-

IBM MACHINE LEARNING

# SUPERVISED MACHINE LEARNING: CLASSIFICATION MODELS FOR EMPLOYEE ATTRITION

**Fatima, Sayeda**

**8/17/2022**

# Table of Contents

## 1) Project Overview

A fundamental issue facing organisations is attraction and retention of best talent. Given the cost of retraining new employees, it is important for a business to prevent loss of good talent. Hence, identification of key factors driving employee churning or turnover is important for the organization's Human Resource (HR) Department.

It is here that machine Learning models can be very useful to gain deeper insight into underlying factors and their relationship in driving employee turnover.

Hence, the main aim of the following machine learning modelling and analysis is to enable the business to:

* To identify different factors predict employee churn

* To gain insight into factors contributing to employee churning

* To enable the business  maximize employee attrition

## 2) About the Dataset

### 2a) Brief description of the data set you chose:

This project uses a hypothetical dataset 'IBM HR Analytics Employee Attrition & Performance' which was downloaded from the following link:

https://www.kaggle.com/datasets/pavansubhasht/ibm-hr-analytics-attrition-dataset?resource=download

### 2b) Summary of Data Attributes

The dataset exhibits 1,470 data points (rows) and 35 features (columns) reflecting on employees' background and characteristics and can be downloaded from the following link:

The data also comes with ‘Attrition’ Column to show current employees and leavers which represents the Class we are trying to predict.

### 2c) Main Objectives of Analysis

Organizational performance is largely dependent on its employees, their quality and experience. Hence, organizations are continuously faced with the challenge to reduce employee attrition and increase retention. Consequently, this analysis is targeted towards answering the following queries

- What are the various factors contributory to employee attrition?
- Which business units face higher employee attrition rate?

As a consequence, implementation of the model will enable the organization to:

- devise suitable measures to increase employee retention
- to save valuable resources in retraining new employees hired in place of leavers

## 3) Data Exploration, Data Cleansing and Features Engineering

Since the quality of any machine learning model highly depends on quality of data, hence, this stage is not only most important but is also time consuming. Hence, it was conducted in a step-by-step process.

### 3a) Data Exploration:

- Data was first loaded into pandas dataframe

**Load & Read Dataset**

```
# Load the dataset
url = ("C:/Users/fatima.s/Documents/PythonScripts/DATA SCIENCE/IBM Machine Learning Intermediate/MODULE 3 SUPERVISED MACHINE
df = pd.read_csv(url, index_col=False) # keep_default_na = False # na_filter=False,
df.head(100)
```

| | Age | Attrition | BusinessTravel | DailyRate | Department | DistanceFromHome | Education | EducationField | EmployeeCount | EmployeeNumber | ... | Relationship |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 41 | Yes | Travel_Rarely | 1102 | Sales | 1 | 2 | Life Sciences | 1 | 1 | ... | |
| 1 | 49 | No | Travel_Frequently | 279 | Research & Development | 8 | 1 | Life Sciences | 1 | 2 | ... | |
| 2 | 37 | Yes | Travel_Rarely | 1373 | Research & Development | 2 | 2 | Other | 1 | 4 | ... | |

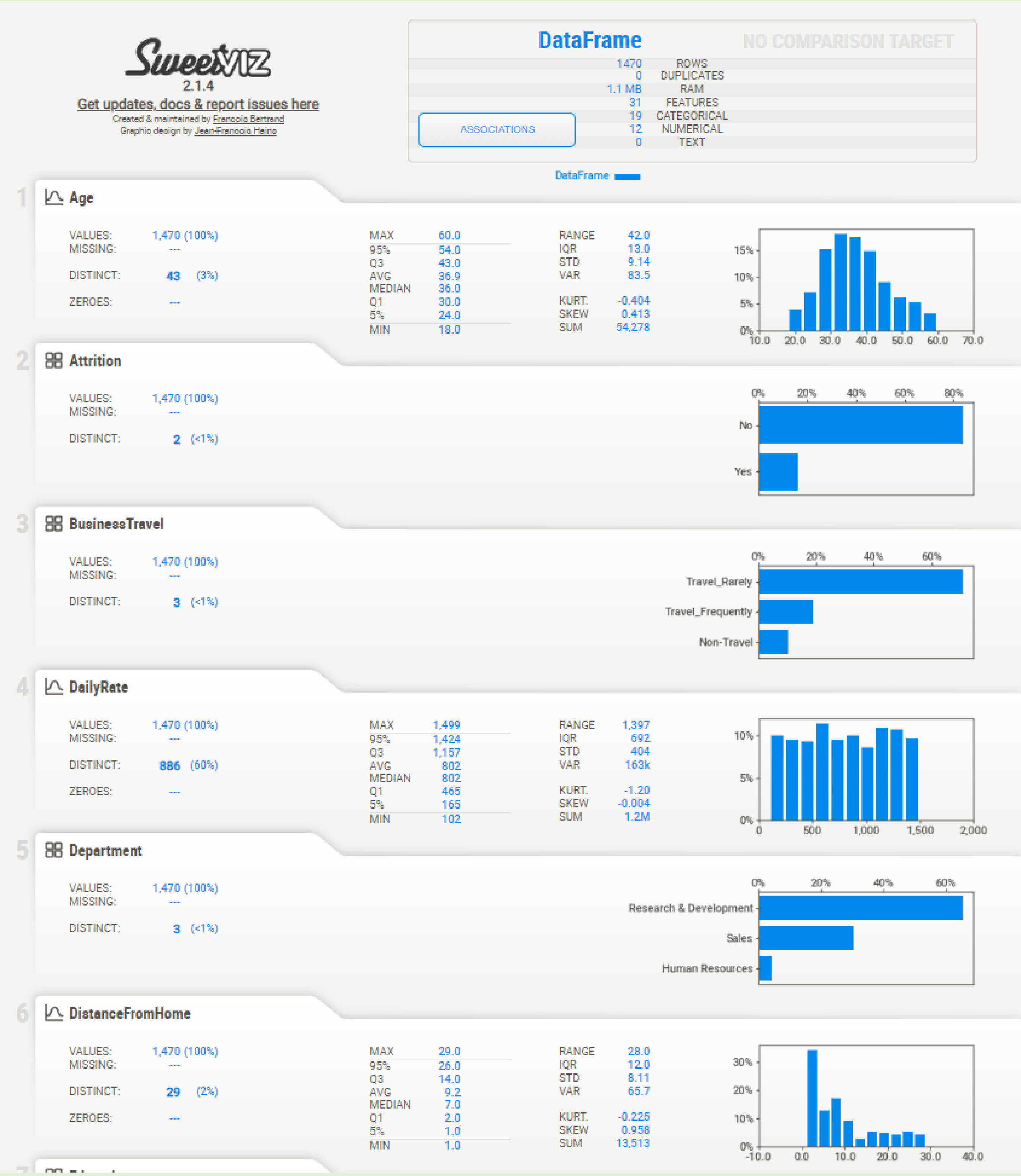
- Column types were explored

**Check data set column types**

```
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
```

- Automated Exploratory Data Analysis was performed using Sweetviz to check



- Descriptive statistics were computed to summarize shape of a dataset's distribution, its dispersion and central tendency

**Compute Descriptive Statistics:** To summarize shape of a dataset's distribution, its dispersion and central tendency.

```
#To get description of all columns
df.describe(include = 'all')
```

| | Age | Attrition | BusinessTravel | DailyRate | Department | DistanceFromHome | Education | EducationField | EmployeeCount | EmployeeNumber |
|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 1470.000000 | 1470 | 1470 | 1470.000000 | 1470 | 1470.000000 | 1470.000000 | 1470 | 1470.0 | 1470.000000 |
| **unique** | NaN | 2 | 3 | NaN | 3 | NaN | NaN | 6 | NaN | NaN |
| **top** | NaN | No | Travel_Rarely | NaN | Research & Development | NaN | NaN | Life Sciences | NaN | NaN |
| **freq** | NaN | 1233 | 1043 | NaN | 961 | NaN | NaN | 606 | NaN | NaN |
| **mean** | 36.923810 | NaN | NaN | 802.485714 | NaN | 9.192517 | 2.912925 | NaN | 1.0 | 1024.865306 |
| **std** | 9.135373 | NaN | NaN | 403.509100 | NaN | 8.106864 | 1.024165 | NaN | 0.0 | 602.024335 |
| **min** | 18.000000 | NaN | NaN | 102.000000 | NaN | 1.000000 | 1.000000 | NaN | 1.0 | 1.000000 |
| **25%** | 30.000000 | NaN | NaN | 465.000000 | NaN | 2.000000 | 2.000000 | NaN | 1.0 | 491.250000 |
| **50%** | 36.000000 | NaN | NaN | 802.000000 | NaN | 7.000000 | 3.000000 | NaN | 1.0 | 1020.500000 |
| **75%** | 43.000000 | NaN | NaN | 1157.000000 | NaN | 14.000000 | 4.000000 | NaN | 1.0 | 1555.750000 |
| **max** | 60.000000 | NaN | NaN | 1499.000000 | NaN | 29.000000 | 5.000000 | NaN | 1.0 | 2068.000000 |

## 3b) Data Cleansing Actions:

- Empty or nearly empty columns were removed using "drop_thresh" to drop columns if 90% of data was empty

**Drop Columns if 90% data is empty**

drop_thresh = df.shape[0]*.10
df = df.loc[:, df.isin([' ','NULL','Nan',0]).mean() < drop_thresh]
df = df.dropna(thresh=drop_thresh, how='all', axis='columns').copy()
df.info()

```
print(df.isin([' ','NULL','NaN', 0]).mean())
drop_thresh = .90
df = df.loc[:, df.isin([' ','NULL', 'NaN',0]).mean() < drop_thresh]
print(df.isin([' ','NULL','NaN',0]).mean())
```

```
Age                         0.000000
Attrition                   0.000000
BusinessTravel              0.000000
DailyRate                   0.000000
Department                  0.000000
DistanceFromHome            0.000000
Education                   0.000000
EducationField              0.000000
EmployeeCount               0.000000
EmployeeNumber              0.000000
EnvironmentSatisfaction     0.000000
Gender                      0.000000
HourlyRate                  0.000000
JobInvolvement              0.000000
JobLevel                    0.000000
JobRole                     0.000000
JobSatisfaction             0.000000
MaritalStatus               0.000000
MonthlyIncome               0.000000
MonthlyRate                 0.000000
NumCompaniesWorked          0.134014
Over18                      0.000000
OverTime                    0.000000
```

- Duplicates were dropped using pandas "df.drop_duplicates()" method

**Handle Missing Values:** Replace remaining ["None","nan", "NaN", ""] values with Zero

```
df = df.replace(["None","nan", "NaN", ""], "0") # Replace all Nan Values with Zero
null = (df.isin(["None","nan", "NaN", ""]).sum()) # Sum as series
null_df=pd.DataFrame({'cols':null.index, 'sum':null.values}).sort_values(by=['sum'],ascending=False)

print(colored("Data has ", 'green', attrs=['bold'])
      +colored((null_df.at[0,'sum']), 'red', attrs=['bold'])
      +colored(" null values.\n ", 'green', attrs=['bold'])
      +colored(null_df.tail(35), 'red', attrs=['bold'])) # print first two rows
```

```
Data has 0 null values.
                         cols  sum
0                         Age    0
26              StandardHours    0
20         NumCompaniesWorked    0
21                     Over18    0
22                   OverTime    0
23          PercentSalaryHike    0
24          PerformanceRating    0
25   RelationshipSatisfaction    0
27           StockOptionLevel    0
18              MonthlyIncome    0
28          TotalWorkingYears    0
29      TrainingTimesLastYear    0
30            WorkLifeBalance    0
31             YearsAtCompany    0
32         YearsInCurrentRole    0
33    YearsSinceLastPromotion    0
19                MonthlyRate    0
17              MaritalStatus    0
1                   Attrition    0
8               EmployeeCount    0
2              BusinessTravel    0
3                   DailyRate    0
4                  Department    0
5            DistanceFromHome    0
6                   Education    0
7              EducationField    0
9              EmployeeNumber    0
16            JobSatisfaction    0
10    EnvironmentSatisfaction    0
11                     Gender    0
12                 HourlyRate    0
13             JobInvolvement    0
14                   JobLevel    0
15                    JobRole    0
34       YearsWithCurrManager    0
```

- Null values were summed and Data was found to exhibit zero null values. Thus, no filling of null values was required

## 3c) Features Engineering

In machine learning, feature selection is the method to reduce the number of input variables during developing predictive modelling. This reduction in input variables is necessary not only to minimize computational cost of modeling but also to achieve performance improvement of the model.

Among widely practices feature selection approaches include statistical-based feature selection methods which use statistical measures to evaluate relationship between each input variable and the target variable and then select those exhibiting strongest relationship with the latter. While these methods can be both speedy and effective, however, the ultimate choice of statistical measure is largely dependant on data types of both of these variables.

Irrespective of the statistical measure being employed, two dominant feature selection techniques, that is supervised and unsupervised, exist where the former can be further categorized into wrapper, filter and intrinsic techniques. Filter-based feature selection methods employs statistical measures to evaluate correlation between input and output variables so that those exhibiting highest correlations are selected. Statistical measures employed in filter-based feature selection are normally univariate in nature since they evaluate relationship of single input variables one by one with target variable, disregarding their interaction with each other.

Consequently, adopting filter-based feature selection methods, the employee attrition model approached filter engineering in three steps. Firstly, unique values for all columns were computed after which columns with unique values less than 2 were dropped.

**1) Assessing Columns for Feature Selection:**

Get unique counts to determine threshold for dropping columns
Drop Columns from dataframe if uniqueness is less than threshold (eg. 2)

```
unique_counts = pd.DataFrame.from_records([(col, df[col].nunique()) for col in df.columns], # get unique counts
                                columns=['Column_Name', 'Unique']).sort_values(by=['Unique'])
print(colored("\nThis can help us determine threshold for which columns to exclude from Features.\n\n ", 'blue', attrs=['bol
      + colored(type(unique_counts), 'green', attrs=['bold'])
      + colored("\n\n", 'green', attrs=['bold'])
      + colored(unique_counts, 'red', attrs=['bold'])
     )

unique = unique_counts[(unique_counts['Unique'] < 2)] #If threshold is lesss than 2 then
drop_unique = (unique['Column_Name'].tolist()) # list of columns to drop

cols_to_exclude = ['EmployeeNumber']
cols_to_exclude = ['EmployeeNumber'] + drop_unique

print(colored("\n\n ", 'blue', attrs=['bold'])
      + colored(type(unique), 'green', attrs=['bold'])
      + colored("\n", 'green', attrs=['bold'])
      + colored(unique, 'red', attrs=['bold'])

      + colored("\n\nList of columns to drop\n", 'blue', attrs=['bold'])
      + colored(cols_to_exclude , 'red', attrs=['bold'])
     )

#Function to Drop Columns & Convert to Categories
for col in df.columns:
    if col in cols_to_exclude:
        df = df.loc[:, ~df.columns.isin(cols_to_exclude)]
df.info()
```

```
This can help us determine threshold for which columns to exclude from Features.

 <class 'pandas.core.frame.DataFrame'>

          Column_Name  Unique
21             Over18       1
26      StandardHours       1
8       EmployeeCount       1
11             Gender       2
1           Attrition       2
24  PerformanceRating       2
22           OverTime       2
17      MaritalStatus       3
```

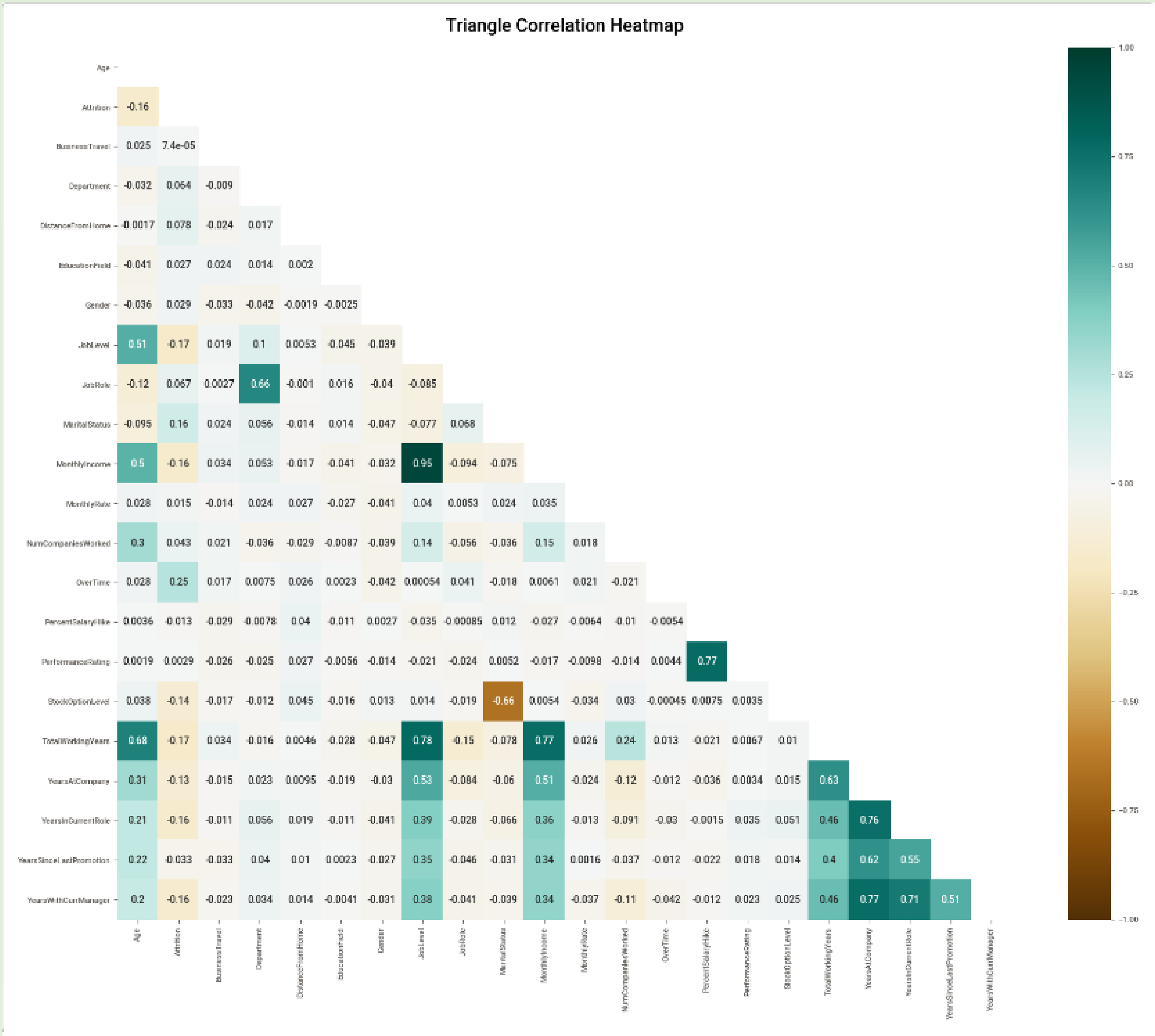
Prior to final features selection Data Encoding of Object or String Columns was carried out to facilitate any statistical computation during features selection process. Hence, after deep copying of original dataset, a function was created and employed to encode object data using Scikit-learn label encoder.

**2) Data Encoding of Object/String Columns:**

* List all Object/String Columns
* Deep copy the original data
* Create Function to employ Scikit-learn label encoding to encode object data
* Create a new dataframe with encoded data description to attach to model outcomes

```
#Function to encode object/string columns

#List all Object/String Columns
from sklearn import preprocessing
cat_columns = df.select_dtypes(include=[object]) # Get Object Type Columns to Convert to Encoded Categories
#cat_columns.info()

categorical_column = list(cat_columns.columns)# List of columns to for label encoding

print(colored("\n\nColumns Requiring Encoding: \n", 'blue', attrs=['bold'])
              + colored(categorical_column, 'green', attrs=['bold']))

#Deep copy the original data
df_encoded = df.copy(deep=True)

# Make Empty Dataframe to decode encoded data later
decode_features = pd.DataFrame()

##### Employ Scikit-Learn label encoding to encode object data #####
lab_enc = preprocessing.LabelEncoder()
for col in categorical_column:
        df_encoded[col] = lab_enc.fit_transform(df[col])
        le_name_mapping = dict(zip(lab_enc.classes_, lab_enc.transform(lab_enc.classes_)))

        ##### Decode Encoded Data #####
        feature_df = pd.DataFrame([le_name_mapping])
        feature_df = feature_df.astype(str)
        print(feature_df)
        feature_df= (col + "_" + feature_df.iloc[0:])
        feature_df["Feature"] = col
        print(feature_df)
        decode_features = decode_features.append(feature_df)# Append Dictionaries to Empty Dataframe for Later Decoding

        ##### Print Encoded Data #####
        print(colored("Feature: \n", 'blue', attrs=['bold'])
              + colored(col, 'red', attrs=['bold'])
              + colored("\nMapping: \n", 'blue', attrs=['bold'])
              + colored(le_name_mapping, 'green', attrs=['bold'])
              + colored("\n\n", 'blue', attrs=['bold'])
              )
df_encoded.head(3)

##### Make Decoded Factor Dataframe with Description #####
#print(decode_features)
factor_list = decode_features.T # Transpose Dataframe and place in new dataframe
factor_list = factor_list.replace(np.nan, "/") # nan values with forward slash
factor_list["Factors"] = factor_list.astype(str).agg("".join,axis=1).replace(r'[^\w\s]|/', '', regex=True) # Aggregate All (
factor_list.reset_index() # Reset index before copying/assigning it to a new column
factor_list['Description'] = factor_list.index  # Assign index to column
```

Statistical measures were then employed with supervised filter-based feature selection technique. Using Pearson's Correlation, the first set of features are selected based on the strength of positive correlation with taget variable 'Attrition'. Additionaly, Pearson's Correlation Matrix was also computed to select feature pairs exhibiting positive correlations with each other.



All feature lists were then combined to filter out dataframe columns not included in the 'final_features' list.

```
df_encoded = df_encoded.filter(final_features)
df_encoded.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1470 entries, 0 to 1469
Data columns (total 22 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Age                      1470 non-null   int64
 1   Attrition                1470 non-null   int32
 2   BusinessTravel           1470 non-null   int32
 3   Department               1470 non-null   int32
 4   DistanceFromHome         1470 non-null   int64
 5   EducationField           1470 non-null   int32
 6   Gender                   1470 non-null   int32
 7   JobLevel                 1470 non-null   int64
 8   JobRole                  1470 non-null   int32
 9   MaritalStatus            1470 non-null   int32
 10  MonthlyIncome            1470 non-null   int64
 11  MonthlyRate              1470 non-null   int64
 12  NumCompaniesWorked       1470 non-null   int64
 13  OverTime                 1470 non-null   int32
 14  PercentSalaryHike        1470 non-null   int64
 15  PerformanceRating        1470 non-null   int64
 16  StockOptionLevel         1470 non-null   int64
 17  TotalWorkingYears        1470 non-null   int64
 18  YearsAtCompany           1470 non-null   int64
 19  YearsInCurrentRole       1470 non-null   int64
 20  YearsSinceLastPromotion  1470 non-null   int64
 21  YearsWithCurrManager     1470 non-null   int64
dtypes: int32(8), int64(14)
memory usage: 218.2 KB
```

# 4) Summary of Training Different Classifier Models

## 4a) Machine Learning Algorithm Approaches

Since EDA revealed a highly imbalanced class distribution, this necessitated achieving appropriate class balancing. While data level and algorithm ensemble approaches do exist for dealing with imbalanced datasets, nevertheless, an automated optimal parameter search method was created to achieve best class reweighting along with isolating other optimal model parameters. This approach was employed because best hyper-parameters are not automatically learnt within estimators and its manual search not only slows down model development but may also lead to ineffective model construction. Hence, exhaustive cv grid search approach was used to pass parameter arguments to the constructor in order to find optimal parameters for each model.

### Logistic Regression (LR) Models

Grid Search Method to Find 'Best Parameters'to 'Build Logistic Regression WITH Best Class Weights'

```python
# Grid Search Method to find Best Hyperparameters for a Logistic Regression Model
def grid_search_lr(X_train, y_train):
    # Parameters
    params_grid = {
    'class_weight': [{0:0.1, 1:0.9}, {0:0.2, 1:0.8}, {0:0.3, 1:0.7}],
    'solver': ['lbfgs', 'saga', 'liblinear', 'newton-cg', 'sag']
    }
    # LR Model
    lr_model = LogisticRegression(random_state=rs, max_iter=1000)

    # Search Best Parameters
    grid_search = GridSearchCV(estimator = lr_model,
                               param_grid = params_grid,
                               scoring='f1',
                               cv = 5, verbose = 1)
    # Train Model with Best Parameters
    grid_search.fit(X_train, y_train)

    # Get Best/optimal parameters
    best_lrparams = grid_search.best_params_
    return best_lrparams
```

Get Optimal Parameters for LR Model using Grid Search LR Method above

```python
final_lrprams = grid_search_lr(X_train, y_train) # From the cell above, Call grid_search_rf(X_train, y_train)

final_lrprams_df = pd.DataFrame([final_lrprams]) # Dictionary To dataframe
print(final_lrprams_df)

# Make Optimal Variables
optimal_lr_class_weight = (final_lrprams_df.at[0,'class_weight'])
optimal_solver = (final_lrprams_df.at[0,'solver'])
print('Optimal LR Class Weights: ', optimal_lr_class_weight)
print('Optimal Solver: ', optimal_solver)

# Define Optimal Parameters
optimal_lr_params = {'class_weight': optimal_lr_class_weight, 'solver': optimal_solver}
print(optimal_lr_params)
```

```
Fitting 5 folds for each of 15 candidates, totalling 75 fits
       class_weight     solver
0  {0: 0.2, 1: 0.8}  newton-cg
Optimal LR Class Weights:  {0: 0.2, 1: 0.8}
Optimal Solver:  newton-cg
{'class_weight': {0: 0.2, 1: 0.8}, 'solver': 'newton-cg'}
```

## Random Forest (RF) Models

### Grid Search Method to Find 'Best Parameters'to 'Build Random Forest WITH Class Weights'

```
# Method for Grid Search Hyperparameters for a Random Forest Model
def grid_search_rf(X_train, y_train):
    # Parameters
    params_grid = {
    'max_depth': [2*n+1 for n in range(10) ], #[5, 10, 15, 20],
    'n_estimators': [2*n+1 for n in range(20)], #[25, 50, 100],
    'min_samples_split': [2, 5],
    'class_weight': [{0:0.1, 1:0.9}, {0:0.2, 1:0.8}, {0:0.3, 1:0.7}]
    }
    # RF Model
    rf_model = RandomForestClassifier(random_state=rs)

    # Search Best Parameters
    grid_search = GridSearchCV(estimator = rf_model,
                               param_grid = params_grid,
                               scoring='f1',
                               cv = 5, verbose = 1)
    # Train Model with Best Parameters
    grid_search.fit(X_train, y_train)

    # Get Best/optimal parameters
    best_params = grid_search.best_params_
    #Best_Score = grid_search.best_score_
    #accuracy = get_accuracy(X_train, X_test, y_train, y_test, grid_search.best_estimator_)
    return best_params
```

### Get `Optimal Parameters` for RF Model using Grid Search RF Method above ¶

```
#Calculate StartTime to Measure Script Execution Time at the End of Script
start_time = datetime.now()

#Get Optimal Parameters for RF Model using "grid_search_rf" Method to Find 'Best/Optimal Parameters'
best_params = grid_search_rf(X_train, y_train) # From the cell above, Call grid_search_rf(X_train, y_train)

best_params_df = pd.DataFrame([best_params]) # Dictionary To dataframe
print(best_params_df)

# Make Optimal Parameter Variables
optimal_class_weight = (best_params_df.at[0,'class_weight'])
print(optimal_class_weight)
optimal_max_depth = (best_params_df.at[0,'max_depth'])
print(optimal_max_depth)
optimal_min_samples_split = (best_params_df.at[0,'min_samples_split'])
print(optimal_min_samples_split)
optimal_n_estimators = (best_params_df.at[0,'n_estimators'])
print(optimal_n_estimators)

# Define Optimal Parameters
optimal_rf_params = {'bootstrap': True,
                     'class_weight': optimal_class_weight,
                     'max_depth': optimal_max_depth,
                     'min_samples_split': optimal_min_samples_split,
                     'n_estimators': optimal_n_estimators}
print(optimal_rf_params)

#Print Total Execution Time
print('Model Execution Time: ', datetime.now() - start_time)
```

```
Fitting 5 folds for each of 1200 candidates, totalling 6000 fits
       class_weight  max_depth  min_samples_split  n_estimators
0  {0: 0.1, 1: 0.9}          7                  5            23
{0: 0.1, 1: 0.9}
7
5
23
{'bootstrap': True, 'class_weight': {0: 0.1, 1: 0.9}, 'max_depth': 7, 'min_samples_split': 5, 'n_estimators': 23}
```

## eXtreme Gradient Boosting (XGB) Model

Grid Search Method to Find 'Best Parameters'to 'Build XGB WITH Best Class Weights'

```
#Calculate StartTime to Measure Model Execution Time at the End
xgb_start_time = datetime.now()

# Method for Grid Search Hyperparameters for a Random Forest Model
def grid_search_xgb(X_train, y_train):
    # Parameters
    params_grid = {
    'max_depth': [5, 10, 15, 20], #[2*n+1 for n in range(10) ],
    'n_estimators': [100, 300, 500],#[2*n+1 for n in range(20)],
    'min_samples_split': [2, 5, 8],
    }
    # RF Model
    xgb_model = GradientBoostingClassifier(random_state=rs)

    # Search Best Parameters
    grid_search = GridSearchCV(estimator = xgb_model,
                           param_grid = params_grid,
                           scoring='f1',
                           cv = 5, verbose = 1)
    # Train Model with Best Parameters
    grid_search.fit(X_train, y_train)

    # Get Best/optimal parameters
    best_params = grid_search.best_params_
    #Best_Score = grid_search.best_score_
    #accuracy = get_accuracy(X_train, X_test, y_train, y_test, grid_search.best_estimator_)
    return best_params
```

Get `Optimal Parameters` for XGB Model using Grid Search XGB Method above

```
xgb_params = grid_search_xgb(X_train, y_train) # From the cell above, Call grid_search_xgb(X_train, y_train)
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
```

```
xgb_params_df = pd.DataFrame([xgb_params]) # Dictionary To dataframe
print(xgb_params_df)

# Make Optimal Parameter Variables
xgb_opt_max_depth = (xgb_params_df.at[0,'max_depth'])
print(xgb_opt_max_depth)
xgb_opt_min_samples_split = (xgb_params_df.at[0,'min_samples_split'])
print(xgb_opt_min_samples_split)
xgb_opt_n_estimators = (xgb_params_df.at[0,'n_estimators'])
print(xgb_opt_n_estimators)

# Define Optimal Parameters
xgb_optimal_params = {#'bootstrap': True,
                         #'class_weight': xgb_opt_class_weight,
                         'max_depth': xgb_opt_max_depth,
                         'min_samples_split': xgb_opt_min_samples_split,
                         'n_estimators': xgb_opt_n_estimators}
print(xgb_optimal_params)
```

```
   max_depth  min_samples_split  n_estimators
0          5                  2           500
5
2
500
{'max_depth': 5, 'min_samples_split': 2, 'n_estimators': 500}
```

Additionally, the following approaches were also combined with optimal parameters to find a model with best scores:

**I) Data Level Approaches:**

**i) Synthetic Minority Over-sampling Technique (SMOTE):**

Due to highly imbalance class distribution, employee data contains very few instances of minority class for any classification model to explicitly learn decision boundary. A popular approach to tackle this problem is oversampling minority class examples which are close in the feature space using SMOTE. This approach allowed us to achieve a balanced class distribution.

**ii) Random under-sampling:**

Using random under-sampling examples from majority class were deleted to achieve a balanced class distribution.

**iii) Random over-sampling:**

Random oversampling was employed to duplicate examples from minority class to achieve a balanced class distribution.

**II) Algorithm Ensemble Approach:**

**i) Boosting:**

Using boosting, a sequential aggregate of base classifier was created on weighted versions of training data set which focused on misclassified samples at every stage of creating new classifiers based on sample weights that were altered as per classifier's performance. Boosting was achieved using XGB Classifier.

## 4b) Summarizing Employed Models

Following three main classifier models have been used to predict employee attrition.

### 1) Logistic Regression (LR) Models:

Due to the binary nature of predictive variable 'y' where the output can only result in whether the employees will fall in attrition class or not, logistic regression classification algorithm was employed. A single method was created to measure predictive capability of the following two LR models:

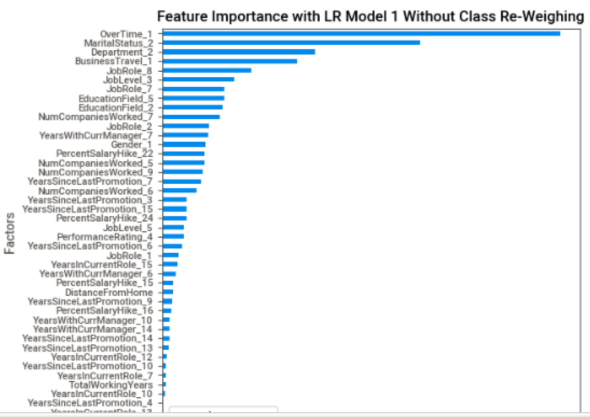
Method to 'Build Logistic Regression WITH & Without Optimal Class Weights'

```python
# Build a logistic regression model with Optimal Class Weights
def build_op_lr(X_train, y_train, X_test, threshold=0.5, best_params=None):

    model = LogisticRegression(random_state=rs,
                               max_iter = 1000)
    # If best parameters are provided
    if best_params:
        model = LogisticRegression(penalty = '12',
                               random_state=rs,
                               max_iter = 1000,
                               class_weight = optimal_lr_class_weight,
                               solver = optimal_solver
                                   )
    # Train the model
    model.fit(X_train, y_train)
    # If predicted probability is largr than threshold (default value is 0.5), generate a positive label
    predicted_proba = model.predict_proba(X_test)
    yp = (predicted_proba [:,1] >= threshold).astype('int')
    return yp, model
```

**1a) LR Model 1 Without Class Re-Weighing:** A simple algorithm was created without achieving any class balance or hyperparameter tuning.

LR Model 1 Without `Class Re-Weighing`

```python
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
lr_preds, lr_model = build_op_lr(X_train, y_train, X_test, best_params=None) # Call Method to 'Build Logistic Regression WIT
lr_results = evaluate(y_test, lr_preds, eval_type="Log Reg")
lr_results["Factors"] = 'LR1_features_df'
#print(lr_results)

# Get Factors Contributing to Attrition
lr1_features = lr_model.coef_[0]##Logistic regression does not have an attribute for ranking feature so we cannot use featur

# Convert to Series and Plot Features
LR1_features = pd.Series(lr1_features, index=X.columns) # Attach Importance to data before split

# Get Features into Dataframe
LR1_features_df = LR1_features.to_frame()
LR1_features_df.reset_index(inplace=True)# Reset index before copying index to new column
LR1_features_df = LR1_features_df.rename(columns = {'index':'Factors', 0: 'Importance'}).sort_values(by=['Importance'],ascen
LR1_features_df = LR1_features_df[LR1_features_df['Importance'] > 0] # Filter Features with Negative Correlation

# Plot Features
LR1_features_df.set_index('Factors', inplace=True)
LR1_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with LR Model 1 Without Class Re-Weighing')

# Merge Description to Encoded Features/Factors
LR1_features_df = LR1_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
LR1_features_df = pd.merge(LR1_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
LR1_features_df['Description'] = np.where(LR1_features_df['Description'] == '/', LR1_features_df['Factors'], LR1_features_df
LR1_features_df['Description'] = LR1_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
LR1_features_df = LR1_features_df.drop(columns=[col for col in LR1_features_df if col not in final_columns]) # Drop Columns
```

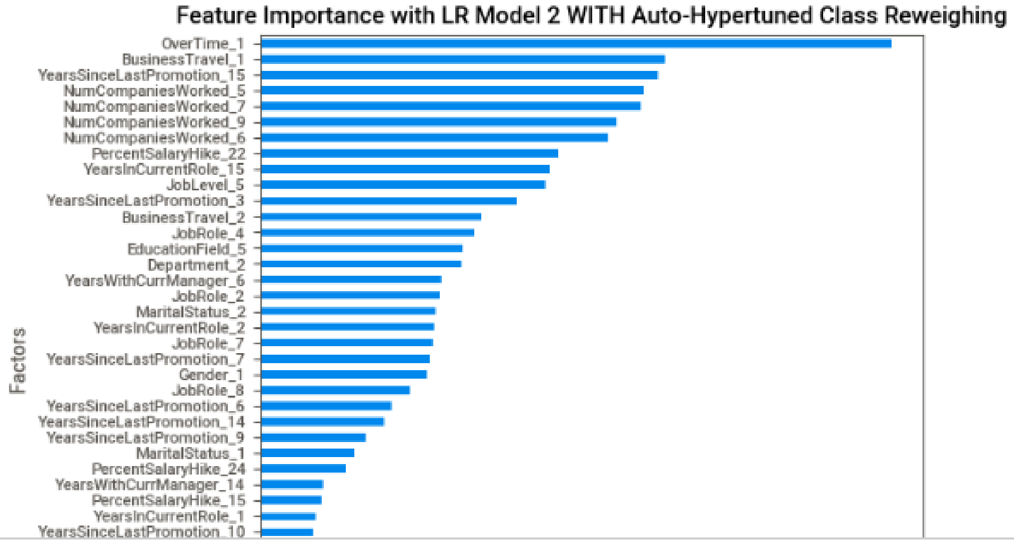


**1b) LR Model 2 WITH Auto-Hypertuned Class Reweighing:** A modified algorithm with auto-hypertuned parameters was created using CV grid search method described above.

LR Model 2 WITH Auto-Hypertuned Class Reweighing

```
#Running the Model...Call Method to 'Build LR With Optimal Class Weights'
op_lr_preds, op_lr_model = build_op_lr(X_train, y_train, X_test, best_params=optimal_lr_params) # Call Method to 'Build Logi
op_lr_results = evaluate(y_test, op_lr_preds, eval_type="LogReg Opt")
op_lr_results["Factors"] = 'LR2_features_df'
print(op_lr_results)

# Get Factors Contributing to Attrition
lr2_features = op_lr_model.coef_[0]##Logistic regression does not have an attribute for ranking feature so we cannot use fea

# Convert to Series and Plot Features
LR2_features = pd.Series(lr2_features, index=X.columns) # Attach Importance to data before split

# Get Features into Dataframe
LR2_features_df = LR2_features.to_frame()
LR2_features_df.reset_index(inplace=True)# Reset index before copying index to new column
LR2_features_df = LR2_features_df.rename(columns = {'index':'Factors', 0: 'Importance'}).sort_values(by=['Importance'],ascen
LR2_features_df = LR2_features_df[LR2_features_df['Importance'] > 0] # Filter Features with Negative Correlation

# Plot Features
LR2_features_df.set_index('Factors', inplace=True)
LR2_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with LR Model 2 WITH Auto-Hypertuned Class Reweighing')

# Merge Description to Encoded Features/Factors
LR2_features_df = LR2_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
LR2_features_df = pd.merge(LR2_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
LR2_features_df['Description'] = np.where(LR2_features_df['Description'] == '/', LR2_features_df['Factors'], LR2_features_df
LR2_features_df['Description'] = LR2_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
LR2_features_df = LR2_features_df.drop(columns=[col for col in LR2_features_df if col not in final_columns]) # Drop Columns
```

```
{'type': 'LogReg Opt', 'accuracy': 0.7891156462585034, 'recall': 0.6808510638297872, 'precision': 0.4050632911392405, 'fscore':
0.6634768740031898, 'auc': 0.7452838315100354, 'Factors': 'LR2_features_df'}
```



## 2) Random Forest Models:

A single method was employed to run random forest models with and without optimal parameters and class weighing:

Method to 'Build Random Forest WITH or WITHOUT Class Weights'

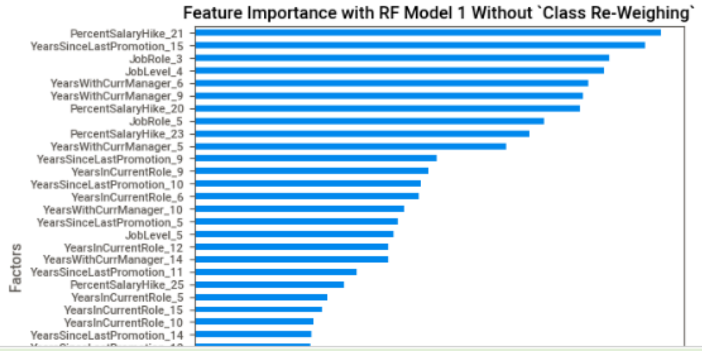
```
def build_op_rf(X_train, y_train, X_test, threshold=0.5, best_params=None):
    model = RandomForestClassifier(random_state = rs)
    # If best parameters are provided
    if best_params:
        model = RandomForestClassifier(random_state = rs,
                                       # If bootstrap sampling is used
                                       bootstrap = True, # False if Bootstaping is being kept as None
                                       # Max depth of each tree
                                       max_depth = optimal_rf_params['max_depth'],
                                       # Class weight parameters
                                       class_weight=optimal_rf_params['class_weight'],
                                       # Number of trees
                                       n_estimators=optimal_rf_params['n_estimators'],
                                       # Minimal samples to split
                                       min_samples_split=optimal_rf_params['min_samples_split'])
    # Train the model
    model.fit(X_train, y_train)
    # If predicted probability is largr than threshold (default value is 0.5), generate a positive label
    predicted_proba = model.predict_proba(X_test)
    yp = (predicted_proba [:,1] >= threshold).astype('int')
    return yp, model
```

**2a) RF Model 1 Without Class Re-Weighing:** A simple algorithm was created without achieving any class balance or hyperparameter tuning.

RF Model 1 Without `Class Re-Weighing`

```
#Running the Model...Call Method to 'Build Simple Random Forest Without Class Weights'
preds, model = build_op_rf(X_train, y_train, X_test, best_params=None) # Call Method to 'Build Simple Random Forest Without 

#Collect Results
original_results = evaluate(y_test, preds, eval_type="RForest")
original_results["Factors"] = 'RF1_features_df'
print(original_results)

# Get Factors Contributing to Attrition in Dataframe
RF1_features_df = feature_importance(X, model)[:40]
RF1_features_df = RF1_features_df[RF1_features_df['Importance'] > 0] # Filter Features with Negative Correlation
RF1_features_df = RF1_features_df.sort_values(by=['Importance'],ascending=True) # Attach Importance to data before split

# Capture Model Drivers in  Dataframe
RF1_drivers = [['Uses train test set to achieve best hypertuninge to resample data from train set to achieve balanced class.
               ['No optimal parameters were used.'],
               ['Runs random forest model with above drivers to get top contributory features.']] # initialize list of lists
RF1_drivers = pd.DataFrame(RF1_drivers, columns=['Main Drivers behind RF1 model are as follows:']) # Create the pandas DataF
RF1_drivers.index = np.arange(1,len(RF1_drivers)+1)

# Plot Features
RF1_features_df.set_index('Factors', inplace=True)
RF1_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with RF Model 1 Without `Class Re-Weighing`')

# Merge Description to Encoded Features/Factors
RF1_features_df = RF1_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
RF1_features_df = pd.merge(RF1_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
RF1_features_df['Description'] = np.where(RF1_features_df['Description'] == '/', RF1_features_df['Factors'], RF1_features_df
RF1_features_df['Description'] = RF1_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
RF1_features_df = RF1_features_df.drop(columns=[col for col in RF1_features_df if col not in final_columns]) # Drop Columns 
```

```
{'type': 'RForest', 'accuracy': 0.8537414965986394, 'recall': 0.1276595744680851, 'precision': 0.75, 'fscore': 0.13186813186813
184, 'auc': 0.5597812042380912, 'Factors': 'RF1_features_df'}
```

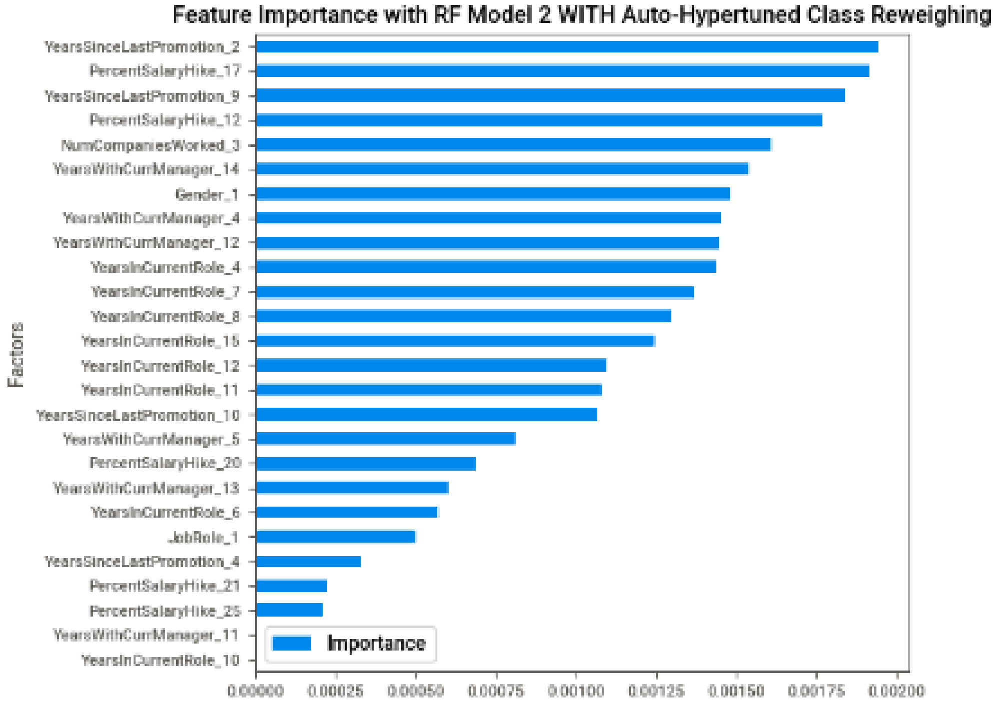


**2b) RF Model 2 WITH Auto-Hypertuned Class Reweighing:** A modified algorithm with auto-hypertuned parameters was created using CV grid search method described above.

RF Model 2 WITH Auto-Hypertuned Class Reweighing

```
#Running the Model...Call Method to 'Build Simple Random Forest With Class Weights'
op_preds, op_model = build_op_rf(X_train, y_train, X_test, best_params=optimal_rf_params) # Call Method to 'Build Simple Ran

#Collect Results
print(original_results)
adjusted_results = evaluate(y_test, op_preds, eval_type="RForest Opt")
adjusted_results["Factors"] = 'RF2_features_df'
print(adjusted_results)

# Get Factors Contributing to Attrition in Dataframe
RF2_features_df = feature_importance(X, op_model)[:40]
RF2_features_df = RF2_features_df[RF2_features_df['Importance'] > 0] # Filter Features with Negative Correlation
RF2_features_df = RF2_features_df.sort_values(by=['Importance'],ascending=True) # Attach Importance to data before split

# Capture Model Drivers in  Dataframe
RF2_drivers = [['Uses train test set to achieve best hypertuninge to resample data from train set to achieve balanced class'
               ['Uses optimal parameters to achieve best hypertuning'],
               ['Runs random forest model with above drivers to get top contributory features']] # initialize list of lists
RF2_drivers = pd.DataFrame(RF2_drivers, columns=['Main Drivers behind RF2 model are as follows:']) # Create the pandas DataF
RF2_drivers.index = np.arange(1,len(RF2_drivers)+1)

# Plot Features
RF2_features_df.set_index('Factors', inplace=True)
RF2_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with RF Model 2 WITH Auto-Hypertuned Class Reweighing')

# Merge Description to Encoded Features/Factors
RF2_features_df = RF2_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
RF2_features_df = pd.merge(RF2_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
RF2_features_df['Description'] = np.where(RF2_features_df['Description'] == '/', RF2_features_df['Factors'], RF2_features_df
RF2_features_df['Description'] = RF2_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
RF2_features_df = RF2_features_df.drop(columns=[col for col in RF2_features_df if col not in final_columns]) # Drop Columns
```

```
{'type': 'RForest', 'accuracy': 0.8537414965986394, 'recall': 0.12765957446808851, 'precision': 0.75, 'fscore': 0.13186813186813
184, 'auc': 0.5597812042380912, 'Factors': 'RF1_features_df'}
{'type': 'RForest Opt', 'accuracy': 0.6972789115646258, 'recall': 0.574468085106383, 'precision': 0.28125, 'fscore': 0.55232100
70810387, 'auc': 0.6475579291928676, 'Factors': 'RF2_features_df'}
```

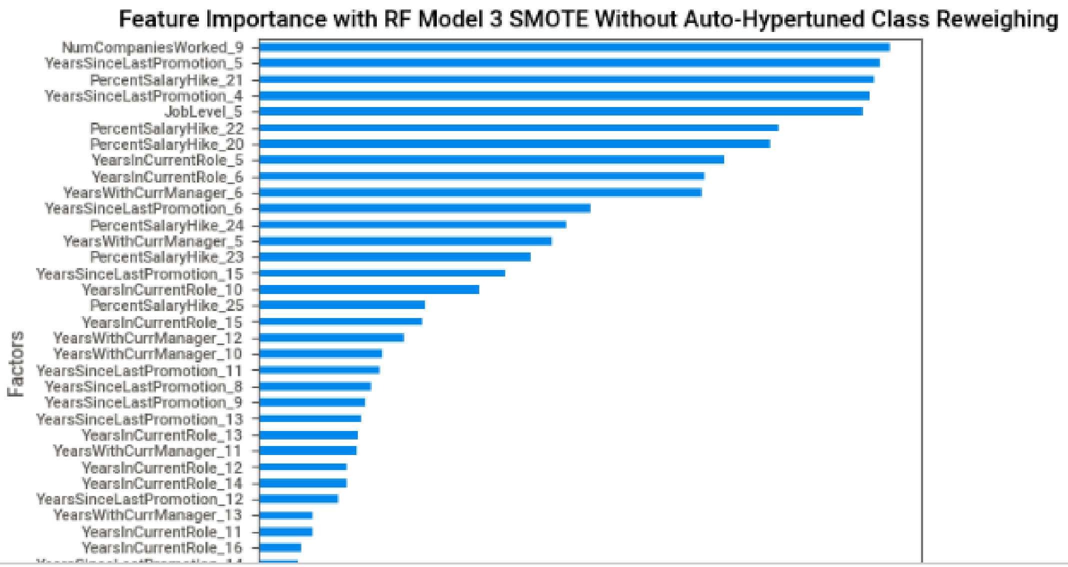


**2c) RF Model 3 SMOTE Without Auto-Hypertuned Class Reweighing:** A simple algorithm was created without achieving any class balance or hyperparameter tuning which employed smote sampling to achieve class balance.

RF Model 3 SMOTE Without Auto-Hypertuned Class Reweighing

```
smote_preds, smo_model = build_op_rf(X_smo, y_smo, X_test, best_params = None)
smote_results = evaluate(y_test, smote_preds, eval_type="RForest Smo")
smote_results["Factors"] = 'RF3_features_df'# Add Important Factors to Dictionary
print(smote_results)

# Get Factors Contributing to Attrition in Dataframe
RF3_features_df = feature_importance(X, smo_model)[:40]
RF3_features_df = RF3_features_df[RF3_features_df['Importance'] > 0] # Filter Features with Negative Correlation
RF3_features_df = RF3_features_df.sort_values(by=['Importance'],ascending=True) # Attach Importance to data before split

# Capture Model Drivers in  Dataframe
RF3_drivers = [['Uses smot parameters to resample data from train set to achieve balanced class'],
               ['Runs random forest model without optimal parameters with above drivers to get top contributory features']]
RF3_drivers = pd.DataFrame(RF3_drivers, columns=['Main Drivers behind RF4 model are as follows:']) # Create the pandas DataF
RF3_drivers.index = np.arange(1,len(RF3_drivers)+1)

# Plot Features
RF3_features_df.set_index('Factors', inplace=True)
RF3_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with RF Model 3 SMOTE Without Auto-Hypertuned Class Reweighing')

# Merge Description to Encoded Features/Factors
RF3_features_df = RF3_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
RF3_features_df = pd.merge(RF3_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
RF3_features_df['Description'] = np.where(RF3_features_df['Description'] == '/', RF3_features_df['Factors'], RF3_features_df
RF3_features_df['Description'] = RF3_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
RF3_features_df = RF3_features_df.drop(columns=[col for col in RF3_features_df if col not in final_columns]) # Drop Columns
```

```
{'type': 'RForest Smo', 'accuracy': 0.8469387755102041, 'recall': 0.23404255319148937, 'precision': 0.55, 'fscore': 0.239330543
93305436, 'auc': 0.5988026531139633, 'Factors': 'RF3_features_df'}
```

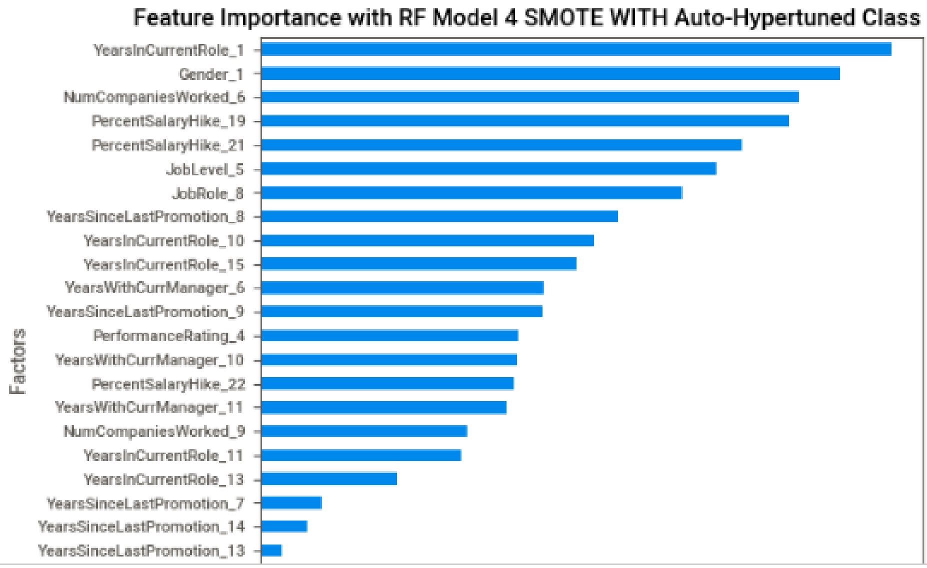


**2d) RF Model 4 SMOTE WITH Auto-Hypertuned Class Reweighing:** A modified algorithm with auto-hypertuned parameters was created using CV grid search method described above which employed smote sampling to achieve class balance.

RF Model 4 SMOTE WITH Auto-Hypertuned Class Reweighing

```
op_smote_preds, op_smo_model = build_op_rf(X_smo, y_smo, X_test, best_params=optimal_rf_params)
op_smote_results = evaluate(y_test, op_smote_preds, eval_type="RForest Opt Smote")
op_smote_results["Factors"] = 'RF4_features_df'# Add Important Factors to Dictionary
print(op_smote_results)

# Get Factors Contributing to Attrition in Dataframe
RF4_features_df = feature_importance(X, op_smo_model)[:40]
RF4_features_df = RF4_features_df[RF4_features_df['Importance'] > 0] # Filter Features with Negative Correlation
RF4_features_df = RF4_features_df.sort_values(by=['Importance'],ascending=True) # Attach Importance to data before split

# Capture Model Drivers in  Dataframe
RF4_drivers = [['Uses smot parameters to achieve best hypertuninge to resample data from train set to achieve balanced class
               ['Uses optimal parameters to achieve best hypertuning'],
               ['Runs random forest model with above drivers to get top contributory features']] # initialize list of lists
RF4_drivers = pd.DataFrame(RF4_drivers, columns=['Main Drivers behind RF4 model are as follows:']) # Create the pandas DataF
RF4_drivers.index = np.arange(1,len(RF4_drivers)+1)

# Plot Features
RF4_features_df.set_index('Factors', inplace=True)
RF4_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with RF Model 4 SMOTE WITH Auto-Hypertuned Class Reweighing')

# Merge Description to Encoded Features/Factors
RF4_features_df = RF4_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
RF4_features_df = pd.merge(RF4_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
RF4_features_df['Description'] = np.where(RF4_features_df['Description'] == '/', RF4_features_df['Factors'], RF4_features_df
RF4_features_df['Description'] = RF4_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
RF4_features_df = RF4_features_df.drop(columns=[col for col in RF4_features_df if col not in final_columns]) # Drop Columns
```

```
{'type': 'RForest Opt Smote', 'accuracy': 0.45918367346938777, 'recall': 0.8085106382978723, 'precision': 0.20212765957446807,
'fscore': 0.7248716067498167, 'auc': 0.6006115944525799, 'Factors': 'RF4_features_df'}
```



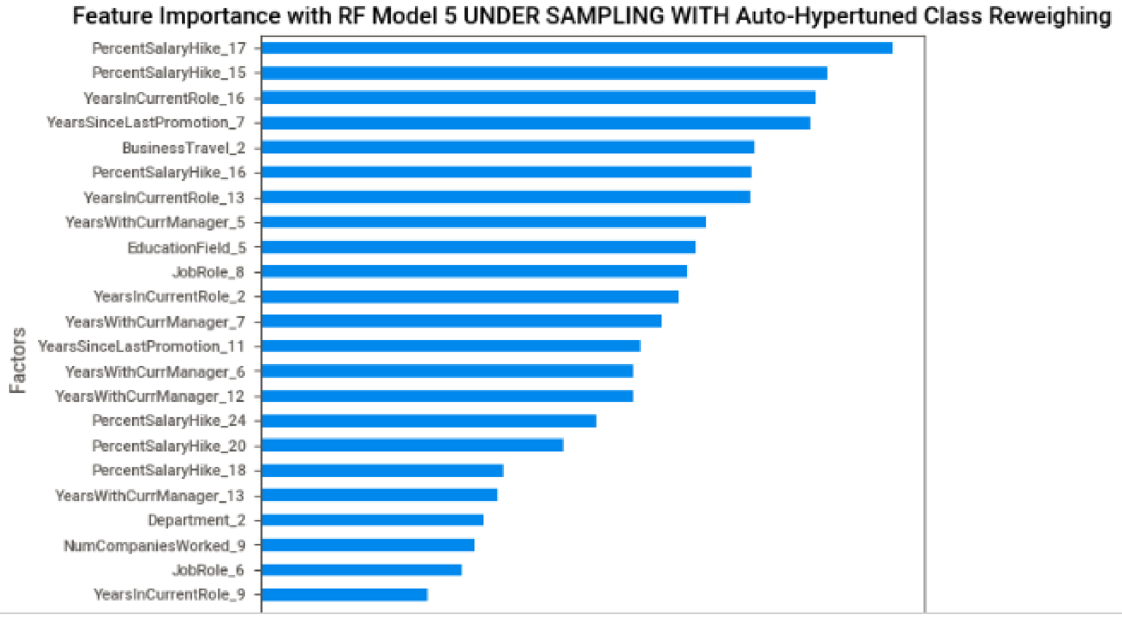
## 2e) RF Model 5 UNDER SAMPLING WITH Auto-Hypertuned Class Reweighing:

A modified algorithm with auto-hypertuned parameters was created using CV grid search method described above which employed random under sampling to achieve class balance.

RF Model 5 UNDER SAMPLING WITH Auto-Hypertuned Class Reweighing

```
under_preds, under_model = build_op_rf(X_under, y_under, X_test, best_params=optimal_rf_params)
under_results = evaluate(y_test, under_preds, eval_type="RForest Opt Under")
under_results["Factors"] = 'RF5_features_df'# Add Important Factors to Dictionary
print(under_results)

# Get Factors Contributing to Attrition in Dataframe
RF5_features_df = feature_importance(X, under_model)[:40]
RF5_features_df = RF5_features_df[RF5_features_df['Importance'] > 0] # Filter Features with Negative Correlation
RF5_features_df = RF5_features_df.sort_values(by=['Importance'],ascending=True) # Attach Importance to data before split

# Capture Model Drivers in  Dataframe
RF5_drivers = [['Uses undersampling to delete examples from majority class in order to achieve balance among classes'],
               ['Uses cv grid method to find optimal parameters to achieve best hypertuning'],
               ['Runs random forest model with above drivers to get top contributory features']] # initialize list of lists
RF5_drivers = pd.DataFrame(RF5_drivers, columns=['Main Drivers behind RF5 model are as follows:']) # Create the pandas DataF
RF5_drivers.index = np.arange(1,len(RF5_drivers)+1)

# Plot Features
RF5_features_df.set_index('Factors', inplace=True)
RF5_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with RF Model 5 UNDER SAMPLING WITH Auto-Hypertuned Class Reweighing')

# Merge Description to Encoded Features/Factors
RF5_features_df = RF5_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
RF5_features_df = pd.merge(RF5_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "/") # Merge
RF5_features_df['Description'] = np.where(RF5_features_df['Description'] == '/', RF5_features_df['Factors'], RF5_features_df
RF5_features_df['Description'] = RF5_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
RF5_features_df = RF5_features_df.drop(columns=[col for col in RF5_features_df if col not in final_columns]) # Drop Columns
```

```
{'type': 'RForest Opt Under', 'accuracy': 0.29931972789115646, 'recall': 0.9148936170212766, 'precision': 0.17551020408163265,
'fscore': 0.7873239436619718, 'auc': 0.5485399259195451, 'Factors': 'RF5_features_df'}
```

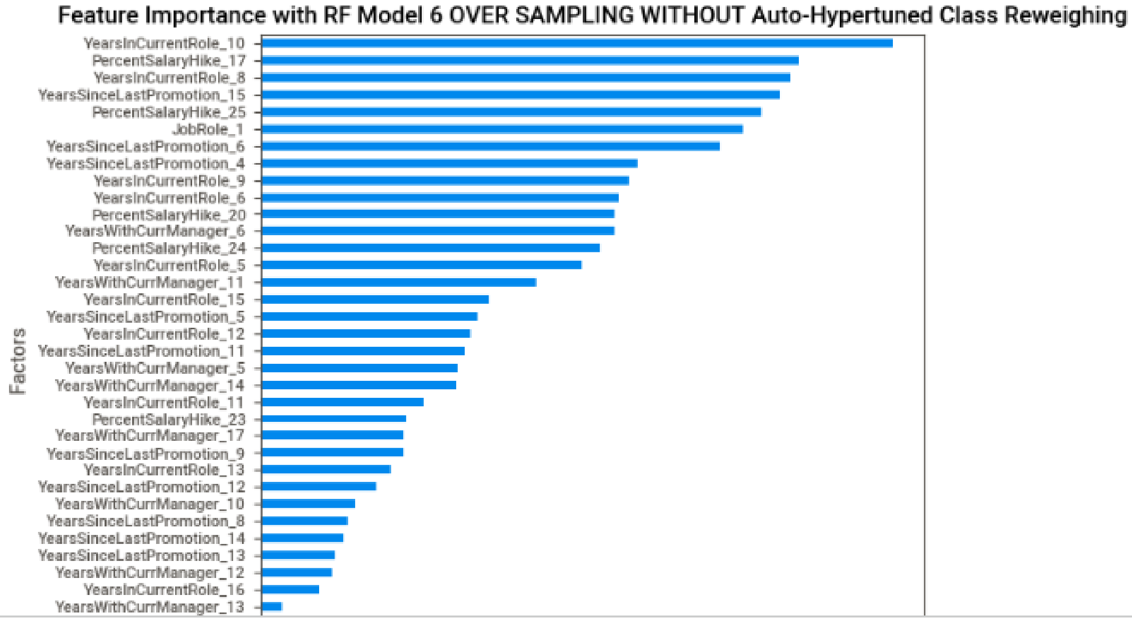


**2f) RF Model 6 OVER SAMPLING WITHOUT Auto-Hypertuned Class Reweighing:** A modified algorithm was created without achieving any class balance or hyperparameter tuning which employed random over sampling to achieve class balance.

RF Model 6 OVER SAMPLING WITHOUT Auto-Hypertuned Class Reweighing

```
over_preds, over_model = build_op_rf(X_over, y_over, X_test, best_params=None)
over_results = evaluate(y_test, over_preds, eval_type="RForest Over")
over_results["Factors"] = 'RF6_features_df'# Add Important Factors to Dictionary
print(over_results)

# Get Factors Contributing to Attrition in Dataframe
RF6_features_df = feature_importance(X, over_model)[:40]
RF6_features_df = RF6_features_df[RF6_features_df['Importance'] > 0] # Filter Features with Negative Correlation
RF6_features_df = RF6_features_df.sort_values(by=['Importance'],ascending=True) # Attach Importance to data before split

# Capture Model Drivers in  Dataframe
RF6_drivers = [['Uses oversampling to randomly selected examples from majority class in order to achieve balance among class
               ['Runs random forest model with above drivers to get top contributory features']] # initialize list of lists
RF6_drivers = pd.DataFrame(RF6_drivers, columns=['Main Drivers behind RF6 model are as follows:']) # Create the pandas DataF
RF6_drivers.index = np.arange(1,len(RF6_drivers)+1)

# Plot Features
RF6_features_df.set_index('Factors', inplace=True)
RF6_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with RF Model 6 OVER SAMPLING WITHOUT Auto-Hypertuned Class Reweighing')

# Merge Description to Encoded Features/Factors
RF6_features_df = RF6_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
RF6_features_df = pd.merge(RF6_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
RF6_features_df['Description'] = np.where(RF6_features_df['Description'] == '/', RF6_features_df['Factors'], RF6_features_df
RF6_features_df['Description'] = RF6_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
RF6_features_df = RF6_features_df.drop(columns=[col for col in RF6_features_df if col not in final_columns]) # Drop Columns
```

```
{'type': 'RForest Over', 'accuracy': 0.6768707482993197, 'recall': 0.723404255319149, 'precision': 0.29310344827586204, 'fscor
e': 0.6847405112316035, 'auc': 0.6957102248255663, 'Factors': 'RF6_features_df'}
```



## 3) XGB Model

A single method was employed to run XGB models with and without optimal parameters and class weighing:

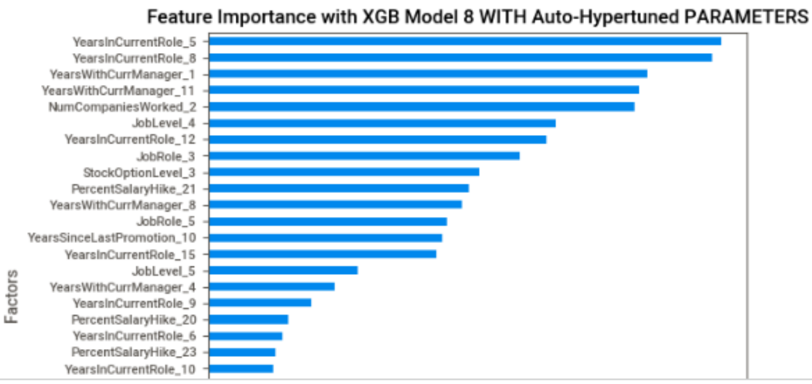
Method to 'Build XGB WITH or WITHOUT Class Weights'

```
def build_xgb(X_train, y_train, X_test, threshold=0.5, best_params=None):
    xgb_model = GradientBoostingClassifier(random_state = rs)
    # If best parameters are provided
    if best_params:
        xgb_model = GradientBoostingClassifier(random_state = rs,
                                   # Max depth of each tree
                                   max_depth = xgb_optimal_params['max_depth'],
                                   # Class weight parameters
                                   #class_weight=optimal_params['class_weight'],
                                   # Number of trees
                                   n_estimators=xgb_optimal_params['n_estimators'],
                                   # Minimal samples to split
                                   min_samples_split=xgb_optimal_params['min_samples_split'])
    # Train the model
    xgb_model.fit(X_train, y_train)
    # If predicted probability is largr than threshold (default value is 0.5), generate a positive label
    predicted_proba = xgb_model.predict_proba(X_test)
    yp = (predicted_proba [:,1] >= threshold).astype('int')
    return yp, xgb_model
```

XGB Model 8 WITH Auto-Hypertuned PARAMETERS

```
xgb_preds, xgb_model = build_xgb(X_train,y_train, X_test, best_params=xgb_optimal_params)
xgb_results = evaluate(y_test, xgb_preds, eval_type="XGB Opt")
xgb_results["Factors"] = 'XGB_features_df'# Add Important Factors to Dictionary
print(xgb_results)

# Get Factors Contributing to Attrition in Dataframe
XGB_features_df = feature_importance(X, xgb_model)[:40]
XGB_features_df = XGB_features_df[XGB_features_df['Importance'] > 0] # Filter Features with Negative Correlation
XGB_features_df = XGB_features_df.sort_values(by=['Importance'],ascending=True) # Attach Importance to data before split

# Capture Model Drivers in  Dataframe
XGB_drivers = [['Uses cv grid method to find optimal parameters to achieve best hypertuning'],
               ['Uses train test and optimal parameters to achieve right balance among classes'],
               ['Runs XGB model with above drivers to get top contributory features']] # initialize list of lists
XGB_drivers = pd.DataFrame(XGB_drivers, columns=['Main Drivers behind RF6 model are as follows:']) # Create the pandas DataF
XGB_drivers.index = np.arange(1,len(XGB_drivers)+1)

# Plot Features
XGB_features_df.set_index('Factors', inplace=True)
XGB_features_df.plot(kind='barh', figsize=(6, 6))
plt.title('Feature Importance with XGB Model 8 WITH Auto-Hypertuned PARAMETERS')

# Merge Description to Encoded Features/Factors
XGB_features_df = XGB_features_df.rename_axis('Factors').reset_index() # Reset index before copying it to column
XGB_features_df = pd.merge(XGB_features_df,factor_list,on='Factors',how='left',indicator=True).replace(np.nan, "") # Merge D
XGB_features_df['Description'] = np.where(XGB_features_df['Description'] == '/', XGB_features_df['Factors'], XGB_features_df
XGB_features_df['Description'] = XGB_features_df['Description'].replace(r'_', ' ', regex=True) # Replace Underscore with sin
final_columns = ['Factors', 'Importance', 'Description'] # List of Columns to keep
XGB_features_df = XGB_features_df.drop(columns=[col for col in XGB_features_df if col not in final_columns]) # Drop Columns

# Get Model Execution Time
print('XGB Model Execution Time: ', datetime.now() - xgb_start_time)
```

```
{'type': 'XGB Opt', 'accuracy': 0.8469387755102041, 'recall': 0.2553191489361702, 'precision': 0.5454545454545454, 'fscore': 0.
2606516290726817, 'auc': 0.6074166594883279, 'Factors': 'XGB_features_df'}
XGB Model Execution Time:  4:02:32.350523
```



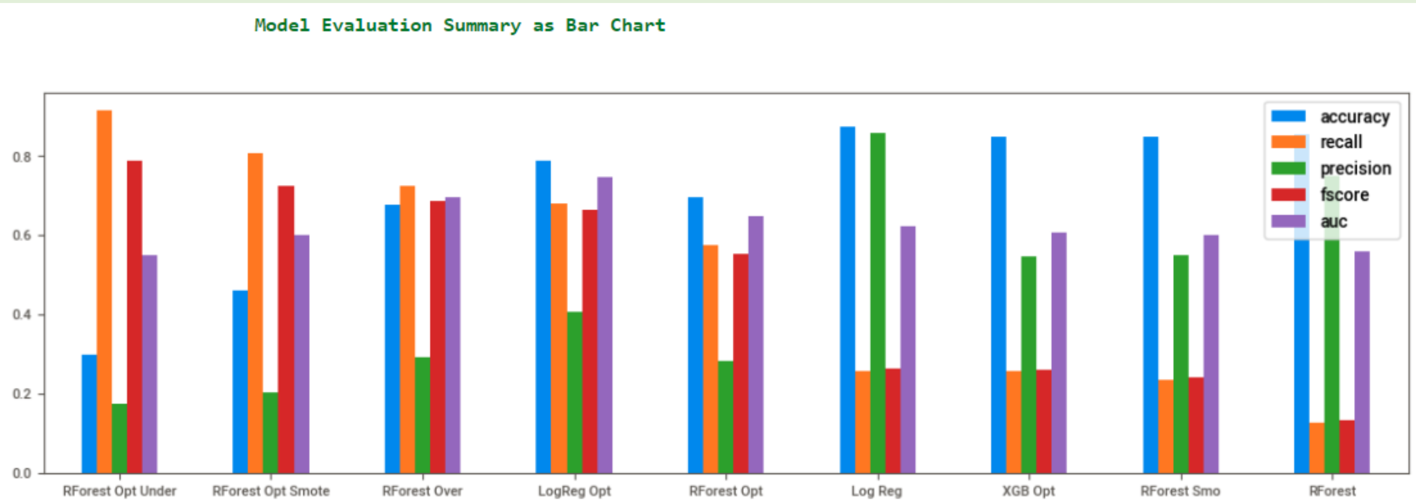
## 5) Recommended Model

### 5a) Result Summary

Result Summary is given below:

| | type | accuracy | recall | precision | fscore |
|---|---|---|---|---|---|
| 0 | RForest Opt Under | 0.299320 | 0.914894 | 0.175510 | 0.787324 |
| 1 | RForest Opt Smote | 0.459184 | 0.808511 | 0.202128 | 0.724872 |
| 2 | RForest Over | 0.676871 | 0.723404 | 0.293103 | 0.684741 |
| 3 | LogReg Opt | 0.789116 | 0.680851 | 0.405063 | 0.663477 |
| 4 | RForest Opt | 0.697279 | 0.574468 | 0.281250 | 0.552321 |
| 5 | Log Reg | 0.874150 | 0.255319 | 0.857143 | 0.262405 |
| 6 | XGB Opt | 0.846939 | 0.255319 | 0.545455 | 0.260652 |
| 7 | RForest Smo | 0.846939 | 0.234043 | 0.550000 | 0.239331 |
| 8 | RForest | 0.853741 | 0.127660 | 0.750000 | 0.131868 |

### 5b) Overall Visual Summary

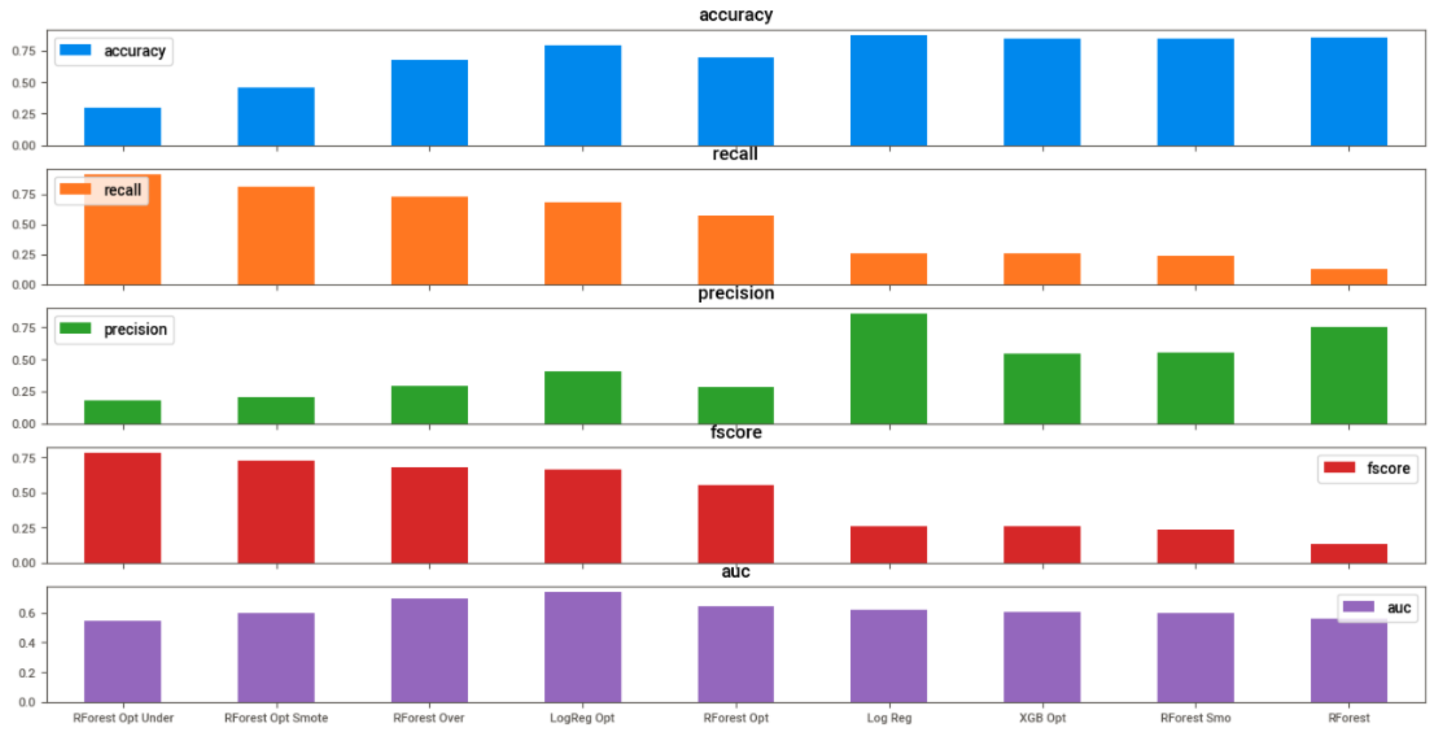


## 5c) Individual Model Visual Summary

**3. Visualize Result Metrics Summary as Individual Bar Charts**

```
axes = chart.plot.bar(rot=0, subplots=True, figsize=(16, 8))
axes[1].legend(loc=2)
```

```
<matplotlib.legend.Legend at 0x2e9ec702e80>
```




## 5d) Model Choice and Justification

For many machine learning tasks with imbalanced datasets, like Employee Attrition, we normally care more about Recall than precision. As a baseline, we want the model to be able to find all possible factors and so, we would allow the model to make false-positive errors because the cost of false positives is usually not very high (maybe it will just cost a false notification email or phone call to confirm with employee).

On the other hand, failing to recognize positive examples (such as employee wanting to leave) can be too costly for the organization. As such, our first priority is to improve model's recall; then we will also want to keep precision as high as possible.

**In this case, the Model with Best Recall and F-Score is RForest Opt Under.** Hence, we will select this model for Employee Attrition Prediction.

## 6) Summary Key Findings and Insights

### 6a) Summarizing Model Drivers:

Main Drivers behind top performing RF5 model are as follows:

- Uses under-sampling to delete examples from majority class in order to achieve balance among classes
- Uses cv grid method to find optimal parameters to achieve best hyper-tuning
- Runs random forest model with above drivers to get top contributory features

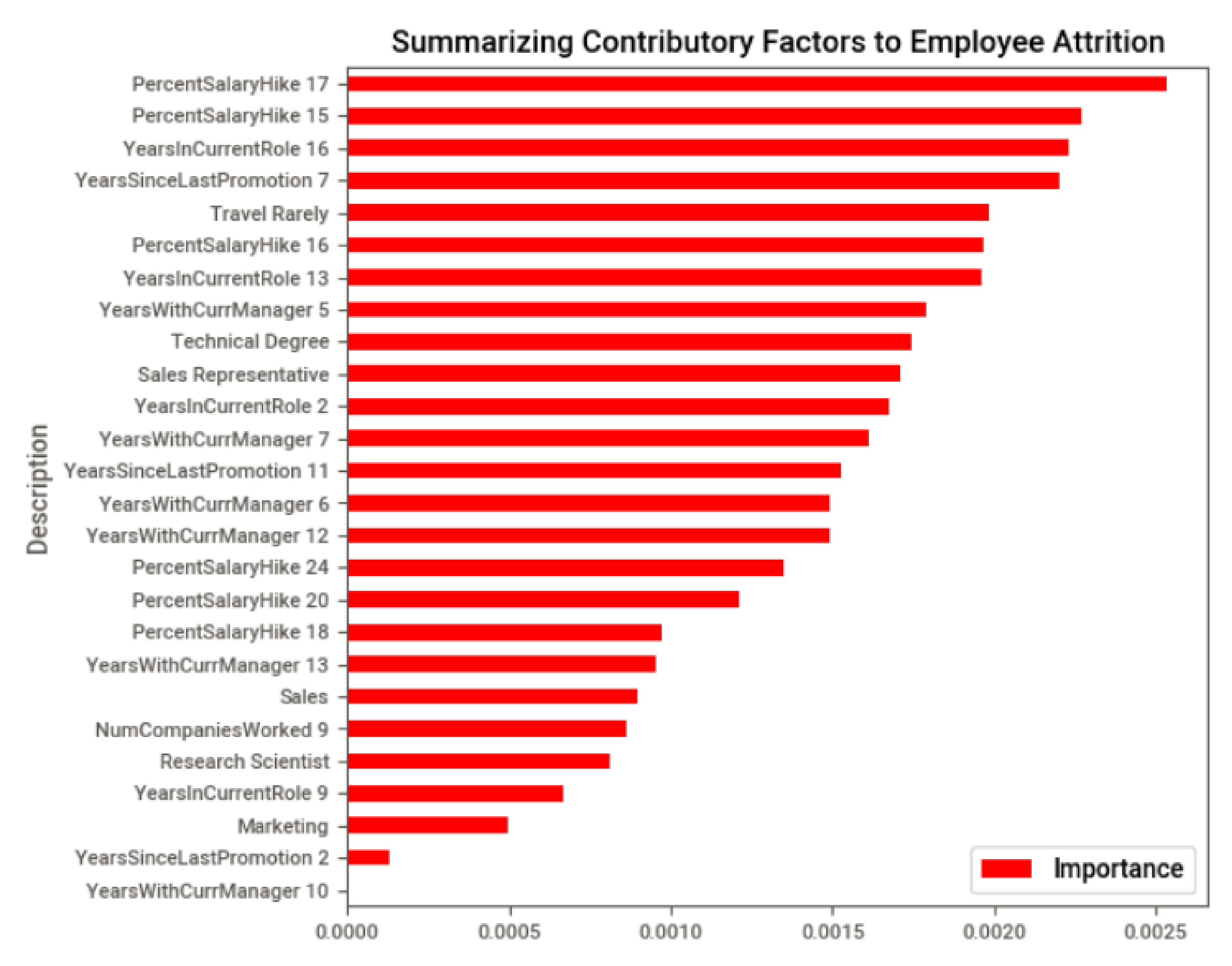
### 6b) Enlisting Top Contributory Factors

Top Factors Contributing to Employee Attrition in ascending order of importance are as follows:

Top Contributory Factors:

| | Factors | Importance | Description |
|---|---|---|---|
| 0 | YearsWithCurrManager_10 | 2.707586e-18 | YearsWithCurrManager 10 |
| 1 | YearsSinceLastPromotion_2 | 1.274117e-04 | YearsSinceLastPromotion 2 |
| 2 | EducationField_2 | 4.953221e-04 | Marketing |
| 3 | YearsInCurrentRole_9 | 6.702776e-04 | YearsInCurrentRole 9 |
| 4 | JobRole_6 | 8.088280e-04 | Research Scientist |
| 5 | NumCompaniesWorked_9 | 8.604488e-04 | NumCompaniesWorked 9 |
| 6 | Department_2 | 8.957229e-04 | Sales |
| 7 | YearsWithCurrManager_13 | 9.515419e-04 | YearsWithCurrManager 13 |
| 8 | PercentSalaryHike_18 | 9.725873e-04 | PercentSalaryHike 18 |
| 9 | PercentSalaryHike_20 | 1.214221e-03 | PercentSalaryHike 20 |
| 10 | PercentSalaryHike_24 | 1.346836e-03 | PercentSalaryHike 24 |
| 11 | YearsWithCurrManager_12 | 1.492372e-03 | YearsWithCurrManager 12 |
| 12 | YearsWithCurrManager_6 | 1.492726e-03 | YearsWithCurrManager 6 |
| 13 | YearsSinceLastPromotion_11 | 1.524371e-03 | YearsSinceLastPromotion 11 |
| 14 | YearsWithCurrManager_7 | 1.609366e-03 | YearsWithCurrManager 7 |
| 15 | YearsInCurrentRole_2 | 1.675148e-03 | YearsInCurrentRole 2 |
| 16 | JobRole_8 | 1.711574e-03 | Sales Representative |
| 17 | EducationField_5 | 1.744438e-03 | Technical Degree |
| 18 | YearsWithCurrManager_5 | 1.788844e-03 | YearsWithCurrManager 5 |
| 19 | YearsInCurrentRole_13 | 1.962220e-03 | YearsInCurrentRole 13 |
| 20 | PercentSalaryHike_16 | 1.969210e-03 | PercentSalaryHike 16 |
| 21 | BusinessTravel_2 | 1.982832e-03 | Travel Rarely |
| 22 | YearsSinceLastPromotion_7 | 2.201833e-03 | YearsSinceLastPromotion 7 |
| 23 | YearsInCurrentRole_16 | 2.228832e-03 | YearsInCurrentRole 16 |
| 24 | PercentSalaryHike_15 | 2.271719e-03 | PercentSalaryHike 15 |
| 25 | PercentSalaryHike_17 | 2.533883e-03 | PercentSalaryHike 17 |

## 6c) Visualizing Top Contributory Factors to Employee Attrition



## 7) Link to Other Useful Models

a) https://github.com/IBM/employee-attrition-aif360/blob/master/notebooks/employee-attrition.ipynb
b) https://github.com/JNYH/employee_attrition/blob/master/employee_attrition.ipynb
c) https://github.com/elastic/examples/tree/master/Machine%20Learning/Analytics%20Jupyter%20Notebooks
d) https://github.com/ganesh10-india/HR_Analytics-Employee_Attrition-Classification-Models/blob/main/HR_Analytics_Employee_Attrition_Classification_Models.ipynb

## 8) Github Link to Assignment Notebook

https://github.com/FATIMASP/IBM-MACHINE-LEARNING-CERTIFICATION/tree/main/Supervised%20Machine%20Learning:%20Classification