Modeling Attrition Prediction

Will Byrd June 2024

Introduction

In this notebook, we will take a look at an HR Analytics dataset and perform a binary classification to determine Attrition. Attrition is the departure of an employee from an organization fro any reason. In this dataset we do not have context on why the emlpoyee leaves the company-they may have been fired or they may have resigned.

For companies and orgnizations-the employees are the most important asset. Therefore, it is vitally important to be able to predict behavior of these employees. Knowing which employees will stay with the company can:

- Reduce turnover costs
    - Hiring and firing employees is expensive
- Increase employee engagement and morale
    - It can be difficult to maintain strong company culture if turnover is high
- Assist with resource planning
    - Companies can add resources to employees they believe will contribute to longterm growth
    - Conversely, companies can take proactive measures to retain at risk employees
- Improve customer relationships
    - Customer percieve companies that retain talent more positively
- Enhance company reputation
    - Company brand is often times tied to the employees interacting with the customers

Data

HR Analytics data was collected here. This is a 228 kB, publicly available csv file with 1479 rows and 35 columns from Kaggle. Important features are going to be our target value **'Attrition'**, and various features such as:

- Job Satisfaction
- Age
- Sex
- Job Title
- Salary

Data Preperation

This dataset is already cleaned, but some processing still needs to occur. We will need to:

- Engineer Features
    - Combining, Transforming columns
    - Label Encoding, One-Hot Encoding categorical features
- Dropping Columns
- Balancing Classes
    - The class imblance issue will be a limitation of this dataset and a tradeoff between overall accuracy and Recall will happen

Goals

The main goal of our model is to maximize **accuracy**. Accuracy will be important here as being able to accurately predict which employees will stay and which will leave is important. It's also important to understand that since we don't know if employees are fired and which ones quit, simply targeting the positive Attrition class (Recall) will still not tell the entire story.

Here are the metrics we will be looking at for this business case:

- Accuracy-Overall accuracy of our model

- Precision-Accuracy of positive predictions made by our model
  - High Precision indicates that an employee will leave, it is usually correct. **High Precision minimizes false positives.**
- Recall-The ability of our model to identify the actual positive observations
  - High Recall is going to be tough with this dataset specifically since we are battling a **class imbalance** issue. As we will see in our dataset, nearly 15% of our data is the positive **Attrition** class.
- F1-Score that factors in both Precision and Recall

Loading in the Data

As mentioned earlier, this is publicly available dataset. Run these cells below to download the data so you can run this notebook anywhere!

We will load in the data and then add our own libraries for manipulation and analysis.

```
!pip install kaggle
```

```
Requirement already satisfied: kaggle in /usr/local/lib/python3.10/dist-packages (1.6.14)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.10/dist-packages (from kaggle) (1.16.0)
Requirement already satisfied: certifi>=2023.7.22 in /usr/local/lib/python3.10/dist-packages (from kaggle) (2024.6.2)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.10/dist-packages (from kaggle) (2.8.2)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from kaggle) (2.31.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from kaggle) (4.66.4)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.10/dist-packages (from kaggle) (8.0.4)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.10/dist-packages (from kaggle) (2.0.7)
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from kaggle) (6.1.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages (from bleach->kaggle) (0.5.1)
Requirement already satisfied: text-unidecode>=1.3 in /usr/local/lib/python3.10/dist-packages (from python-slugify->kaggle) (1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->kaggle) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->kaggle) (3.7)
```

```
!kaggle datasets download -d pavansubhasht/ibm-hr-analytics-attrition-dataset
```

```
Dataset URL: https://www.kaggle.com/datasets/pavansubhasht/ibm-hr-analytics-attrition-dataset
License(s): DbCL-1.0
Downloading ibm-hr-analytics-attrition-dataset.zip to /content
  0% 0.00/50.1k [00:00<?, ?B/s]
100% 50.1k/50.1k [00:00<00:00, 43.1MB/s]
```

Importing all of our libraries and modules for data analysis, feature engineering and modeling.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report, accuracy_score, precision_score, roc_auc_score, recall_score, confusion_matrix, Confus
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
from imblearn.combine import SMOTEENN
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
```

Reading in our dataset as df.

```
!unzip -o ibm-hr-analytics-attrition-dataset.zip
```

```
Archive:  ibm-hr-analytics-attrition-dataset.zip
  inflating: WA_Fn-UseC_-HR-Employee-Attrition.csv
```

```
Start coding or generate with AI.
```

```
import os
print(os.listdir())

```

```
['.config', 'ibm-hr-analytics-attrition-dataset.zip', 'WA_Fn-UseC_-HR-Employee-Attrition.csv', 'sample_data']
```

```python
# Load the dataset into a DataFrame
df = pd.read_csv('WA_Fn-UseC_-HR-Employee-Attrition.csv')

# Display the first few rows of the DataFrame to verify
df.head()

```

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

5 rows × 35 columns

```python
# Load the dataset
df = pd.read_csv('WA_Fn-UseC_-HR-Employee-Attrition.csv')

# Display the first few rows of the dataset
print(df.head())
```

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

   DistanceFromHome  Education EducationField  EmployeeCount  EmployeeNumber  \
0                 1          2  Life Sciences              1               1   
1                 8          1  Life Sciences              1               2   
2                 2          2          Other              1               4   
3                 3          4  Life Sciences              1               5   
4                 2          1        Medical              1               7   

   ...  RelationshipSatisfaction StandardHours  StockOptionLevel  \
0  ...                         1            80                 0   
1  ...                         4            80                 1   
2  ...                         2            80                 0   
3  ...                         3            80                 0   
4  ...                         4            80                 1   

   TotalWorkingYears  TrainingTimesLastYear WorkLifeBalance  YearsAtCompany  \
0                  8                      0               1               6   
1                 10                      3               3              10   
2                  7                      3               3               0   
3                  8                      3               3               8   
4                  6                      3               3               2   

  YearsInCurrentRole  YearsSinceLastPromotion  YearsWithCurrManager  
0                  4                        0                     5  
1                  7                        1                     7  
2                  0                        0                     0  
3                  7                        3                     0  
4                  2                        2                     2  

[5 rows x 35 columns]
```

EDA

Let's explore the data to enhance our analysis. This is a crucial step in the modeling process as we can add context to the data.

We will perform the following:

- View the info
- Dropping unnecesary columns

- Feature engineering
- Creating Visualizations to better understand the data

Quick view of info. Notice no NaN values in our dataset!

```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1470 entries, 0 to 1469
Data columns (total 35 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   DailyRate                 1470 non-null   int64
 4   Department                1470 non-null   object
 5   DistanceFromHome          1470 non-null   int64
 6   Education                 1470 non-null   int64
 7   EducationField            1470 non-null   object
 8   EmployeeCount             1470 non-null   int64
 9   EmployeeNumber            1470 non-null   int64
 10  EnvironmentSatisfaction   1470 non-null   int64
 11  Gender                    1470 non-null   object
 12  HourlyRate                1470 non-null   int64
 13  JobInvolvement            1470 non-null   int64
 14  JobLevel                  1470 non-null   int64
 15  JobRole                   1470 non-null   object
 16  JobSatisfaction           1470 non-null   int64
 17  MaritalStatus             1470 non-null   object
 18  MonthlyIncome             1470 non-null   int64
 19  MonthlyRate               1470 non-null   int64
 20  NumCompaniesWorked        1470 non-null   int64
 21  Over18                    1470 non-null   object
 22  OverTime                  1470 non-null   object
 23  PercentSalaryHike         1470 non-null   int64
 24  PerformanceRating         1470 non-null   int64
 25  RelationshipSatisfaction  1470 non-null   int64
 26  StandardHours             1470 non-null   int64
 27  StockOptionLevel          1470 non-null   int64
 28  TotalWorkingYears         1470 non-null   int64
 29  TrainingTimesLastYear     1470 non-null   int64
 30  WorkLifeBalance           1470 non-null   int64
 31  YearsAtCompany            1470 non-null   int64
 32  YearsInCurrentRole        1470 non-null   int64
 33  YearsSinceLastPromotion   1470 non-null   int64
 34  YearsWithCurrManager      1470 non-null   int64
dtypes: int64(26), object(9)
memory usage: 402.1+ KB
```

Confirmation of no NaN values

```
1 df.isna().sum()
```

```
Age                         0
Attrition                   0
BusinessTravel              0
DailyRate                   0
Department                  0
DistanceFromHome            0
Education                   0
EducationField              0
EmployeeCount               0
EmployeeNumber              0
EnvironmentSatisfaction     0
Gender                      0
HourlyRate                  0
JobInvolvement              0
JobLevel                    0
JobRole                     0
JobSatisfaction             0
MaritalStatus               0
MonthlyIncome               0
MonthlyRate                 0
NumCompaniesWorked          0
Over18                      0
OverTime                    0
PercentSalaryHike           0
PerformanceRating           0
RelationshipSatisfaction    0
```

```
StandardHours               0
StockOptionLevel            0
TotalWorkingYears           0
TrainingTimesLastYear       0
WorkLifeBalance             0
YearsAtCompany              0
YearsInCurrentRole          0
YearsSinceLastPromotion     0
YearsWithCurrManager        0
dtype: int64
```

These columns below don't offer any value to our statistical analysis, so they can be removed.

```
df = df.drop(columns=[
    'Over18',
    'EmployeeCount',
    'StandardHours',
    'EmployeeNumber'

])
```

Let's create a new column that shows the relationship between 2 similar features- 'YearsInCurrentRole' and 'YearsAtCompany'. Because some people will have been at the company for less than a year, we need to turn NaN values to 0, since this will affectively have the same effect we are looking for.

```
# Create the new feature with division
df['YearsInCurrentRole_vs_YearsAtCompany'] = df['YearsInCurrentRole'] / df['YearsAtCompany']

# Replace NaN values with 0
df['YearsInCurrentRole_vs_YearsAtCompany'].fillna(0, inplace=True)

# Verify changes
print(df[['YearsInCurrentRole', 'YearsAtCompany', 'YearsInCurrentRole_vs_YearsAtCompany']].head())

```

```
   YearsInCurrentRole  YearsAtCompany  YearsInCurrentRole_vs_YearsAtCompany
0                   4               6                              0.666667
1                   7              10                              0.700000
2                   0               0                              0.000000
3                   7               8                              0.875000
4                   2               2                              1.000000
```

Feature Engineering

Let's create some more new columns to better illustrate the relationship between years in current job status, amount of travel, and overall satisfaction.

```
df['TotalYearsCurrentJob'] = df['YearsInCurrentRole'] + df['YearsWithCurrManager']

```

```
#df['FrequentBusinessTravel'] = df['BusinessTravel'].apply(lambda x: 1 if x == 'Travel_Frequently' else 0)

```

```
df['JobEnvSatisfaction'] = df['JobSatisfaction'] + df['EnvironmentSatisfaction']

```

Quick view of the df again.

```
df.head()
```

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

5 rows × 34 columns

Let's take a look at the distribution of the values in our **'Attrition'** column.

Here is the imbalancing mentioned earlier. We can see the majority class 'No' outweighs our minority class 'yes'.

```
attrition_counts = df['Attrition'].value_counts(normalize=True) * 100

# Print the results
print(attrition_counts)
```

```
Attrition
No     83.877551
Yes    16.122449
Name: proportion, dtype: float64
```

Visualizations

Now let's look at some visualizations. First things first is the distribution of 'Age' across all employees. We can see it has a slight skew to the right, which makes sense, because most employees tend to be entry level or early in their career and then some older executives will skew the distribution.

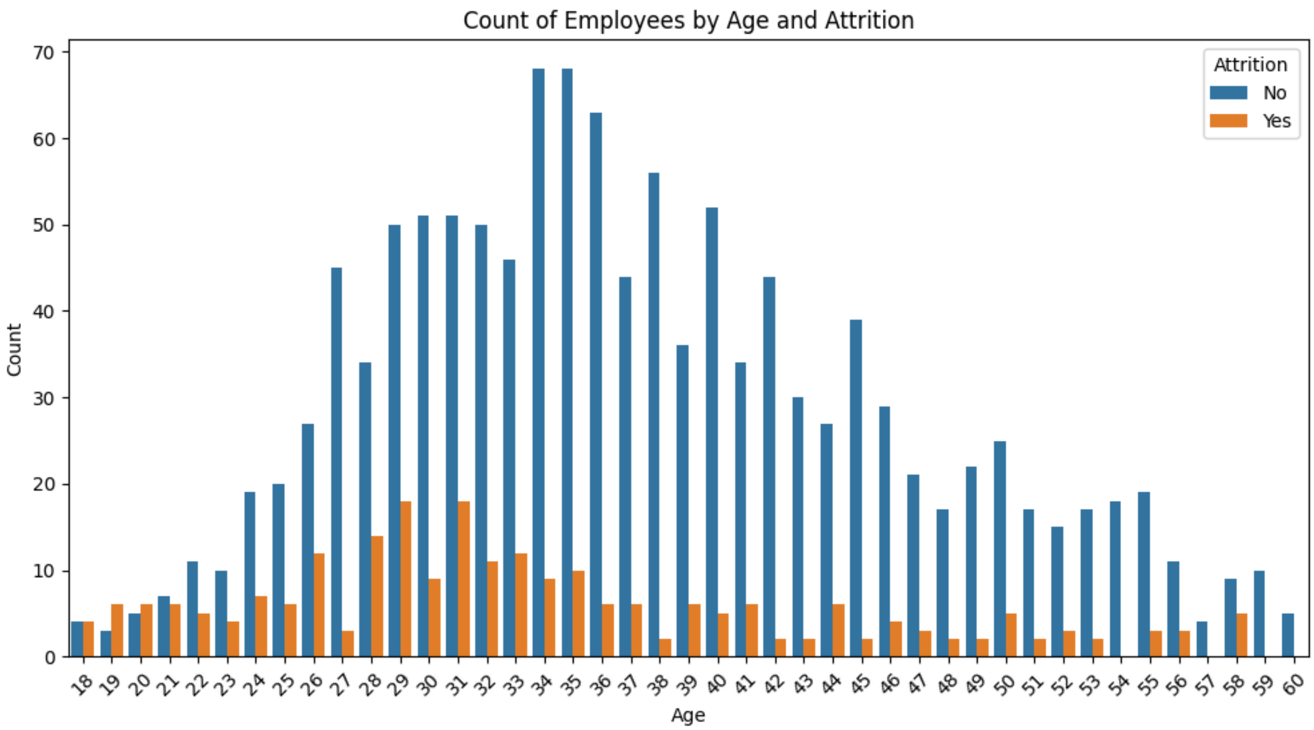
We will use the Seaborn library to build most of these visualizations.

```
sns.histplot(df['Age'], bins=20, kde=True)
plt.title('Age Distribution of Employees')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()
```

Let's see if there is any relationship between 'Age' and 'Attrition'

Based on what we see, it looks like **mostly employees early in their career or in mid-level positions are leaving** the company and **fewer senior/executive aged employees are leaving.**

```
att_age_group = df.groupby(['Attrition', 'Age']) # creating 'att_age_group' for visualizing
group_sizes = att_age_group.size()

# Convert group_sizes to DataFrame for easier manipulation
group_sizes_df = group_sizes.reset_index(name='Count')

# Plotting the bar graph
plt.figure(figsize=(12, 6))
sns.barplot(data=group_sizes_df, x='Age', y='Count', hue='Attrition')
plt.title('Count of Employees by Age and Attrition')
plt.xlabel('Age')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```



Let's take a quick look at the employees who leave the company based on their 'Gender'. While this column name is 'Gender', it is actually referring to sex.

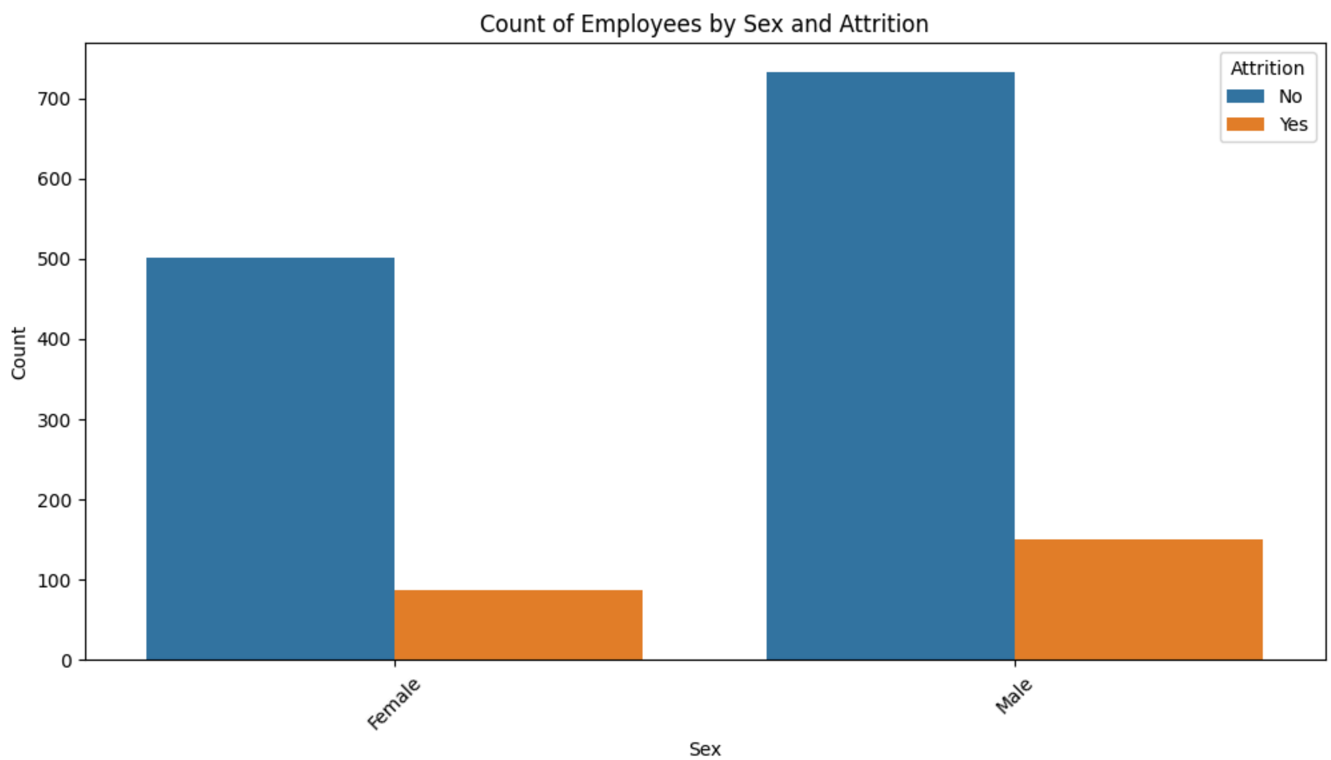
1st thing to notice is there are **more males than females.**

```
att_gen_group = df.groupby(['Attrition', 'Gender'])
att_gen_group = att_gen_group.size()
print(att_gen_group)

```

```
Attrition  Gender
No         Female    501
           Male      732
Yes        Female     87
           Male      150
dtype: int64
```

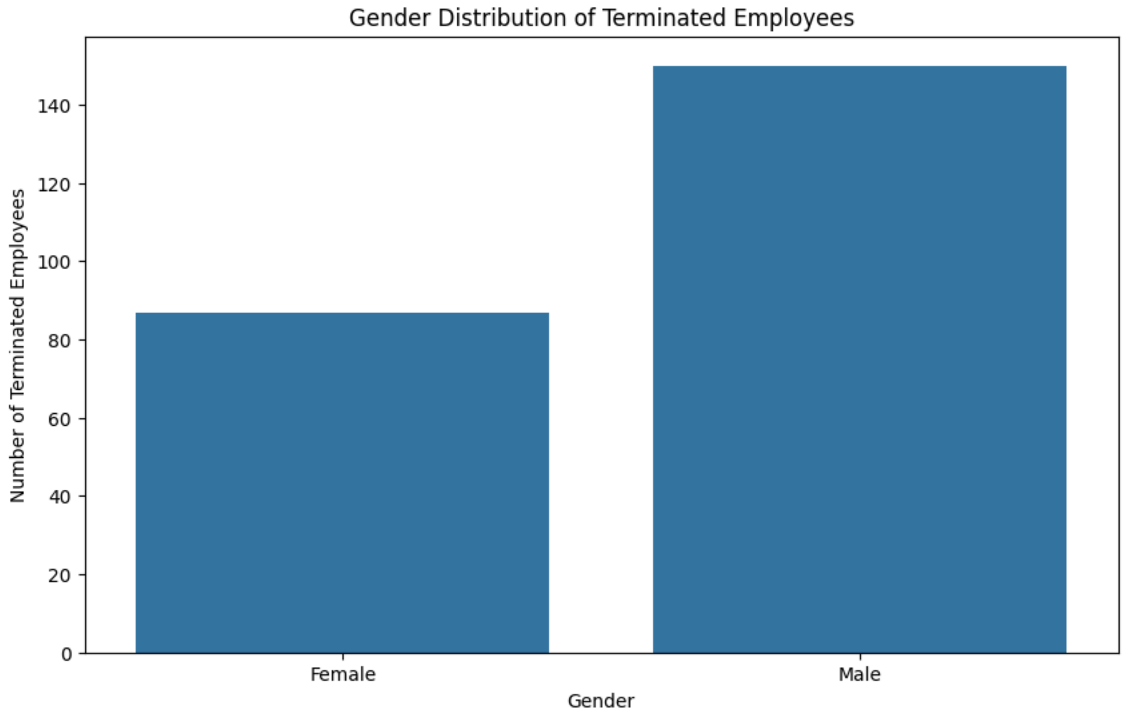
Let's visualize that.

```
# Convert group_sizes to DataFrame for easier manipulation
att_gen_df = att_gen_group.reset_index(name='Count')

# Plotting the bar graph
plt.figure(figsize=(12, 6))
sns.barplot(data=att_gen_df, x='Gender', y='Count', hue='Attrition')
plt.title('Count of Employees by Sex and Attrition')
plt.xlabel('Sex')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```



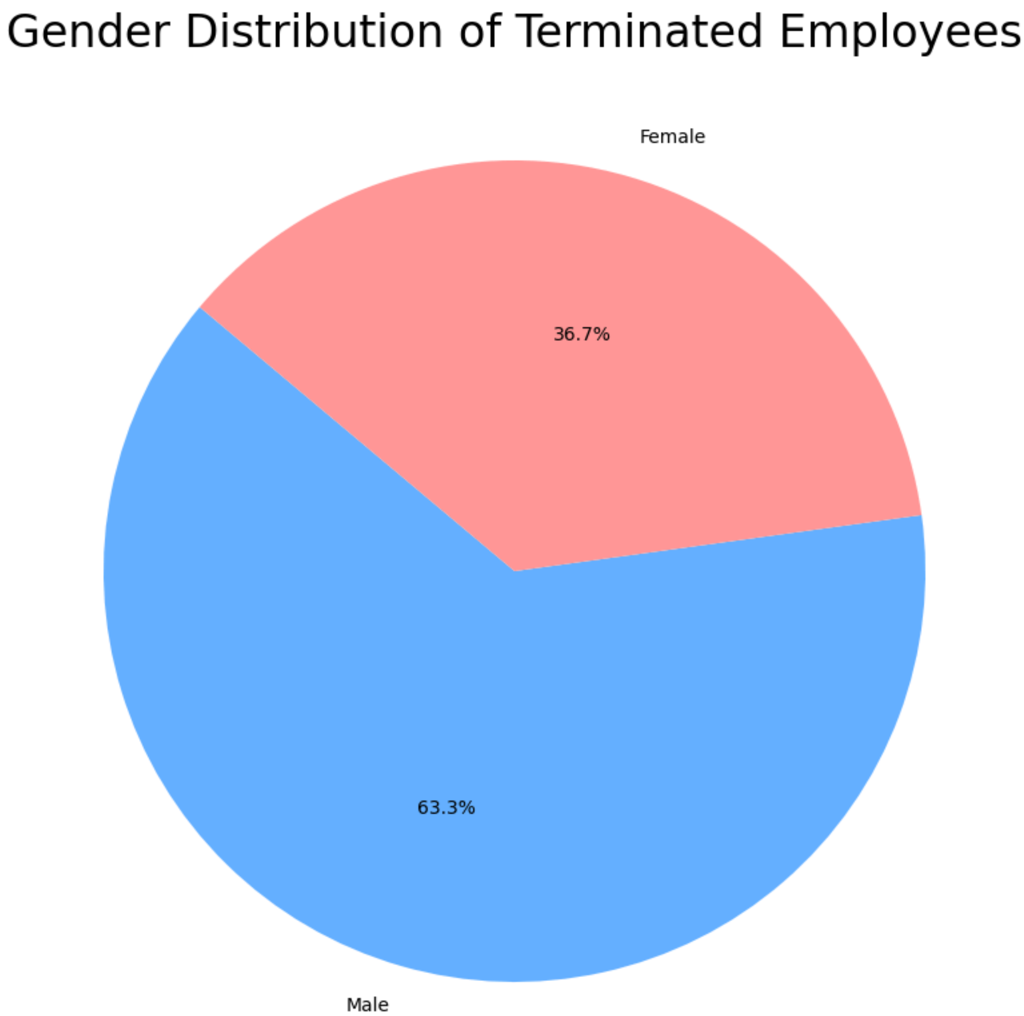
Again, we can visualize, of the sexes, which is leaving the company more and it is Male. We will also create a new **'terminated_df'** that holds only the rows where the **'Attrition'** value is **Yes**.

```
terminated_df = df[df['Attrition'] == 'Yes']

# Plot the gender distribution of terminated employees
plt.figure(figsize=(10, 6))
sns.countplot(data=terminated_df, x='Gender')
plt.title('Gender Distribution of Terminated Employees')
plt.xlabel('Gender')
plt.ylabel('Number of Terminated Employees')
plt.show()
```



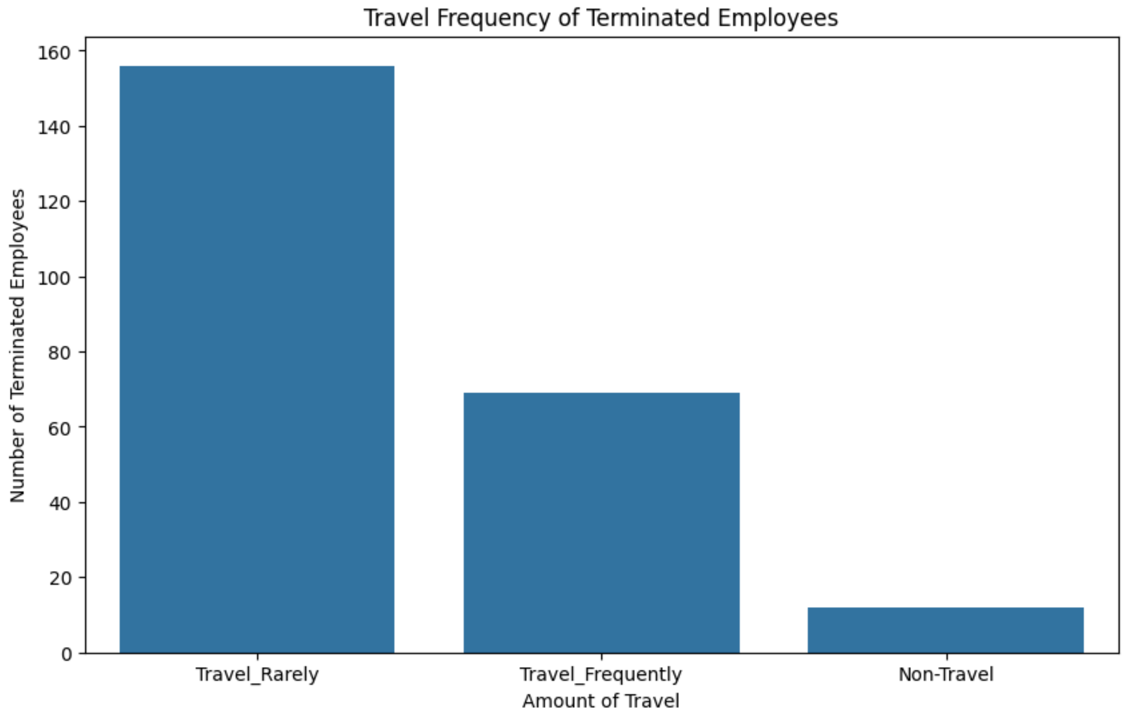
And let's visualize this another way.

```
# Calculate the distribution of genders
gender_counts = terminated_df['Gender'].value_counts()

# Plot the gender distribution of terminated employees as a pie chart
plt.figure(figsize=(10, 10))
plt.pie(gender_counts, labels=gender_counts.index, autopct='%1.1f%%', startangle=140, colors=['#66b3ff','#ff9999'])
plt.title('Gender Distribution of Terminated Employees', fontsize=24)

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



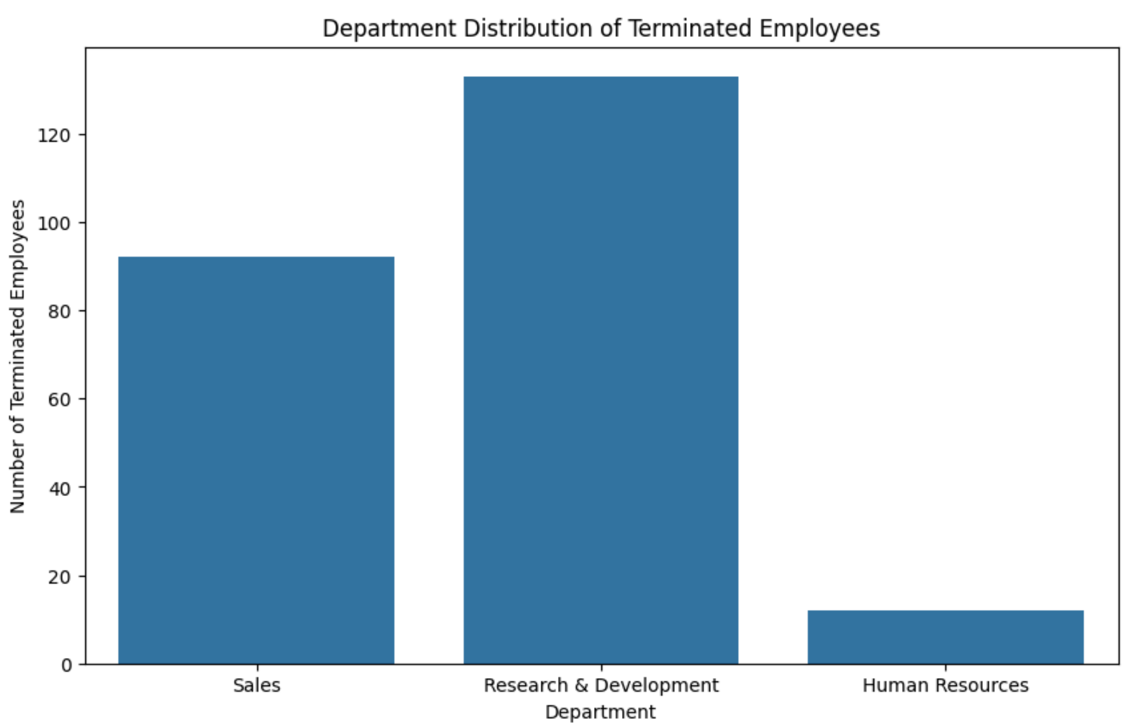
Next, let's take a look at Frequency of travel amongst employees who left the company.

```
terminated_df = df[df['Attrition'] == 'Yes']

# Plot the gender distribution of terminated employees
plt.figure(figsize=(10, 6))
sns.countplot(data=terminated_df, x='BusinessTravel')
plt.title('Travel Frequency of Terminated Employees')
plt.xlabel('Amount of Travel')
plt.ylabel('Number of Terminated Employees')
plt.show()
```



Now let's take a look at the distribution of of terminated employees based on the department they work in. The **Research and Development Department has seen the most amount of employees who 'Attrition'.**

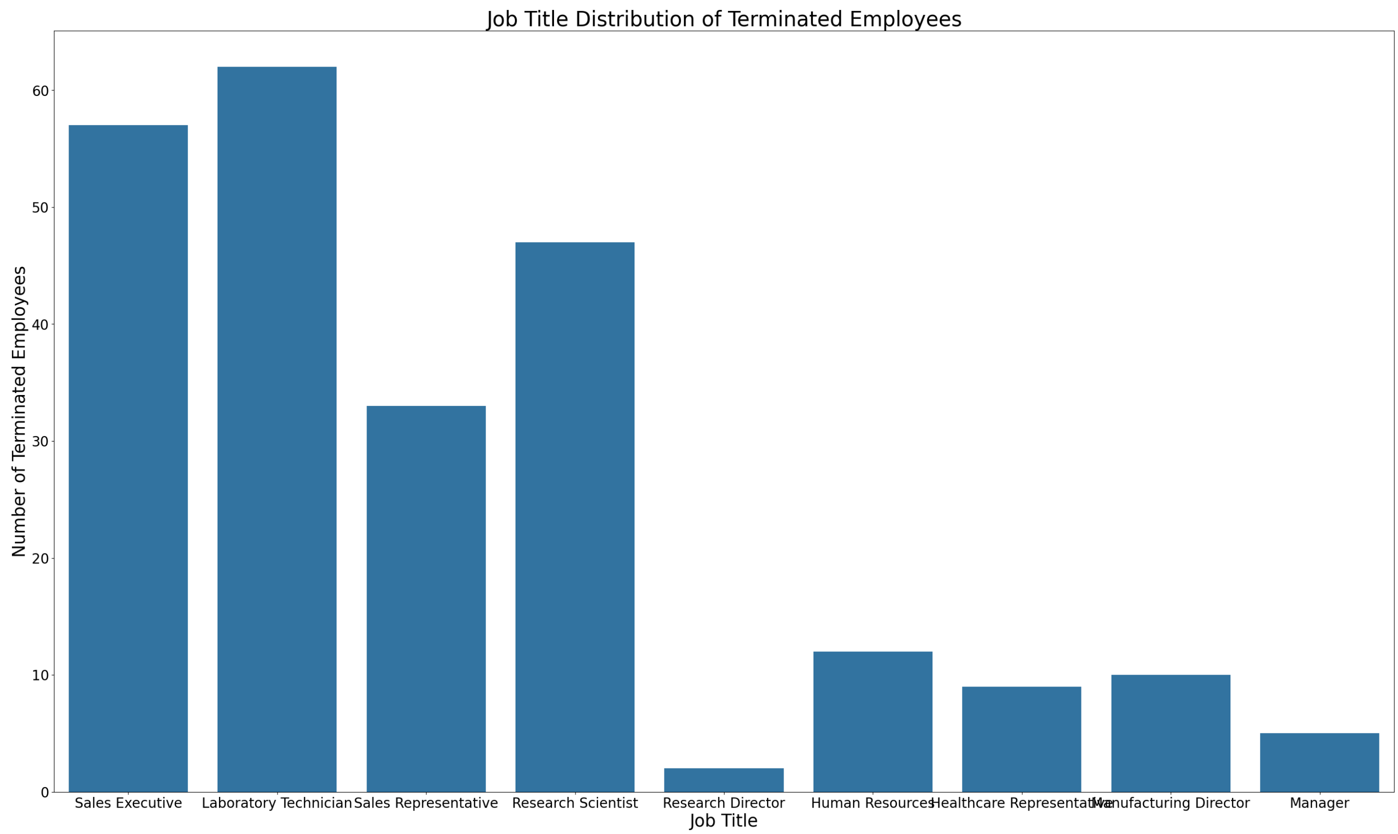
```
terminated_df = df[df['Attrition'] == 'Yes']

# Plot the gender distribution of terminated employees
plt.figure(figsize=(10, 6))
sns.countplot(data=terminated_df, x='Department')
plt.title('Department Distribution of Terminated Employees')
plt.xlabel('Department')
plt.ylabel('Number of Terminated Employees')
plt.show()
```



Now let's take a look at the **distribution of employees who leave the company based on their job title.** Below are the top 3 titles of employees of leave the company.

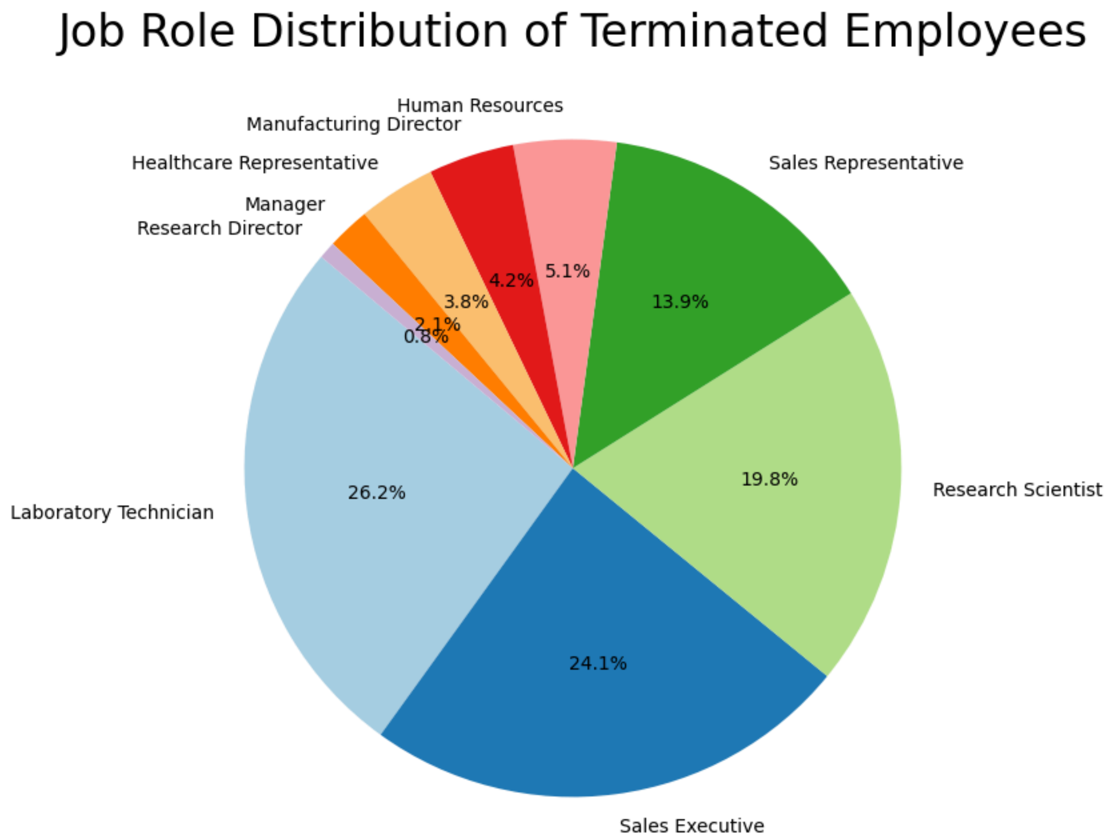
- Lab Tech
- Sales Exec
- Research Scientist

```
# Plot the gender distribution of terminated employees
plt.figure(figsize=(35, 20))
sns.countplot(data=terminated_df, x='JobRole')
plt.title('Job Title Distribution of Terminated Employees', fontsize=30)
plt.xlabel('Job Title', fontsize=25)
plt.ylabel('Number of Terminated Employees', fontsize=25)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.show()
```

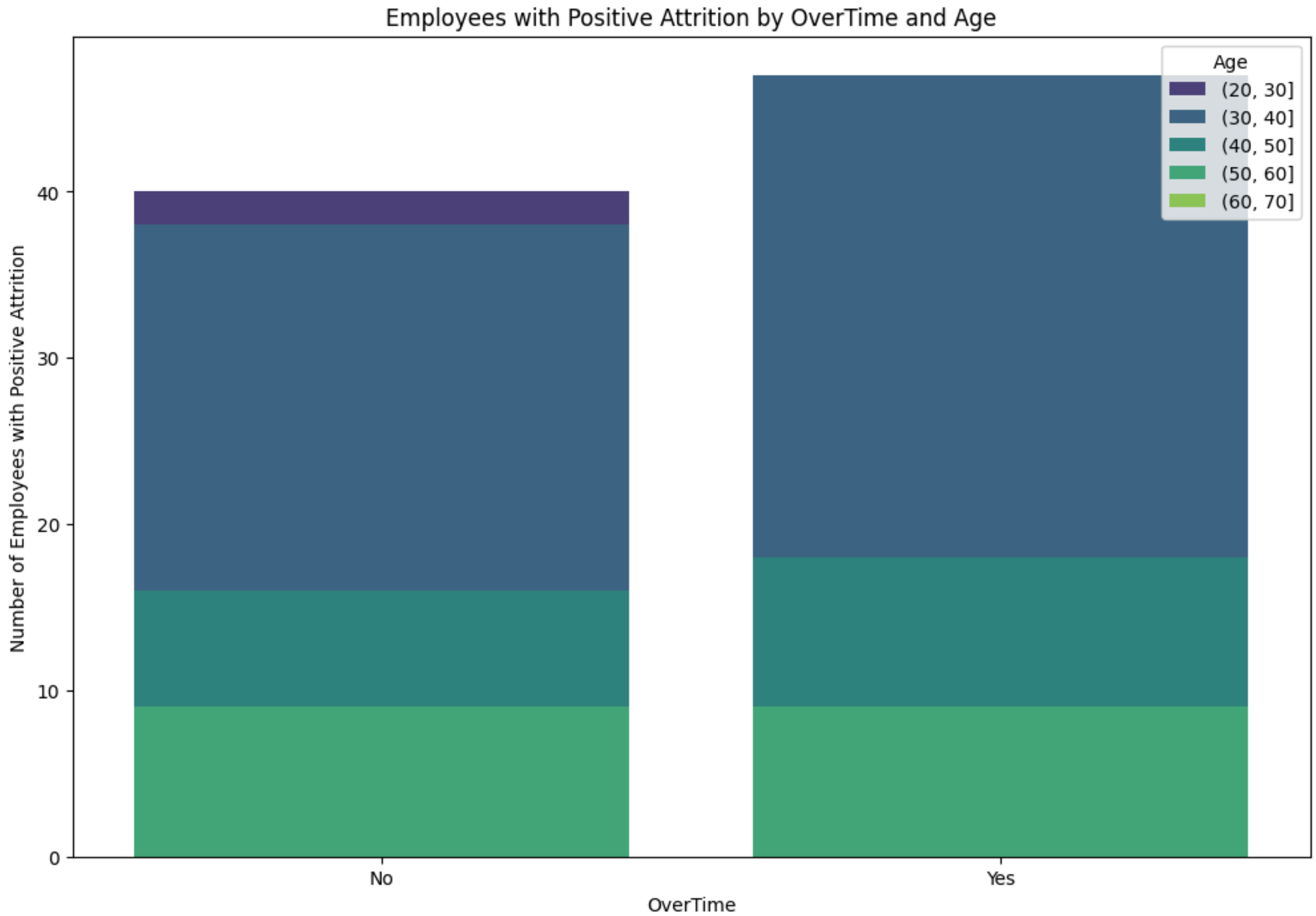


Next, let's create a **pie chart** to view that same data above, but as percentages of the whole.

```
# Calculate the distribution of job roles
jobrole_counts = terminated_df['JobRole'].value_counts()

# Plot the job role distribution of terminated employees as a pie chart
plt.figure(figsize=(12, 8))
plt.pie(jobrole_counts, labels=jobrole_counts.index, autopct='%1.1f%%', startangle=140, colors=plt.cm.Paired.colors)
plt.title('Job Role Distribution of Terminated Employees', fontsize=24)

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

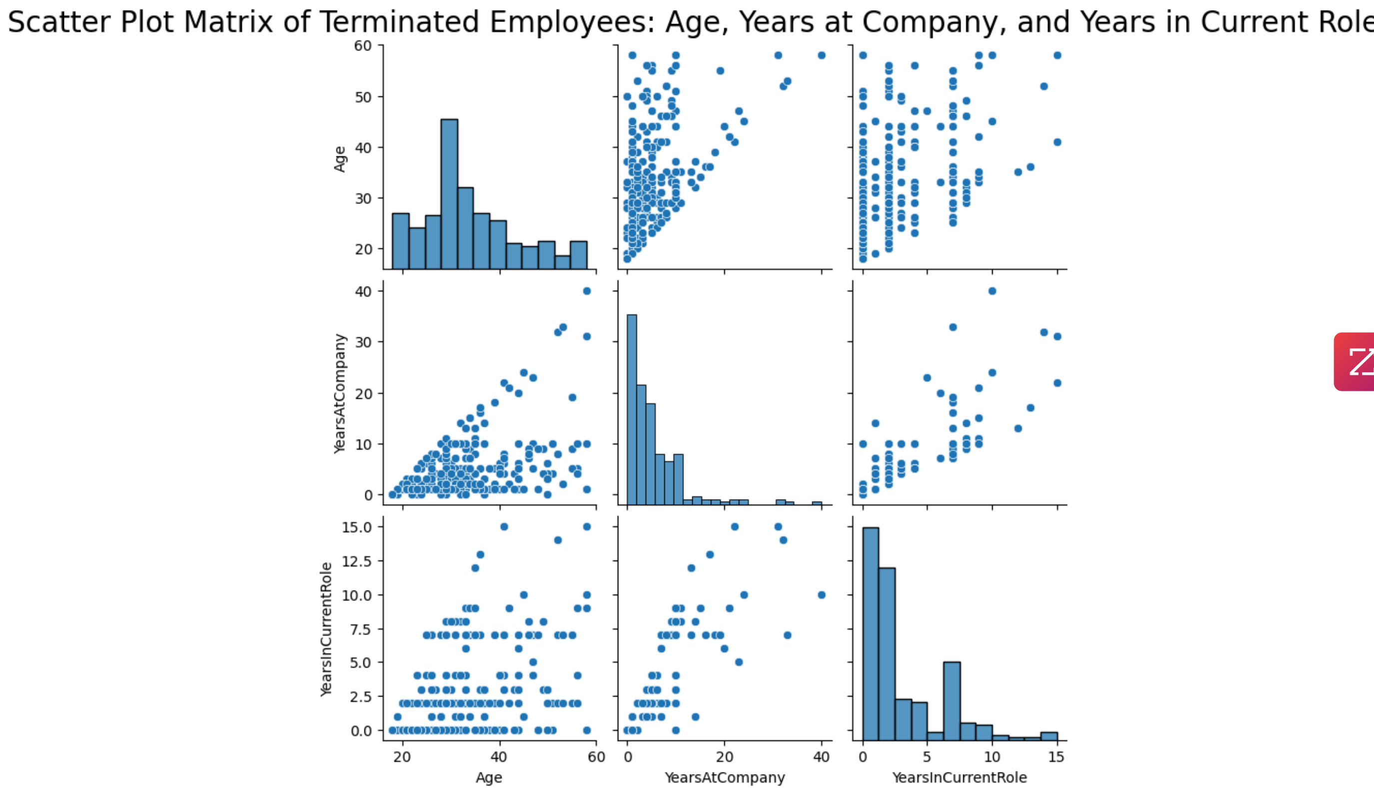



```
attrition_overtime_counts = terminated_df.groupby(['OverTime', pd.cut(terminated_df['Age'], bins=[20, 30, 40, 50, 60, 70])]).size().reset

# Plotting the bar chart
plt.figure(figsize=(12, 8))
sns.barplot(data=attrition_overtime_counts, x='OverTime', y='Count', hue='Age', palette='viridis', dodge=False)
plt.xlabel('OverTime')
plt.ylabel('Number of Employees with Positive Attrition')
plt.title('Employees with Positive Attrition by OverTime and Age')
plt.legend(title='Age', loc='upper right')
plt.show()
```



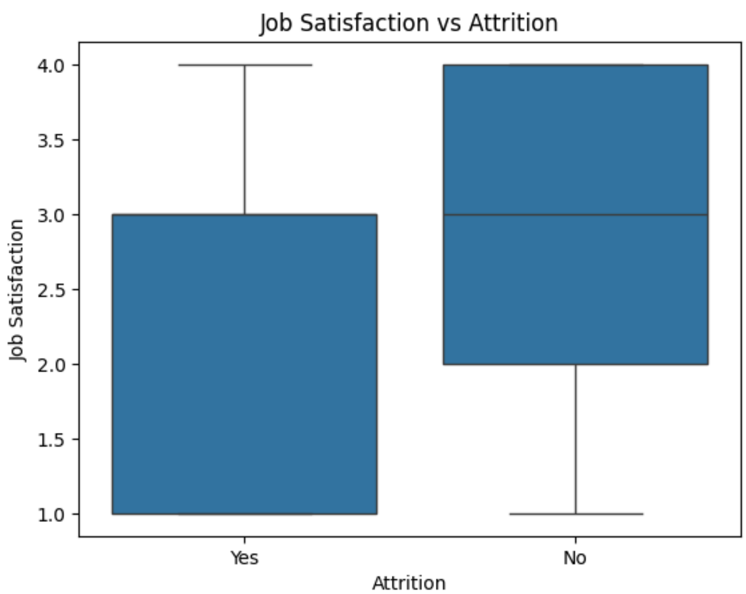
Let's use pairplot from the Seaborn library to view a variety of graphs.

```
# Plot using pairplot
sns.pairplot(terminated_df[['Age', 'YearsAtCompany', 'YearsInCurrentRole']])
plt.suptitle('Scatter Plot Matrix of Terminated Employees: Age, Years at Company, and Years in Current Role', y=1.02, fontsize=20)
plt.show()
```



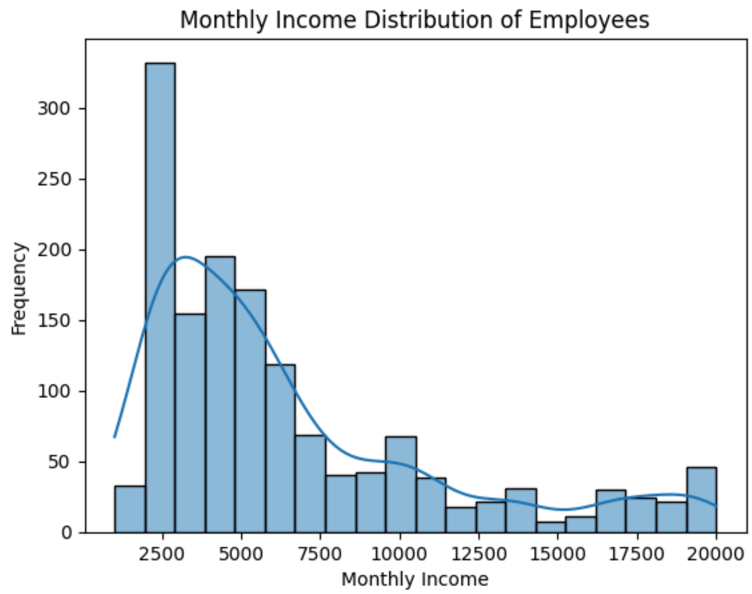
We can see **'Attrition' based on 'Job Satisfaction'** and how most of the employees who leave the company have lower ratings of 'Job Satisfaction'.

```
sns.boxplot(data=df, x='Attrition', y='JobSatisfaction')
plt.title('Job Satisfaction vs Attrition')
plt.xlabel('Attrition')
plt.ylabel('Job Satisfaction')
plt.show()
```



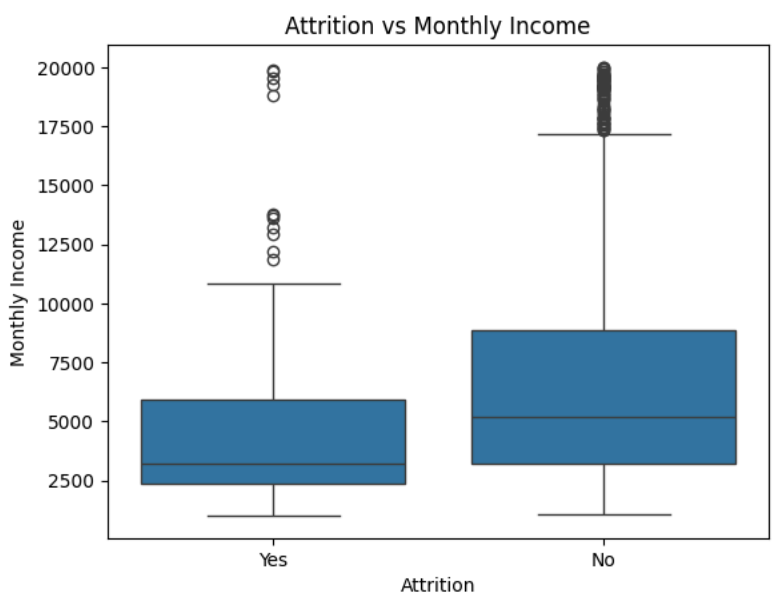
Next, let's take a look at **'Monthly Income'**. This is really just to get an idea of the distribution across the entire dataset.

```
sns.histplot(df['MonthlyIncome'], bins=20, kde=True)
plt.title('Monthly Income Distribution of Employees')
plt.xlabel('Monthly Income')
plt.ylabel('Frequency')
plt.show()
```



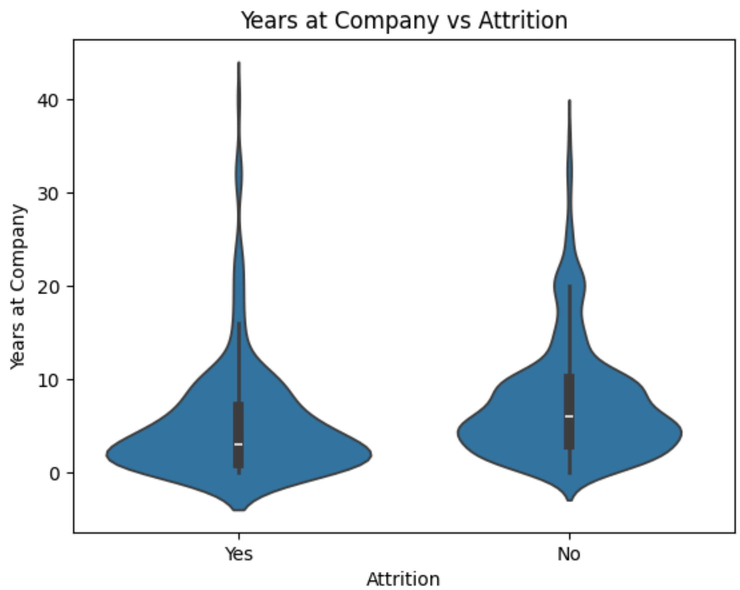
Now let's visualize that data as it relates to **'Attrition'**. We can see employees who leave the company tend to have lower **'Monthly Income'**.

```
sns.boxplot(data=df, x='Attrition', y='MonthlyIncome')
plt.title('Attrition vs Monthly Income')
plt.xlabel('Attrition')
plt.ylabel('Monthly Income')
plt.show()
```



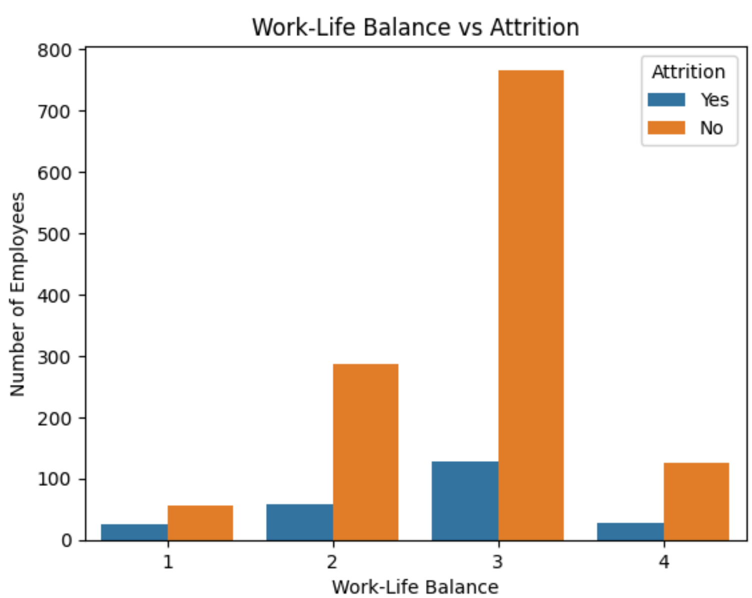
This below now looks at **'Attrition' vs Years at Company'.** It's interesting how similar these 2 graphs for **'Monthly Income'** and **'Years at Company'** are. This all leads us to believe that fewer employees are sr level/executive and most are junior level or mid level.

```
sns.violinplot(data=df, x='Attrition', y='YearsAtCompany')
plt.title('Years at Company vs Attrition')
plt.xlabel('Attrition')
plt.ylabel('Years at Company')
plt.show()
```

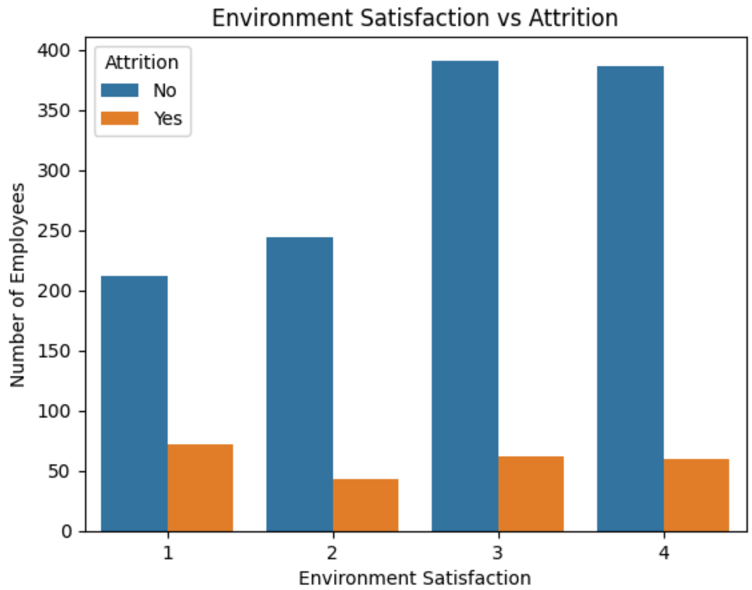


let's take a look at **'Work-Life Balance'**, **'Environment Satisfaction'**, and then one of our engineered features **'Environment/Job Satisfaction'**. Overall, it looks like these features don't seem to indicate if an employee leaves the company or not.

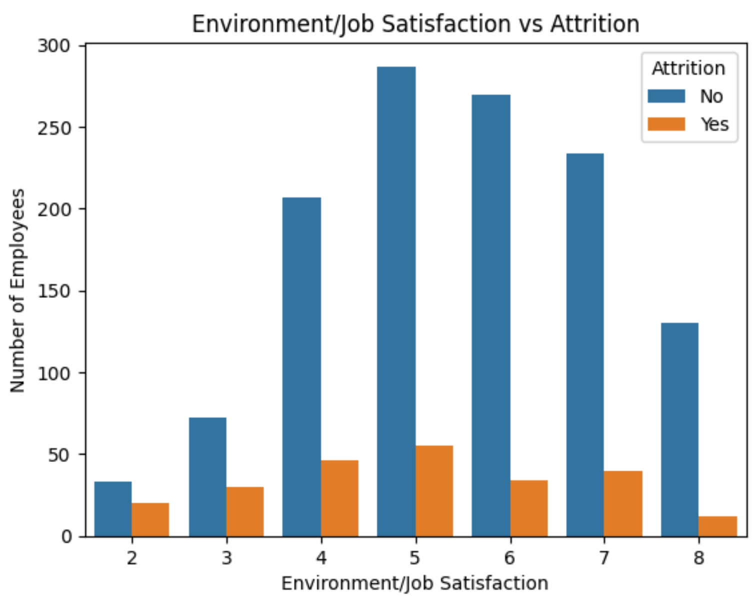
```
sns.countplot(data=df, x='WorkLifeBalance', hue='Attrition')
plt.title('Work-Life Balance vs Attrition')
plt.xlabel('Work-Life Balance')
plt.ylabel('Number of Employees')
plt.show()
```



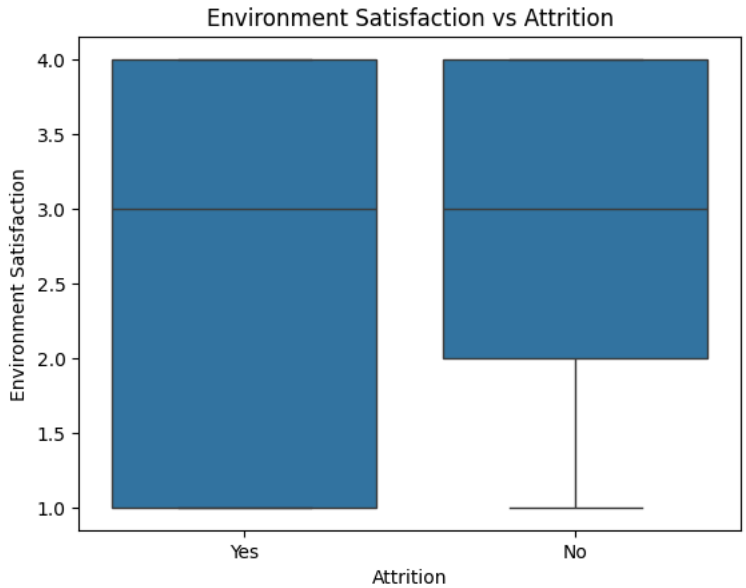

```
sns.countplot(data=df, x='EnvironmentSatisfaction', hue='Attrition')
plt.title('Environment Satisfaction vs Attrition')
plt.xlabel('Environment Satisfaction')
plt.ylabel('Number of Employees')
plt.show()
```




```
sns.countplot(data=df, x='JobEnvSatisfaction', hue='Attrition')
plt.title('Environment/Job Satisfaction vs Attrition')
plt.xlabel('Environment/Job Satisfaction')
plt.ylabel('Number of Employees')
plt.show()

```




```
sns.boxplot(data=df, x='Attrition', y='EnvironmentSatisfaction')
plt.title('Environment Satisfaction vs Attrition')
plt.xlabel('Attrition')
plt.ylabel('Environment Satisfaction')
plt.show()
```




```
# Filter the data to include only employees with positive attrition
df_attrition = df[df['Attrition'] == 'Yes']

# Count the number of employees with positive attrition based on 'OverTime'
attrition_overtime_counts = df_attrition['OverTime'].value_counts()

# Plotting the bar chart
plt.figure(figsize=(10, 6))
sns.barplot(x=attrition_overtime_counts.index, y=attrition_overtime_counts.values, palette='viridis')
plt.xlabel('OverTime')
plt.ylabel('Number of Employees with Positive Attrition')
plt.title('Employees with Positive Attrition and OverTime')
plt.show()
```

Ok, so here is what we learned:

- Most employees are junior or mid level
- Most employees are under 40 and most employees who leave the company are under 40
- Mostly males work here and are therefore more often leaving the company
- Most employees leaving rarely travel
- Research and Sales departments experience more Attrition
- Job Satisfaction has minimal impact on Attrition
- Employees making less money are more likely to experience Attrition
- Less experienced employees are more likely to experience Attrition
- Work Life Balance has minimal impact on Attrition

Alright, we're almost ready to model. Let's take a look at the data one more time. As we can see, we need to encode our categorical features before we move on to the modeling. Let's use **LabelEncoder and One-Hot Encoding**.

```
df.head()
```

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

5 rows × 34 columns

Label Encoder is great when there is an ordinal relationship in the data such as travel or overtime or Attrition.

```
label_encoder = LabelEncoder()

columns_to_encode = [
    'BusinessTravel',
    'OverTime',
    'Attrition'
]

for column in columns_to_encode:
#    df.loc[:, column] = label_encoder.fit_transform(df[column]).astype(int)
    df[column] = label_encoder.fit_transform(df[column])
    df[column] = pd.to_numeric(df[column], errors='coerce').astype(int)
# Print df_clean to verify the encoding
df.head()
df.dtypes
```

```
Age                                   int64
Attrition                             int64
BusinessTravel                        int64
DailyRate                             int64
Department                           object
DistanceFromHome                      int64
Education                             int64
EducationField                       object
EnvironmentSatisfaction               int64
Gender                               object
HourlyRate                            int64
JobInvolvement                        int64
JobLevel                              int64
JobRole                              object
JobSatisfaction                       int64
MaritalStatus                        object
MonthlyIncome                         int64
MonthlyRate                           int64
NumCompaniesWorked                    int64
OverTime                              int64
PercentSalaryHike                     int64
PerformanceRating                     int64
RelationshipSatisfaction              int64
StockOptionLevel                      int64
TotalWorkingYears                     int64
TrainingTimesLastYear                 int64
WorkLifeBalance                       int64
YearsAtCompany                        int64
YearsInCurrentRole                    int64
YearsSinceLastPromotion               int64
YearsWithCurrManager                  int64
YearsInCurrentRole_vs_YearsAtCompany float64
TotalYearsCurrentJob                  int64
JobEnvSatisfaction                    int64
dtype: object
```

And One-Hot Encoding is best when there is no ordinal relationship amongst the values, like in Department or Job Role.

```
columns_to_encode = ['Department',
                     'EducationField',
                     'Gender',
                     'JobRole',
                     'MaritalStatus']

# Create an instance of OneHotEncoder
ohe = OneHotEncoder(drop='first', sparse_output=False)

# Fit and transform the data
encoded_data = ohe.fit_transform(df[columns_to_encode])

# Create a DataFrame with the encoded data
encoded_df = pd.DataFrame(encoded_data, columns=ohe.get_feature_names_out(columns_to_encode))

# Drop the original columns and concatenate the encoded columns
df_encoded = df.drop(columns_to_encode, axis=1).reset_index(drop=True)
encoded_df = encoded_df.reset_index(drop=True)
df_cleaned = pd.concat([df_encoded, encoded_df], axis=1)

# Display the DataFrame
df_cleaned
```

	Age	Attrition	BusinessTravel	DailyRate	DistanceFromHome	Education	EnvironmentSatisfaction	HourlyRate	JobInvolvement	JobLev
0	41	1	2	1102	1	2	2	94	3	
1	49	0	1	279	8	1	3	61	2	
2	37	1	2	1373	2	2	4	92	2	
3	33	0	1	1392	3	4	4	56	3	
4	27	0	2	591	2	1	1	40	3	
...	...	...	...	...	...	...	...	...	...	
1465	36	0	1	884	23	2	3	41	4	
1466	39	0	2	613	6	1	4	42	2	
1467	27	0	2	155	4	3	2	87	4	
1468	49	0	1	1023	2	3	4	63	2	
1469	34	0	2	628	8	3	2	82	4	

1470 rows × 47 columns

One thing to note. With all imbalanced datasets, we need to verify that balancing the classes **Actually** improves our model. As noted in this [Oxford Academic Article](), sometimes adjusting for **class imbalance**s can have a detrimental impact on modeling.

```
# Initialize the scaler
scaler = StandardScaler()

# Select numeric columns to scale
numeric_columns = df_cleaned.select_dtypes(include=['int64', 'float64']).columns
numeric_columns = numeric_columns.drop('Attrition', errors='ignore')  # Exclude 'Attrition'

# Apply the scaler to the numeric columns
df_cleaned[numeric_columns] = scaler.fit_transform(df_cleaned[numeric_columns])
```

Let's take a look at correlations in our dataset.

```
df_cleaned.corr()
```

	Age	Attrition	BusinessTravel	DailyRate	DistanceFromHome	Education	EnvironmentSatisfa
Age	1.000000	-0.159205	0.024751	0.010661	-0.001686	0.208034	0.0
Attrition	-0.159205	1.000000	0.000074	-0.056652	0.077924	-0.031373	-0.1
BusinessTravel	0.024751	0.000074	1.000000	-0.004086	-0.024469	0.000757	0.0
DailyRate	0.010661	-0.056652	-0.004086	1.000000	-0.004985	-0.016806	0.0
DistanceFromHome	-0.001686	0.077924	-0.024469	-0.004985	1.000000	0.021042	-0.0
Education	0.208034	-0.031373	0.000757	-0.016806	0.021042	1.000000	-0.0
EnvironmentSatisfaction	0.010146	-0.103369	0.004174	0.018355	-0.016075	-0.027128	[illegible]
HourlyRate	0.024287	-0.006846	0.026528	0.023381	0.031131	0.016775	[illegible]
JobInvolvement	0.029820	-0.130016	0.039062	0.046135	0.008783	0.042438	-0.0
JobLevel	0.509604	-0.169105	0.019311	0.002966	0.005303	0.101589	0.0
JobSatisfaction	-0.004892	-0.103481	-0.033962	0.030571	-0.003669	-0.011296	-0.0
MonthlyIncome	0.497855	-0.159840	0.034319	0.007707	-0.017014	0.094961	-0.0
MonthlyRate	0.028051	0.015170	-0.014107	-0.032182	0.027473	-0.026084	0.0
NumCompaniesWorked	0.299635	0.043494	0.020875	0.038153	-0.029251	0.126317	0.0
OverTime	0.028062	0.246118	0.016543	0.009135	0.025514	-0.020322	0.0
PercentSalaryHike	0.003634	-0.013478	-0.029377	0.022704	0.040235	-0.011111	-0.0
PerformanceRating	0.001904	0.002889	-0.026341	0.000473	0.027110	-0.024539	-0.0
RelationshipSatisfaction	0.053535	-0.045872	-0.035986	0.007846	0.006557	-0.009118	0.0
StockOptionLevel	0.037510	-0.137145	-0.016727	0.042143	0.044872	0.018422	0.0
TotalWorkingYears	0.680381	-0.171063	0.034226	0.014515	0.004628	0.148280	-0.0
TrainingTimesLastYear	-0.019621	-0.059478	0.015240	0.002453	-0.036942	-0.025100	-0.0
WorkLifeBalance	-0.021490	-0.063939	-0.011256	-0.037848	-0.026556	0.009819	0.0
YearsAtCompany	0.311309	-0.134392	-0.014575	-0.034055	0.009508	0.069114	0.0
YearsInCurrentRole	0.212901	-0.160545	-0.011497	0.009932	0.018845	0.060236	0.0
YearsSinceLastPromotion	0.216513	-0.033019	-0.032591	-0.033229	0.010029	0.054254	0.0
YearsWithCurrManager	0.202089	-0.156199	-0.022636	-0.026363	0.014406	0.069065	-0.0
YearsInCurrentRole_vs_YearsAtCompany	-0.007213	-0.130324	-0.006886	0.048004	0.017989	0.023963	0.0
TotalYearsCurrentJob	0.224158	-0.171075	-0.018388	-0.008724	0.017975	0.069792	0.0
JobEnvSatisfaction	0.003681	-0.146763	-0.021255	0.034752	-0.013970	-0.027213	0.7
Department_Research & Development	0.017883	-0.085293	0.002598	0.014871	-0.008117	-0.018604	0.0
Department_Sales	-0.027549	0.080855	-0.006534	-0.003616	0.014085	0.014215	-0.0
EducationField_Life Sciences	0.016824	-0.032703	-0.023132	0.004028	-0.024499	0.013184	-0.0
EducationField_Marketing	0.038162	0.055781	0.037568	-0.064449	0.039294	0.072405	0.0
EducationField_Medical	-0.006354	-0.046999	-0.008520	0.034202	0.013486	-0.072335	-0.0
EducationField_Other	-0.041466	-0.017898	0.018654	-0.003893	-0.007969	0.038043	0.0
EducationField_Technical Degree	-0.027604	0.069355	0.010059	0.030869	-0.014802	-0.026742	0.0
Gender_Male	-0.036311	0.029453	-0.032981	-0.011716	-0.001851	-0.016547	0.0
JobRole_Human Resources	-0.029856	0.036215	0.013346	-0.021156	-0.024089	-0.005295	-0.0
JobRole_Laboratory Technician	-0.143176	0.098290	-0.014328	-0.006728	0.012369	-0.063566	-0.0
JobRole_Manager	0.294248	-0.083316	0.012221	-0.013224	-0.039190	0.028453	0.0
JobRole_Manufacturing Director	0.049726	-0.082994	0.006567	-0.005302	0.011848	-0.005290	0.0
JobRole_Research Director	0.185891	-0.088870	0.033365	-0.000021	-0.022351	0.049694	-0.0
JobRole_Research Scientist	-0.146518	-0.000360	0.011829	-0.002624	-0.010986	0.000709	0.0
JobRole_Sales Executive	-0.002001	0.019774	-0.022251	-0.000513	0.030761	0.053398	-0.0
JobRole_Sales Representative	-0.175785	0.157234	-0.001866	0.005375	-0.015994	-0.091465	0.0

MaritalStatus_Married	0.083919	-0.090984	0.057808	0.040035	0.030232	-0.001865	-0.(
MaritalStatus_Single	-0.119185	0.175419	-0.012097	-0.075835	-0.027445	0.004168	0.(

47 rows × 47 columns

Feature Scaling is not necessary for Decision Tree Classifiers (our baseline model), but this is still a valuable tool for models built using K-Nearest Neighbors.

Modeling

Let's begin our iterative process to build the best model for our data. 1st thing we will need to do is create our train and test sets. Before that, lets discuss our plan.

We will end up building **20 models!** Here is a quick synopsis of our methodology:

- baseline model
- baseline model addressing **class imbalance**
- Baseline model addressing **class imbalance** with GridSearchCV performed to optimize hyperparameters
- model with unimportant features dropped
- model with unimportant features dropped addressing **class imbalance**
- model with unimportant features dropped addressing **class imbalance** with GridSearchCV performed to optimize hyperparameters

This methodology will be applied to all 3 types of models we are building:

- Decision Tree
- Random Forest
- K-Nearest Neighbors

Then we will read in all results and pick the 4 best models. Those 4 best models will be used to create a **Stacking Ensemble** and then a **Stacking Ensemble with GridSearchCV** to finetune hyperparameters. Once our best model is created, we can discuss results.

Decision Tree

A Decision Tree is a supervised learning appraoch that is great for binary classifications because of the way the model splits the features based on simple rules to detect patterns and also because of the way the model handles outliers.

More documentation on Decision Trees can be found [here](#).

1st thing we need to do is create our X and y variables and then build our train and test sets.

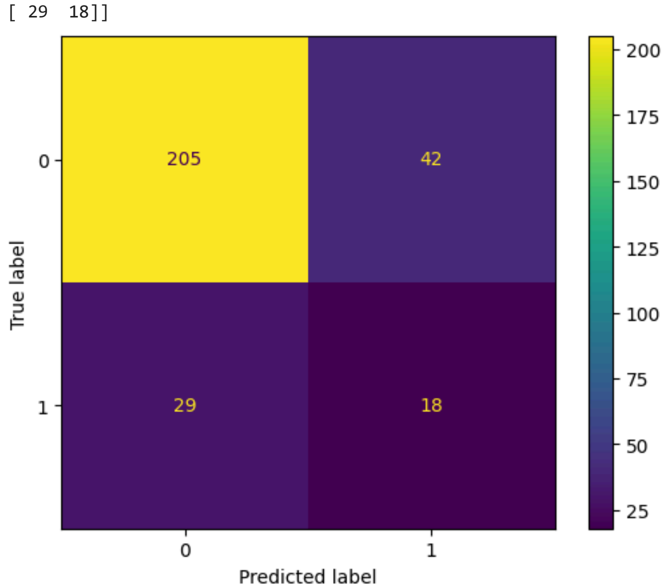
```
# Seperate target and features
X = df_cleaned.drop('Attrition', axis=1) # all columns except the target column our our independent variables
y = df_cleaned['Attrition']

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, shuffle=True, stratify=y)
```

For our first model, we will build a Decision Tree Classifier. These are great for **Binary Classifications** because we can gather **Feature Importances** and they are great at handling **non-linear data**.

This model will be built using our original dataset-**before we address the class imbalancing**.

```python
clf_tree =DecisionTreeClassifier(random_state=42)
clf_tree.fit(X_train, y_train)

y_pred_tree = clf_tree.predict(X_test)


# Evaluate the classifier
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_tree))
print("\nClassification Report:")
print(classification_report(y_test, y_pred_tree))

print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_tree))

print("\nConfusion Matrix:")
cm_tree = confusion_matrix(y_test, y_pred_tree)
print(cm_tree)

# Plot the confusion matrix
ConfusionMatrixDisplay(confusion_matrix=cm_tree, display_labels=clf_tree.classes_).plot()
plt.show()
```

```
Accuracy Score:
0.7585034013605442

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.83      0.85       247
           1       0.30      0.38      0.34        47

    accuracy                           0.76       294
   macro avg       0.59      0.61      0.59       294
weighted avg       0.78      0.76      0.77       294


Accuracy Score:
0.7585034013605442

Confusion Matrix:
[[205  42]
 [ 29  18]]
```



Now we need to address our **Class Imbalance**. As we saw earlier, our minority class is roughly 16% of our data. In a perfect world, the data would be evenly split 50/50. However, we can still work with this data and use sampling techniques to improve our results. We will combine SMOTE oversampling and Undersampling with stratification. Undersampling with stratification reduces the number of samples in the majority class, while maintaining the class distribution. This helps create a more balanced dataset and should improve modeling.

75% accuracy is not a bad start. We can definitely improve this though!

Now let's build a **decision tree with our balanced data.**

```
# Initialize SMOTE and RandomUnderSampler
smote = SMOTE(sampling_strategy=.5, random_state=42)  # Adjust sampling_strategy as needed
under_sampler = RandomUnderSampler(sampling_strategy=1, random_state=42, replacement=False)  # Adjust sampling_strategy as needed

# Apply SMOTE to the training data
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Apply RandomUnderSampler to the training data
X_train_balanced, y_train_balanced = under_sampler.fit_resample(X_train_smote, y_train_smote)
# Print the class distribution after resampling
print("Class distribution before resampling:")
print(y_train.value_counts())

print("\nClass distribution after SMOTE:")
print(y_train_smote.value_counts())

print("\nClass distribution after SMOTE and undersampling:")
print(y_train_balanced.value_counts())

# Print the shapes of the resulting DataFrames
print(f"\nX_train shape after resampling: {X_train_balanced.shape}")
print(f"y_train shape after resampling: {y_train_balanced.shape}")
```

```
Class distribution before resampling:
Attrition
0    986
1    190
Name: count, dtype: int64

Class distribution after SMOTE:
Attrition
0    986
1    493
Name: count, dtype: int64

Class distribution after SMOTE and undersampling:
Attrition
0    493
1    493
Name: count, dtype: int64

X_train shape after resampling: (986, 46)
y_train shape after resampling: (986,)
```

Quick checks below to make sure our splits are shaped properly for modeling.

```
# Print the shapes of the resulting DataFrames
print(f"X_train shape: {X_train_balanced.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train_balanced.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (986, 46)
X_test shape: (294, 46)
y_train shape: (986,)
y_test shape: (294,)
```

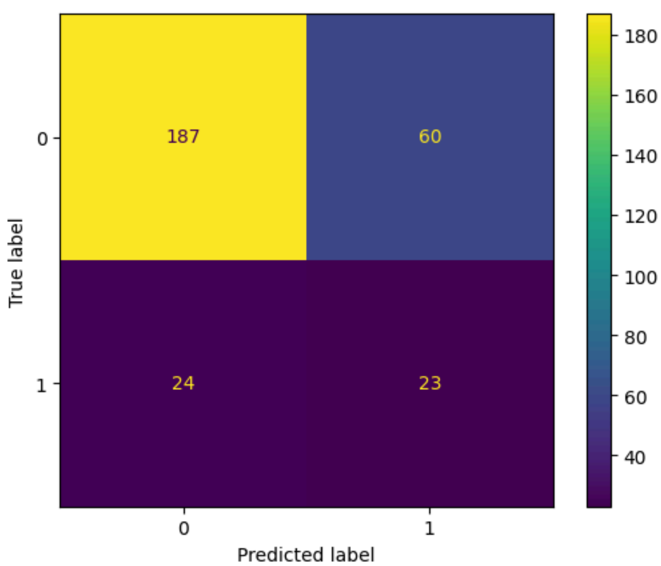
```
# Check the class distribution in y_train and y_test
print("Class distribution in y_train:\n", y_train_balanced.value_counts())
print("Class distribution in y_test:\n", y_test.value_counts())
```

```
Class distribution in y_train:
 Attrition
0    493
1    493
Name: count, dtype: int64
Class distribution in y_test:
 Attrition
0    247
1     47
Name: count, dtype: int64
```

Here is the **Decision Tree with balanced classes.**

```
clf_tree_balanced =DecisionTreeClassifier(random_state=42)
clf_tree_balanced.fit(X_train_balanced, y_train_balanced)

y_pred_tree_balanced = clf_tree_balanced.predict(X_test)


# Evaluate the classifier
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_tree_balanced))
print("\nClassification Report:")
print(classification_report(y_test, y_pred_tree_balanced))

print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_tree_balanced))

print("\nConfusion Matrix:")
cm_tree_balanced = confusion_matrix(y_test, y_pred_tree_balanced)
print(cm_tree_balanced)

# Plot the confusion matrix
ConfusionMatrixDisplay(confusion_matrix=cm_tree_balanced, display_labels=clf_tree_balanced.classes_).plot()
plt.show()
```

```
Accuracy Score:
0.7142857142857143

Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.76      0.82       247
           1       0.28      0.49      0.35        47

    accuracy                           0.71       294
   macro avg       0.58      0.62      0.59       294
weighted avg       0.79      0.71      0.74       294


Accuracy Score:
0.7142857142857143

Confusion Matrix:
[[187  60]
 [ 24  23]]
```

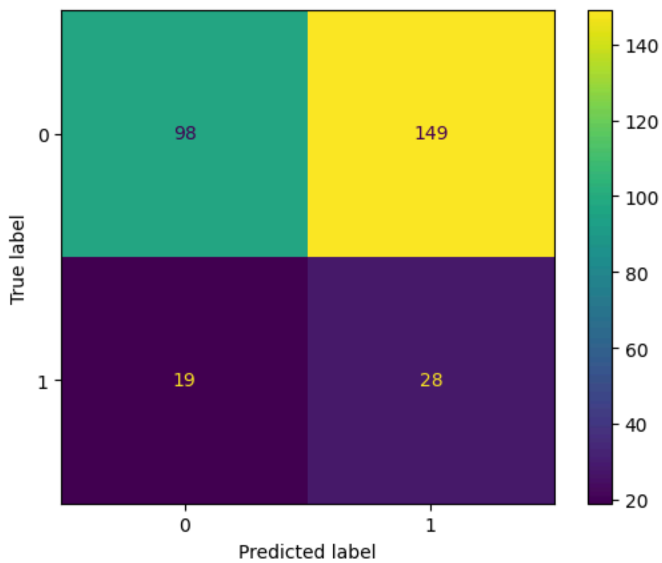


As we anticipated based on the Oxford Article, balancing the classes had a negative impact on overall model accuracy. **Recall and F1 score** for the minority class improved though. This is part of the trade-off we have to make. Since recall of our minority class is important, we will still consider balancing the data for the remainder of this notebook.

Next, let's use **gridsearchCV to finetune the parameters of our Decision Tree.**

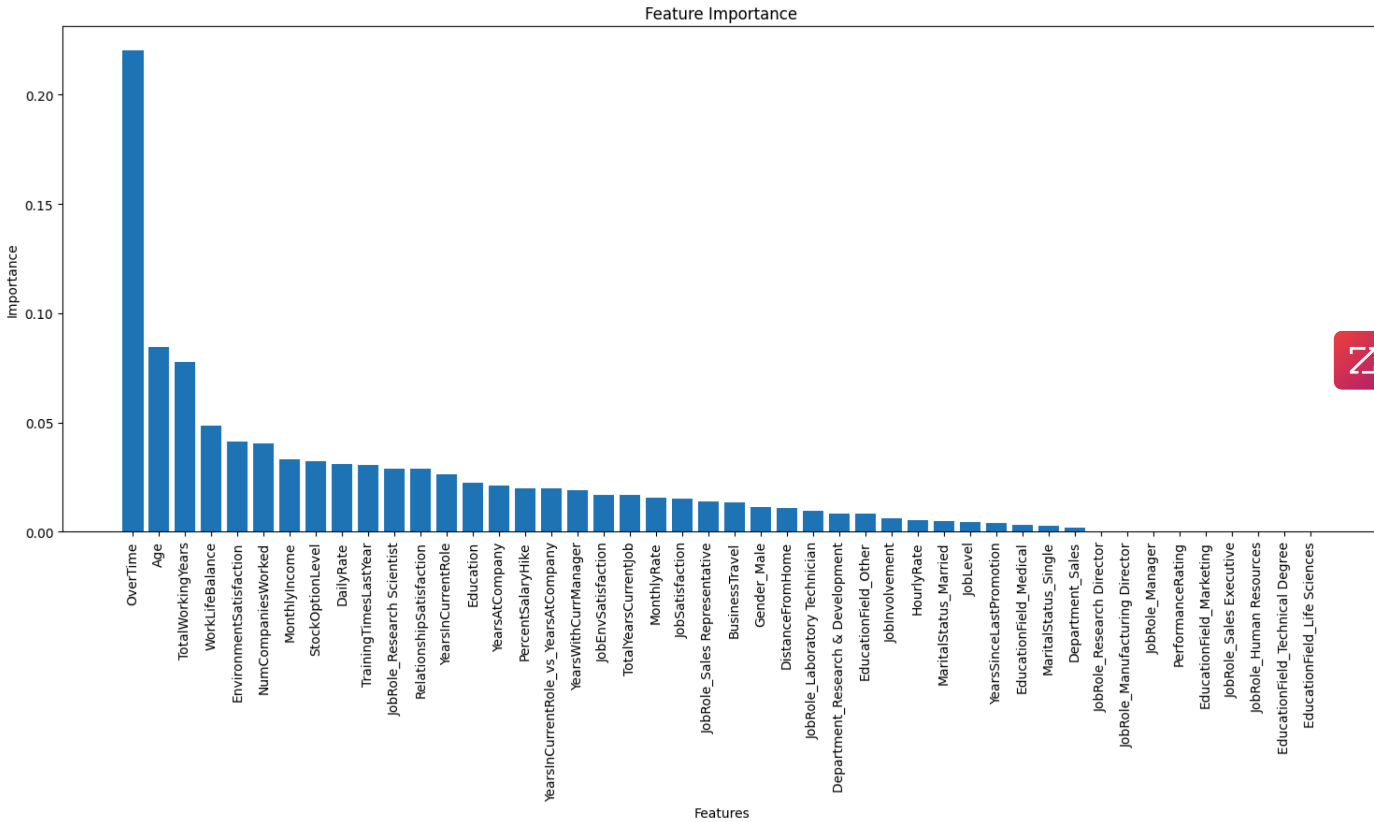
```
# Define the parameter grid
param_grid_tree = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 100],
    'min_samples_split': [2, 15, 20],
    'min_samples_leaf': [1, 10, 15, 20],
    'max_features': [None, 'sqrt', 'log2', 0.5, 0.75],
    'class_weight': [None, 'balanced', {0: 1, 1: 2}, {0: 1, 1: 4}]
}

# Initialize the classifier
clf_tree_balanced_cv = DecisionTreeClassifier(random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=clf_tree_balanced_cv, param_grid=param_grid_tree, cv=5, scoring='recall', n_jobs=-1)

# Fit GridSearchCV
grid_search.fit(X_train_balanced, y_train_balanced)

# Print the best parameters found by GridSearchCV
print("Best parameters found: ", grid_search.best_params_)

# Use the best estimator to make predictions
best_clf_tree_balanced_cv = grid_search.best_estimator_
y_pred_tree_balanced_cv = best_clf_tree_balanced_cv.predict(X_test)

# Evaluate the classifier
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_tree_balanced_cv))
print("\nClassification Report:")
print(classification_report(y_test, y_pred_tree_balanced_cv))

print("\nConfusion Matrix:")
cm_tree_balanced_cv = confusion_matrix(y_test, y_pred_tree_balanced_cv)
print(cm_tree_balanced_cv)

# Plot the confusion matrix
ConfusionMatrixDisplay(confusion_matrix=cm_tree_balanced_cv, display_labels=best_clf_tree_balanced_cv.classes_).plot()
plt.show()
```

```
Best parameters found:  {'class_weight': {0: 1, 1: 4}, 'criterion': 'entropy', 'max_depth': None, 'max_features': 'sqrt', 'min_samples_l

Accuracy Score:
0.42857142857142855

Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.40      0.54       247
           1       0.16      0.60      0.25        47

    accuracy                           0.43       294
   macro avg       0.50      0.50      0.39       294
weighted avg       0.73      0.43      0.49       294


Confusion Matrix:
[[ 98 149]
 [ 19  28]]
```



Now let's take a moment to look a the feature importances from our balanced dataset to see if we can **drop some columns that did not contribute to model performance.**

```
# Function definition
def plot_feature_importance(model, X_train):
    # Get feature importances
    feature_importances = model.feature_importances_

    # Sort feature importances in descending order
    indices = np.argsort(feature_importances)[::-1]

    # Plot the feature importances
    plt.figure(figsize=(15, 9))
    plt.bar(range(X_train.shape[1]), feature_importances[indices], align='center')
    plt.xticks(range(X_train.shape[1]), X_train.columns[indices], rotation=90)
    plt.xlabel('Features')
    plt.ylabel('Importance')
    plt.title('Feature Importance')
    plt.tight_layout()
    plt.show()

# Call the function at the end of the notebook
plot_feature_importance(clf_tree_balanced, X_train_balanced)

```



There are a few features at the left side of the graph that have little to no predicitive relevance here. Let's drop them and see what happens.

Now, let's retrain the model on a new df that drops some of these unimportant features and start this process over.

```
df_fi = df_cleaned.drop(columns=[
    'EducationField_Life Sciences',
    'EducationField_Technical Degree',
    'JobRole_Human Resources',
    'MaritalStatus_Married'
])
```

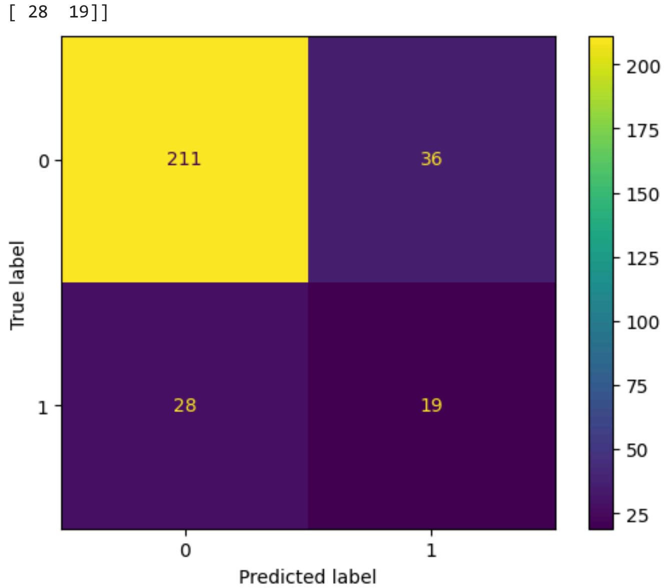
Now we need to split and train our data again, since we have dropped some columns. We will call the new variables **X-fi and y_fi.**

```
# Seperate target and features
X_fi = df_fi.drop('Attrition', axis=1) # all columns except the target column our our independent variables
y_fi = df_fi['Attrition']

# Split the data into train and test sets
X_train_fi, X_test_fi, y_train_fi, y_test_fi = train_test_split(X_fi, y_fi, test_size=0.2, random_state=42, shuffle=True, stratify=y)
```

Below we have our **Decision Tree trained on the new df with dropped features.**

```
clf_tree_fi =DecisionTreeClassifier(random_state=42)
clf_tree_fi.fit(X_train_fi, y_train_fi)

y_pred_tree_fi = clf_tree_fi.predict(X_test_fi)


# Evaluate the classifier
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_tree_fi))
print("\nClassification Report:")
print(classification_report(y_test_fi, y_pred_tree_fi))

print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_tree_fi))

print("\nConfusion Matrix:")
cm_tree_fi = confusion_matrix(y_test_fi, y_pred_tree_fi)
print(cm_tree_fi)

# Plot the confusion matrix
ConfusionMatrixDisplay(confusion_matrix=cm_tree_fi, display_labels=clf_tree_fi.classes_).plot()
plt.show()
```

```
Accuracy Score:
0.782312925170068

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.85      0.87       247
           1       0.35      0.40      0.37        47

    accuracy                           0.78       294
   macro avg       0.61      0.63      0.62       294
weighted avg       0.80      0.78      0.79       294


Accuracy Score:
0.782312925170068

Confusion Matrix:
[[211  36]
 [ 28  19]]
```



Now let's **balance the new X_train_fi and y_train_fi** the same way we trained X_train and y_train above.

```
# Initialize SMOTE and RandomUnderSampler
smote = SMOTE(sampling_strategy=.75, random_state=42)  # Adjust sampling_strategy as needed
under_sampler = RandomUnderSampler(sampling_strategy=.75, random_state=42, replacement=False)  # Adjust sampling_strategy as needed

# Apply SMOTE to the training data
X_train_smote_fi, y_train_smote_fi = smote.fit_resample(X_train_fi, y_train_fi)

```

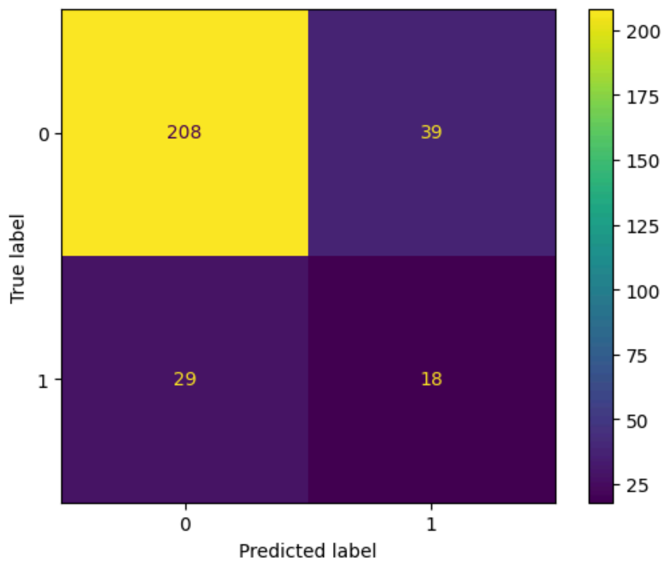
```
# Apply RandomUnderSampler to the training data
X_train_balanced_fi, y_train_balanced_fi = under_sampler.fit_resample(X_train_smote_fi, y_train_smote_fi)
# Print the class distribution after resampling
print("Class distribution before resampling:")
print(y_train_fi.value_counts())

print("\nClass distribution after SMOTE:")
print(y_train_smote_fi.value_counts())

print("\nClass distribution after SMOTE and undersampling:")
print(y_train_balanced_fi.value_counts())

# Print the shapes of the resulting DataFrames
print(f"\nX_train shape after resampling: {X_train_balanced_fi.shape}")
print(f"y_train shape after resampling: {y_train_balanced_fi.shape}")
```

```
Class distribution before resampling:
Attrition
0    986
1    190
Name: count, dtype: int64

Class distribution after SMOTE:
Attrition
0    986
1    739
Name: count, dtype: int64

Class distribution after SMOTE and undersampling:
Attrition
0    985
1    739
Name: count, dtype: int64

X_train shape after resampling: (1724, 42)
y_train shape after resampling: (1724,)
```

Below we have the **Decision Tree with balanced data** from the new df with dropped, unimportant features!

```
clf_tree_fi_balanced =DecisionTreeClassifier(random_state=42)
clf_tree_fi_balanced.fit(X_train_balanced_fi, y_train_balanced_fi)

y_pred_tree_fi_balanced = clf_tree_fi_balanced.predict(X_test_fi)


# Evaluate the classifier
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_tree_fi_balanced))
print("\nClassification Report:")
print(classification_report(y_test_fi, y_pred_tree_fi_balanced))

print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_tree_fi_balanced))

print("\nConfusion Matrix:")
cm_tree_fi_balanced = confusion_matrix(y_test_fi, y_pred_tree_fi_balanced)
print(cm_tree_fi_balanced)

# Plot the confusion matrix
ConfusionMatrixDisplay(confusion_matrix=cm_tree_fi_balanced, display_labels=clf_tree_fi_balanced.classes_).plot()
plt.show()
```

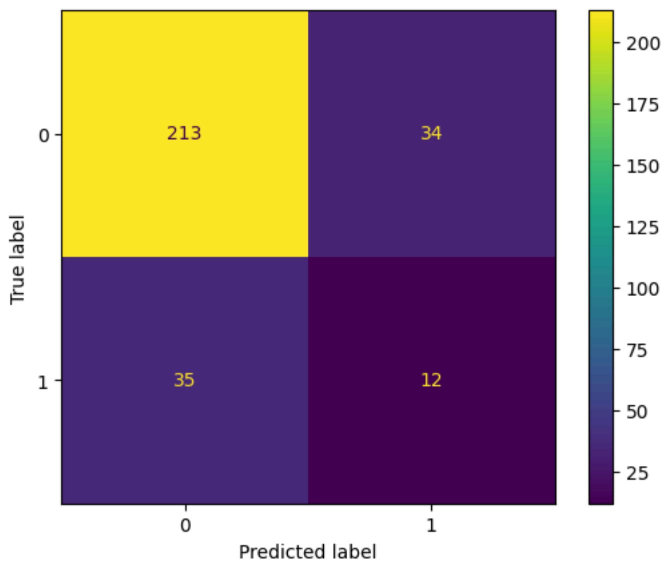
```
Accuracy Score:
0.7687074829931972

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.84      0.86       247
           1       0.32      0.38      0.35        47

    accuracy                           0.77       294
   macro avg       0.60      0.61      0.60       294
weighted avg       0.79      0.77      0.78       294


Accuracy Score:
0.7687074829931972

Confusion Matrix:
[[208  39]
 [ 29  18]]
```



Last step for our Decision Tree is to **finetune hyperparameters via GridSearchCV.**

```
# Define the parameter grid
param_grid_tree_fi_cv = {
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 5, 10],
    'criterion': ['gini', 'entropy']
}

# Initialize the classifier
clf_tree_fi_balanced_cv = DecisionTreeClassifier(random_state=42)

# Initialize GridSearchCV with the classifier and parameter grid
grid_search = GridSearchCV(estimator=clf_tree_fi_balanced_cv, param_grid=param_grid_tree_fi_cv,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=1)

# Fit GridSearchCV to the training data
grid_search.fit(X_train_balanced_fi, y_train_balanced_fi)

# Get the best estimator
best_clf_tree_fi_balanced_cv = grid_search.best_estimator_

# Make predictions with the best estimator
y_pred_tree_fi_balanced_cv = best_clf_tree_fi_balanced_cv.predict(X_test_fi)

# Evaluate the best estimator
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_tree_fi_balanced_cv))
print("\nClassification Report:")
print(classification_report(y_test_fi, y_pred_tree_fi_balanced_cv))

print("\nConfusion Matrix:")
cm_tree_fi_balanced_cv = confusion_matrix(y_test_fi, y_pred_tree_fi_balanced_cv)
print(cm_tree_fi_balanced_cv)

# Plot the confusion matrix
ConfusionMatrixDisplay(confusion_matrix=cm_tree_fi_balanced_cv, display_labels=best_clf_tree_fi_balanced_cv.classes_).plot()
plt.show()
```

```
Fitting 5 folds for each of 192 candidates, totalling 960 fits

Accuracy Score:
0.7653061224489796

Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.86      0.86       247
           1       0.26      0.26      0.26        47

    accuracy                           0.77       294
   macro avg       0.56      0.56      0.56       294
weighted avg       0.76      0.77      0.76       294


Confusion Matrix:
[[213  34]
 [ 35  12]]
```



We finished our process for the Decision Tree. Let's complete this same methodology on our **Random Forest and KNN.**

Random Forest

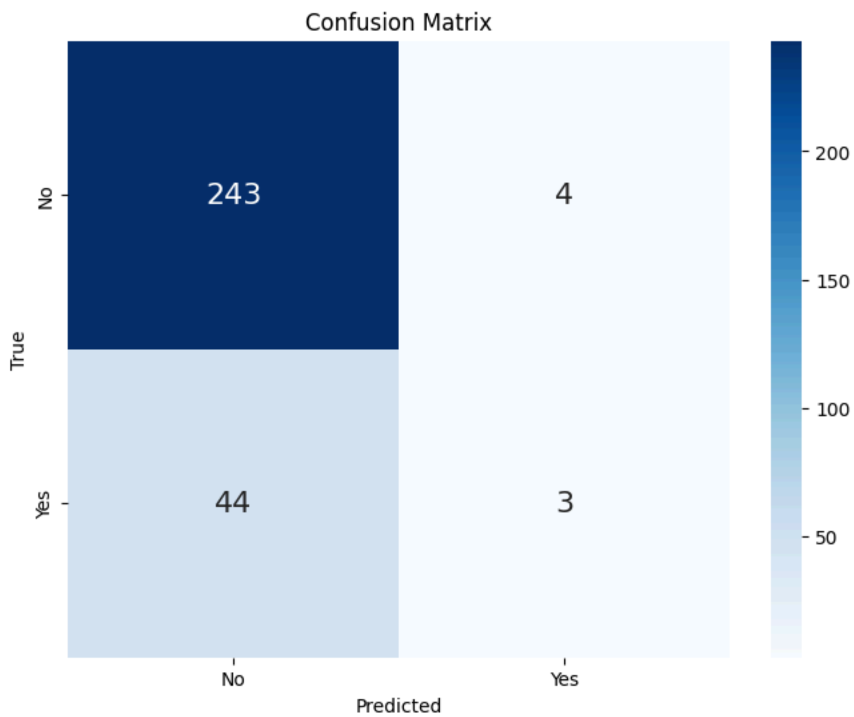
A Random Forest is great for binary classifications for the same great reason Decision Trees are with the added advantage of being able to prevent overfitting.

More documentation on Decision Trees can be found here.

We will start with our original **X_train and y_train.**

```
# Initialize and train your classifier, adjusting class weights if needed
clf_rf = RandomForestClassifier(class_weight='balanced', random_state=42)
clf_rf.fit(X_train, y_train)

# Make predictions and evaluate the model
y_pred_rf = clf_rf.predict(X_test)
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_rf))
print(classification_report(y_test, y_pred_rf))

# Print confusion matrix
print("Confusion Matrix:")
cm_rf = confusion_matrix(y_test, y_pred_rf)
print(cm_rf)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

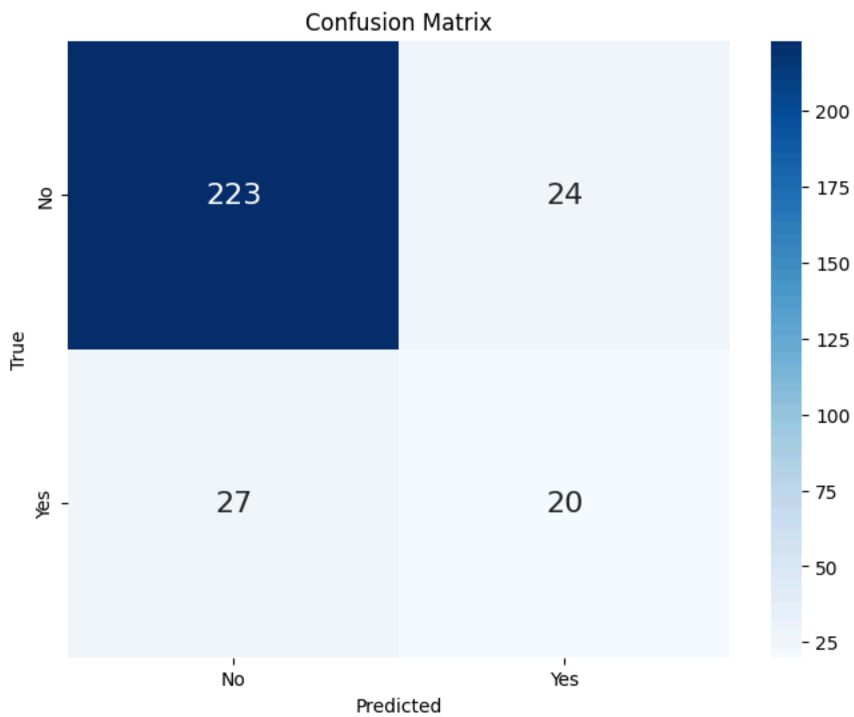
```
Accuracy Score:
0.8367346938775511
              precision    recall  f1-score   support

           0       0.85      0.98      0.91       247
           1       0.43      0.06      0.11        47

    accuracy                           0.84       294
   macro avg       0.64      0.52      0.51       294
weighted avg       0.78      0.84      0.78       294

Confusion Matrix:
[[243   4]
 [ 44   3]]
```



Following the same methodology from the Decision Tree Classifier, let's **build another Random Forest with the balanced data.**

```python
# Initialize and train your classifier, adjusting class weights if needed
clf_rf_balanced = RandomForestClassifier(class_weight='balanced', random_state=42)
clf_rf_balanced.fit(X_train_balanced, y_train_balanced)

# Make predictions and evaluate the model
y_pred_rf_balanced = clf_rf_balanced.predict(X_test)
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_rf_balanced))
print(classification_report(y_test, y_pred_rf_balanced))

# Print confusion matrix
print("Confusion Matrix:")
cm_rf_balanced = confusion_matrix(y_test, y_pred_rf_balanced)
print(cm_rf_balanced)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf_balanced, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

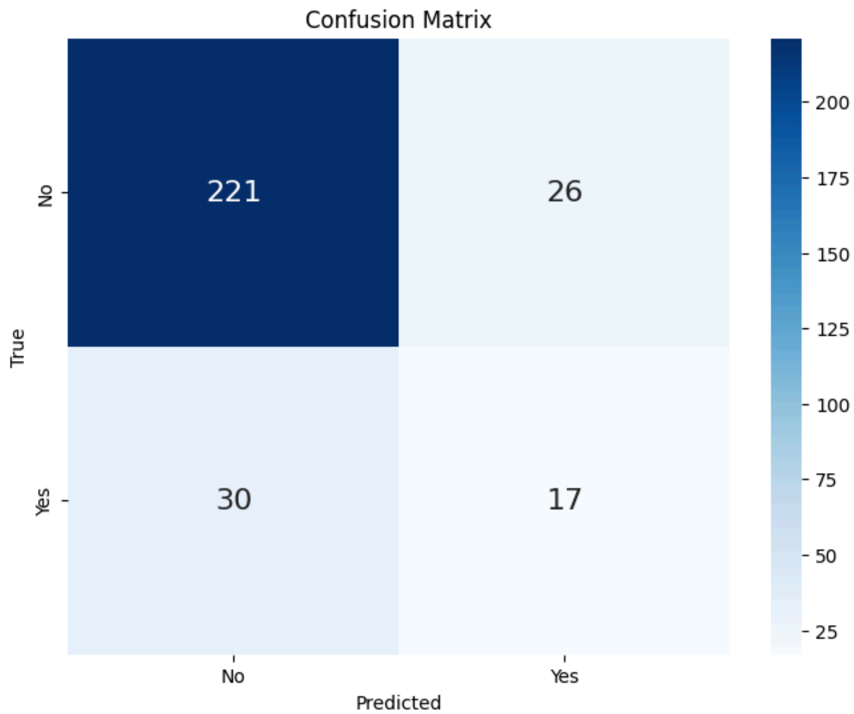
```
Accuracy Score:
0.826530612244898
              precision    recall  f1-score   support

           0       0.89      0.90      0.90       247
           1       0.45      0.43      0.44        47

    accuracy                           0.83       294
   macro avg       0.67      0.66      0.67       294
weighted avg       0.82      0.83      0.82       294

Confusion Matrix:
[[223  24]
 [ 27  20]]
```



Now we can take a look at the **finetuned random forest built with class imbalancing addressed.**

```
# Define the parameter grid
param_grid_rf = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy']
}

# Initialize the classifier
clf_rf_balanced_cv = RandomForestClassifier(class_weight='balanced', random_state=42)

# Initialize GridSearchCV with the classifier and parameter grid
grid_search = GridSearchCV(estimator=clf_rf_balanced, param_grid=param_grid_rf,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=1)

# Fit GridSearchCV to the training data
grid_search.fit(X_train_balanced, y_train_balanced)

# Get the best estimator
best_clf_rf_balanced_cv = grid_search.best_estimator_

# Make predictions with the best estimator
y_pred_rf_balanced_cv = best_clf_rf_balanced_cv.predict(X_test)

# Evaluate the best estimator
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_rf_balanced_cv))
print(classification_report(y_test, y_pred_rf_balanced_cv))

# Print confusion matrix
print("Confusion Matrix:")
cm_rf_balanced = confusion_matrix(y_test, y_pred_rf_balanced_cv)
print(cm_rf_balanced)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf_balanced, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Fitting 5 folds for each of 270 candidates, totalling 1350 fits

Accuracy Score:
0.8095238095238095
              precision    recall  f1-score   support

           0       0.88      0.89      0.89       247
           1       0.40      0.36      0.38        47

    accuracy                           0.81       294
   macro avg       0.64      0.63      0.63       294
weighted avg       0.80      0.81      0.81       294

Confusion Matrix:
[[221  26]
 [ 30  17]]
```

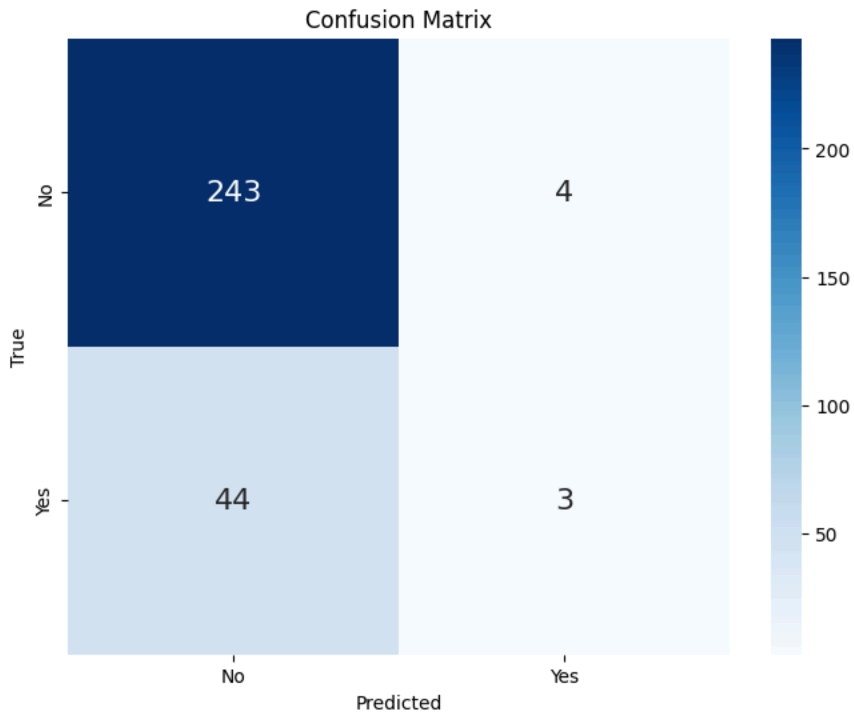


And now we remove the **unimportant features in our Random Forest.**

Notice the X_train_fi and y_train_fi used here when fitting the Random Forest.

```
# Initialize and train your classifier, adjusting class weights if needed
clf_rf_fi = RandomForestClassifier(class_weight='balanced', random_state=42)
clf_rf_fi.fit(X_train_fi, y_train_fi)

# Make predictions and evaluate the model
y_pred_rf_fi = clf_rf_fi.predict(X_test_fi)
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_rf_fi))
print(classification_report(y_test_fi, y_pred_rf_fi))

# Print confusion matrix
print("Confusion Matrix:")
cm_rf_fi = confusion_matrix(y_test_fi, y_pred_rf_fi)
print(cm_rf_fi)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf_fi, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

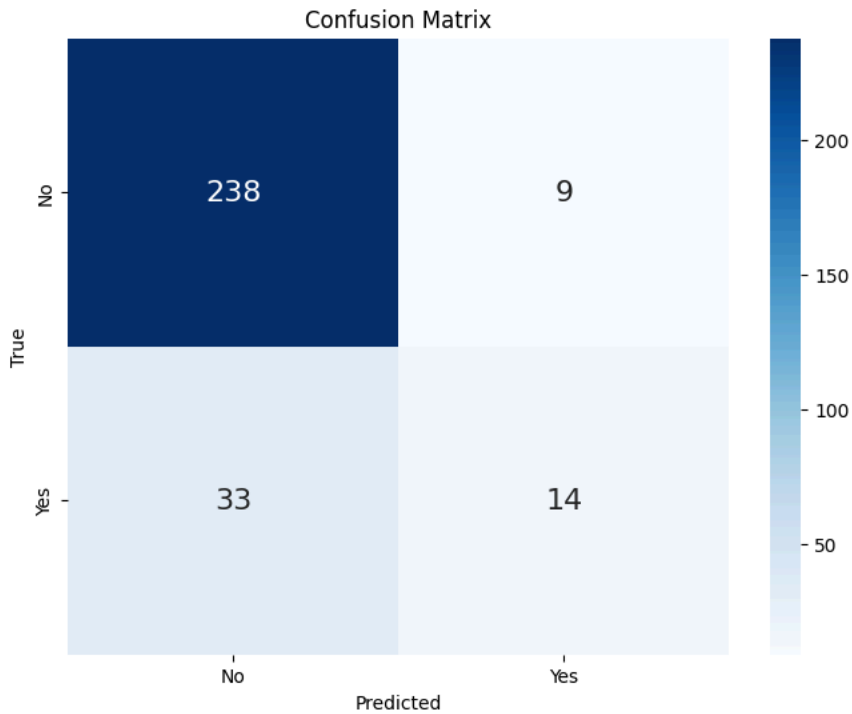
```
Accuracy Score:
0.8367346938775511
              precision    recall  f1-score   support

           0       0.85      0.98      0.91       247
           1       0.43      0.06      0.11        47

    accuracy                           0.84       294
   macro avg       0.64      0.52      0.51       294
weighted avg       0.78      0.84      0.78       294

Confusion Matrix:
[[243   4]
 [ 44   3]]
```



Now addressing **class imbalancing and with our Random Forest that has unimportant features dropped.**

```
# Initialize and train your classifier, adjusting class weights if needed
clf_rf_fi_balanced = RandomForestClassifier(class_weight='balanced', random_state=42)
clf_rf_fi_balanced.fit(X_train_balanced_fi, y_train_balanced_fi)

# Make predictions and evaluate the model
y_pred_rf_fi_balanced = clf_rf_fi_balanced.predict(X_test_fi)
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_rf_fi_balanced))
print(classification_report(y_test_fi, y_pred_rf_fi_balanced))

# Print confusion matrix
print("Confusion Matrix:")
cm_fi_balanced = confusion_matrix(y_test_fi, y_pred_rf_fi_balanced)
print(cm_fi_balanced)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_fi_balanced, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

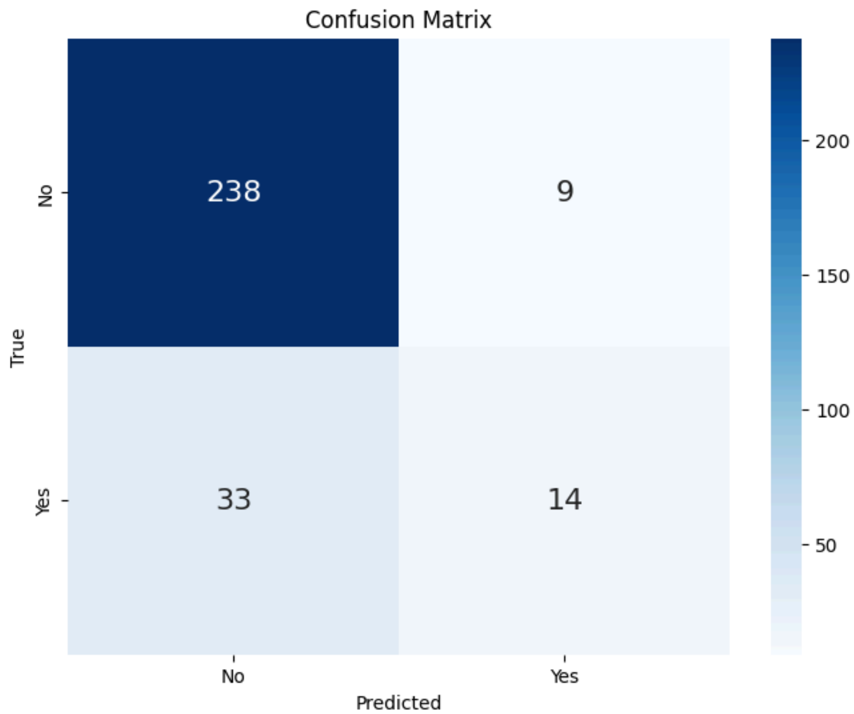
```
Accuracy Score:
0.8571428571428571
              precision    recall  f1-score   support

           0       0.88      0.96      0.92       247
           1       0.61      0.30      0.40        47

    accuracy                           0.86       294
   macro avg       0.74      0.63      0.66       294
weighted avg       0.84      0.86      0.84       294

Confusion Matrix:
[[238   9]
 [ 33  14]]
```



Now let's **finetune the Random Forest after unimportant features are dropped.**

```
# Define the parameter grid
param_grid_rf_fi = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy']
}

# Initialize the classifier
clf_rf_fi_balanced_cv = RandomForestClassifier(class_weight='balanced', random_state=42)

# Initialize GridSearchCV with the classifier and parameter grid
grid_search = GridSearchCV(estimator=clf_rf_fi_balanced_cv, param_grid=param_grid_rf_fi,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=1)

# Fit GridSearchCV to the training data
grid_search.fit(X_train_balanced_fi, y_train_balanced_fi)

# Get the best estimator
best_clf_rf_fi_balanced_cv = grid_search.best_estimator_

# Make predictions with the best estimator
y_pred_rf_fi_balanced_cv = best_clf_rf_fi_balanced_cv.predict(X_test_fi)

# Evaluate the best estimator
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_rf_fi_balanced_cv))
print(classification_report(y_test_fi, y_pred_rf_fi_balanced_cv))

# Print confusion matrix
print("Confusion Matrix:")
cm_rf_fi_balanced_cv = confusion_matrix(y_test_fi, y_pred_rf_fi_balanced_cv)
print(cm_rf_fi_balanced_cv)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf_fi_balanced_cv, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits

Accuracy Score:
0.8571428571428571
              precision    recall  f1-score   support

           0       0.88      0.96      0.92       247
           1       0.61      0.30      0.40        47

    accuracy                           0.86       294
   macro avg       0.74      0.63      0.66       294
weighted avg       0.84      0.86      0.84       294

Confusion Matrix:
[[238   9]
 [ 33  14]]
```



K-Nearest Neighbors

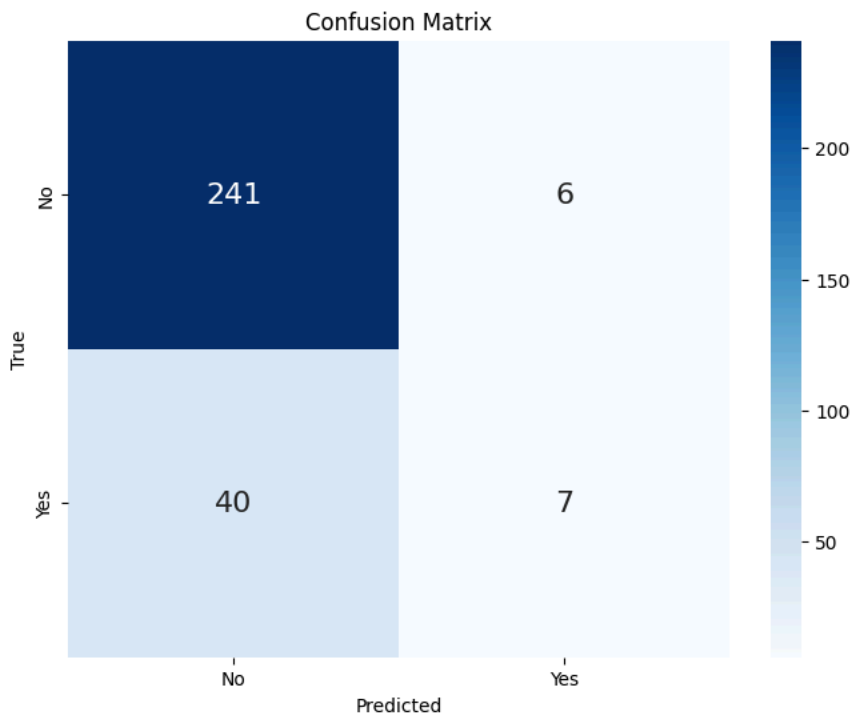
KNN is great for binary classifications because of the way it predicts patterns based on **euclidean distance** of data points. This is a great method for classifications.

More Documentation on KNN can be found here.

Here is our KNN baseline.

```
# Initialize and train the KNN classifier
clf_knn = KNeighborsClassifier()
clf_knn.fit(X_train, y_train)

# Make predictions and evaluate the model
y_pred_knn = clf_knn.predict(X_test)
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn))

# Print confusion matrix
print("Confusion Matrix:")
cm_knn = confusion_matrix(y_test, y_pred_knn)
print(cm_knn)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_knn, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

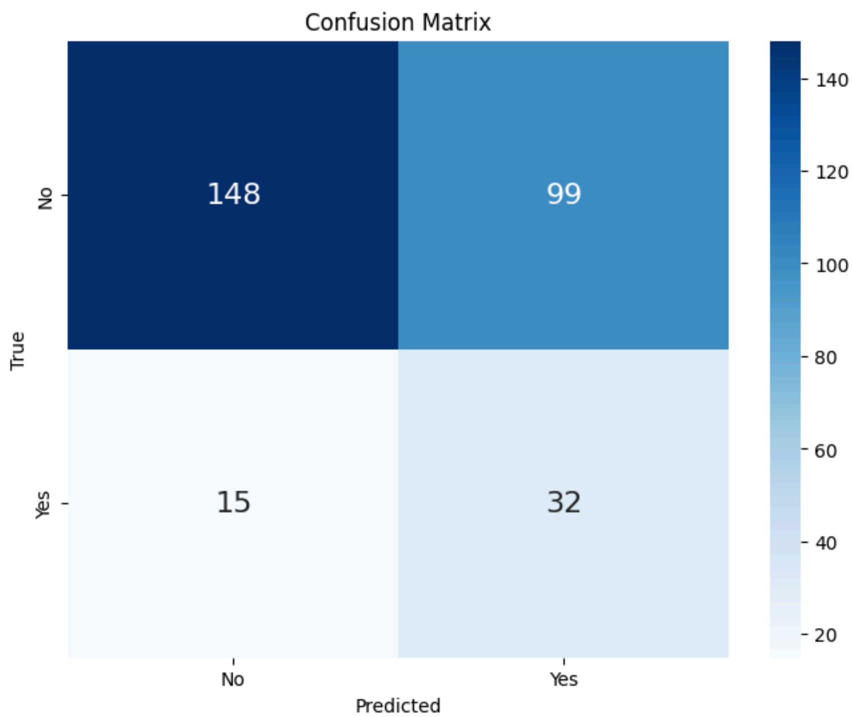
```
Accuracy Score:
0.8435374149659864
              precision    recall  f1-score   support

           0       0.86      0.98      0.91       247
           1       0.54      0.15      0.23        47

    accuracy                           0.84       294
   macro avg       0.70      0.56      0.57       294
weighted avg       0.81      0.84      0.80       294

Confusion Matrix:
[[241   6]
 [ 40   7]]
```



Now let's use the **balanced data for our KNN model.**

```python
# Initialize and train the KNN classifier
clf_knn_balanced = KNeighborsClassifier()
clf_knn_balanced.fit(X_train_balanced, y_train_balanced)

# Make predictions and evaluate the model
y_pred_knn_balanced = clf_knn_balanced.predict(X_test)
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_knn_balanced))
print(classification_report(y_test, y_pred_knn_balanced))

# Print confusion matrix
print("Confusion Matrix:")
cm_knn_balanced = confusion_matrix(y_test, y_pred_knn_balanced)
print(cm_knn_balanced)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_knn_balanced, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

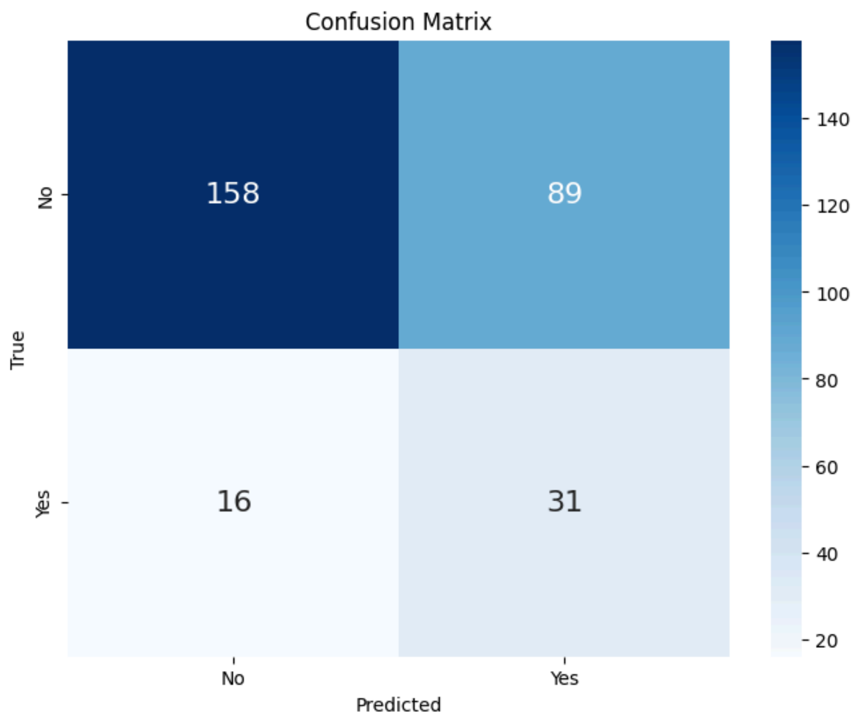
```
Accuracy Score:
0.6122448979591837
              precision    recall  f1-score   support

           0       0.91      0.60      0.72       247
           1       0.24      0.68      0.36        47

    accuracy                           0.61       294
   macro avg       0.58      0.64      0.54       294
weighted avg       0.80      0.61      0.66       294

Confusion Matrix:
[[148  99]
 [ 15  32]]
```



Now performing **GridSearchCV on th KNN model with balanced data.**

```
# Define the parameter grid
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': [10, 20, 30, 40, 50]
}

# Initialize the KNN classifier
clf_knn_balanced_cv = KNeighborsClassifier()

# Initialize GridSearchCV with the classifier and parameter grid
grid_search = GridSearchCV(estimator=clf_knn_balanced_cv, param_grid=param_grid_knn,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=1)

# Fit GridSearchCV to the training data
grid_search.fit(X_train_balanced, y_train_balanced)

# Get the best estimator
best_clf_knn_balanced_cv = grid_search.best_estimator_

# Make predictions with the best estimator
y_pred_knn_balanced_cv = best_clf_knn_balanced_cv.predict(X_test)

# Evaluate the best estimator
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred_knn_balanced_cv))
print("\nClassification Report:")
print(classification_report(y_test, y_pred_knn_balanced_cv))

print("\nConfusion Matrix:")
cm_knn_balanced_cv = confusion_matrix(y_test, y_pred_knn_balanced_cv)
print(cm_knn_balanced_cv)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_knn_balanced_cv, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

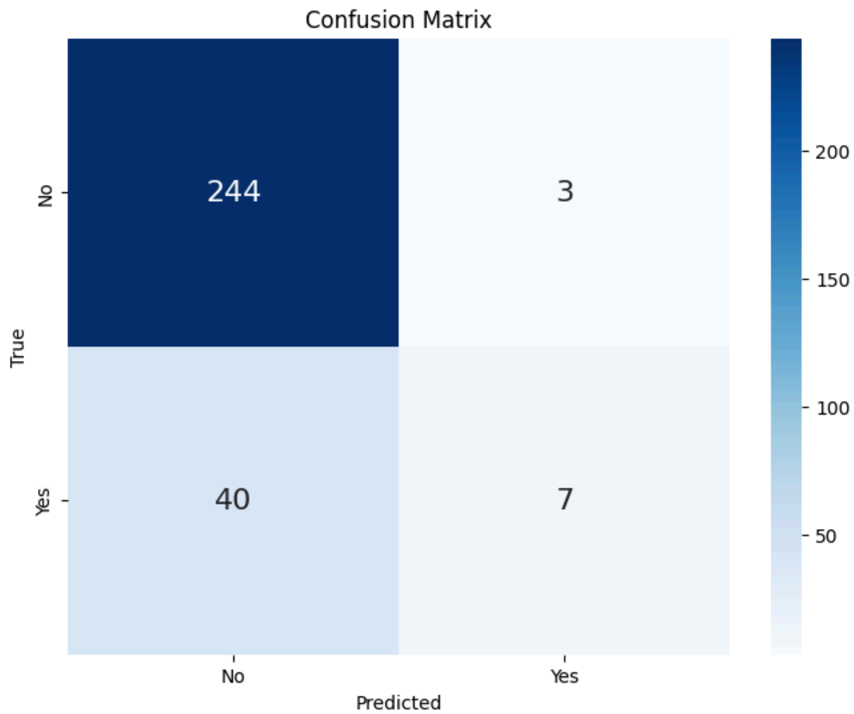
```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits

Accuracy Score:
0.6428571428571429

Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.64      0.75       247
           1       0.26      0.66      0.37        47

    accuracy                           0.64       294
   macro avg       0.58      0.65      0.56       294
weighted avg       0.80      0.64      0.69       294


Confusion Matrix:
[[158  89]
 [ 16  31]]
```



Now let's build a **KNN model based on the data after we dropeed feature importances.**

```
# Initialize and train the KNN classifier
clf_knn_fi = KNeighborsClassifier()
clf_knn_fi.fit(X_train_fi, y_train_fi)

# Make predictions and evaluate the model
y_pred_knn_fi = clf_knn_fi.predict(X_test_fi)
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_knn_fi))
print(classification_report(y_test_fi, y_pred_knn_fi))

# Print confusion matrix
print("Confusion Matrix:")
cm_knn_fi = confusion_matrix(y_test_fi, y_pred_knn_fi)
print(cm_knn_fi)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_knn_fi, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

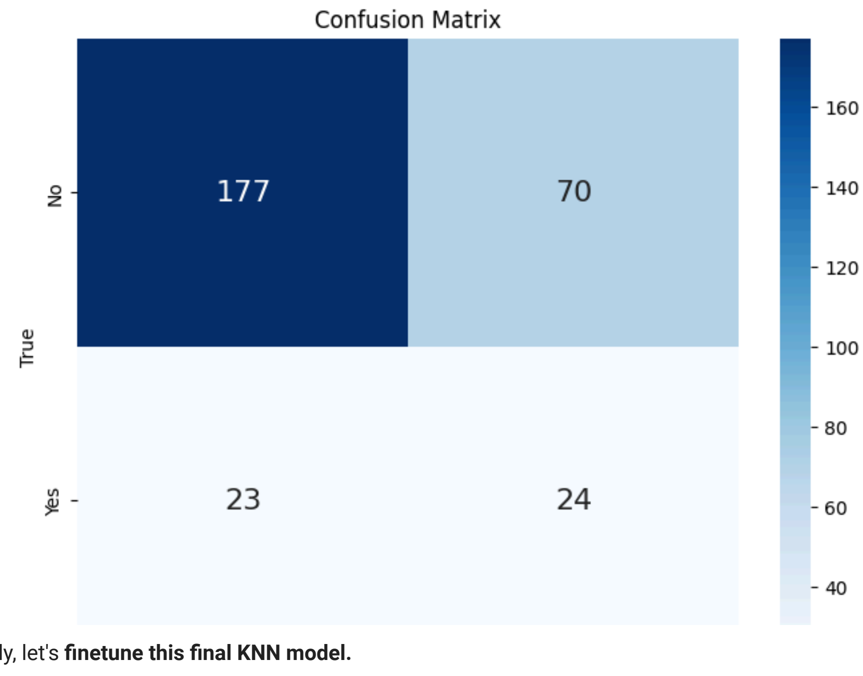
```
Accuracy Score:
0.8537414965986394
              precision    recall  f1-score   support

           0       0.86      0.99      0.92       247
           1       0.70      0.15      0.25        47

    accuracy                           0.85       294
   macro avg       0.78      0.57      0.58       294
weighted avg       0.83      0.85      0.81       294

Confusion Matrix:
[[244   3]
 [ 40   7]]
```



Now we can build a **KNN model with the balanced data after dropping unimportant features.**

```
# Initialize and train the KNN classifier
clf_knn_fi_balanced = KNeighborsClassifier()
clf_knn_fi_balanced.fit(X_train_balanced_fi, y_train_balanced_fi)

# Make predictions and evaluate the model
y_pred_knn_fi_balanced = clf_knn_fi_balanced.predict(X_test_fi)
print("\nAccuracy Score:")
print(accuracy_score(y_test_fi, y_pred_knn_fi_balanced))
print(classification_report(y_test_fi, y_pred_knn_fi_balanced))

# Print confusion matrix
print("Confusion Matrix:")
cm_knn_fi_balanced = confusion_matrix(y_test_fi, y_pred_knn_fi_balanced)
print(cm_knn_fi_balanced)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm_knn_fi_balanced, annot=True, cmap='Blues', fmt='g', annot_kws={"size": 16},
            xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy Score:
0.6836734693877551
              precision    recall  f1-score   support

           0       0.89      0.72      0.79       247
           1       0.26      0.51      0.34        47

    accuracy                           0.68       294
   macro avg       0.57      0.61      0.57       294
weighted avg       0.78      0.68      0.72       294

Confusion Matrix:
[[177  70]
 [ 23  24]]
```



Lastly, let's **finetune this final KNN model.**

```
# Define the parameter grid
param_grid_knn_fi = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': [10, 20, 30, 40, 50]
}

# Initialize the KNN classifier
clf_knn_fi_balanced_cv = KNeighborsClassifier()

# Initialize GridSearchCV with the classifier and parameter grid
grid_search = GridSearchCV(estimator=clf_knn_fi_balanced_cv, param_grid=param_grid_knn_fi,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=1)

# Fit GridSearchCV to the training data
grid_search.fit(X_train_balanced_fi, y_train_balanced_fi)

# Get the best estimator
best_clf_knn_fi_balanced_cv = grid_search.best_estimator_

# Make predictions with the best estimator
```