



Artificial Intelligence (ACCA7026)

Supervised and Unsupervised Learning

Dr. Seena Joseph

Hands-on Project

Python Demo

Python code snippet

Hands-on Project

Python Demo

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# To scale the data using z-score
from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

# Algorithms to use
from sklearn.linear_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import BaggingClassifier

from sklearn.ensemble import RandomForestClassifier

# Metrics to evaluate the model
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report, precision_recall_curve, recall_score
from sklearn import tree

# For tuning the model
from sklearn.model_selection import GridSearchCV

# To ignore warnings
import warnings
warnings.filterwarnings("ignore")
```

Hands-on Project

Python Demo

Loading the Dataset

```
# Loading the dataset  
df = pd.read_excel('HR_Employee_Attrition_Dataset.xlsx')  
  
df.head()
```

Checking the info of the dataset

```
: # Let us see the info of the data  
df.info()
```

```
| # Checking unique values in each column  
df.nunique()
```

Hands-on Project

Python Demo

Let's drop the columns mentioned above and define lists for numerical and categorical columns to explore them separately.

```
In [6]: # Dropping the columns
df=df.drop(['EmployeeNumber','Over18','StandardHours'],axis=1)
```

```
In [7]: # Creating numerical columns
num_cols=['DailyRate','Age','DistanceFromHome','MonthlyIncome','MonthlyRate','PercentSalaryHike','TotalWorkingYears',
          'YearsAtCompany','NumCompaniesWorked','HourlyRate',
          'YearsInCurrentRole','YearsSinceLastPromotion','YearsWithCurrManager','TrainingTimesLastYear']

# Creating categorical variables
cat_cols=['Attrition','OverTime','BusinessTravel','Department','Education','EducationField','JobSatisfaction','EnvironmentSatisfaction','WorkLifeBalance',
          'StockOptionLevel','Gender','PerformanceRating','JobInvolvement','JobLevel','JobRole','MaritalStatus','RelationshipSatisfaction']
```

Hands-on Project

Python Demo

Exploratory Data Analysis and Data Preprocessing

Univariate analysis of numerical columns

```
# Checking summary statistics  
df[num_cols].describe().T
```

Let's explore these variables in some more depth by observing their distributions

```
# Creating histograms  
df[num_cols].hist(figsize=(14,14))  
plt.show()
```

Hands-on Project

Python Demo

Univariate analysis for categorical variables

```
: # Printing the % sub categories of each category
for i in cat_cols:
    print(df[i].value_counts (normalize=True))
    print('*'*40)
```

Hands-on Project

Python Demo

Bivariate and Multivariate analysis

We have analyzed different categorical and numerical variables. Let's now check how does attrition rate is related with other categorical variables.

```
for i in cat_cols:
    if i != 'Attrition':
        (pd.crosstab(df[i], df['Attrition'], normalize='index') * 100).plot(kind='bar', figsize=(8, 4), stacked=True)
        plt.ylabel('Percentage Attrition %')
```

Let's check the relationship between different numerical variables

```
# Plotting the correlation between numerical variables
plt.figure(figsize=(15, 8))
sns.heatmap(df[num_cols].corr(), annot=True, fmt='0.2f', cmap='YlOrRd')
```

Hands-on Project

Python Demo

Model Building

Data preparation

Creating dummy variables for categorical Variables

```
] : # Creating list of dummy columns
to_get_dummies_for = ['BusinessTravel', 'Department', 'Education', 'EducationField', 'EnvironmentSatisfaction', 'Gender', 'JobInvolvement', 'JobLevel', 'JobRole', 'MaritalStatus' ]

# Creating dummy variables
df = pd.get_dummies(data = df, columns = to_get_dummies_for, drop_first = True)

# Mapping overtime and attrition
dict_Overtime = {'Yes': 1, 'No': 0}
dict_attrition = {'Yes': 1, 'No': 0}

df['OverTime'] = df.OverTime.map(dict_Overtime)
df['Attrition'] = df.Attrition.map(dict_attrition)
```

Hands-on Project

Python Demo

Separating the independent variables (X) and the dependent variable (Y)

```
# Separating target variable and other variables
```

```
Y= df.Attrition  
X= df.drop(columns = ['Attrition'])
```

Splitting the data into 70% train and 30% test set

Some classification problems can exhibit a large imbalance in the distribution of the target classes: for instance there could be several times more negative samples than positive samples. In such cases it is recommended to use the **stratified sampling** technique to ensure that relative class frequencies are approximately preserved in each train and validation fold.

```
# Splitting the data  
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.3, random_state = 1, stratify = Y)
```

Hands-on Project

Python Demo

```
# Scaling the data
sc=StandardScaler()

# Fit_transform on train data
X_train_scaled=sc.fit_transform(X_train)
X_train_scaled=pd.DataFrame(X_train_scaled, columns=X.columns)

# Transform on test data
X_test_scaled=sc.transform(X_test)
X_test_scaled=pd.DataFrame(X_test_scaled, columns=X.columns)
```

Hands-on Project

Python Demo

```
# Creating metric function
def metrics_score(actual, predicted):
    print(classification_report(actual, predicted))

    cm = confusion_matrix(actual, predicted)
    plt.figure(figsize=(8,5))

    sns.heatmap(cm, annot=True,  fmt='.2f', xticklabels=['Not Attrite', 'Attrite'], yticklabels=['Not Attrite', 'Attrite'])
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()
```

Hands-on Project

Python Demo

Decision Tree

- We will build our model using the **DecisionTreeClassifier** function.
- If the frequency of class A is 17% and the frequency of class B is 83%, then class B will become the dominant class and **the decision tree will become biased toward the dominant class.**
- **class_weight** is a **hyperparameter** for the decision tree classifier, and in this case, we can pass a dictionary {0:0.17, 1:0.83} to the model to specify the weight of each class and the decision tree will give more weightage to class 1.

```
In [33]: ▶ # Building decision tree model
dt = DecisionTreeClassifier(class_weight = {0: 0.17, 1: 0.83}, random_state = 1)
```

```
In [34]: ▶ # Fitting decision tree model
dt.fit(X_train, y_train)
```

```
Out[34]: ▼ DecisionTreeClassifier
DecisionTreeClassifier(class_weight={0: 0.17, 1: 0.83}, random_state=1)
```

Hands-on Project

Python Demo

Let's check the model performance of decision tree

```
In [35]: ▶ # Checking performance on the training dataset
y_train_pred_dt = dt.predict(X_train)

metrics_score(y_train, y_train_pred_dt)
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	1726
1	1.00	1.00	1.00	332
accuracy			1.00	2058
macro avg	1.00	1.00	1.00	2058
weighted avg	1.00	1.00	1.00	2058

Hands-on Project

Python Demo

```
In [36]: ▶ # Checking performance on the test dataset
y_test_pred_dt = dt.predict(X_test)

metrics_score(y_test, y_test_pred_dt)
```

	precision	recall	f1-score	support
0	0.96	0.94	0.95	740
1	0.73	0.80	0.77	142
accuracy			0.92	882
macro avg	0.85	0.87	0.86	882
weighted avg	0.92	0.92	0.92	882

Hands-on Project

Python Demo

Observations:

- The Decision Tree works well on the training data but **not so well on the test data** as the recall is 0.80 in comparison to 1 for the training dataset, i.e., the Decision Tree is overfitting the training data.
- The precision on the test data suggests that there's a 27% ($1 - 0.73$) chance that the model will predict that a person is going to leave even though he/she would not, and **the company may waste their time and energy on these employees who are not at risk of attrition.**

Hands-on Project

Python Demo

```
In [39]: ▶ # Plot the feature importance

importances = dt.feature_importances_

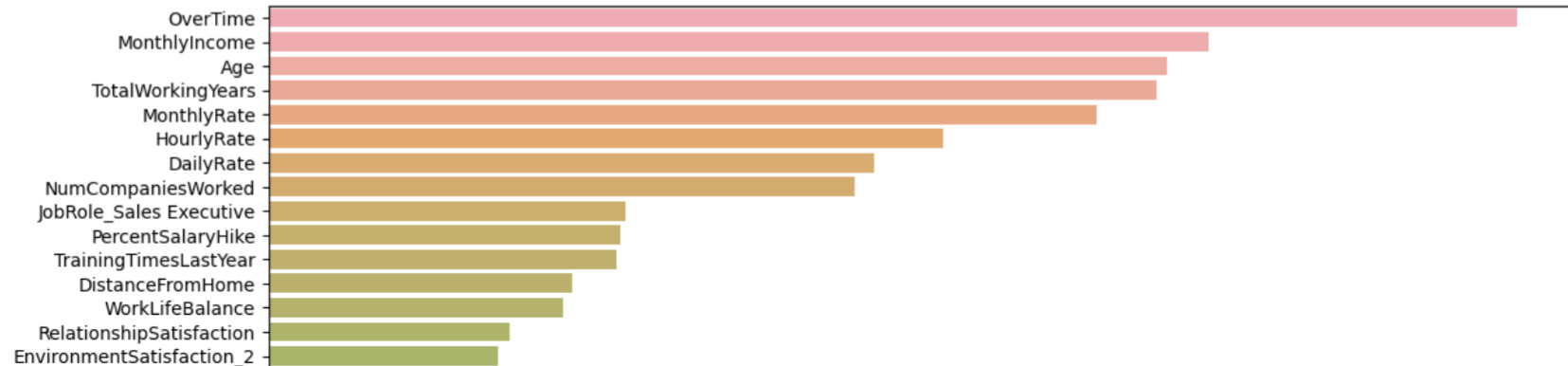
columns = X.columns

importance_df = pd.DataFrame(importances, index = columns, columns = ['Importance']).sort_values(by = 'Importance', ascending

plt.figure(figsize = (13, 13))

sns.barplot(x=importance_df.Importance,y=importance_df.index, alpha=0.8)|
```

Out[39]: <Axes: xlabel='Importance'>



Hands-on Project

Python Demo

Let's plot the tree and check:

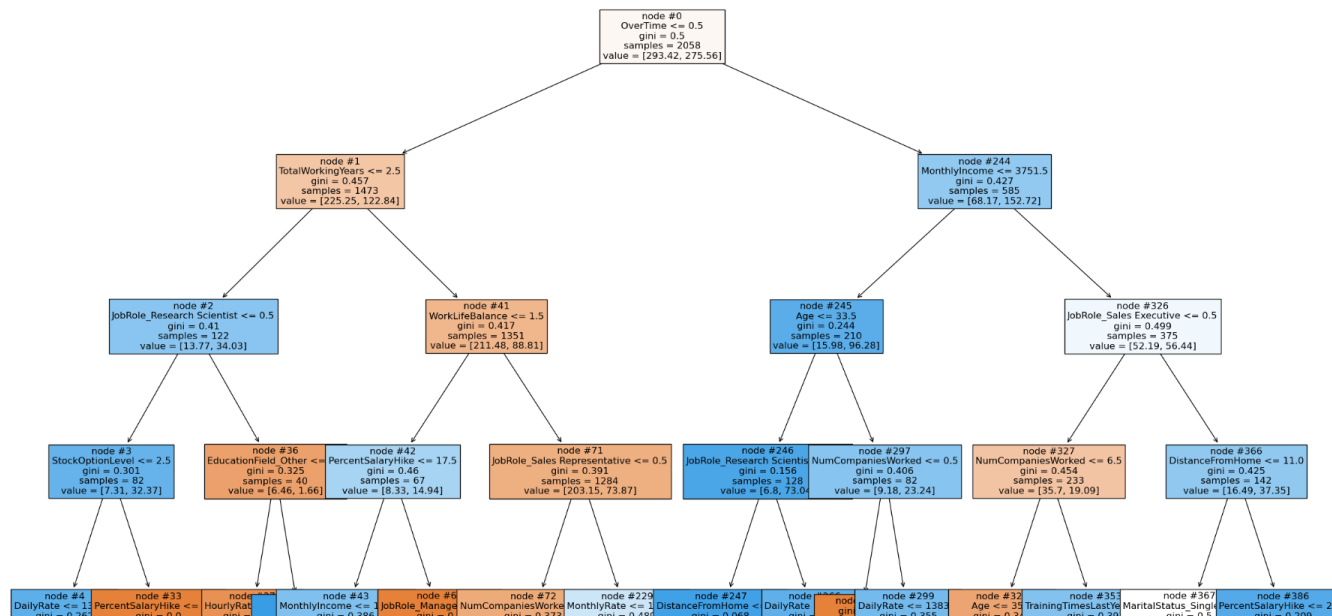
As we know the decision tree keeps growing until the nodes are homogeneous, i.e., it has only one class, and the dataset here has a lot of features, it would be hard to visualize the whole tree with so many features. Therefore, we are only visualizing the tree up to **max_depth = 4**.

```
In [43]: features = list(X.columns)

plt.figure(figsize = (30, 20))

tree.plot_tree(dt, max_depth = 4, feature_names = features, filled = True, fontsize = 12, node_ids = True, class_names = None)

plt.show()
```



Hands-on Project

Python Demo

Parameter Tuning ?

Hands-on Project

Python Demo

Random Forest

- **Random Forest is a bagging algorithm where the base models are Decision Trees.** Samples are taken from the training data and on each sample, a decision tree makes a prediction.
- The results from all the **decision trees** are combined and the final prediction is made using **voting** (for classification problems) or **averaging** (for regression problems).**

```
: # Fitting the Random Forest classifier on the training data
rf_estimator = RandomForestClassifier(class_weight = {0: 0.17, 1: 0.83}, random_state = 1)

rf_estimator.fit(X_train, y_train)

: # Checking performance on the training data
y_pred_train_rf = rf_estimator.predict(X_train)

metrics_score(y_train, y_pred_train_rf)
```

Observation:

- For all the metrics in the training dataset, the Random Forest gives a 100% score.

```
: # Checking performance on the testing data
y_pred_test_rf = rf_estimator.predict(X_test)

metrics_score(y_test, y_pred_test_rf)
```

Observations:

- The Random Forest classifier **seems to be overfitting the training data**. The recall on the training data is 1, while the recall on the test data is only ~ 0.80 for class 1.
- Precision is high for the test data as well.

Hands-on Project

Python Demo

Let's check the feature importance of the Random Forest

```
importances = rf_estimator.feature_importances_  
  
columns = X.columns  
  
importance_df = pd.DataFrame(importances, index = columns, columns = ['Importance']).sort_values(by = 'Importance', ascending = False)  
  
plt.figure(figsize = (13, 13))  
  
sns.barplot(importance_df.Importance, importance_df.index)
```

Observations:

- The Random Forest further verifies the results from the decision tree that the most important features are `MonthlyIncome`, `Age`, `OverTime`.
- We can say that the people appear to be leaving the organization because of the overtime they are doing and this is because they are not paid accordingly. These might be mostly junior-level and mid-level employees with less experience.
- Distance from home is also a key feature, probably as employees living far from the office have to travel a lot, making their schedules hectic.
- Not having stock options is also a driver for attrition - this feature seems to have good importance in both the decision tree and random forest models. This could be related to the junior level employees and their lack of stock options - with the additional burden of a lower salary and working overtime, those without stock options could also be attriting more.
- Other features like, number of companies worked and percent salary hike also seem to be intuitive in explaining attrition likelihood, as people who have worked in a large number of companies are probably not going to stay loyal to the current organization and may have a high risk of attrition, while if an employee is not getting enough of a salary hike, that might demotivate them and lead to a higher likelihood of attriting as well.
- Other features such as job satisfaction, environment satisfaction and their job level also play a crucial role in knowing whether an employee will attrite or not.

Hands-on Project

Python Demo

Types of Hyperparameter Tuning

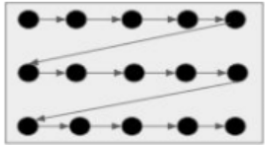
Some models consist of a huge number of hyperparameters, and finding the optimal set of hyperparameters can be a very time-consuming process. To make the process efficient, the most commonly used methods are,

- Grid Search
- Random Search

Grid Search

Grid search is a technique used to find the optimal set of hyperparameters for a model from the provided search space.

Let's understand how grid search works, with an example



- Let the grey box above be set of all possible hyperparameters
- Let these black circles indicate the search space
- Grid search will iterate over all black circles in a sequence
- And finally gives the best set of hyperparameters based on the best score obtained

Grid Search doesn't work well on large search spaces. It will find the best set of hyperparameters but at a high cost. Grid search is best used when we have small search space. We can use a grid search to get the best possible results when we don't have any time constraints, but when we have time constraints, it's better to go with the random search.

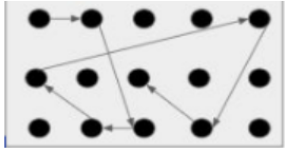
Hands-on Project

Python Demo

Random Search

Random Search is another technique to find the best set of hyperparameters which takes lesser time than grid search

Random search is very similar to grid search, the difference is that in the random search,



Here we will define the number of iterations to search. Not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. Also, the set of hyperparameters is not searched sequentially. So out of the entire search space of hyperparameters, only a certain number of sets of hyperparameters will be checked randomly.

Random Search Works well on large search spaces and gives better results than grid search but it doesn't guarantee to find the best set of hyperparameters.

Hands-on Project

Python Demo

Tuning Models

We will tune Decision Trees and Random Forest models. For tuning, we will only use Grid Search.

Decision Tree

Please refer to the sklearn link to learn more about the hyperparameters and values that the algorithm can take <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>.

Here are some of the parameters,

1) **Criterion**{“gini”, “entropy”}

The function is to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.

2) **max_depth**

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

3) **min_samples_leaf**

The minimum number of samples is required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

Hands-on Project

Python Demo

```
# Choose the type of classifier
dtree_estimator = DecisionTreeClassifier(class_weight = {0: 0.17, 1: 0.83}, random_state = 1)

# Grid of parameters to choose from
parameters = {'max_depth': np.arange(2, 7),
              'criterion': ['gini', 'entropy'],
              'min_samples_leaf': [5, 10, 20, 25]
            }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(recall_score, pos_label = 1)

# Run the grid search
gridCV = GridSearchCV(dtree_estimator, parameters, scoring = scorer, cv = 10)

# Fitting the grid search on the train data
gridCV = gridCV.fit(X_train, y_train)

# Set the classifier to the best combination of parameters
dtree_estimator = gridCV.best_estimator_

# Fit the best estimator to the data
dtree_estimator.fit(X_train, y_train)
```

Hands-on Project

Python Demo

```
# Choose the type of classifier
dtree_estimator = DecisionTreeClassifier(class_weight = {0: 0.17, 1: 0.83}, random_state = 1)

# Grid of parameters to choose from
parameters = {'max_depth': np.arange(2, 7),
              'criterion': ['gini', 'entropy'],
              'min_samples_leaf': [5, 10, 20, 25]
            }

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(recall_score, pos_label = 1)

# Run the grid search
gridCV = GridSearchCV(dtree_estimator, parameters, scoring = scorer, cv = 10)

# Fitting the grid search on the train data
gridCV = gridCV.fit(X_train, y_train)

# Set the classifier to the best combination of parameters
dtree_estimator = gridCV.best_estimator_

# Fit the best estimator to the data
dtree_estimator.fit(X_train, y_train)
```

Hands-on Project

Python Demo

```
DecisionTreeClassifier  
DecisionTreeClassifier(class_weight={0: 0.17, 1: 0.83}, max_depth=2,  
                        min_samples_leaf=5, random_state=1)
```

```
# Checking performance on the training dataset  
y_train_pred_dt = dtree_estimator.predict(X_train)  
  
metrics_score(y_train, y_train_pred_dt)
```

```
# Checking performance on the test dataset  
y_test_pred_dt = dtree_estimator.predict(X_test)  
  
metrics_score(y_test, y_test_pred_dt)
```

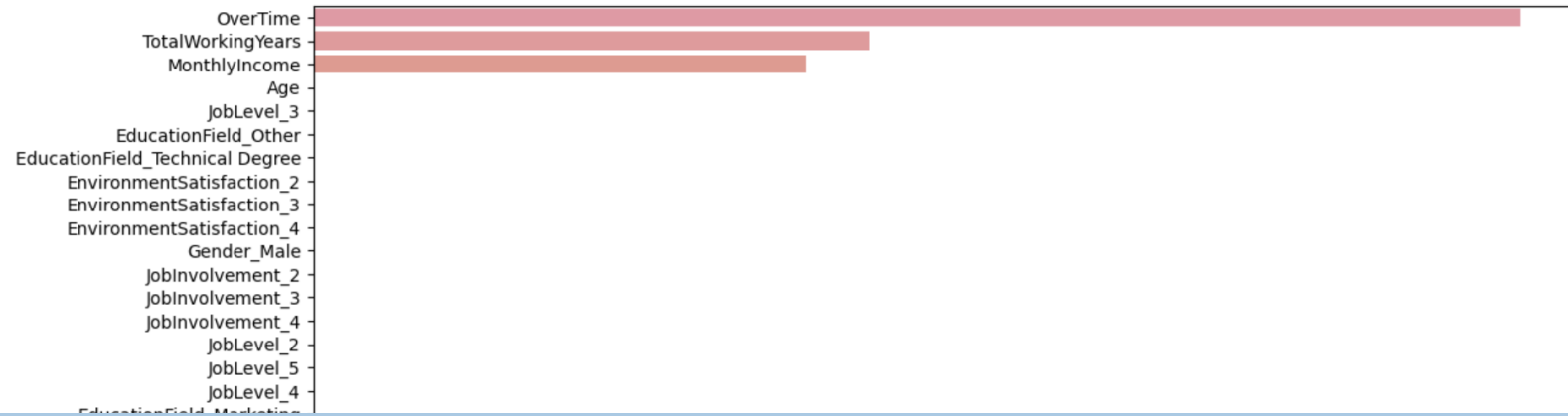
Hands-on Project

Python Demo

Let's look at the feature importance of this model and try to analyze why this is happening.

```
importances = dtree_estimator.feature_importances_  
  
columns = X.columns  
  
importance_df = pd.DataFrame(importances, index = columns, columns = ['Importance']).sort_values(by = 'Importance', ascending = False)  
  
plt.figure(figsize = (13, 13))  
  
sns.barplot(x=importance_df.Importance, y=importance_df.index)
```

Out[50]: <Axes: xlabel='Importance'>



Hands-on Project

Python Demo

Random Forest

Please refer to the sklearn link to learn more about the parameters and values that the algorithm can take <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>.

Here are some of the parameters,

n_estimators: The number of trees in the forest.

min_samples_split: The minimum number of samples required to split an internal node.

min_samples_leaf: The minimum number of samples required to be at a leaf node.

max_features{"auto", "sqrt", "log2", 'None'}: The number of features to consider when looking for the best split.

- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features) (same as "auto").
- If "log2", then max_features=log2(n_features).
- If None, then max_features=n_features.

Hands-on Project

Python Demo

```
# Choose the type of classifier
rf_estimator_tuned = RandomForestClassifier(class_weight = {0: 0.17, 1: 0.83}, random_state = 1)

# Grid of parameters to choose from
params_rf = {
    "n_estimators": [100, 250, 500],
    "min_samples_leaf": np.arange(1, 4, 1),
    "max_features": [0.7, 0.9, 'auto'],
}

# Type of scoring used to compare parameter combinations - recall score for class 1
scorer = metrics.make_scorer(recall_score, pos_label = 1)

# Run the grid search
grid_obj = GridSearchCV(rf_estimator_tuned, params_rf, scoring = scorer, cv = 5)

grid_obj = grid_obj.fit(X_train, y_train)

# Set the classifier to the best combination of parameters
rf_estimator_tuned = grid_obj.best_estimator_

rf_estimator_tuned.fit(X_train, y_train)
```

Hands-on Project

Python Demo

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

▼ RandomForestClassifier

```
RandomForestClassifier(class_weight={0: 0.17, 1: 0.83}, max_features=0.9,  
                        min_samples_leaf=3, n_estimators=250, random_state=1)
```

```
# Checking performance on the training data  
y_pred_train_rf_tuned = rf_estimator_tuned.predict(X_train)  
  
metrics_score(y_train, y_pred_train_rf_tuned)
```

```
# Checking performance on the test data  
y_pred_test_rf_tuned = rf_estimator_tuned.predict(X_test)  
  
metrics_score(y_test, y_pred_test_rf_tuned)
```

Hands-on Project

Python Demo

```
# Plotting feature importance
importances = rf_estimator_tuned.feature_importances_

columns = X.columns

importance_df = pd.DataFrame(importances, index = columns, columns = ['Importance']).sort_values(by = 'Importance', ascending = False)

plt.figure(figsize = (13, 13))

sns.barplot(x=importance_df.Importance, y=importance_df.index)
```

Out[56]: <Axes: xlabel='Importance'>

