Problem Statement :

- Build predictive model to predict the customer is buyer for product ‘ABC’ or not.
- Explain the model performance using different performance metrics?
- Explain the confusion matrix of model?
- Assign probability of purchase to each record.
- Identify which variables are influencing on the purchase of product ‘ABC’?
- Explain trained model using **SHAP** values?
- Identify the group of customer to whom we can reach out to increase sales of product

Hypothesis Generation To Prediction :

Once you have understood the problem statement and gathered the required domain knowledge. The next step comes, the hypothesis generation. This will directly spring from the problem statement. Whatever set of analysis we can think of at this stage we should write it down.

The structured thinking approach will help us here. Let me state some hypotheses from our problem statement.

1. Male customers are more tend to buy vehicle insurance than females.
2. The middle-aged customers would be more interested in the insurance offer.
3. Customers having a driving license are more prone to convert.
4. Those with new vehicles would be more interested in getting insurance.
5. The customers who already have vehicle insurance won’t be interested in getting another.
6. If the Customer got his/her vehicle damaged in the past, they would be more interested in buying insurance.

Implementation of model Prediction in Python :

#Import Libraries

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

#!pip install imblearn

from sklearn.metrics import accuracy_score, f1_score,auc
```

```
from sklearn.linear_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier
```

#Reading the dataset

```
df= pd.read_csv("/Users/rajkumar/Desktop/DSdataset.csv")
df.shap
```

Now, the first step is to look at the top 5 rows in the dataframe. This will give us an initial picture of the data.

```
df.head()
df.info()
```

Here, we will see the basic details of the features in the given dataset. Like the columns, non-null values in each column, and the respective data type.

Now, we will look for any of the missing values in the given dataset.

```
df.isna()
```

Exploratory Data Analysis :

Before jumping into modelling and creating a machine learning-based solution for the given problem, it is important to understand the basic traits of the data.

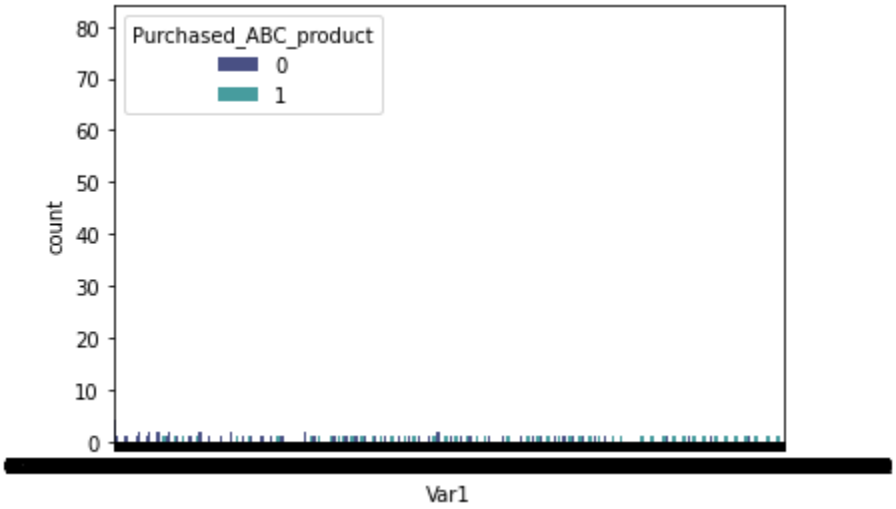
```
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
sns.countplot(ax=axes[0,0],x='Group',hue='Purchased_ABC_product',data=df,palette="mako")
sns.countplot(ax=axes[0,1],x='Category',hue='Purchased_ABC_product',data=df,palette="mako")
sns.countplot(ax=axes[1,0],x='Rating',hue='Purchased_ABC_product',data=df,palette="mako")
sns.countplot(ax=axes[1,1],x='Customer_ID',hue='Purchased_ABC_product',data=df,palette="mako")
sns.countplot(x='Var1',hue='Purchased_ABC_product',data=df,palette="mako")
sns.countplot(x='Var2',hue='Purchased_ABC_product',data=df,palette="mako")
```

It is also important to look at the target column, as it will tell us whether the problem is a balanced problem or an imbalanced problem. This will define our approach further.

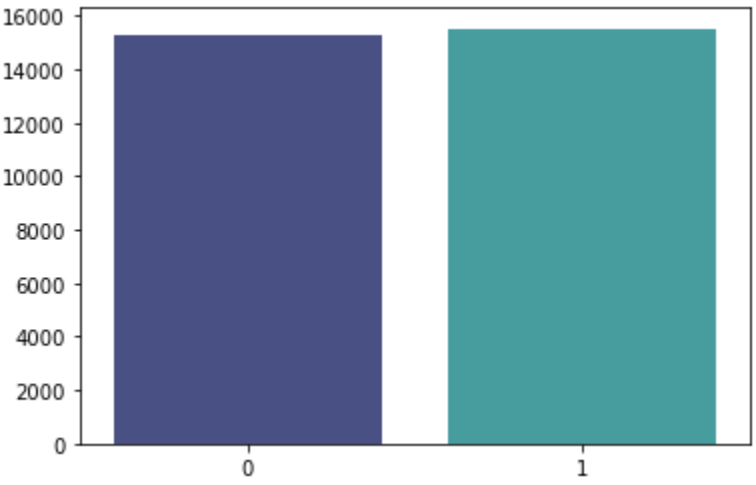
The given problem is an imbalance problem as the Response variable with the value 1 is significantly lower than the value zero.

```
Purchased_ABC_product = df.loc[:,"Purchased_ABC_product"].value_counts().rename('Count')

plt.xlabel("Purchased_ABC_product")

plt.ylabel('Count')
```

sns.barplot(Purchased_ABC_product.index , Purchased_ABC_product.values,palette="mako")



sns.displot(df['Var1'])
sns.distplot(df['Var2'])



Data preprocessing :

The next step in the project is to prepare the data for the modelling. The following preprocessing techniques are being used here

1. Convert the categorical features into dummies or doing categorical encoding.
2. Binning the numerical features.
3. dropping the unnecessary columns like ids.
   Here we a have user-defined function. We just need to pass the raw dataframe and we will get the preprocessed one.

```
def data_prep(df):

    df= df.drop(columns=['Customer_ID','Category','Rating'])

    df=pd.get_dummies(df,columns=['Var1'] ,prefix='Var')

    df=pd.get_dummies(df,columns=['Var2'] ,prefix='Var')
    df["Purchased_ABC_product"] = pd.cut(df['Purchased_ABC_product'], bins=[0, 1])

    df['Purchased_ABC_product']= df['Purchased_ABC_product'].cat.codes
    df.drop(columns=['Region_Code'],inplace= True)
    return df
df1=data_prep(df)
df1.head()
```

Select Feature :

In the following code, we will select those features only we want to use in our model training.

```
Features= ['Customer_ID','Category','Var1','Var2','Rating']
```

Train-Test split :

In the next step, we will split the whole data in our hands into train data and test data.
The train data, as the name suggests will be used for training our machine learning model. On the other hand test, data will be used to make predictions and test the trained model.
Here, I have kept 30% of the total data for testing and the remaining 70% will be used for model training.

```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(df1[Features],df1['Purchased_ABC_product'],
                                test_size = 0.3, random_state = 101)
X_train.shape,X_test.shape
from imblearn.under_sampling import RandomUnderSampler
```

```
RUS = RandomUnderSampler(sampling_strategy=.5,random_state=3,)


X_train,Y_train  = RUS.fit_resample(df1[Features],df1['Purchased_ABC_product'])
```

Handle Imbalance Data Problem :

As from the distribution of target variables in the EDA section, we know it is an imbalance problem. The imbalance datasets could have their own challenge.
For example, a disease prediction model may have an accuracy of 99% but it is of no use if it can not classify a patient successfully.

So to handle such a problem, we can resample the data. In the following code, we will be using undersampling.

Undersampling is the method where we will be reducing the occurrence of the majority class up to a given point.

```
from imblearn.under_sampling import RandomUnderSampler


RUS = RandomUnderSampler(sampling_strategy=.5,random_state=3,)


X_train,Y_train  = RUS.fit_resample(df1[Features],df1['Purchased_ABC_product'])
```

Model training and prediction :

Now, it is time to train a model and make predictions. Here, I have written a user-defined function for measuring the performance of the models.

For performance measurement, we will be using the accuracy score and F1 score. It is important to note here that for imbalanced classification problems, the F1 score is a more significant metric.

```
def performance_met(model,X_train,Y_train,X_test,Y_test):


    acc_train=accuracy_score(Y_train, model.predict(X_train))


    f1_train=f1_score(Y_train, model.predict(X_train))
```

```
acc_test=accuracy_score(Y_test, model.predict(X_test))

f1_test=f1_score(Y_test, model.predict(X_test))

print("train score: accuracy:{} f1:{}".format(acc_train,f1_train))

print("test score: accuracy:{} f1:{}".format(acc_test,f1_test))
```

In this section, first, we will train three models

- Logistic Regression
- Decision Tree
- Random Forest

Logistic Regression :

```
model = LogisticRegression()
model.fit(X_train,Y_train)
performance_met(model,X_train,Y_train,X_test,Y_test)
```

```
train score: accuracy:0.7471847570113466 f1:0.6866392463845031
test score: accuracy:0.7187863521467993 f1:0.41871564940700023
```

Decision Tree :

```
model_DT=DecisionTreeClassifier(random_state=1)
model_DT.fit(X_train,Y_train)
performance_met(model_DT,X_train,Y_train,X_test,Y_test)
```

```
train score: accuracy:0.7578106044387355 f1:0.6603414800136111
test score: accuracy:0.7739410318980522 f1:0.43228045512454427
```

Random forest :

```
Forest= RandomForestClassifier(random_state=1)
Forest.fit(X_train,Y_train)
performance_met(Forest,X_train,Y_train,X_test,Y_test)
```

```
train score: accuracy:0.7578106044387355 f1:0.6604434305839035
test score: accuracy:0.7737573578931717 f1:0.43215595021184106
```

Hyperparameter tuning :

For this project, the last step is to do some hyperparameter tuning. It is a process to find the best performing hyper-parameters.

Here, we will be using a GridSearch algorithm for finding the best parameters of a random forest classifier.

```
rf= RandomForestClassifier(random_state=1)


parameters = {


    'bootstrap': [True],


'max_depth': [20, 25],


'min_samples_leaf': [3, 4],


'min_samples_split': [100,300],


}


grid_search_1 = GridSearchCV(rf, parameters, cv=3, verbose=2, n_jobs=-1)


grid_search_1.fit(X_train, Y_train)


performance_met(grid_search_1,X_train,Y_train,X_test,Y_test)
```

We can see that after using some basic hyperparameter tuning, the f1 score has slightly improved.

```
train score: accuracy:0.7577106972097338 f1:0.660588611644274
test score: accuracy:0.7736873868436934 f1:0.4324786699712675
```

Code to Calculate SHAP Values :

Calculate SHAP values using the wonderful SHAP library.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

data = pd.read_csv("/Users/rajkumar/Desktop/DSdataset.csv")
y = (data['Purchased_ABC_product'] == "Yes")  # Convert from string "Yes"/"No" to binary
feature_names = [i for i in data.columns if data[i].dtype in [np.int64, np.int64]]
X = data[feature_names]
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state=1)
my_model = RandomForestClassifier(random_state=0).fit(train_X, train_y)
```

SHAP values for a single row of the dataset. For context, we'll look at the raw predictions before looking at the SHAP values.

```
row_to_show = 5
data_for_prediction = val_X.iloc[row_to_show]  # use 1 row of data here. Could use multiple rows if desired
data_for_prediction_array = data_for_prediction.values.reshape(1, -1)


my_model.predict_proba(data_for_prediction_array)
```

It is 70%likely to Purchased_ABC_product

Now, we'll move onto the code to get SHAP values for that single prediction.

```
import shap  # package used to calculate Shap values

# Create object that can calculate shap values
explainer = shap.TreeExplainer(my_model)

# Calculate Shap values
shap_values = explainer.shap_values(data_for_prediction)
```

The shap_values object above is a list with two arrays. The first array is the SHAP values for a negative outcome (don't win the award), and the second array is the list of SHAP values for the positive outcome (wins the award). I think about predictions in terms of the prediction of a positive outcome, so we'll pull out SHAP values for positive outcomes (pulling out shap_values[1]).

It's cumbersome to review raw arrays, but the shap package has a nice way to visualize the results.

```
shap.initjs()

shap.force_plot(explainer.expected_value[1], shap_values[1], data_for_predic
```