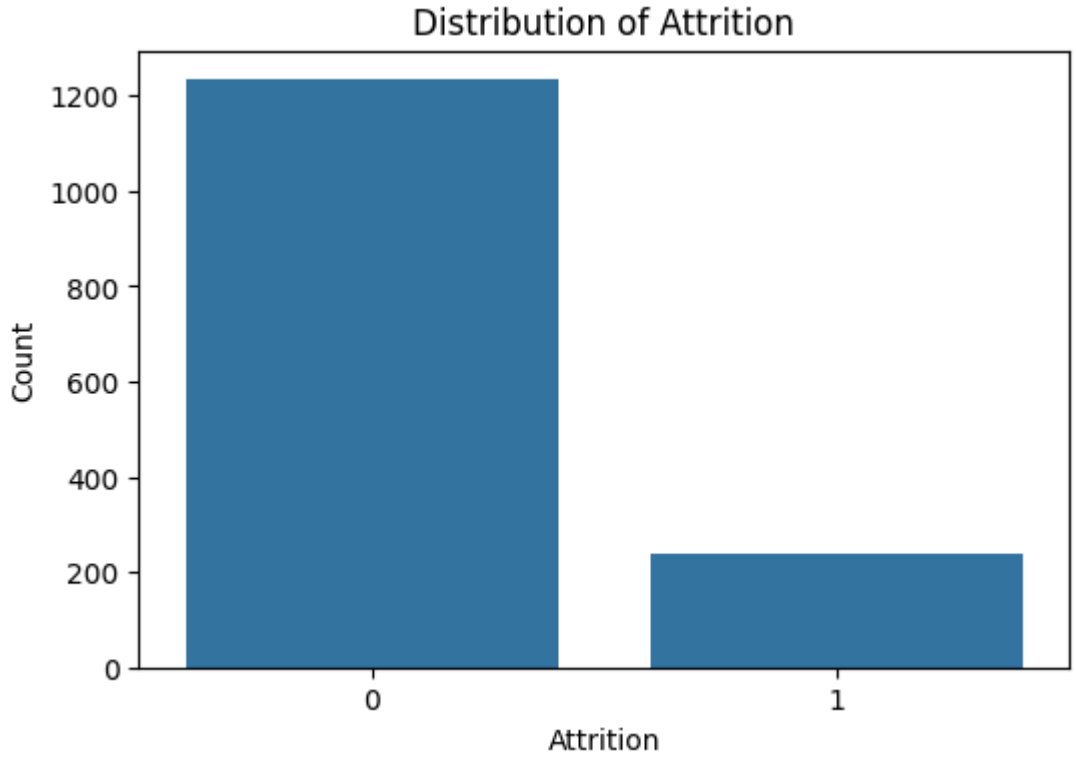
Attriting employees are also seen to be working with one company for lesser number of years compared to the non attriting employees. Which means they might be the ones who change companies more frequently.

We do not see any effect of percent salary hike. We would normally expect employees with higher salary hike to be staying and lower salary hike attriting, but no clear evidence from the boxplot.

Attriting employees are seen to be working with their current manager for lesser number of years than employees that do not attrite. It is possible that employees who are working with the same manager for a longer period of time, have good relations with their manager and are less likely to leave the company.

In [32]:
```
# Create a bar plot
plt.figure(figsize=(6, 4))  # Adjust size as needed
sns.countplot(data=df, x='Attrition')

# Add labels and title
plt.xlabel('Attrition')
plt.ylabel('Count')
plt.title('Distribution of Attrition')
```

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



In [33]:
```
df["Attrition"].value_counts()/df["Attrition"].count()
```

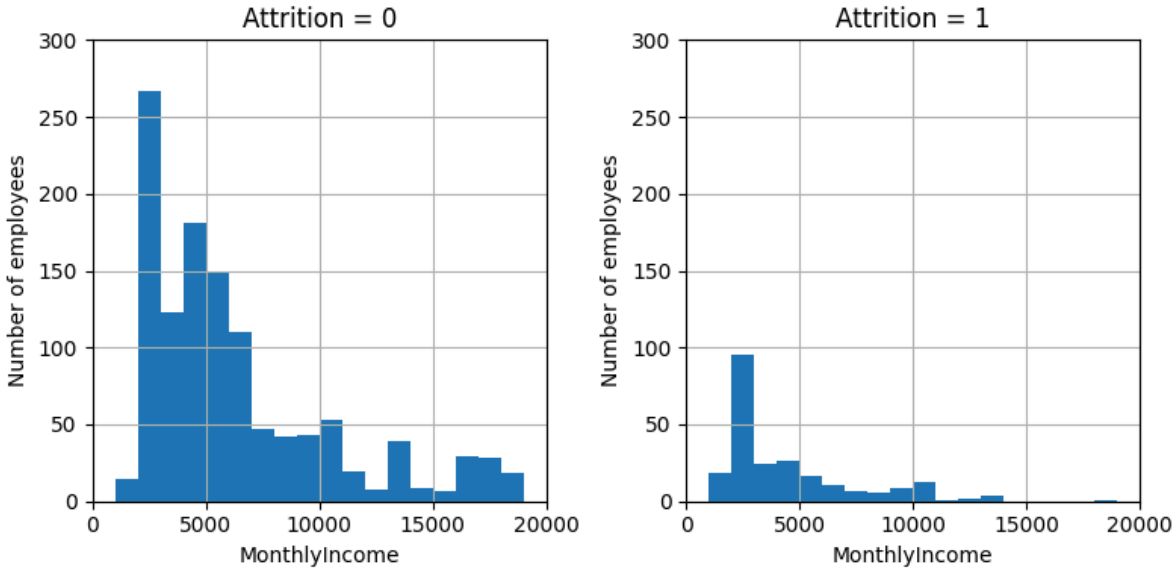
Out[33]:
```
Attrition
0    0.838776
1    0.161224
Name: count, dtype: float64
```

BASE RATE is 16%. Since if we take out a random sample from the data, the probability that the employee has attrited is 16%.

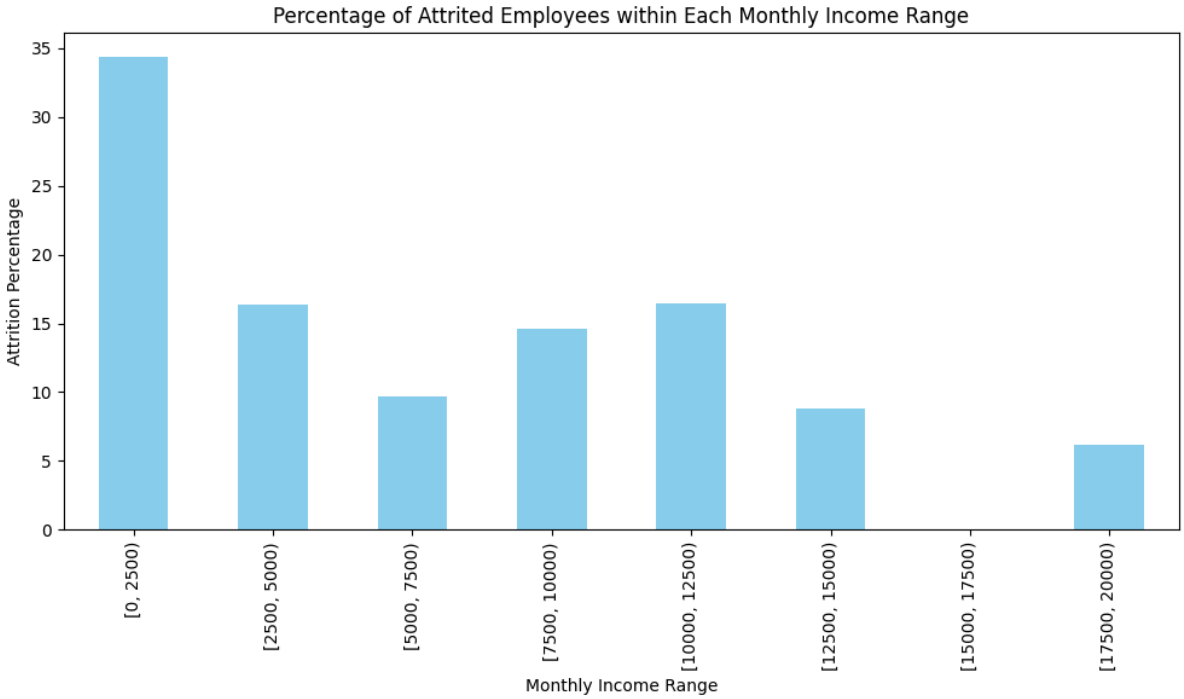
In [34]:
```
df['Attrition'].value_counts()
```

Out[34]:
```
Attrition
0    1233
1     237
Name: count, dtype: int64
```

In [35]:
```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 2, figsize=(8, 4))

x_max = 20000

for r, ax in enumerate(axs):
    hist = df[df.Attrition == r].hist('MonthlyIncome', bins=range(0, x_max, 1000),
    ax.set_title("Attrition = " + str(r))
    ax.set_ylim(0, 300)
    ax.set_xlabel("MonthlyIncome")
    ax.set_xlim([0, x_max])
    ax.set_ylabel('Number of employees')

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



In [36]:
```
# Define the bin edges based on the minimum and maximum monthly income values
bin_edges = list(range(0, df['MonthlyIncome'].max() + 2500, 2500))

# Create a new DataFrame with monthly income bins and attrition status
df['MonthlyIncomeBin'] = pd.cut(df['MonthlyIncome'], bins=bin_edges, right=False)

# Group by monthly income bins and calculate the percentage of attrited employees
attrition_percentage = df.groupby('MonthlyIncomeBin')['Attrition'].mean() * 100

# Plot the bar chart
plt.figure(figsize=(10, 6))
attrition_percentage.plot(kind='bar', color='skyblue')

# Add labels and title
plt.xlabel('Monthly Income Range')
plt.ylabel('Attrition Percentage')
plt.title('Percentage of Attrited Employees within Each Monthly Income Range')

# Show the plot
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()

# Remove the 'MonthlyIncomeBin' column from the DataFrame to revert it back to the
df.drop(columns=['MonthlyIncomeBin'], inplace=True)
```



We can see in the above graph that maximum attrition happens in the bracket of income below 2500

CALCULATING THE ENTROPY AND INFORMATION GAIN FOR DECISION TREE

In [37]:
```
def entropy(target_column):
    """
        computes -sum_i p_i * log_2 (p_i) for each i
    """
    # get the counts of each target value
    target_counts = target_column.value_counts().astype(float).values
    total = target_column.count()
    # compute probas
    probas = target_counts/total
    # p_i * log_2 (p_i)
    entropy_components = probas * np.log2(probas)
    # return negative sum
    return - entropy_components.sum()

def information_gain(df, info_column, target_column, threshold):
    """
        computes H(target) - H(target | info > thresh) - H(target | info <= thresh)
    """
    # split data
    data_above_thresh = df[df[info_column] > threshold]
    data_below_thresh = df[df[info_column] <= threshold]
    # get entropy
    H = entropy(df[target_column])
    entropy_above = entropy(data_above_thresh[target_column])
    entropy_below = entropy(data_below_thresh[target_column])
    # compute weighted average
    ct_above = data_above_thresh.shape[0]
    ct_below = data_below_thresh.shape[0]
    tot = float(df.shape[0])
    return H - entropy_above*ct_above/tot - entropy_below*ct_below/tot
```

FINDING THE RIGHT SPLIT FOR A PARTICULAR FEATURE

In [38]:
```
def best_threshold(df, info_column, target_column, criteria=information_gain):
    maximum_ig = 0
    maximum_threshold = 0

    for thresh in df[info_column].unique():
        IG = criteria(df, info_column, target_column, thresh)
        if IG > maximum_ig:
            maximum_ig = IG
            maximum_threshold = thresh

    return (maximum_threshold, maximum_ig)

col = "YearsInOneCompany"
maximum_threshold, maximum_ig = best_threshold(df, col, "Attrition")

print ("the maximum Information Gain we can achieve splitting on %s is %.4f using a
```

```
the maximum Information Gain we can achieve splitting on YearsInOneCompany is 0.04
21 using a threshold of 1.57
```

FINDING THE BEST SPLIT AMONG ALL THE FEATURES

In [39]:
```
from sklearn.preprocessing import StandardScaler
X = df.drop('Attrition', axis=1)
y = df.Attrition

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [40]:
```
categorical_cols = [
    "OverTime", "Department_Research & Development", "Department_Sales", "BusinessT
    "MaritalStatus_Married", "MaritalStatus_Single", "EducationField_Life Sciences'
    "EducationField_Other", "EducationField_Technical Degree", "JobRole_Human Resou
    "JobRole_Manufacturing Director", "JobRole_Research Director", "JobRole_Researc
    "EnvironmentSatisfaction", "JobLevel","RelationshipSatisfaction", "StockOptionL
    "WorkLifeBalance", "JobInvolvement", "isMale"

]

continuous_cols = [
    "Age", "DistanceFromHome", "MonthlyIncome", "NumCompaniesWorked", "PercentSalar
    "YearsAtCompany", "YearsInCurrentRole", "YearsSinceLastPromotion", "YearsWithCu
    "TrainingTimesLastYear"
]

target_col = "Attrition"
predictor_cols = categorical_cols + continuous_cols

def best_split(df, info_columns, target_column, criteria=information_gain):
    best_information_gains = {}
    maximum_ig = 0
    maximum_threshold = 0
    maximum_column = ""

    for info_column in info_columns:
        thresh, ig = best_threshold(df, info_column, target_col, criteria)
        best_information_gains[info_column] = ig

        if ig > maximum_ig:
            maximum_ig = ig
            maximum_threshold = thresh
```

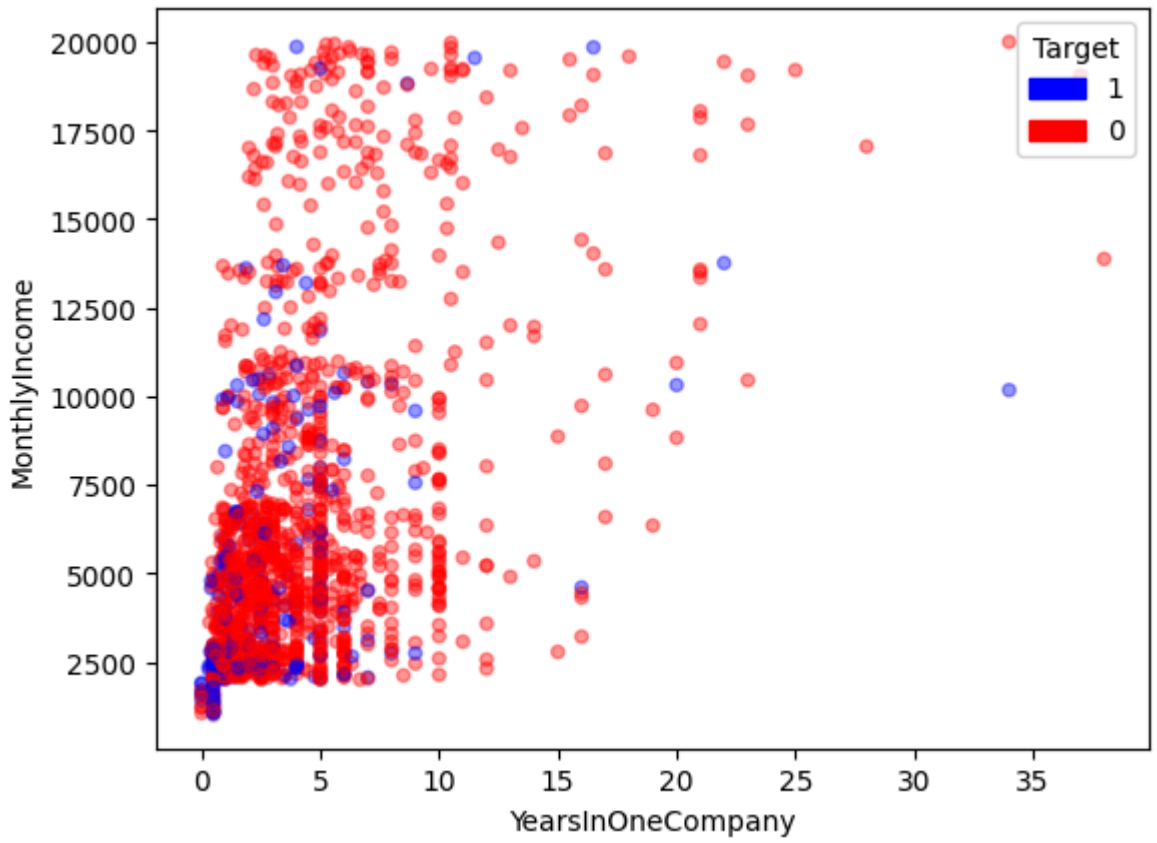
```
            maximum_column = info_column

    # Sort the dictionary by information gain in descending order
    sorted_information_gains = sorted(best_information_gains.items(), key=lambda x:

    return maximum_column, maximum_threshold, maximum_ig, sorted_information_gains

max_col, max_threshold, max_ig, sorted_information_gains = best_split(df, predictor

print("The best column to split on is %s giving us a Information Gain of %.4f using
print("\nInformation Gains on all attributes for the best threshold for each attrib
for column, ig in sorted_information_gains:
    print("%s, Information Gain: %.4f" % (column, ig))
```

```
The best column to split on is YearsInOneCompany giving us a Information Gain of
0.0421 using a threshold of 1.57

Information Gains on all attributes for the best threshold for each attribute:
YearsInOneCompany, Information Gain: 0.0421
OverTime, Information Gain: 0.0399
JobLevel, Information Gain: 0.0314
MonthlyIncome, Information Gain: 0.0300
TotalWorkingYears, Information Gain: 0.0291
StockOptionLevel, Information Gain: 0.0274
YearsAtCompany, Information Gain: 0.0272
YearsWithCurrManager, Information Gain: 0.0264
Age, Information Gain: 0.0237
MaritalStatus_Single, Information Gain: 0.0210
YearsInCurrentRole, Information Gain: 0.0189
JobRole_Sales Representative, Information Gain: 0.0140
EnvironmentSatisfaction, Information Gain: 0.0099
BusinessTravel_Travel_Frequently, Information Gain: 0.0087
JobRole_Research Director, Information Gain: 0.0082
JobInvolvement, Information Gain: 0.0081
JobRole_Manager, Information Gain: 0.0065
JobRole_Laboratory Technician, Information Gain: 0.0064
MaritalStatus_Married, Information Gain: 0.0061
JobRole_Manufacturing Director, Information Gain: 0.0060
WorkLifeBalance, Information Gain: 0.0059
JobSatisfaction, Information Gain: 0.0059
DistanceFromHome, Information Gain: 0.0057
Department_Research & Development, Information Gain: 0.0051
Department_Sales, Information Gain: 0.0045
NumCompaniesWorked, Information Gain: 0.0042
EducationField_Technical Degree, Information Gain: 0.0031
YearsSinceLastPromotion, Information Gain: 0.0027
RelationshipSatisfaction, Information Gain: 0.0024
TrainingTimesLastYear, Information Gain: 0.0024
EducationField_Marketing, Information Gain: 0.0021
BusinessTravel_Travel_Rarely, Information Gain: 0.0017
EducationField_Medical, Information Gain: 0.0016
PercentSalaryHike, Information Gain: 0.0010
Education, Information Gain: 0.0009
JobRole_Human Resources, Information Gain: 0.0009
EducationField_Life Sciences, Information Gain: 0.0008
isMale, Information Gain: 0.0006
JobRole_Sales Executive, Information Gain: 0.0003
EducationField_Other, Information Gain: 0.0002
PerformanceRating, Information Gain: 0.0000
JobRole_Research Scientist, Information Gain: 0.0000
```

In [41]:

```
import matplotlib.patches as mpatches

cmap = {1: 'blue', 0: 'red'}
```

```
df.plot(kind="scatter", x="YearsInOneCompany", y="MonthlyIncome", c=[cmap[c] for c
plt.legend(handles=[mpatches.Patch(color=cmap[k], label=k) for k in cmap], loc=1, t
plt.show()
```

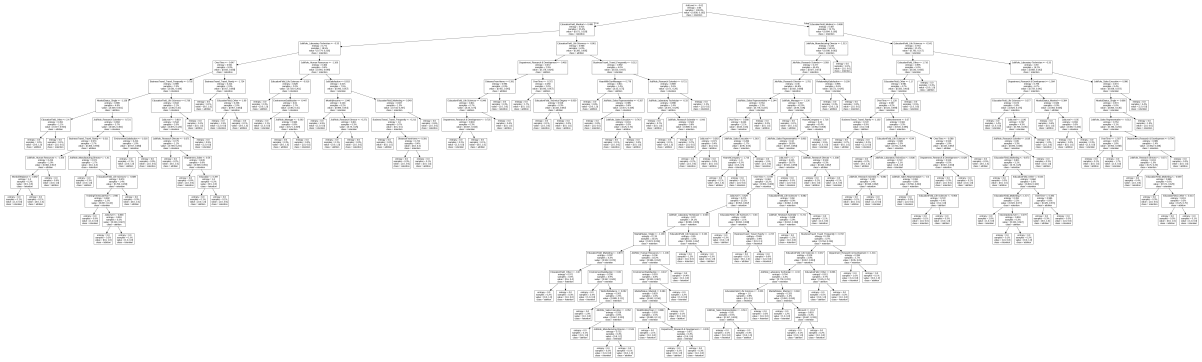


In [42]:
```
from sklearn.tree import DecisionTreeClassifier

decision_tree = DecisionTreeClassifier(max_depth=None, criterion="entropy")   # Loc
# Tell the model what data to use and then train the model
decision_tree.fit(X_train, y_train)
```

Out[42]:
```
DecisionTreeClassifier
DecisionTreeClassifier(criterion='entropy')
```

In [43]:
```
import os
from IPython.display import Image
from sklearn.tree import export_graphviz

def visualize_tree(decision_tree, feature_names, class_names, directory="./images",
    os.makedirs(directory, exist_ok=True)

    dot_name = "%s/%s.dot" % (directory, name)
    dot_file = export_graphviz(decision_tree, out_file=dot_name,
                               feature_names=feature_names, class_names=class_names
    image_name = "%s/%s.png" % (directory, name)
    os.system("dot -T png %s -o %s" % (dot_name, image_name))
    return Image(filename=image_name)


visualize_tree(decision_tree, predictor_cols, ["retention", "attrition"])
```

Out[43]:

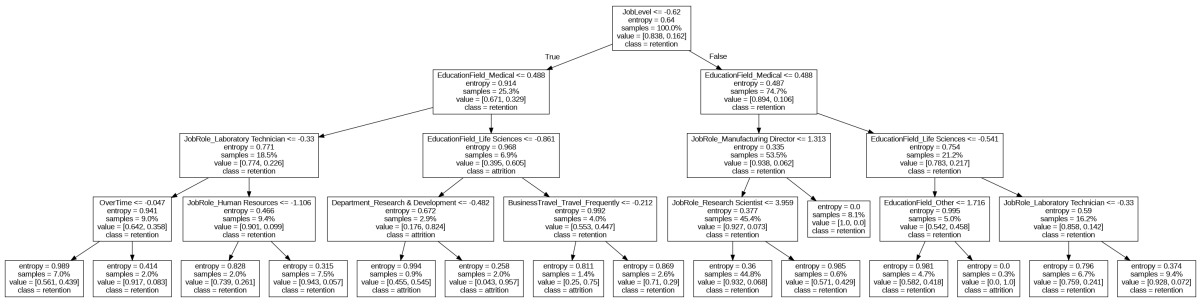


In [44]:
```
# user grid search to find optimized tree
param_grid = {
    'max_depth': [3,4,5,6,7,8],
    'min_impurity_decrease': [0.005,0.01,0.02,0.03],
    'min_samples_split': [8,9,10,11,12],
}
gridSearch = GridSearchCV(DecisionTreeClassifier(), param_grid, cv=5, n_jobs=-1)
gridSearch.fit(X_train, y_train)
print('First level optimal parameters: ', gridSearch.best_params_)
```

```
First level optimal parameters:  {'max_depth': 4, 'min_impurity_decrease': 0.005,
'min_samples_split': 8}
```

In [45]:
```
decision_tree = DecisionTreeClassifier(max_depth=4, criterion="entropy")
decision_tree.fit(X_train , y_train)

visualize_tree(decision_tree, predictor_cols, ["retention", "attrition"])
```

Out[45]:



In [46]:
```
from sklearn import metrics
predictions = decision_tree.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print ( "Decision Tree Classifier Accuracy = ", accuracy)
```

```
Decision Tree Classifier Accuracy =  0.8707482993197279
```

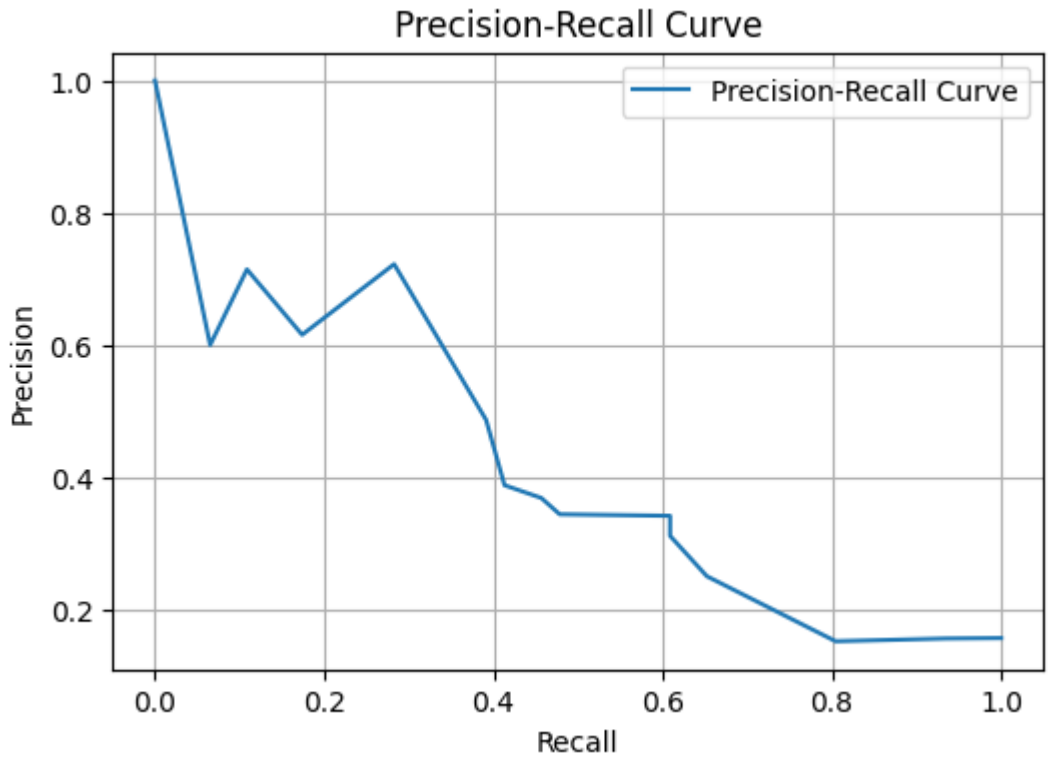
In [47]:
```
from sklearn.metrics import precision_score, recall_score, f1_score

# Calculate precision, recall, and F1 score
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)

print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Precision: 0.7222222222222222
Recall: 0.2826086956521739
F1 Score: 0.40625
```

In [48]:
```
from sklearn.metrics import precision_recall_curve

# Calculate predicted probabilities for the positive class
```

```
y_probs = decision_tree.predict_proba(X_test)[:, 1]

# Calculate precision and recall for different thresholds
precisions, recalls, thresholds = precision_recall_curve(y_test, y_probs)

# Choose the threshold that maximizes F1 score
f1_scores = 2 * (precisions * recalls) / (precisions + recalls)
best_threshold = thresholds[np.argmax(f1_scores)]

# Calculate precision and recall using the best threshold
precision_best = precisions[np.argmax(f1_scores)]
recall_best = recalls[np.argmax(f1_scores)]
# Calculate F1 score at the best threshold
f1_best = f1_scores[np.argmax(f1_scores)]

print("Best Threshold:", best_threshold)
print("Precision at Best Threshold:", precision_best)
print("Recall at Best Threshold:", recall_best)
print("F1 Score at Best Threshold:", f1_best)
# Plot precision-recall curve
plt.figure(figsize=(6,4))
plt.plot(recalls, precisions, label='Precision-Recall Curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.grid(True)
plt.show()
```

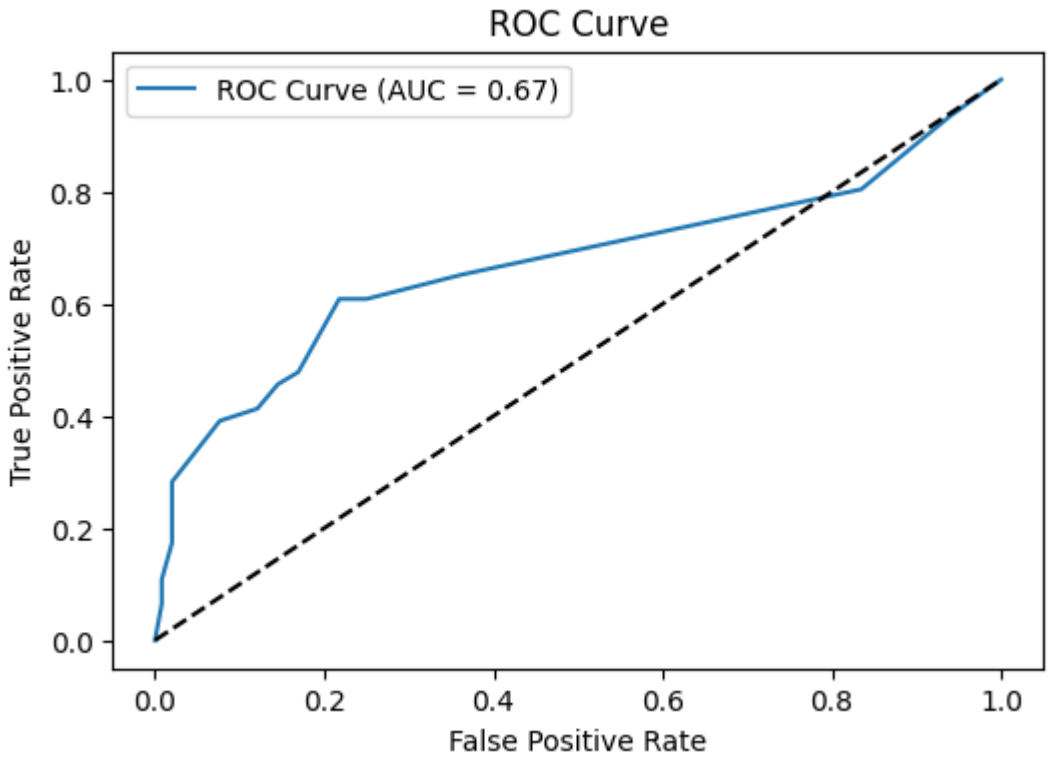
```
Best Threshold: 0.24050632911392406
Precision at Best Threshold: 0.34146341463414637
Recall at Best Threshold: 0.6086956521739131
F1 Score at Best Threshold: 0.4375
```



In [49]:

```
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# Calculate ROC curve and AUC
fpr_dt, tpr_dt, thresholds = roc_curve(y_test, y_probs)
auc = roc_auc_score(y_test, y_probs)
```

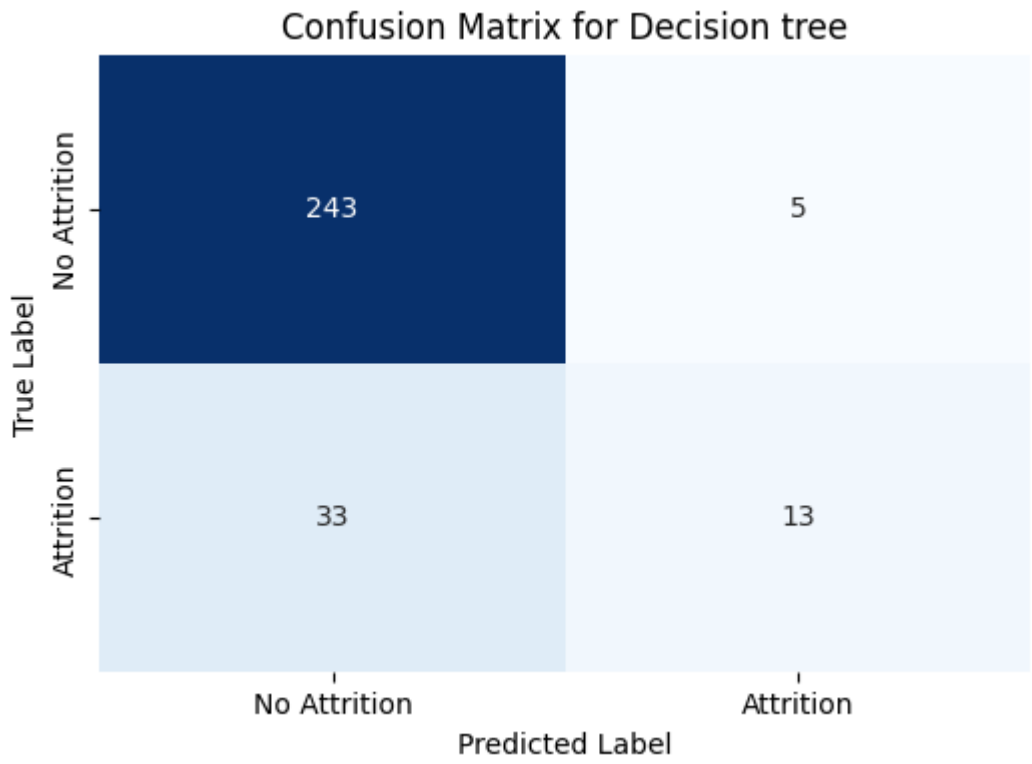
```
# Plot ROC curve
plt.figure(figsize=(6, 4))
plt.plot(fpr_dt, tpr_dt, label='ROC Curve (AUC = {:.2f})'.format(auc))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

print("AUC Score:", auc)
```




```
AUC Score: 0.6710203366058907
```

In [50]:

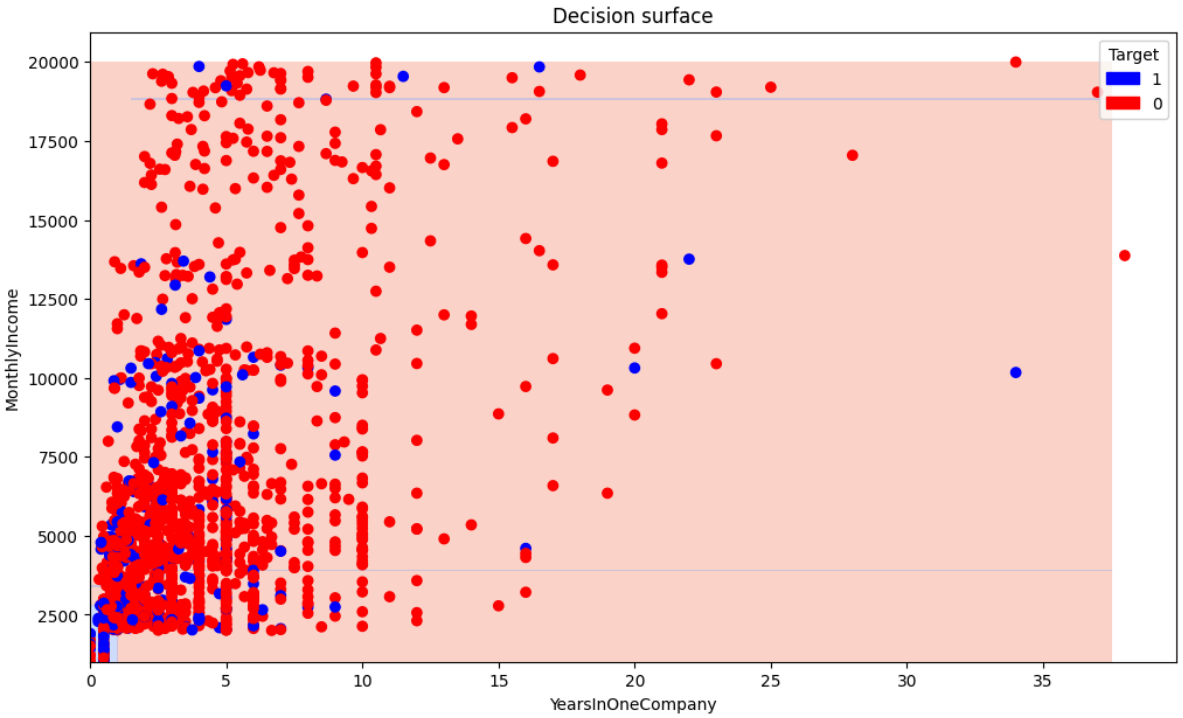
```
from sklearn.metrics import confusion_matrix
# Calculate confusion matrix
conf_matrix_dt = confusion_matrix(y_test, predictions)

# Define labels
labels = ["No Attrition", "Attrition"]

# Plot confusion matrix
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_dt, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix for Decision tree')
plt.ylabel('True Label')
plt.show()
```



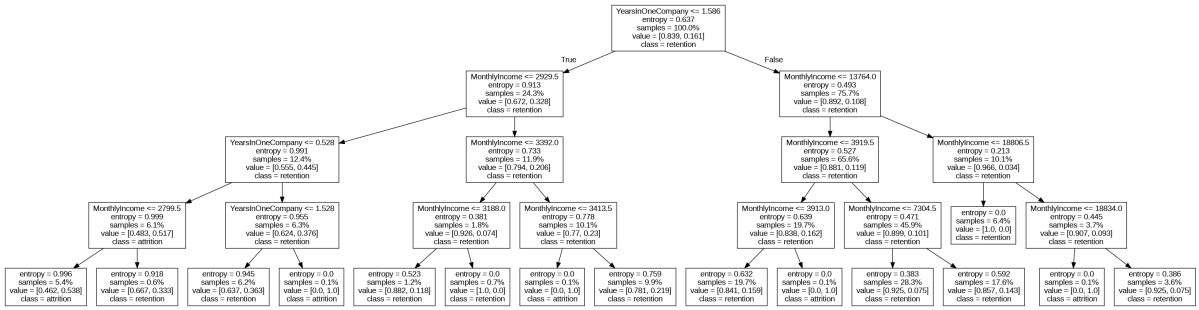
In [51]:
```
def Decision_Surface(data, col1, col2, target, model, probabilities=False):
    # Get bounds
    x_min, x_max = data[col1].min(), data[col1].max()
    y_min, y_max = data[col2].min(), data[col2].max()
    # Create a mesh
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.5), np.arange(y_min, y_max,0.5))
    meshed_data = pd.DataFrame(np.c_[xx.ravel(), yy.ravel()])
    # Get predictions for the mesh
    tdf = data[[col1, col2]]
    model.fit(tdf, target)
    if probabilities:
        Z = model.predict_proba(meshed_data)[:, 1].reshape(xx.shape)
    else:
        Z = model.predict(meshed_data).reshape(xx.shape)
    # Chart details
    plt.figure(figsize=[12,7])
    plt.title("Decision surface")
    plt.xlabel(col1)
    plt.ylabel(col2)
    if probabilities:
        # Color-scale on the contour (surface = separator)
        cs = plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm_r, alpha=0.4)
    else:
        # Only a curve/line on the contour (surface = separator)
        cs = plt.contourf(xx, yy, Z, levels=[-1,0,1], cmap=plt.cm.coolwarm_r, alpha
    # Plot scatter plot
    cmap = {1: 'blue', 0: 'red'}
    colors = [cmap[c] for c in df[target_col]]
    plt.scatter(data[col1], data[col2], color=colors)
    # Build legend
    plt.legend(handles=[mpatches.Patch(color=cmap[k], label=k) for k in cmap], loc=
    plt.show()

# Can change depths here
tree_depth=4
model = DecisionTreeClassifier(max_depth=tree_depth, criterion="entropy")
Decision_Surface(df, "YearsInOneCompany", "MonthlyIncome", df.Attrition, model)
```

```
## just the model on fare and age!!
visualize_tree(model, ["YearsInOneCompany","MonthlyIncome"], ["retention", "attriti
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does n
ot have valid feature names, but DecisionTreeClassifier was fitted with feature na
mes
  warnings.warn(
```



Out[51]:



APPLYING LOGISTIC REGRESSION

In [52]:
```
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
# Make and fit a model on the training data
logistic_regression = LogisticRegression(C=1000000, solver='liblinear')
logistic_regression.fit(X_train, y_train)
y_pred_lr=logistic_regression.predict(X_test)
# Get probabilities of attrition
probabilities = logistic_regression.predict_proba(X_test)[:, 1]
```

In [53]:
```
accuracy_lr = logistic_regression.score(X_test, y_test)
print("Accuracy on logistic regression : ", accuracy_lr)
```

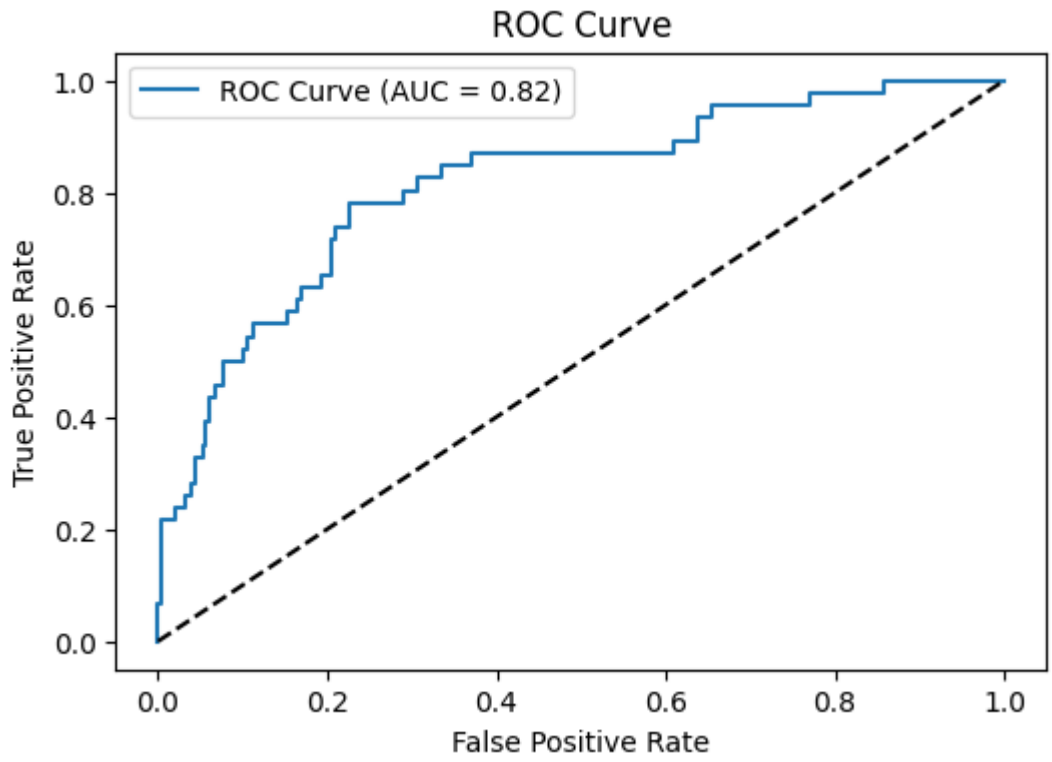
```
Accuracy on logistic regression :  0.8571428571428571
```

In [54]:
```
# Calculate precision, recall, and F1 score using predicted probabilities
precision_lr = metrics.precision_score(y_test, probabilities >= 0.5)
recall_lr = metrics.recall_score(y_test, probabilities >= 0.5)
f1_lr = metrics.f1_score(y_test, probabilities >= 0.5)

print("Precision on logistic regression:", precision_lr)
```

```
print("Recall on logistic regression:", recall_lr)
print("F1 Score on logistic regression:", f1_lr)
```

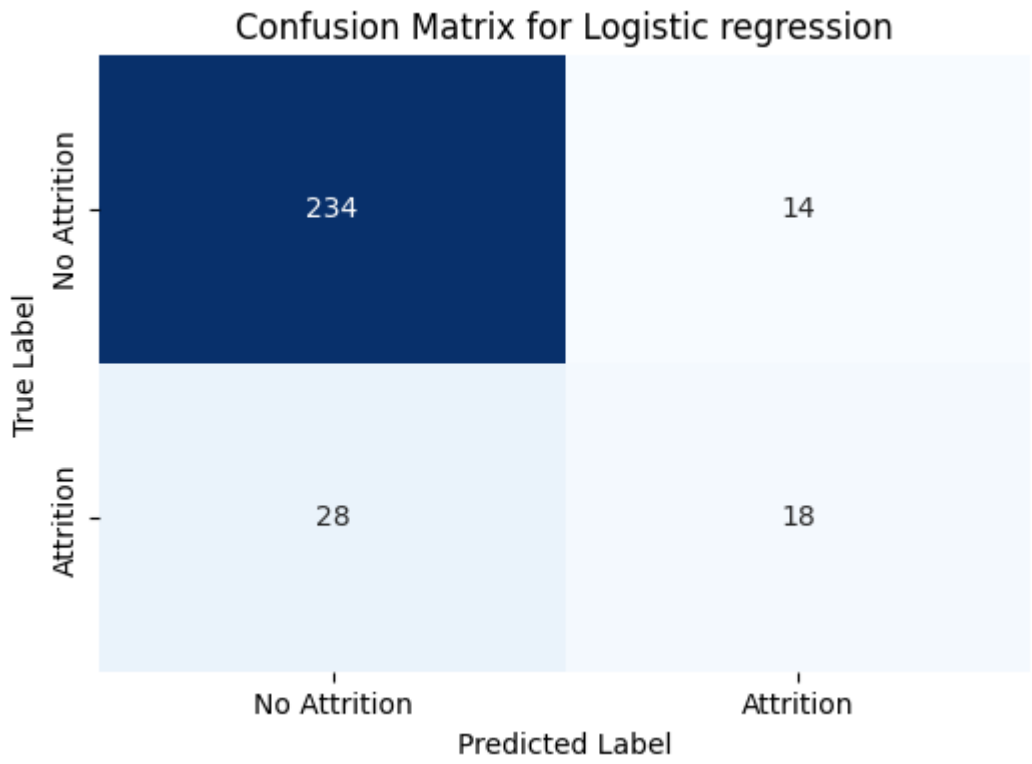
```
Precision on logistic regression: 0.5625
Recall on logistic regression: 0.391304347826087
F1 Score on logistic regression: 0.46153846153846156
```

In [55]:
```
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# Calculate ROC curve and AUC
fpr_lr, tpr_lr, thresholds = roc_curve(y_test, probabilities)
auc = roc_auc_score(y_test, probabilities)

# Plot ROC curve
plt.figure(figsize=(6,4))
plt.plot(fpr_lr, tpr_lr, label='ROC Curve (AUC = {:.2f})'.format(auc))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

print("AUC Score:", auc)
```




```
AUC Score: 0.8160063113604489
```

In [56]:
```
from sklearn.metrics import confusion_matrix
# Calculate confusion matrix for logistic regression
conf_matrix_lr = confusion_matrix(y_test, y_pred_lr)

# Plot confusion matrix for logistic regression
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_lr, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix for Logistic regression')
plt.ylabel('True Label')
plt.show()
```



LDA

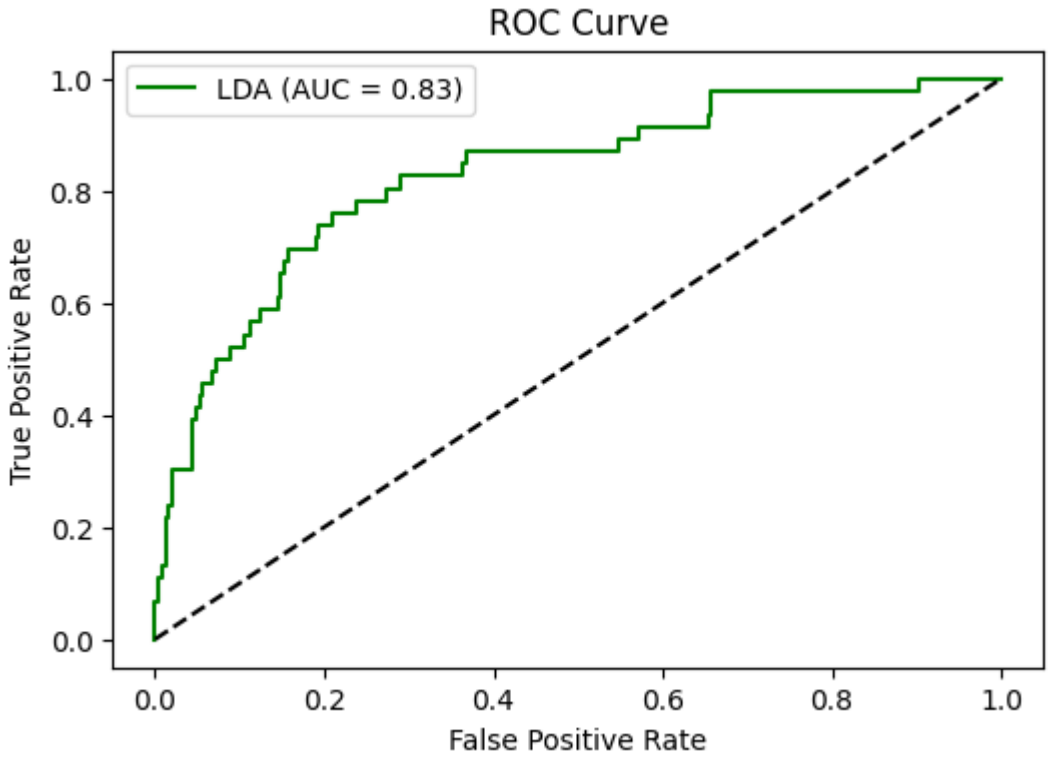
In [57]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Initialize and fit the LDA model
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)

# Calculate accuracy
accuracy_lda = lda.score(X_test, y_test)
print("Accuracy on LDA:", accuracy_lda)

# Define a threshold for binary classification
threshold = 0.21

probabilities_lda = lda.predict_proba(X_test)[:, 1]
# Convert probabilities to binary predictions using the threshold
predictions_binary = (probabilities_lda >= threshold).astype(int)

# Calculate precision, recall, and F1 score using binary predictions
precision_lda = metrics.precision_score(y_test, predictions_binary)
recall_lda = metrics.recall_score(y_test, predictions_binary)
f1_lda = metrics.f1_score(y_test, predictions_binary)

print("Precision on LDA:", precision_lda)
print("Recall on LDA:", recall_lda)
print("F1 Score on LDA:", f1_lda)
```

```
Accuracy on LDA: 0.8673469387755102
Precision on LDA: 0.4146341463414634
Recall on LDA: 0.7391304347826086
F1 Score on LDA: 0.53125
```

Threshold 0.2 has the maximum F1 score

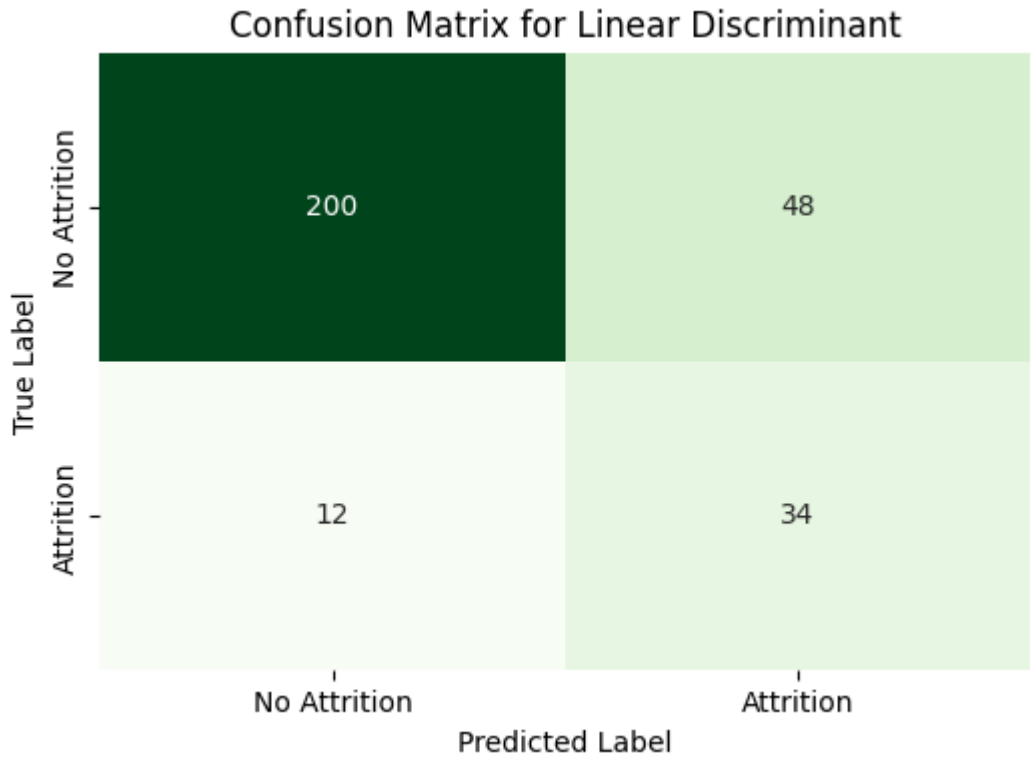
In [58]:
```
# Calculate ROC curve and AUC
fpr_lda, tpr_lda, thresholds_lda = roc_curve(y_test, probabilities_lda)
auc_lda = roc_auc_score(y_test, probabilities_lda)
```

```
# Plot ROC curve
plt.figure(figsize=(6,4))
plt.plot(fpr_lda, tpr_lda, label='LDA (AUC = {:.2f})'.format(auc_lda), color='greer
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```



In [59]:

```
from sklearn.metrics import confusion_matrix
# Build and print a confusion matrix for LDA
prediction_lda = probabilities_lda > 0.21
conf_matrix_lda = confusion_matrix(y_test, prediction_lda)

plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_lda, annot=True, fmt='d', cmap='Greens', cbar=False, xtickl
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix for Linear Discriminant')
plt.ylabel('True Label')
plt.show()
```

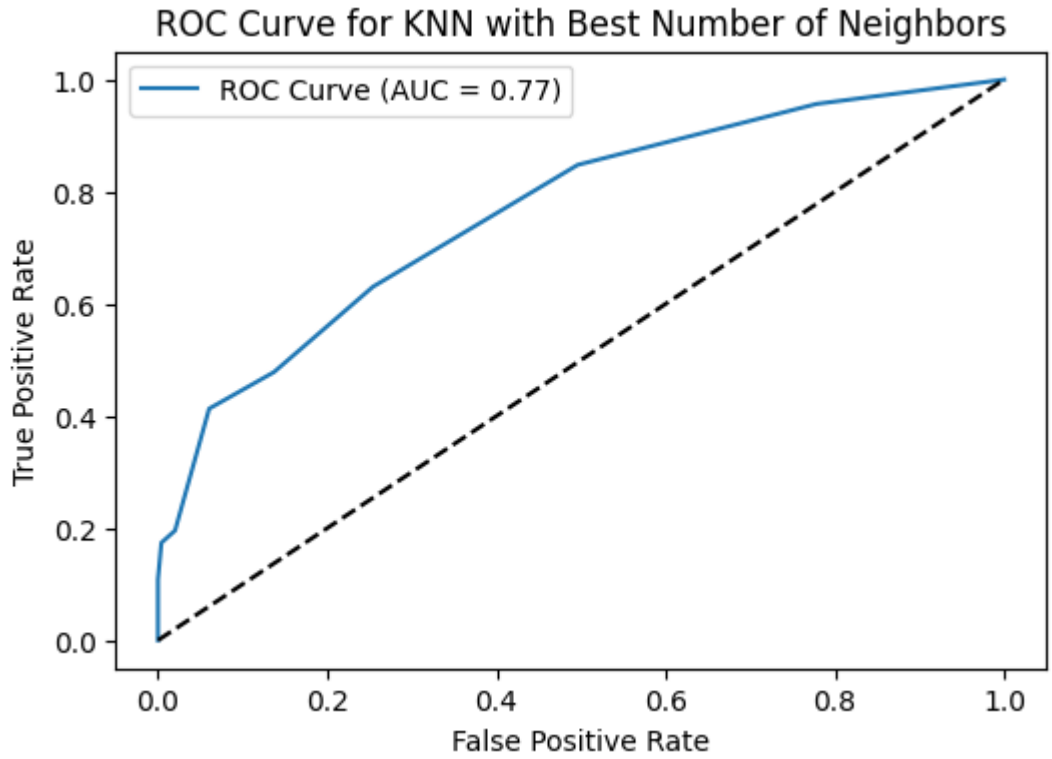


KNN

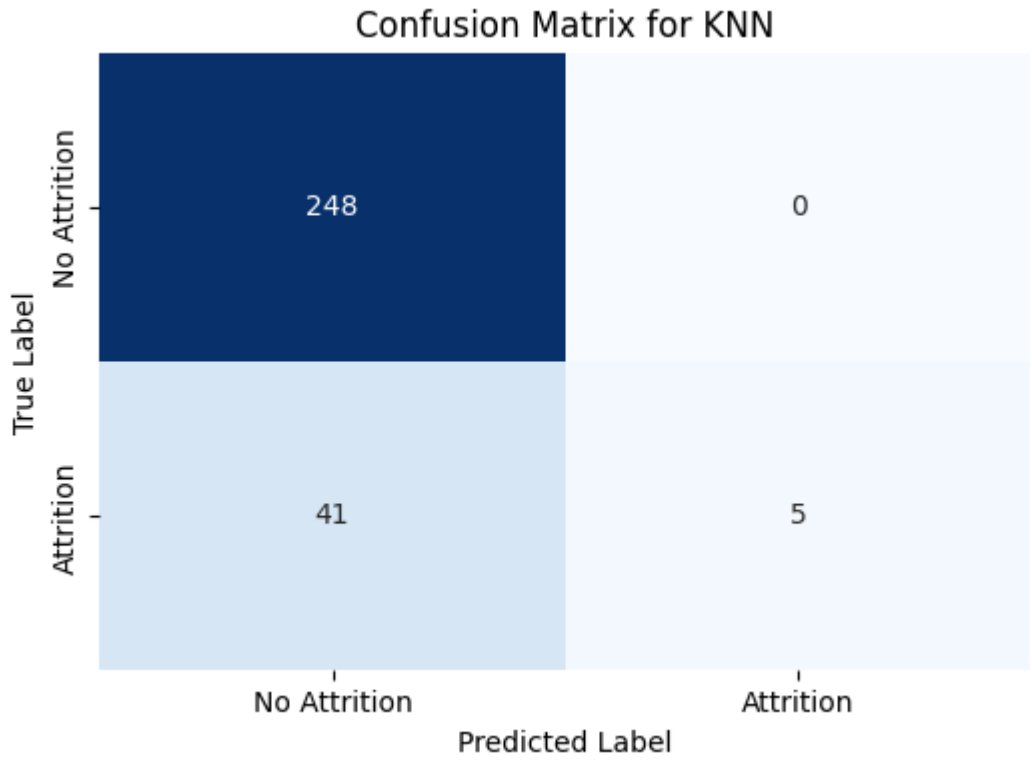
In [60]:
```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {'n_neighbors': [15,17,19,21]}  # You can adjust the list of values as

# Create the grid search object
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)

# Perform the grid search
grid_search.fit(X_train, y_train)

# Print the best parameter
print("Best number of neighbors:", grid_search.best_params_['n_neighbors'])

# Get the best model
best_knn = grid_search.best_estimator_

# Use the best model to predict on test data
y_pred_best_knn = best_knn.predict(X_test)

# Calculate accuracy using the best model
accuracy_best_knn = best_knn.score(X_test, y_test)
print("Accuracy on KNN: ", accuracy_best_knn)

# Calculate precision, recall, and F1 score using the best model
precision_best_knn = metrics.precision_score(y_test, y_pred_best_knn)
recall_best_knn = metrics.recall_score(y_test, y_pred_best_knn)
f1_best_knn = metrics.f1_score(y_test, y_pred_best_knn)

print("Precision on KNN: ", precision_best_knn)
print("Recall on KNN: ", recall_best_knn)
print("F1 Score on KNN: ", f1_best_knn)
```

```
Best number of neighbors: 17
Accuracy on KNN:  0.8605442176870748
Precision on KNN:  1.0
Recall on KNN:  0.10869565217391304
F1 Score on KNN:  0.19607843137254902
```

In [61]:
```
# Calculate probabilities for ROC curve using the best model
y_probs_knn = best_knn.predict_proba(X_test)[:, 1]

# Calculate ROC curve and AUC using the best model
fpr_knn, tpr_knn, thresholds_knn = roc_curve(y_test, y_probs_knn)
auc_best_knn = roc_auc_score(y_test, y_probs_knn)

# Plot ROC curve for KNN with best number of neighbors
plt.figure(figsize=(6, 4))
plt.plot(fpr_knn, tpr_knn, label='ROC Curve (AUC = {:.2f})'.format(auc_best_knn))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for KNN with Best Number of Neighbors')
plt.legend()
plt.show()
```



In [62]:
```
# Build and print a confusion matrix for KNN with best number of neighbors
conf_matrix_best_knn = confusion_matrix(y_test, y_pred_best_knn)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_best_knn, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for KNN')
plt.show()
```



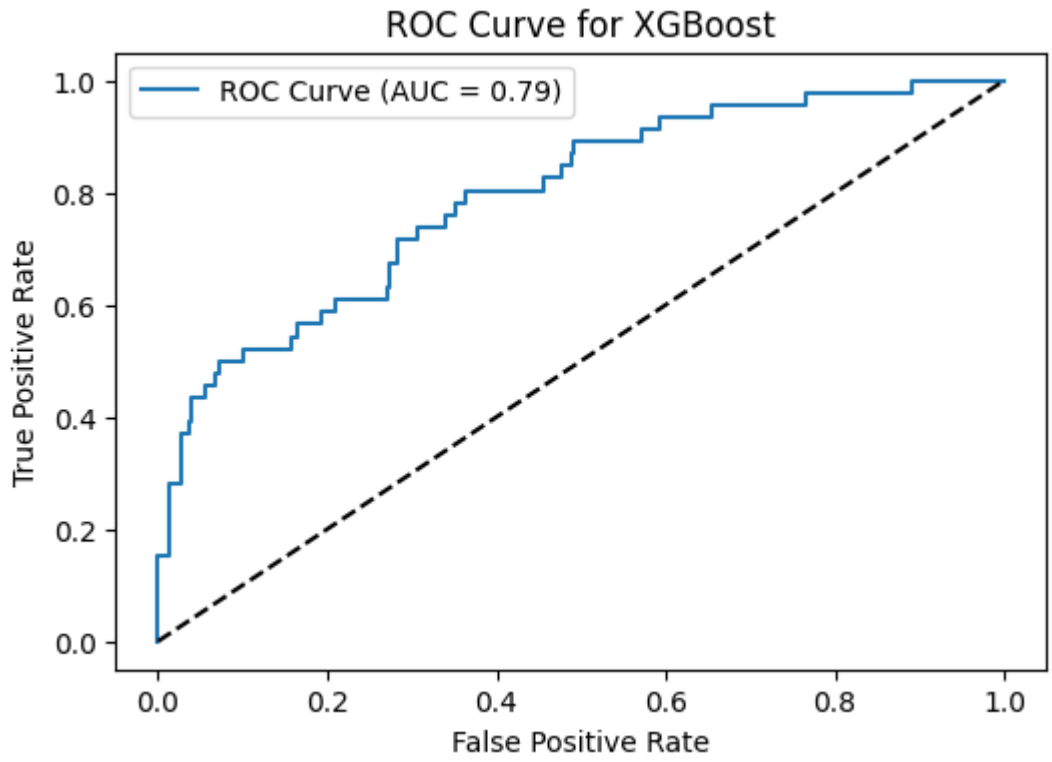
XGBOOST

In [63]:
```
import xgboost as xgb

# Define the XGBoost classifier
xgb_classifier = xgb.XGBClassifier()

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [150, 200, 250],  # Number of trees in the forest
    'max_depth': [5,6,7,8],             # Maximum depth of each tree
    'learning_rate': [0.05, 0.1, 0.2]  # Learning rate
}

# Perform grid search cross-validation to find the best hyperparameters
grid_search_xgb = GridSearchCV(xgb_classifier, param_grid, cv=5)
grid_search_xgb.fit(X_train, y_train)

# Print the best hyperparameters
print("Best hyperparameters for XGBoost:", grid_search_xgb.best_params_)

# Get the best XGBoost model
best_xgb_model = grid_search_xgb.best_estimator_

# Use the best model to make predictions on the test set
y_pred_xgb = best_xgb_model.predict(X_test)

# Calculate accuracy using the best model
accuracy_xgb = accuracy_score(y_test, y_pred_xgb)
print("Accuracy on XGBoost: ", accuracy_xgb)

# Calculate precision, recall, and F1 score using the best model
precision_xgb = precision_score(y_test, y_pred_xgb)
recall_xgb = recall_score(y_test, y_pred_xgb)
f1_xgb = f1_score(y_test, y_pred_xgb)

print("Precision on XGBoost:", precision_xgb)
```

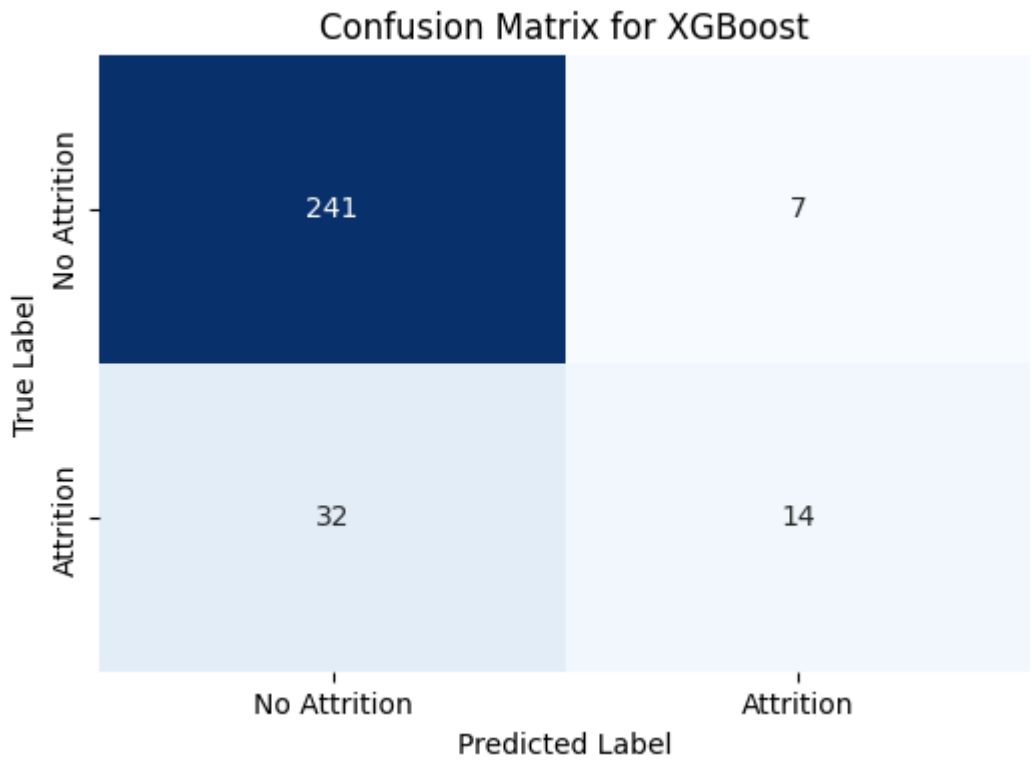
```
print("Recall on XGBoost:", recall_xgb)
print("F1 Score on XGBoost:", f1_xgb)
```

```
Best hyperparameters for XGBoost: {'learning_rate': 0.1, 'max_depth': 8, 'n_estima
tors': 150}
Accuracy on XGBoost:  0.8673469387755102
Precision on XGBoost: 0.6666666666666666
Recall on XGBoost: 0.30434782608695654
F1 Score on XGBoost: 0.41791044776119407
```

In [64]:
```
# Calculate probabilities for ROC curve using the best model
y_probs_xgb = best_xgb_model.predict_proba(X_test)[:, 1]

# Calculate ROC curve and AUC using the best model
fpr_xgb, tpr_xgb, thresholds_xgb = roc_curve(y_test, y_probs_xgb)
auc_xgb = roc_auc_score(y_test, y_probs_xgb)

# Plot ROC curve for XGBoost
plt.figure(figsize=(6, 4))
plt.plot(fpr_xgb, tpr_xgb, label='ROC Curve (AUC = {:.2f})'.format(auc_xgb))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for XGBoost')
plt.legend()
plt.show()
```



In [65]:
```
# Build and print a confusion matrix for XGBoost
conf_matrix_xgb = confusion_matrix(y_test, y_pred_xgb)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_xgb, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for XGBoost')
plt.show()
```

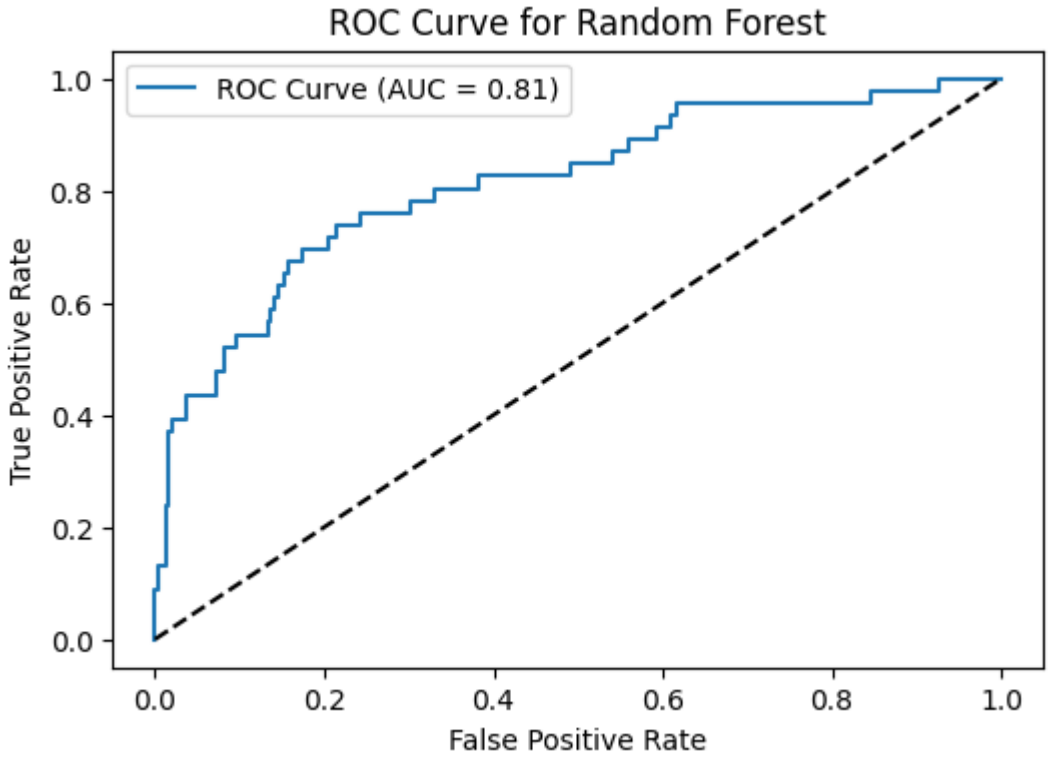


In [66]:
```
from sklearn.ensemble import RandomForestClassifier

# Define the Random Forest classifier
rf_classifier = RandomForestClassifier()

# Define the parameter grid for hyperparameter tuning
param_grid_rf = {
    'n_estimators': [80,90,100,110,120],  # Number of trees in the forest
    'max_depth': [5, 6, 7, 8],            # Maximum depth of each tree
    'min_samples_split': [1,2,3,4],    # Minimum number of samples required to split
    'min_samples_leaf': [2,3]      # Minimum number of samples required at each lea
}

# Perform grid search cross-validation to find the best hyperparameters
grid_search_rf = GridSearchCV(rf_classifier, param_grid_rf, cv=5)
grid_search_rf.fit(X_train, y_train)

# Print the best hyperparameters
print("Best hyperparameters for Random Forest:", grid_search_rf.best_params_)

# Get the best Random Forest model
best_rf_model = grid_search_rf.best_estimator_

# Use the best model to make predictions on the test set
y_pred_rf = best_rf_model.predict(X_test)

# Calculate accuracy using the best model
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print("Accuracy on Random Forest: ", accuracy_rf)

# Calculate precision, recall, and F1 score using the best model
precision_rf = precision_score(y_test, y_pred_rf)
recall_rf = recall_score(y_test, y_pred_rf)
f1_rf = f1_score(y_test, y_pred_rf)

print("Precision on Random Forest:", precision_rf)
print("Recall on Random Forest:", recall_rf)
print("F1 Score on Random Forest:", f1_rf)
```

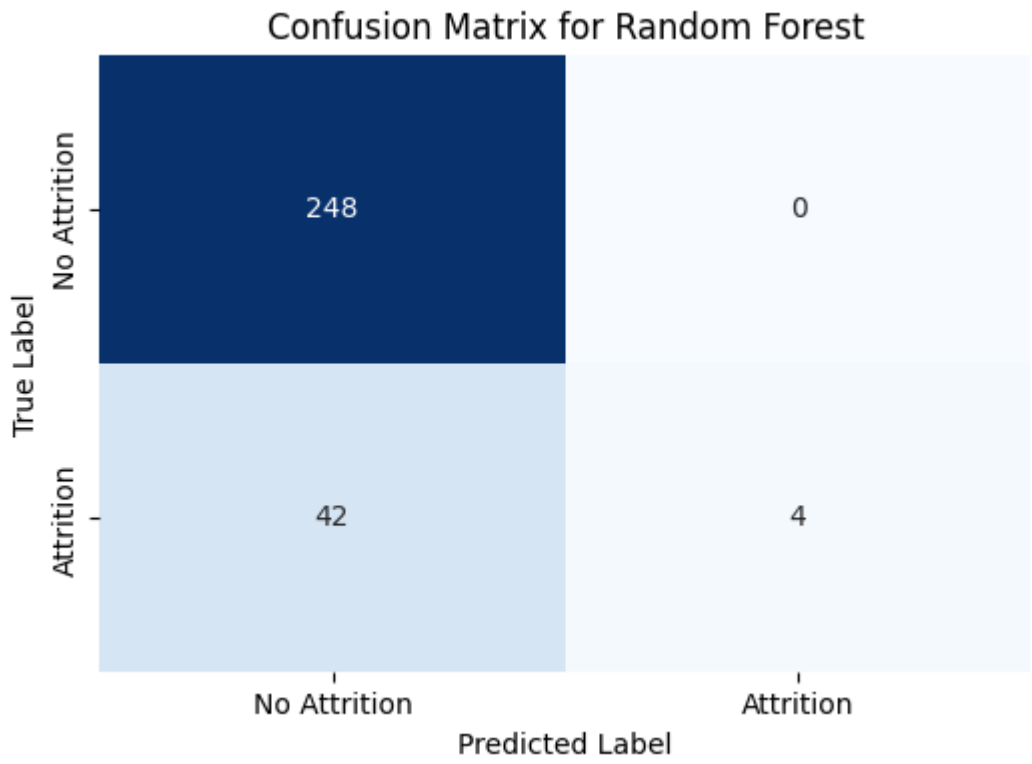
```
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validation.py:37
8: FitFailedWarning:
200 fits failed out of a total of 800.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_sco
re='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
200 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_validatio
n.py", line 686, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_forest.py", line
340, in fit
    self._validate_params()
  File "/usr/local/lib/python3.10/dist-packages/sklearn/base.py", line 600, in _va
lidate_params
    validate_parameter_constraints(
  File "/usr/local/lib/python3.10/dist-packages/sklearn/utils/_param_validation.p
y", line 97, in validate_parameter_constraints
    raise InvalidParameterError(
sklearn.utils._param_validation.InvalidParameterError: The 'min_samples_split' par
ameter of RandomForestClassifier must be an int in the range [2, inf) or a float i
n the range (0.0, 1.0]. Got 1 instead.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_search.py:952: Us
erWarning: One or more of the test scores are non-finite: [       nan        nan
nan        nan        nan 0.85205193
 0.85205193 0.85120087 0.85120087 0.85460512 0.85290299 0.85545618
 0.85375406 0.85630004 0.85204832 0.85035341 0.84865128 0.85375766
 0.8503498  0.85119726        nan        nan        nan        nan
        nan 0.85290299 0.85204832 0.85544897 0.84865128 0.85205193
 0.85544897 0.85290299 0.85120087 0.85460151 0.84949874 0.85290299
 0.85290299 0.85630364 0.85120447 0.85290299        nan        nan
        nan        nan        nan 0.85459791 0.85204832 0.85289578
 0.85375045 0.85460512 0.85630725 0.85459791 0.85460151 0.85120447
 0.85120808 0.85375045 0.85545979 0.85629643 0.85545618 0.85545618
        nan        nan        nan        nan        nan 0.85545258
 0.85545258 0.85291021 0.85459791 0.85375045 0.85290299 0.85460873
 0.85715471 0.85205193 0.85715471 0.85460512 0.85545618 0.85545258
 0.85545258 0.8529066         nan        nan        nan        nan
        nan 0.85545618 0.85970069 0.85800216 0.85460512 0.85544897
 0.85544897 0.85460512 0.85545618 0.85460512 0.85460873 0.85714749
 0.85545618 0.85545258 0.85715831 0.85545258        nan        nan
        nan        nan        nan 0.85460151 0.85801298 0.85629643
 0.85715831 0.85544897 0.85290299 0.85630364 0.85289939 0.85630004
 0.85800577 0.85460151 0.85885323 0.85290299 0.85799856 0.85885683
        nan        nan        nan        nan        nan 0.8571511
 0.85884962 0.85630725 0.85544897 0.85289578 0.85290299 0.85800577
 0.85715471 0.85460151 0.85545258 0.85289939 0.85545258 0.85375406
 0.8545943  0.85800216        nan        nan        nan        nan
        nan 0.85630725 0.85545618 0.85715471 0.85630725 0.85205554
 0.85970069 0.85885683 0.85630364 0.85885323 0.85800938 0.85714749
 0.85545258 0.85460151 0.85544537 0.85800577]
  warnings.warn(
```

```
Best hyperparameters for Random Forest: {'max_depth': 8, 'min_samples_leaf': 3, 'm
in_samples_split': 3, 'n_estimators': 80}
Accuracy on Random Forest:  0.8571428571428571
Precision on Random Forest: 1.0
Recall on Random Forest: 0.08695652173913043
F1 Score on Random Forest: 0.16
```

In [67]:
```
# Calculate probabilities for ROC curve using the best model
y_probs_rf = best_rf_model.predict_proba(X_test)[:, 1]

# Calculate ROC curve and AUC using the best model
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, y_probs_rf)
auc_rf = roc_auc_score(y_test, y_probs_rf)

# Plot ROC curve for Random Forest
plt.figure(figsize=(6, 4))
plt.plot(fpr_rf, tpr_rf, label='ROC Curve (AUC = {:.2f})'.format(auc_rf))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Random Forest')
plt.legend()
plt.show()
```

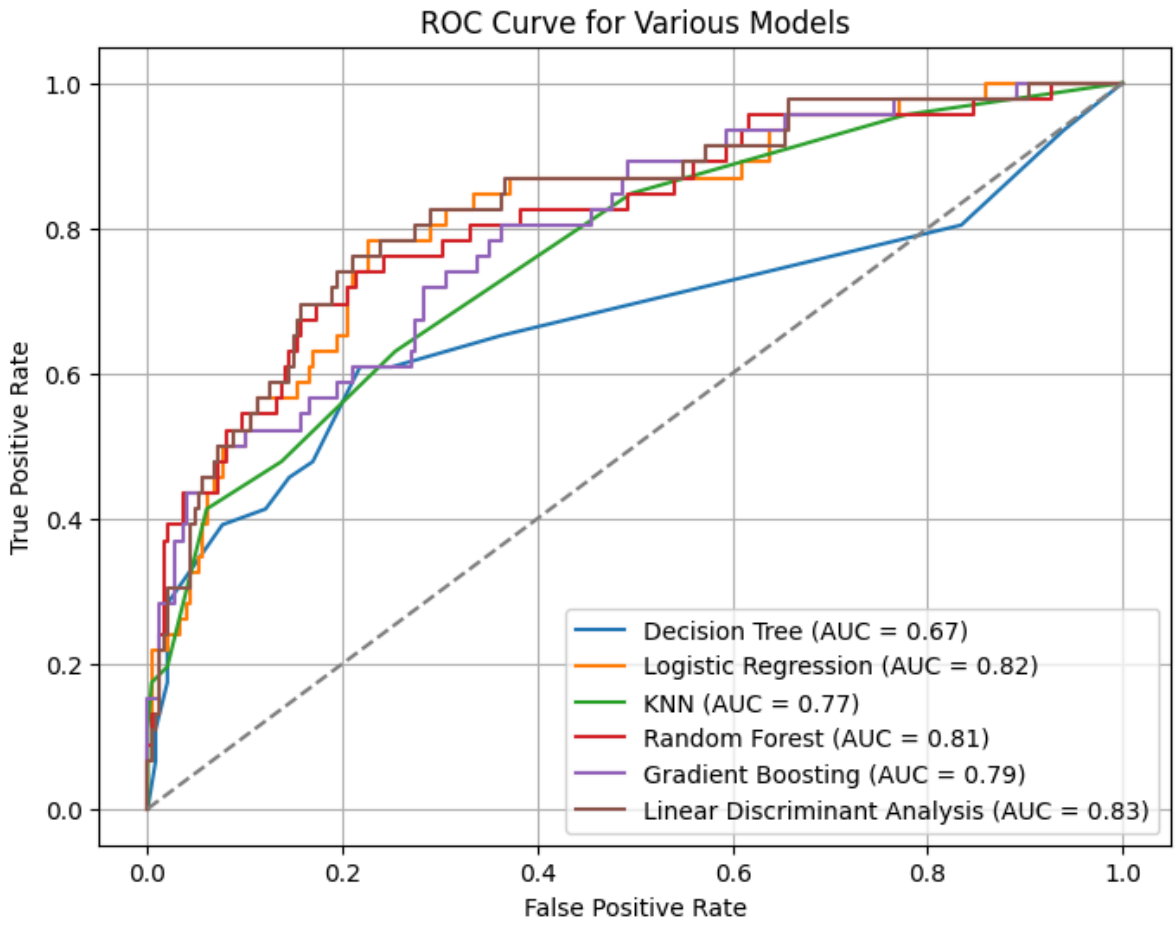


In [68]:
```
# Build and print a confusion matrix for Random Forest
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_rf, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Random Forest')
plt.show()
```



In [69]:
```
from sklearn.metrics import roc_auc_score

# Calculate ROC AUC scores for each model
roc_auc_dt = roc_auc_score(y_test, y_probs)
roc_auc_lr = roc_auc_score(y_test, probabilities)
roc_auc_knn = roc_auc_score(y_test, y_probs_knn)
roc_auc_rf = roc_auc_score(y_test, y_probs_rf)
roc_auc_gb = roc_auc_score(y_test, y_probs_xgb)
roc_auc_lda = roc_auc_score(y_test, probabilities_lda)

# Plot ROC curves for all models
plt.figure(figsize=(8, 6))

# Decision Tree
plt.plot(fpr_dt, tpr_dt, label='Decision Tree (AUC = {:.2f})'.format(roc_auc_dt))

# Logistic Regression
plt.plot(fpr_lr, tpr_lr, label='Logistic Regression (AUC = {:.2f})'.format(roc_auc_

# KNN
plt.plot(fpr_knn, tpr_knn, label='KNN (AUC = {:.2f})'.format(roc_auc_knn))

# Random Forest
plt.plot(fpr_rf, tpr_rf, label='Random Forest (AUC = {:.2f})'.format(roc_auc_rf))

# Gradient Boosting
plt.plot(fpr_xgb, tpr_xgb, label='Gradient Boosting (AUC = {:.2f})'.format(roc_auc_

# Linear Discriminant Analysis
plt.plot(fpr_lda, tpr_lda, label='Linear Discriminant Analysis (AUC = {:.2f})'.form

# Random Guess (diagonal line)
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')

# Add labels and legend
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Various Models')
plt.legend()
```

```
plt.grid(True)
plt.show()
```



Based solely on AUC scores, Linear Discriminant Analysis (LDA) remains the best-performing model.

Logistic Regression (LR): Offers a decent balance between precision and recall with an F1 score of 0.46. It's a good choice if interpretability is important and computational resources are limited.

Linear Discriminant Analysis (LDA): Shows the highest F1 score among the models (0.51), indicating good performance in predicting attrition while maintaining a balance between precision and recall.

K-Nearest Neighbors (KNN): Although it has a high precision score, the low recall score suggests that it might miss many attrition cases. It may not be the best choice unless precision is prioritized over recall.

Random Forest (RF) and XGBoost (XGB): Both models have high precision but low recall, indicating a potential issue with identifying all attrition cases. They could be suitable if minimizing false positives (incorrectly predicting attrition) is a priority.

Imbalanced Classes: Employee attrition datasets often have a significant class imbalance, where the number of employees who stay (no attrition) is much larger than those who leave (attrition). In such cases, a model that predicts all employees to stay would still achieve high accuracy but fail to identify the minority class (attrition cases).

Misleading Performance: Accuracy alone doesn't provide insight into how well a model performs on each class. A model with high accuracy may still have poor performance in correctly identifying attrition cases (low recall), which is crucial for the organization's decision-making process.

Business Impact: In predicting employee attrition, correctly identifying employees who are likely to leave (attrition cases) is often more critical than correctly predicting those who stay. False negatives (missed attrition cases) could result in increased costs associated with recruitment, training, and lost productivity.

AUC-ROC as a Better Measure: Area Under the ROC Curve (AUC-ROC) considers the trade-off between true positive rate (recall) and false positive rate across different thresholds. It provides a comprehensive measure of model performance across various decision thresholds and is more robust to imbalanced datasets.

In [70]:
```
avg_onemonthincome = 4800
cost_hiring = 15000

cost_matrix = pd.DataFrame([[-avg_onemonthincome, - avg_onemonthincome], [-cost_hir
print ("Cost matrix")
print (cost_matrix)
```

```
Cost matrix
       p     n
Y  -4800 -4800
N -15000     0
```

If the employee is at a risk of attrition (probability > 0.2), we give them one month income as bonus. Average one month income of the employees that have attrited is approximately $4800

Cost of hiring is $15000 per employee

With this cost-benefit matrix, our focus should be on minimizing false negatives (employees who are predicted as not going to attrite but actually do). This way, we can reduce the cost associated with hiring new employees to replace those who attrite unexpectedly.

Given this priority, we would choose a model that minimizes false negatives, even if it comes at the expense of potentially higher false positives. This means we should select a model with higher recall (true positive rate) or adjusting the decision threshold to prioritize sensitivity over specificity.

Linear Discriminant Analysis (LDA) seems to be a suitable choice, as it seems to perform well in terms of precision, recall, and F1 score. It also has the highest auc among all the models that we tested.

Accuracy on LDA: 0.867

Precision on LDA: 0.415

Recall on LDA: 0.739

F1 Score on LDA: 0.53

In [71]:
```
prediction = probabilities_lda > 0.21
#prediction = y_probs > 0.5
#prediction = probabilities > 0.5
#prediction = y_probs_knn > 0.5
#prediction = y_probs_rf > 0.5
#prediction = y_probs_xgb > 0.5
# Build and print a confusion matrix
confusion_matrix_final = pd.DataFrame(metrics.confusion_matrix(y_test, prediction,
                                columns=['p', 'n'], index=['Y', 'N'])
print (confusion_matrix_final)
profit = np.sum((confusion_matrix_final * cost_matrix).values)
print ("Profit or loss on test set with the threshold with best f score is $%.2f."
```

```
    p    n
Y  34   48
N  12  200
Profit or loss on test set with the threshold with best f score is $-573600.00.
```

In [72]:
```
targeted_matrix = pd.DataFrame([[1,1], [0, 0]], columns=['p', 'n'], index=['Y', 'N'
print ("Targeted matrix")
print (targeted_matrix)
```

```
Targeted matrix
   p  n
Y  1  1
N  0  0
```

In [73]:
```
# Calculate profit for threshold
prediction = probabilities_lda > 0.38
conf_matrix_try = pd.DataFrame(metrics.confusion_matrix(y_test, prediction, labels=
                                columns=['p', 'n'], index=['Y', 'N'])
profit = np.sum((conf_matrix_try * cost_matrix).values)
print ("Profit or loss on test set with the given threshold is $%.2f." % profit)
```

```
Profit or loss on test set with the given threshold is $-561000.00.
```

CALCULATING PROFIT OR LOSS ON A RANDOM MODEL

In [74]:
```
# Define the confusion matrix values
TP = 38
FN = 199
FP = 199
TN = 1034

# Define the cost matrix
cost_matrix_random = {
    ('Y', 'p'): -4800,
    ('Y', 'n'): -4800,
    ('N', 'p'): -15000,
    ('N', 'n'): 0
}

# Calculate total profit or loss
total_profit_loss = (TP * cost_matrix_random[('Y', 'p')]) + (FN * cost_matrix_randc

print("Total profit or loss in random model:", total_profit_loss)
```

```
Total profit or loss in random model: -4122600
```

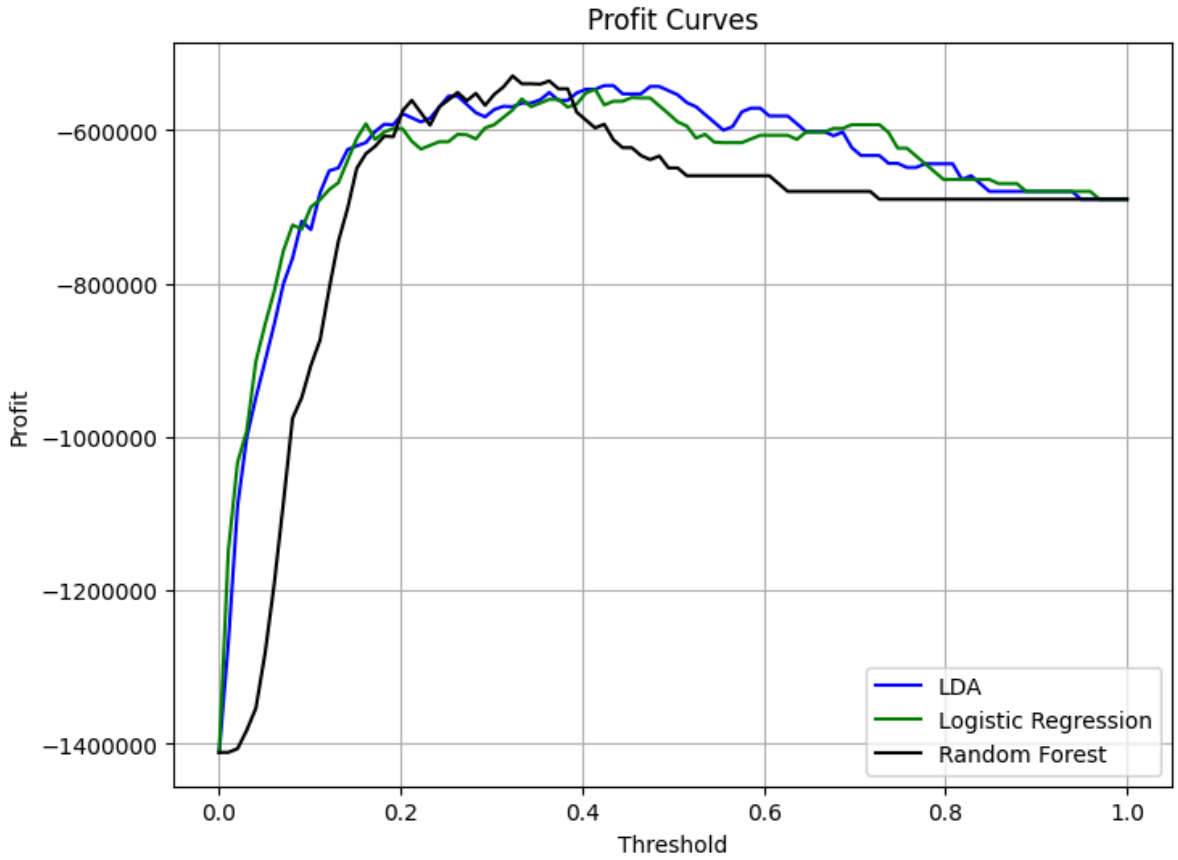
In [75]:
```
# Calculate metrics for random model
accuracy_random = (TP + TN) / (TP + TN + FP + FN)
precision_random = TP / (TP + FP)
recall_random = TP / (TP + FN)
f1_score_random = 2 * precision * recall / (precision + recall)
```

```
# Print metrics
print("Accuracy for random model:", accuracy)
print("Precision for random model:", precision)
print("Recall for random model:", recall)
print("F1-Score for random model:", f1_score)
```

```
Accuracy for random model: 0.8707482993197279
Precision for random model: 0.7222222222222222
Recall for random model: 0.2826086956521739
F1-Score for random model: <function f1_score at 0x7f34c740e7a0>
```

In [76]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# Calculate profit for different thresholds
thresholds = np.linspace(0, 1, 100)
profits_lda = []
profits_logistic = []
profits_rf = []

for threshold in thresholds:
    # For LDA
    prediction_lda = probabilities_lda > threshold
    conf_matrix_lda = confusion_matrix(y_test, prediction_lda, labels=[1, 0]).T
    profit_lda = np.sum((conf_matrix_lda * cost_matrix).values)
    profits_lda.append(profit_lda)

    # For Logistic Regression
    prediction_logistic = probabilities > threshold
    conf_matrix_logistic = confusion_matrix(y_test, prediction_logistic, labels=[1,
    profit_logistic = np.sum((conf_matrix_logistic * cost_matrix).values)
    profits_logistic.append(profit_logistic)

    # For Random Forest
    prediction_rf = y_probs_rf > threshold
    conf_matrix_rf = confusion_matrix(y_test, prediction_rf, labels=[1, 0]).T
    profit_rf = np.sum((conf_matrix_rf * cost_matrix).values)
    profits_rf.append(profit_rf)

# Plot the profit curves
plt.figure(figsize=(8, 6))
plt.plot(thresholds, profits_lda, label='LDA', color='blue')
plt.plot(thresholds, profits_logistic, label='Logistic Regression', color='green')
plt.plot(thresholds, profits_rf, label='Random Forest', color='black')

plt.xlabel('Threshold')
plt.ylabel('Profit')
plt.title('Profit Curves')
plt.grid(True)
plt.ticklabel_format(style='plain', axis='y')
plt.legend()
plt.show()
```



We can see from the above profit loss curve that optimal threshold is between 0.2 to 0.4

In [78]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Initialize and fit the LDA model
lda_final = LinearDiscriminantAnalysis()
lda_final.fit(X, y)

# Calculate accuracy
accuracy_lda_final = lda_final.score(X, y)
print("Accuracy on LDA:", accuracy_lda_final)

# Define a threshold for binary classification
threshold = 0.21

probabilities_lda_final = lda_final.predict_proba(X)[:, 1]
# Convert probabilities to binary predictions using the threshold
predictions_binary = (probabilities_lda_final >= threshold).astype(int)

# Calculate precision, recall, and F1 score using binary predictions
precision_lda_final = metrics.precision_score(y, predictions_binary)
recall_lda_final = metrics.recall_score(y, predictions_binary)
f1_lda_final = metrics.f1_score(y, predictions_binary)

print("Precision on LDA:", precision_lda_final)
print("Recall on LDA:", recall_lda_final)
print("F1 Score on LDA:", f1_lda_final)
```

```
Accuracy on LDA: 0.891156462585034
Precision on LDA: 0.4763231197771588
Recall on LDA: 0.7215189873417721
F1 Score on LDA: 0.5738255033557047
```

For final delivery of the model, we train it on the entire dataset which gives us improved metrics.