"Visit with us" Purchase Prediction

Problem Statement:

The Policy Maker of the company wants to enable and establish a viable business model to expand the customer base.

A viable business model is a central concept that helps you to understand the existing ways of doing the business and how to change the ways for the benefit of the tourism sector.

One of the ways to expand the customer base is to introduce a new offering of packages.

Currently, there are 5 types of packages the company is offering - Basic, Standard, Deluxe, Super Deluxe, King. Looking at the data of the last year, we observed that 18% of the customers purchased the packages.

However, the marketing cost was quite high because customers were contacted at random without looking at the available information.

The company is now planning to launch a new product i.e. Wellness Tourism Package. Wellness Tourism is defined as Travel that allows the traveler to maintain, enhance or kick-start a healthy lifestyle, and support or increase one's sense of well-being.

However, this time company wants to harness the available data of existing and potential customers to make the marketing expenditure more efficient.

Objective:

To predict which customer is more likely to purchase the newly introduced travel package.

- ProdTaken will be the dependent variable.

Data Description:

- CustomerID: Unique customer ID
- ProdTaken: Whether the customer has purchased a package or not (0: No, 1: Yes)
- Age: Age of customer
- TypeofContact: How customer was contacted (Company Invited or Self Inquiry)
- CityTier: City tier depends on the development of a city, population, facilities, and living standards. The categories are ordered i.e. Tier 1 > Tier 2 > Tier 3
- Occupation: Occupation of customer
- Gender: Gender of customer

- NumberOfPersonVisiting: Total number of persons planning to take the trip with the customer
- PreferredPropertyStar: Preferred hotel property rating by customer
- MaritalStatus: Marital status of customer
- NumberOfTrips: Average number of trips in a year by customer
- Passport: The customer has a passport or not (0: No, 1: Yes)
- OwnCar: Whether the customers own a car or not (0: No, 1: Yes)
- NumberOfChildrenVisiting: Total number of children with age less than 5 planning to take the trip with the customer
- Designation: Designation of the customer in the current organization
- MonthlyIncome: Gross monthly income of the customer

Customer interaction data:

- PitchSatisfactionScore: Sales pitch satisfaction score
- ProductPitched: Product pitched by the salesperson
- NumberOfFollowups: Total number of follow-ups has been done by the salesperson after the sales pitch
- DurationOfPitch: Duration of the pitch by a salesperson to the customer

Libraries

In [1]:
```
%load_ext nb_black
# Library to suppress warnings or deprecation notes
import warnings

warnings.filterwarnings("ignore")

# Libraries to help with reading and manipulating data
import numpy as np
import pandas as pd

# Libraries to help with data visualization
import matplotlib.pyplot as plt

%matplotlib inline
import seaborn as sns

# Libraries to split data, impute missing values
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

# Libraries to import decision tree classifier and different ensemble classifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.ensemble import StackingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
```

```python
# Libtune to tune model, get different metric scores
from sklearn import metrics
from sklearn.metrics import (
    confusion_matrix,
    classification_report,
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    roc_auc_score,
)
from sklearn.model_selection import GridSearchCV
```

Read Dataset

In [2]:

```python
data = pd.read_excel("Tourism.xlsx", sheet_name="Tourism")

df = data.copy()
```

Data Info/Details

In [3]:

```python
df.head()
```

Out[3]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender	N
0	200000	1	41.0	Self Enquiry	3	6.0	Salaried	Female	
1	200001	0	49.0	Company Invited	1	14.0	Salaried	Male	
2	200002	1	37.0	Self Enquiry	1	8.0	Free Lancer	Male	
3	200003	0	33.0	Company Invited	1	9.0	Salaried	Female	
4	200004	0	NaN	Self Enquiry	1	8.0	Small Business	Male	

In [4]:

```python
df.tail()
```

Out[4]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
4883	204883	1	49.0	Self Enquiry	3	9.0	Small Business	Ma
4884	204884	1	28.0	Company Invited	1	31.0	Salaried	Ma
4885	204885	1	52.0	Self Enquiry	3	17.0	Salaried	Fema
4886	204886	1	19.0	Self Enquiry	3	16.0	Small Business	Ma
4887	204887	1	36.0	Self Enquiry	1	14.0	Salaried	Ma

In [5]:
```
np.random.seed(2)
df.sample(10)
```

Out[5]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
2055	202055	1	23.0	Self Enquiry	1	12.0	Salaried	Mal
3626	203626	0	37.0	Company Invited	1	7.0	Small Business	Femal
4812	204812	0	44.0	Self Enquiry	1	10.0	Salaried	Mal
3047	203047	0	46.0	Self Enquiry	1	9.0	Salaried	Femal
121	200121	0	33.0	Company Invited	3	28.0	Small Business	Mal
278	200278	0	26.0	Company Invited	1	6.0	Salaried	Femal
3507	203507	0	30.0	Self Enquiry	3	26.0	Salaried	Mal
3536	203536	0	46.0	Self Enquiry	1	35.0	Large Business	Mal
1623	201623	0	45.0	Self Enquiry	1	15.0	Salaried	Mal
1740	201740	0	55.0	Self Enquiry	1	6.0	Small Business	Mal

In [6]:
```
print(f"There are {df.shape[0]} rows and {df.shape[1]} columns.")
```

```
There are 4888 rows and 20 columns.
```

In [7]:
```
df[data.duplicated()].count()
```

Out[7]:
```
CustomerID                 0
ProdTaken                  0
Age                        0
TypeofContact              0
CityTier                   0
DurationOfPitch            0
Occupation                 0
Gender                     0
NumberOfPersonVisiting     0
NumberOfFollowups          0
ProductPitched             0
PreferredPropertyStar      0
MaritalStatus              0
NumberOfTrips              0
Passport                   0
PitchSatisfactionScore     0
OwnCar                     0
NumberOfChildrenVisiting   0
Designation                0
MonthlyIncome              0
dtype: int64
```

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4888 entries, 0 to 4887
Data columns (total 20 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   CustomerID                4888 non-null   int64  
 1   ProdTaken                 4888 non-null   int64  
 2   Age                       4662 non-null   float64
 3   TypeofContact             4863 non-null   object 
 4   CityTier                  4888 non-null   int64  
 5   DurationOfPitch           4637 non-null   float64
 6   Occupation                4888 non-null   object 
 7   Gender                    4888 non-null   object 
 8   NumberOfPersonVisiting    4888 non-null   int64  
 9   NumberOfFollowups         4843 non-null   float64
 10  ProductPitched            4888 non-null   object 
 11  PreferredPropertyStar     4862 non-null   float64
 12  MaritalStatus             4888 non-null   object 
 13  NumberOfTrips             4748 non-null   float64
 14  Passport                  4888 non-null   int64  
 15  PitchSatisfactionScore    4888 non-null   int64  
 16  OwnCar                    4888 non-null   int64  
 17  NumberOfChildrenVisiting  4822 non-null   float64
 18  Designation               4888 non-null   object 
 19  MonthlyIncome             4655 non-null   float64
dtypes: float64(7), int64(7), object(6)
memory usage: 763.9+ KB
```

In [9]:
```
df.isnull().sum()
```

Out[9]:
```
CustomerID                    0
ProdTaken                     0
Age                         226
TypeofContact                25
CityTier                      0
DurationOfPitch             251
Occupation                    0
Gender                        0
NumberOfPersonVisiting        0
NumberOfFollowups            45
ProductPitched                0
PreferredPropertyStar        26
MaritalStatus                 0
NumberOfTrips               140
Passport                      0
PitchSatisfactionScore        0
OwnCar                        0
NumberOfChildrenVisiting     66
Designation                   0
MonthlyIncome               233
dtype: int64
```

In [10]:
```
df.describe().T
```

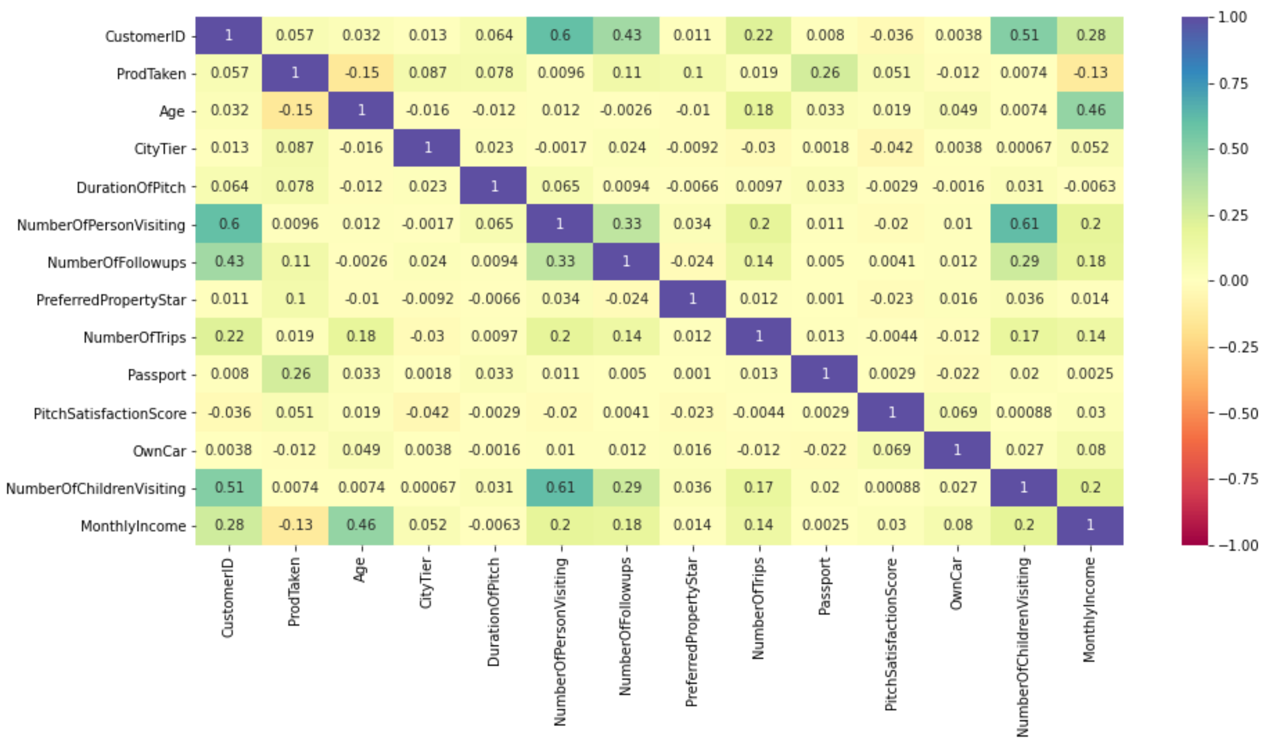
Out[10]:

	count	mean	std	min	25%	50%	2(
CustomerID	4888.0	202443.500000	1411.188388	200000.0	201221.75	202443.5	2(
ProdTaken	4888.0	0.188216	0.390925	0.0	0.00	0.0	
Age	4662.0	37.622265	9.316387	18.0	31.00	36.0	
CityTier	4888.0	1.654255	0.916583	1.0	1.00	1.0	

	count	mean	std	min	25%	50%
DurationOfPitch	4637.0	15.490835	8.519643	5.0	9.00	13.0
NumberOfPersonVisiting	4888.0	2.905074	0.724891	1.0	2.00	3.0
NumberOfFollowups	4843.0	3.708445	1.002509	1.0	3.00	4.0
PreferredPropertyStar	4862.0	3.581037	0.798009	3.0	3.00	3.0
NumberOfTrips	4748.0	3.236521	1.849019	1.0	2.00	3.0
Passport	4888.0	0.290917	0.454232	0.0	0.00	0.0
PitchSatisfactionScore	4888.0	3.078151	1.365792	1.0	2.00	3.0
OwnCar	4888.0	0.620295	0.485363	0.0	0.00	1.0
NumberOfChildrenVisiting	4822.0	1.187267	0.857861	0.0	1.00	1.0
MonthlyIncome	4655.0	23619.853491	5380.698361	1000.0	20346.00	22347.0

Initial EDA

In [11]:
```
plt.figure(figsize=(15, 7))
sns.heatmap(df.corr(), annot=True, vmin=-1, vmax=1, cmap="Spectral")
plt.show()
```



- Passport, Age, and MonthlyIncome have the most correlation to ProdTaken.

In [12]:
```
my_tab = pd.crosstab(index=df["ProdTaken"], columns="count")
my_tab
```

Out[12]:

col_0	count
ProdTaken	
0	3968
1	920

In [13]:
```
my_tab = pd.crosstab(index=df["Age"], columns="count")
my_tab
```

Out[13]:

col_0	count
Age	
18.0	14
19.0	32
20.0	38
21.0	41
22.0	46
23.0	46
24.0	56
25.0	74
26.0	106
27.0	138
28.0	147
29.0	178
30.0	199
31.0	203
32.0	197
33.0	189
34.0	211
35.0	237
36.0	231
37.0	185
38.0	176
39.0	150
40.0	146
41.0	155
42.0	142
43.0	130
44.0	105

col_0	count
Age	
45.0	116
46.0	121
47.0	88
48.0	65
49.0	65
50.0	86
51.0	90
52.0	68
53.0	66
54.0	61
55.0	64
56.0	58
57.0	29
58.0	31
59.0	44
60.0	29
61.0	9

In [14]:
```
my_tab = pd.crosstab(index=df["TypeofContact"], columns="count")
my_tab
```

Out[14]:

col_0	count
TypeofContact	
Company Invited	1419
Self Enquiry	3444

In [15]:
```
my_tab = pd.crosstab(index=df["CityTier"], columns="count")
my_tab
```

Out[15]:

col_0	count
CityTier	
1	3190
2	198
3	1500

In [16]:
```
my_tab = pd.crosstab(index=df["DurationOfPitch"], columns="count")
my_tab
```

Out[16]:

col_0	count
DurationOfPitch	
5.0	6
6.0	307
7.0	342
8.0	333
9.0	483
10.0	244
11.0	205
12.0	195
13.0	223
14.0	253
15.0	269
16.0	274
17.0	172
18.0	75
19.0	57
20.0	65
21.0	73
22.0	89
23.0	79
24.0	70
25.0	73
26.0	72
27.0	72
28.0	61
29.0	74
30.0	95
31.0	83
32.0	74
33.0	57
34.0	50
35.0	66
36.0	44

col_0	count
DurationOfPitch	
126.0	1
127.0	1

In [17]:
```
my_tab = pd.crosstab(index=df["Occupation"], columns="count")
my_tab
```

Out[17]:

col_0	count
Occupation	
Free Lancer	2
Large Business	434
Salaried	2368
Small Business	2084

In [18]:
```
my_tab = pd.crosstab(index=df["Gender"], columns="count")
my_tab
```

Out[18]:

col_0	count
Gender	
Fe Male	155
Female	1817
Male	2916

- Have to correct this data entry error.

In [19]:
```
my_tab = pd.crosstab(index=df["NumberOfPersonVisiting"], columns="count")
my_tab
```

Out[19]:

col_0	count
NumberOfPersonVisiting	
1	39
2	1418
3	2402
4	1026
5	3

In [20]:
```
my_tab = pd.crosstab(index=df["NumberOfFollowups"], columns="count")
my_tab
```

Out[20]:

col_0	count
NumberOfFollowups	
1.0	176
2.0	229
3.0	1466
4.0	2068
5.0	768
6.0	136

In [21]:

```
my_tab = pd.crosstab(index=df["ProductPitched"], columns="count")
my_tab
```

Out[21]:

col_0	count
ProductPitched	
Basic	1842
Deluxe	1732
King	230
Standard	742
Super Deluxe	342

In [22]:

```
my_tab = pd.crosstab(index=df["PreferredPropertyStar"], columns="count")
my_tab
```

Out[22]:

col_0	count
PreferredPropertyStar	
3.0	2993
4.0	913
5.0	956

In [23]:

```
my_tab = pd.crosstab(index=df["MaritalStatus"], columns="count")
my_tab
```

Out[23]:

col_0	count
MaritalStatus	
Divorced	950
Married	2340
Single	916
Unmarried	682

In [24]:
```
my_tab = pd.crosstab(index=df["NumberOfTrips"], columns="count")
my_tab
```

Out[24]:

col_0	count
NumberOfTrips	
1.0	620
2.0	1464
3.0	1079
4.0	478
5.0	458
6.0	322
7.0	218
8.0	105
19.0	1
20.0	1
21.0	1
22.0	1

In [25]:
```
my_tab = pd.crosstab(index=df["Passport"], columns="count")
my_tab
```

Out[25]:

col_0	count
Passport	
0	3466
1	1422

In [26]:
```
my_tab = pd.crosstab(index=df["PitchSatisfactionScore"], columns="count")
my_tab
```

Out[26]:

col_0	count
PitchSatisfactionScore	
1	942
2	586
3	1478
4	912
5	970

In [27]:
```
my_tab = pd.crosstab(index=df["OwnCar"], columns="count")
my_tab
```

Out[27]:

col_0	count
OwnCar	
0	1856
1	3032

In [28]:
```
my_tab = pd.crosstab(index=df["NumberOfChildrenVisiting"], columns="count")
my_tab
```

Out[28]:

col_0	count
NumberOfChildrenVisiting	
0.0	1082
1.0	2080
2.0	1335
3.0	325

In [29]:
```
my_tab = pd.crosstab(index=df["Designation"], columns="count")
my_tab
```

Out[29]:

col_0	count
Designation	
AVP	342
Executive	1842
Manager	1732
Senior Manager	742
VP	230

In [30]:
```
my_tab = pd.crosstab(index=df["MonthlyIncome"], columns="count")
my_tab
```

Out[30]:

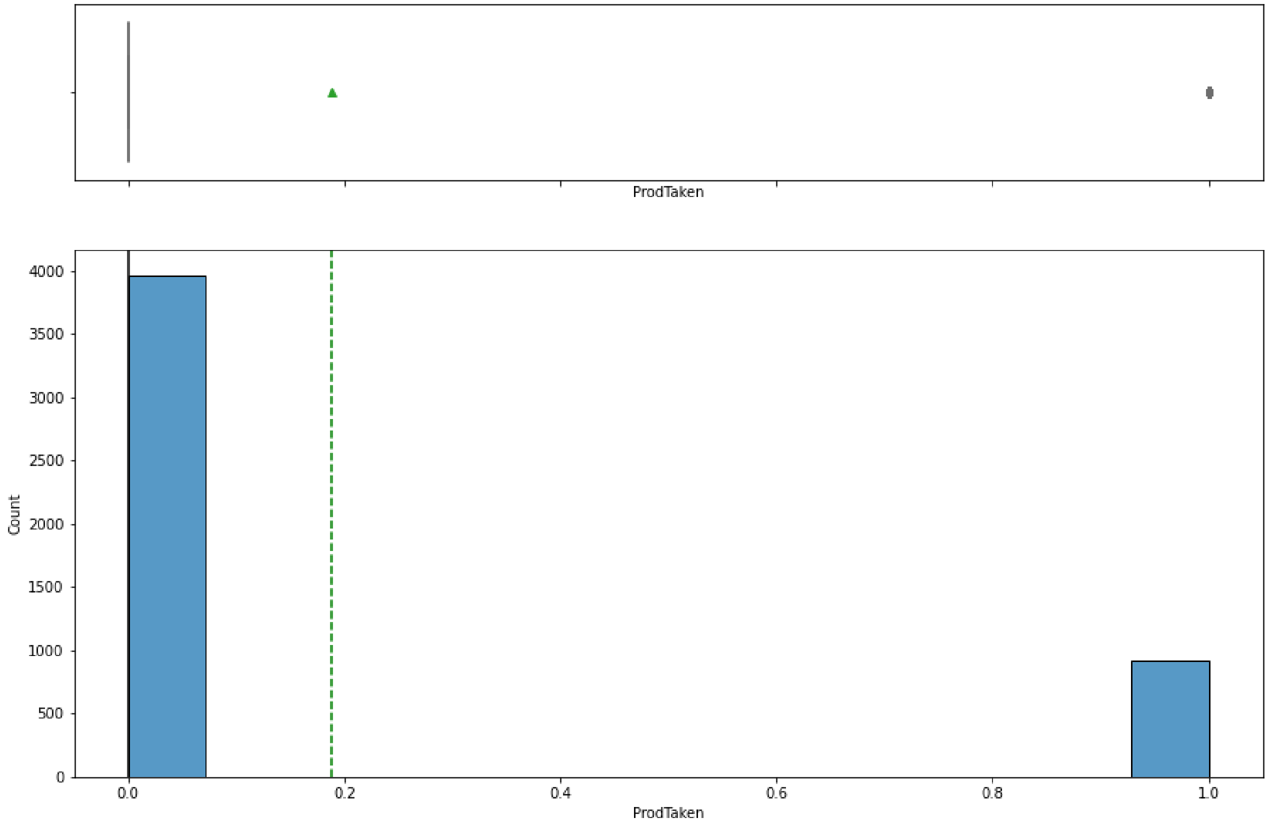
col_0	count
MonthlyIncome	
1000.0	1
4678.0	1
16009.0	2
16051.0	2
16052.0	2

col_0	count
MonthlyIncome	
...	...
38621.0	2
38651.0	2
38677.0	2
95000.0	1
98678.0	1

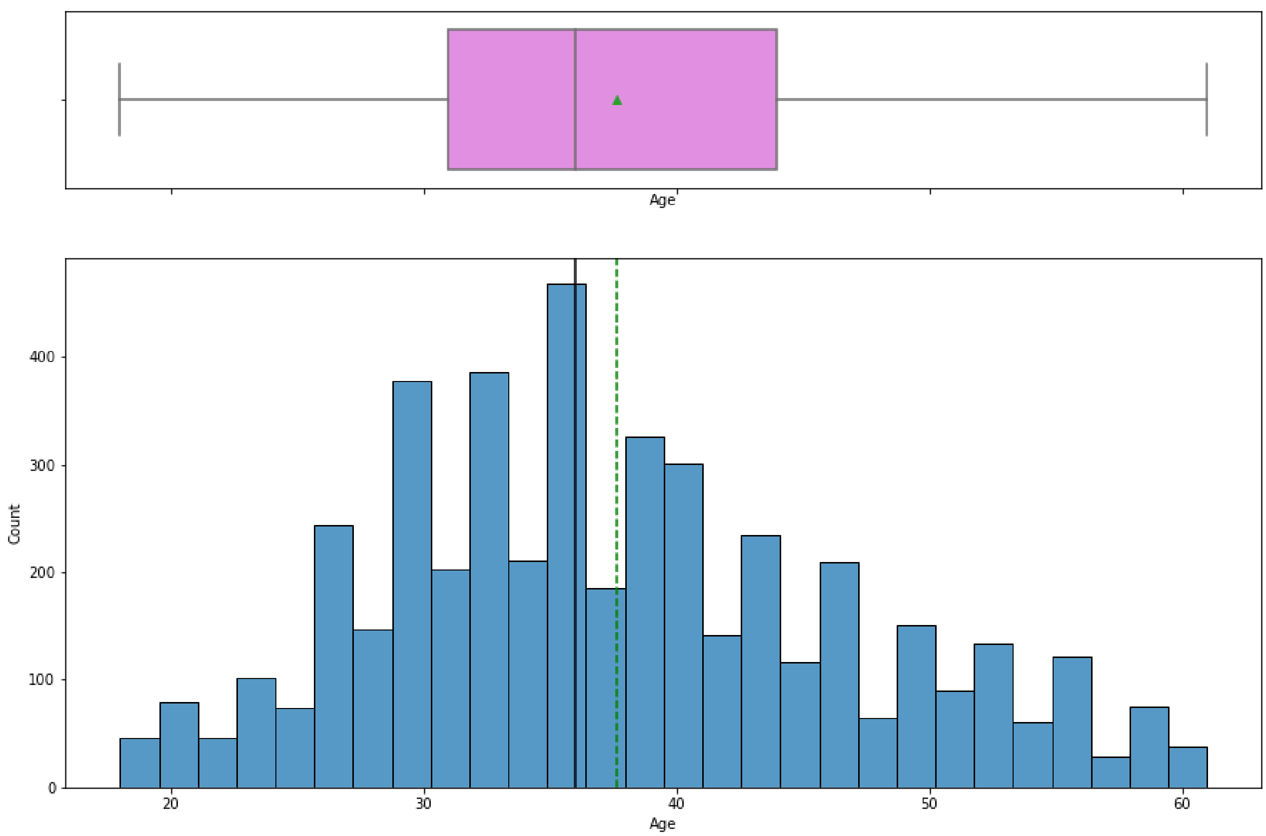
2475 rows × 1 columns

In [31]:
```
def histogram_boxplot(data, feature, figsize=(15, 10), kde=False, bins=None):
    """
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (15,10))
    kde: whether to show the density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows=2,
        sharex=True,
        gridspec_kw={"height_ratios": (0.25, 0.75)},
        figsize=figsize,
    )
    sns.boxplot(data=data, x=feature, ax=ax_box2, showmeans=True, color="violet"
    sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2, bins=bins, palette="winter"
    ) if bins else sns.histplot(data=data, x=feature, kde=kde, ax=ax_hist2)
    ax_hist2.axvline(data[feature].mean(), color="green", linestyle="--")
    ax_hist2.axvline(data[feature].median(), color="black", linestyle="-")
```

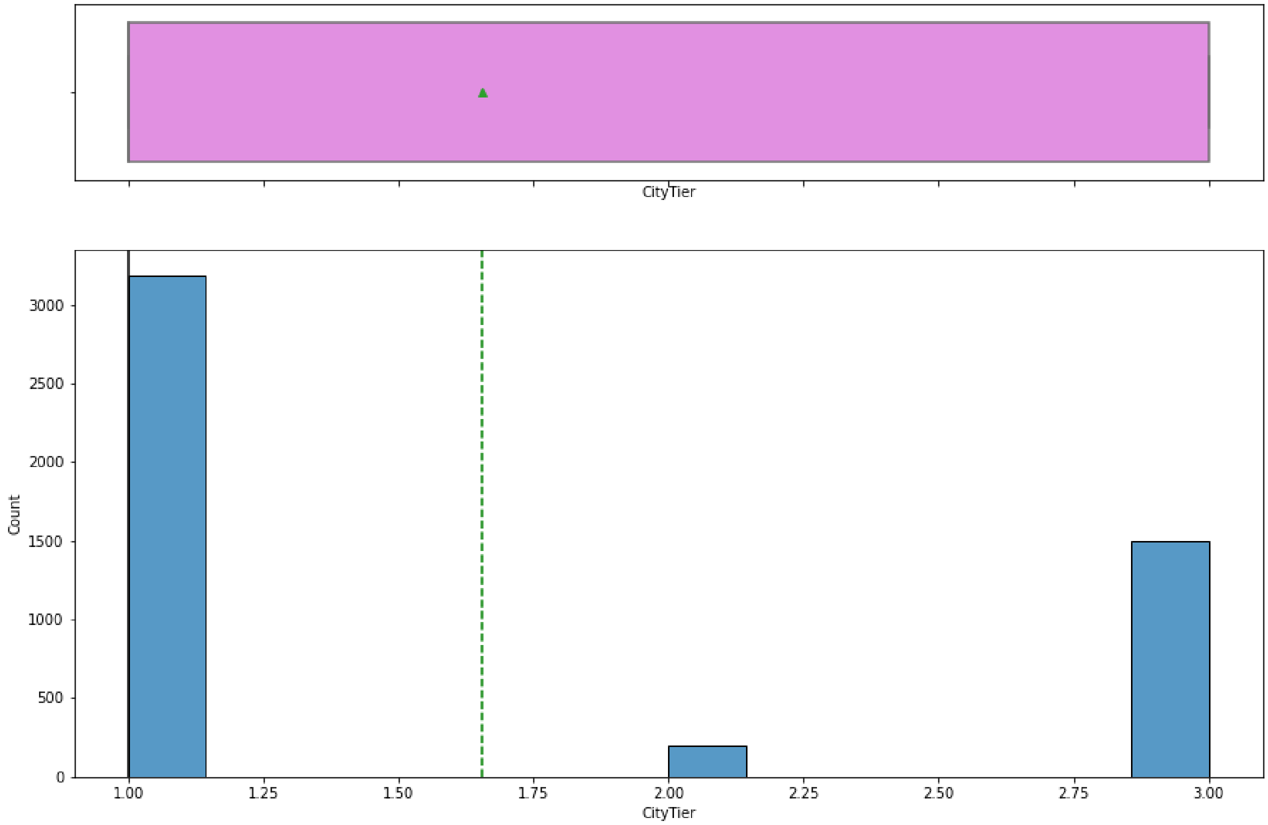
In [32]:
```
histogram_boxplot(data, "ProdTaken")
```



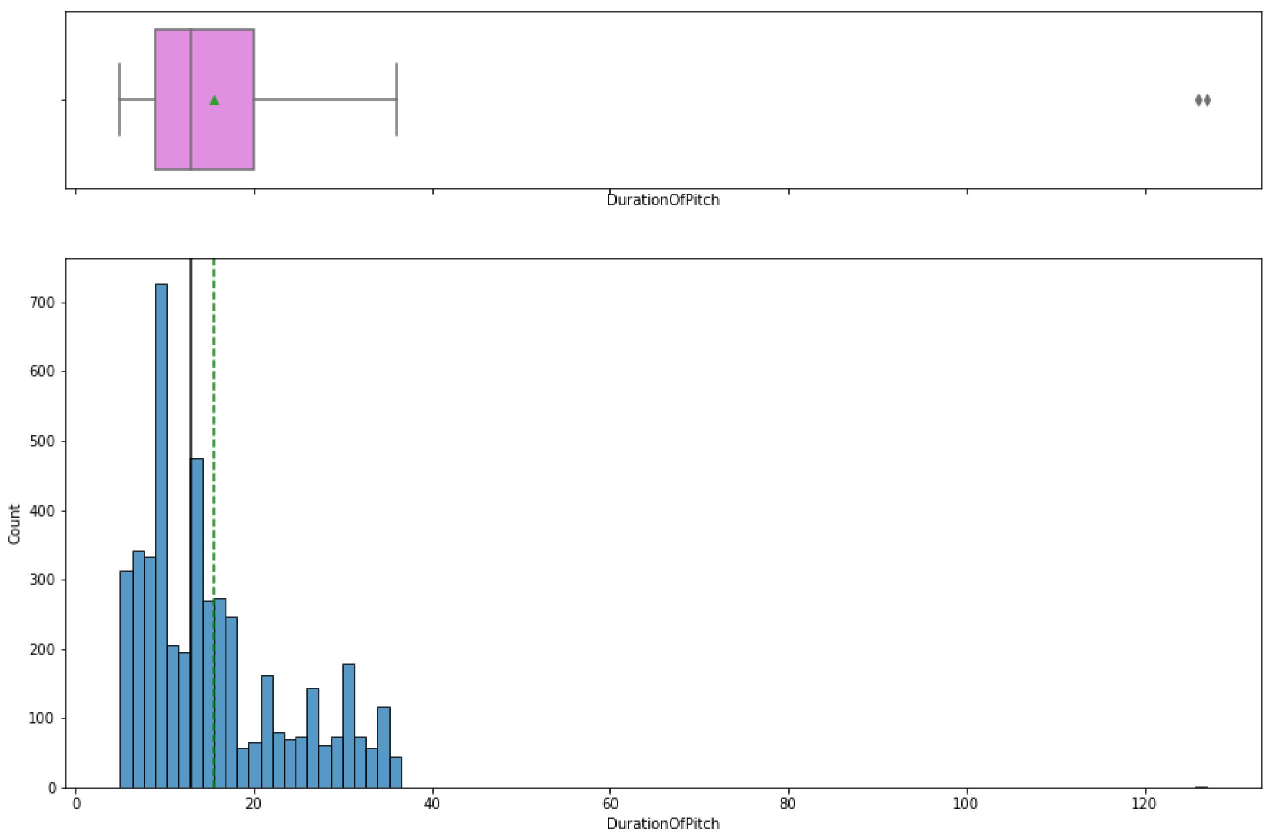
In [33]:
```
histogram_boxplot(data, "Age")
```



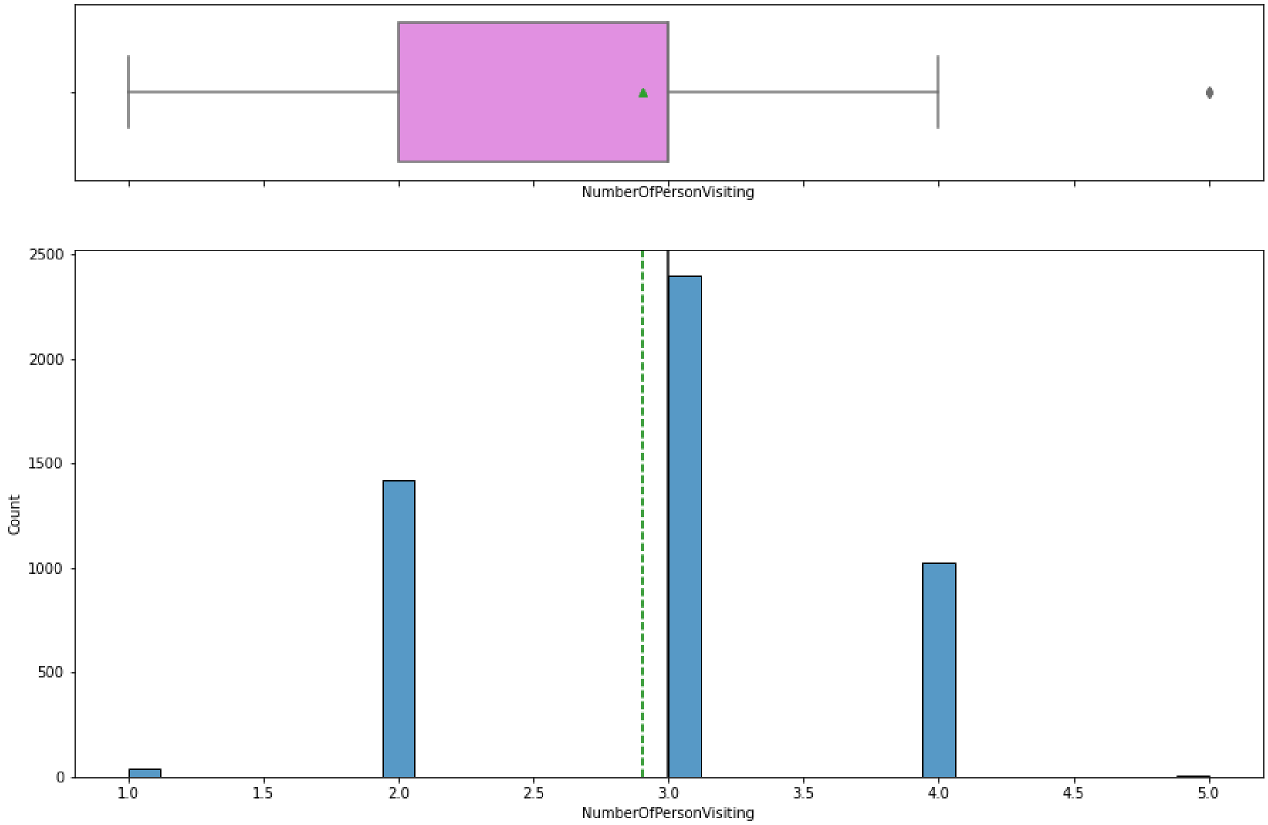
In [34]:
```
histogram_boxplot(data, "CityTier")
```



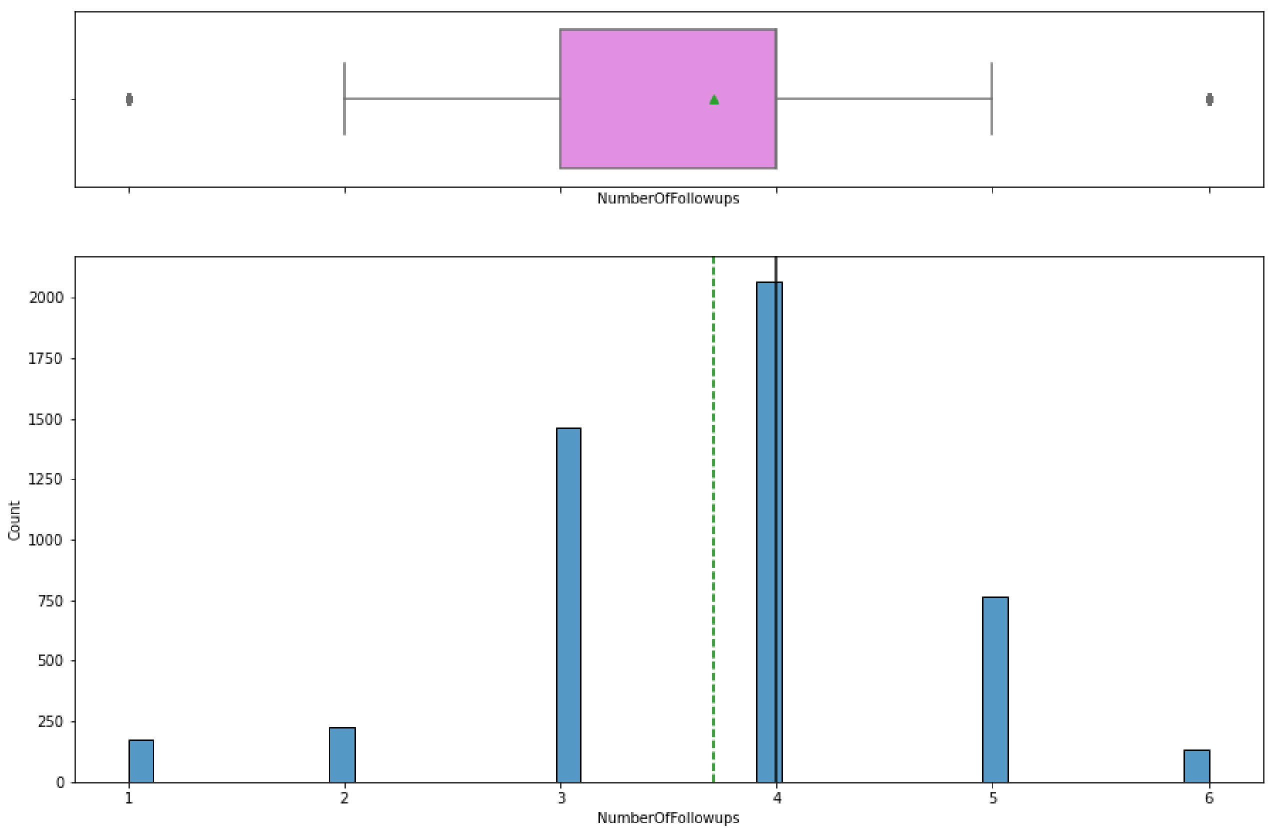
In [35]:
```
histogram_boxplot(data, "DurationOfPitch")
```



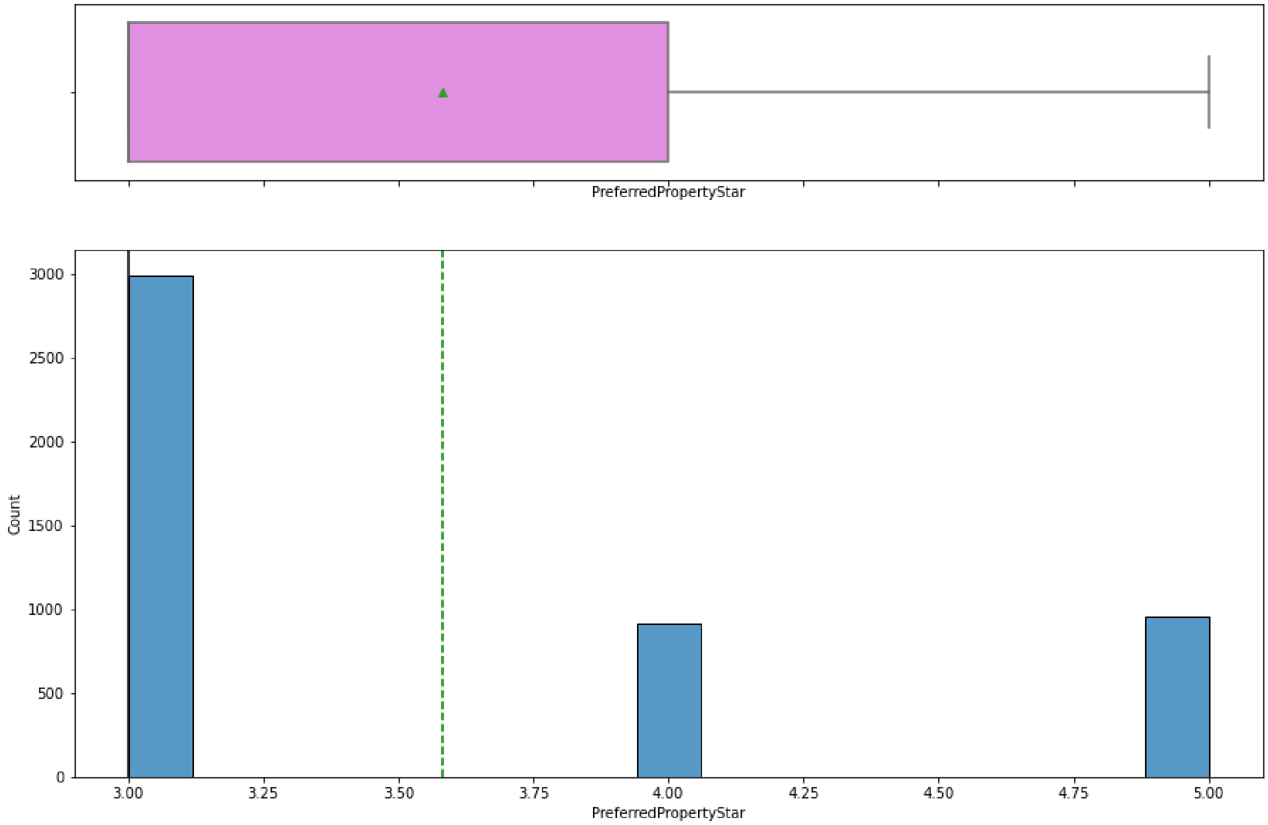
In [36]:
```
histogram_boxplot(data, "NumberOfPersonVisiting")
```



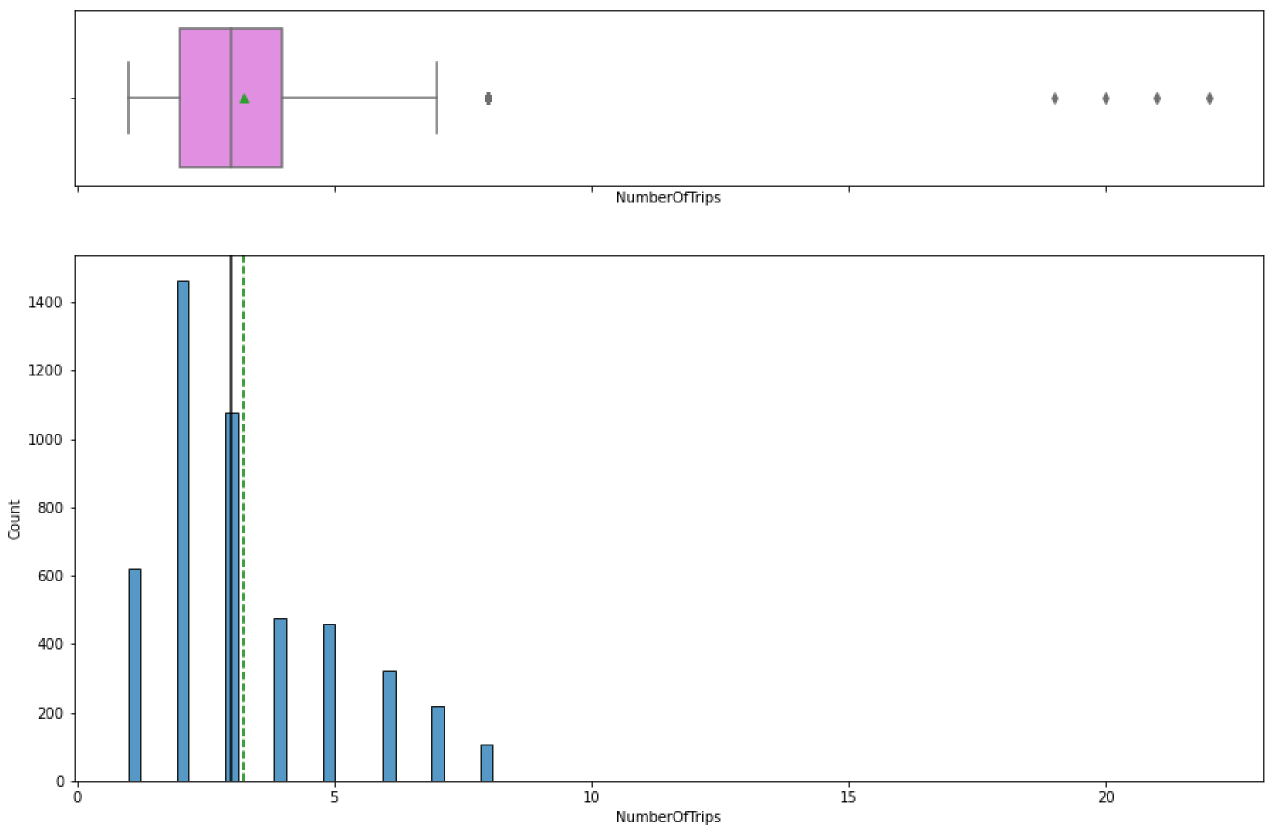
In [37]:
```
histogram_boxplot(data, "NumberOfFollowups")
```



In [38]:
```
histogram_boxplot(data, "PreferredPropertyStar")
```

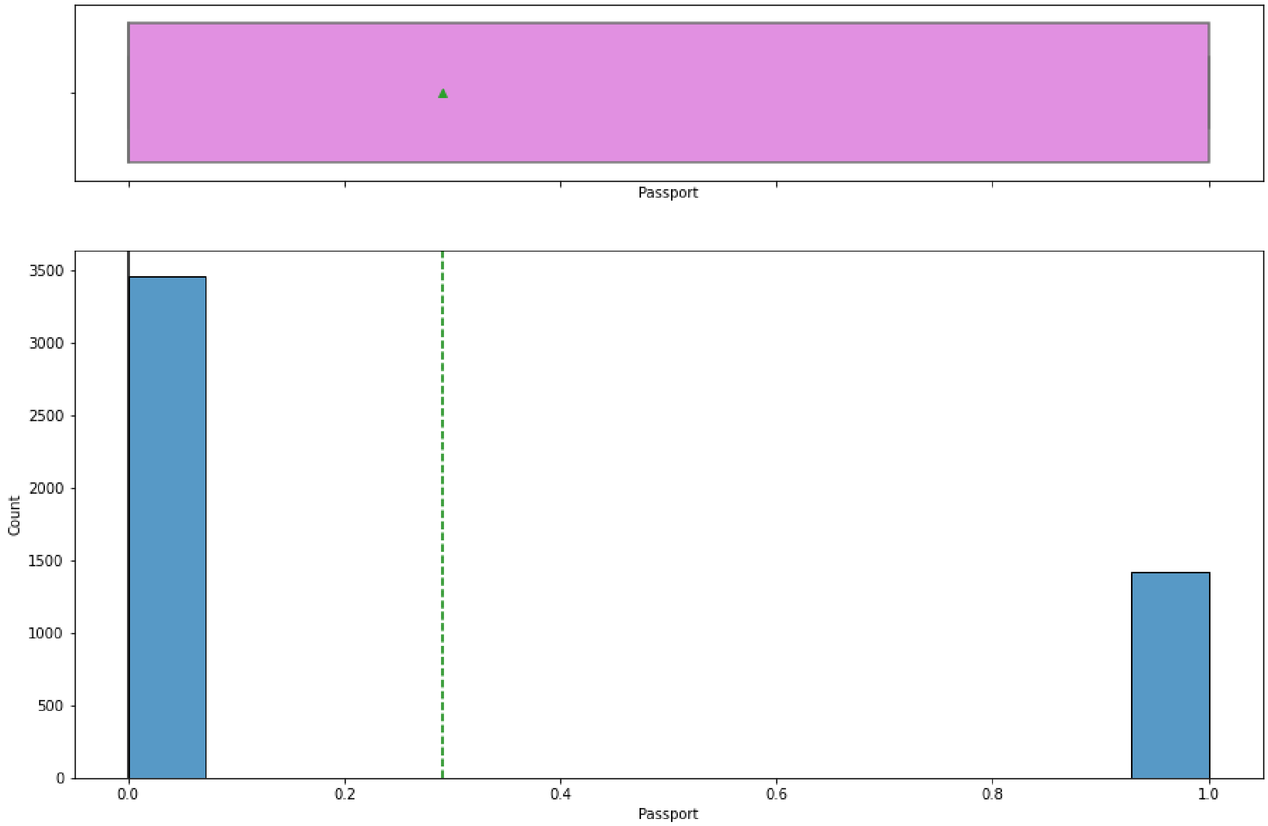


In [39]:
```
histogram_boxplot(data, "NumberOfTrips")
```

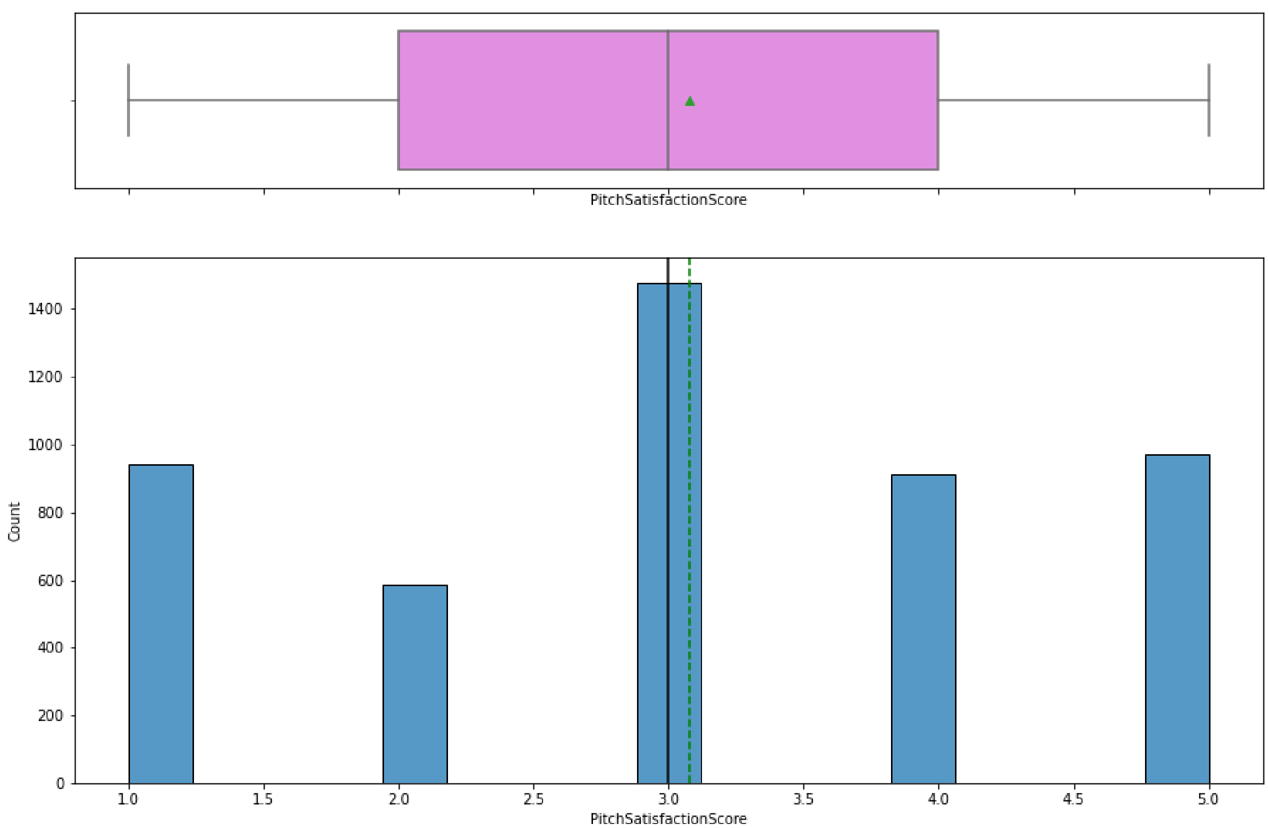


- There are a few outliers I will look into.

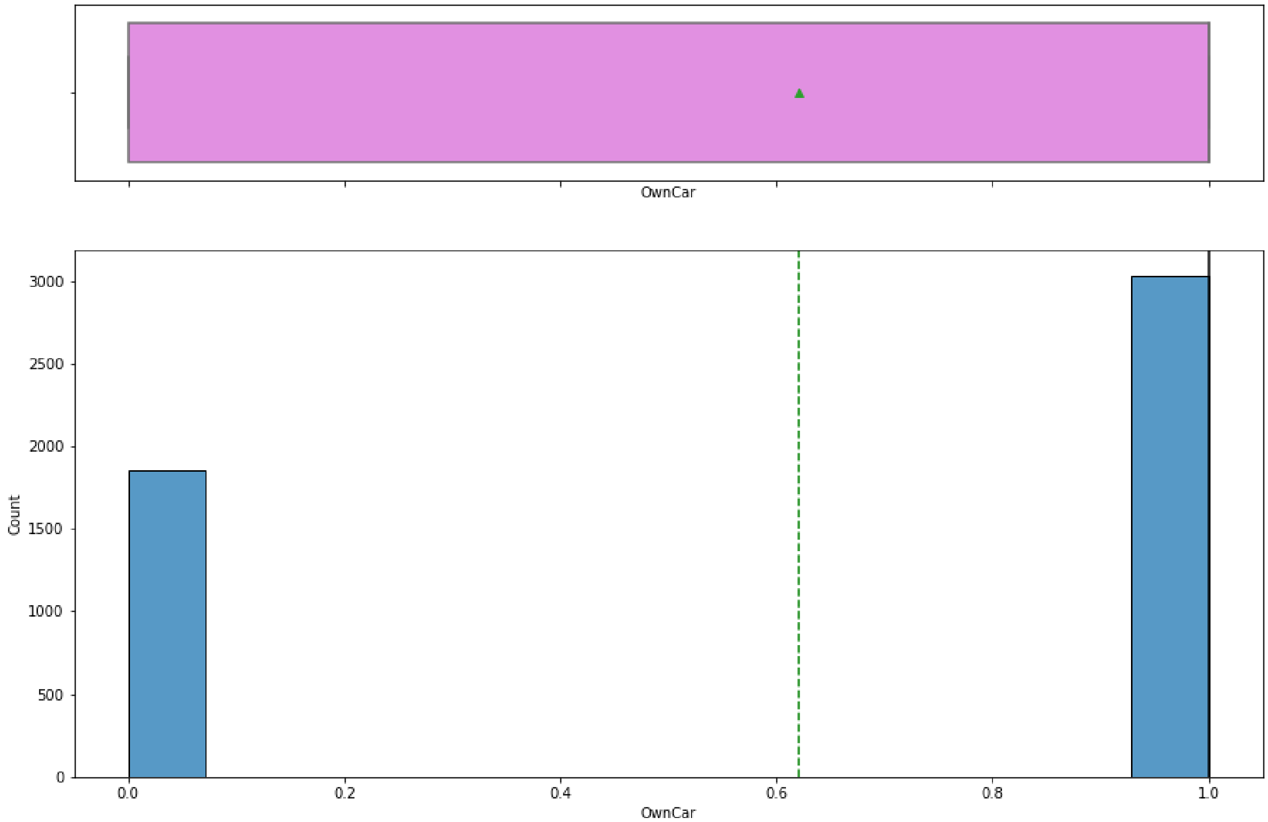
In [40]:
```
histogram_boxplot(data, "Passport")
```



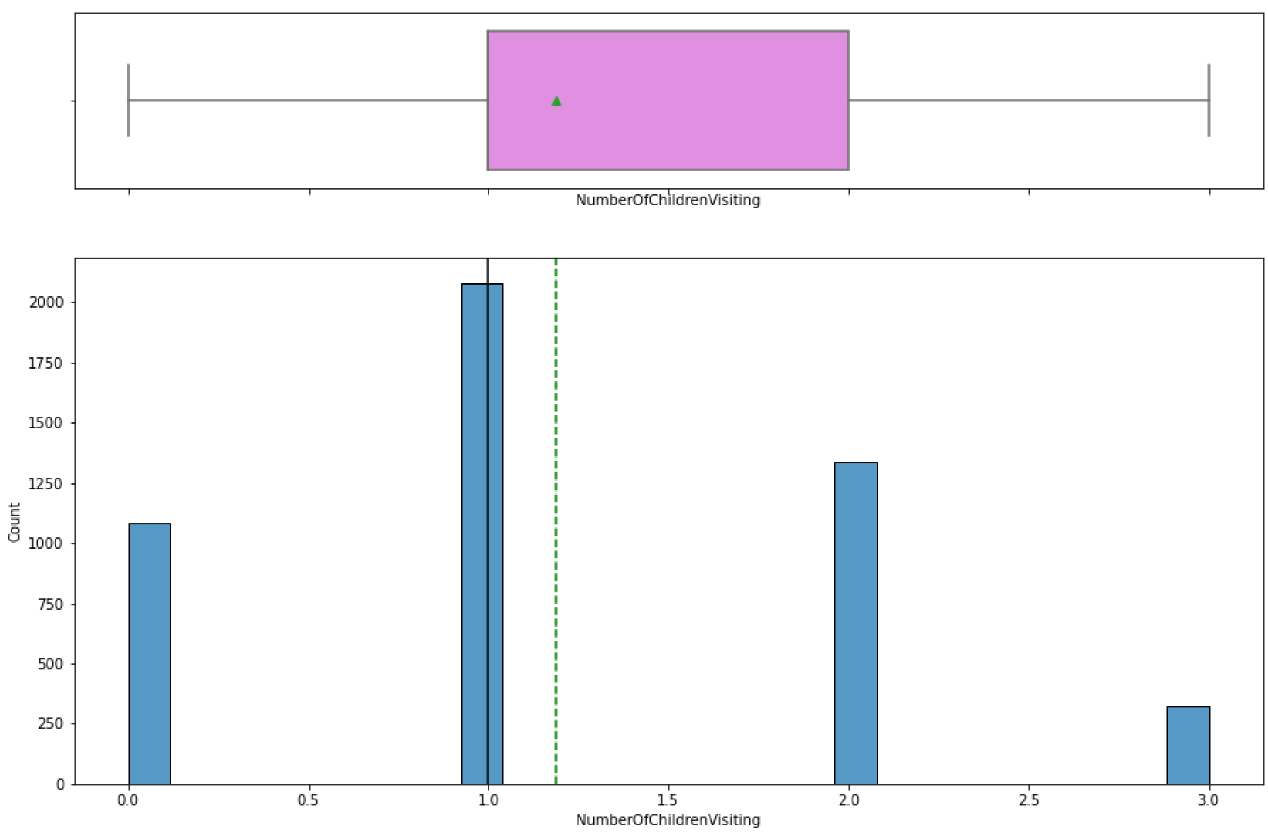
In [41]:
```
histogram_boxplot(data, "PitchSatisfactionScore")
```



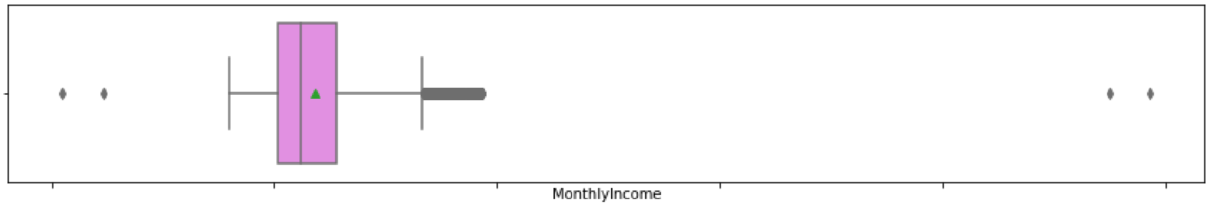
In [42]:
```
histogram_boxplot(data, "OwnCar")
```

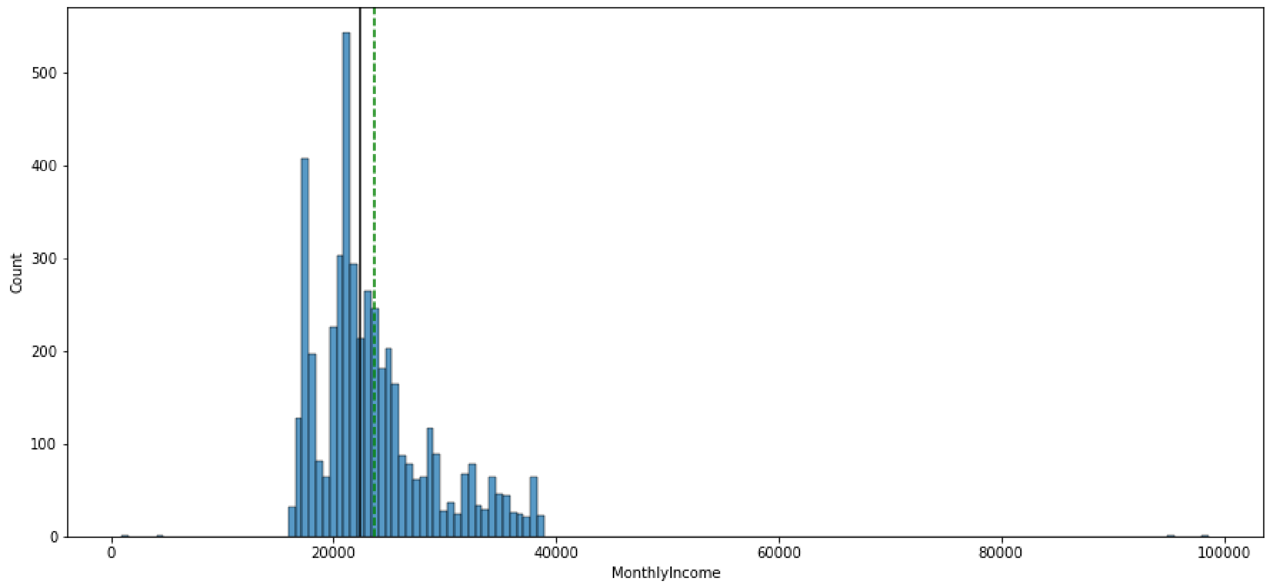


In [43]:
```
histogram_boxplot(data, "NumberOfChildrenVisiting")
```



In [44]:
```
histogram_boxplot(data, "MonthlyIncome")
```



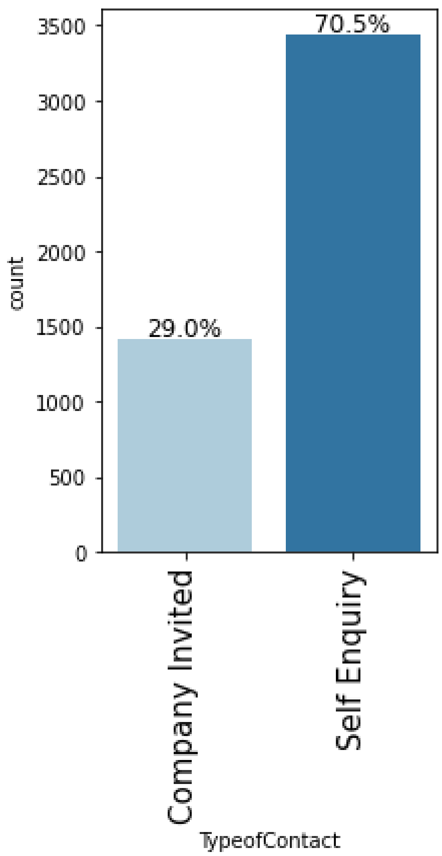


- There are a few outliers I will look into.

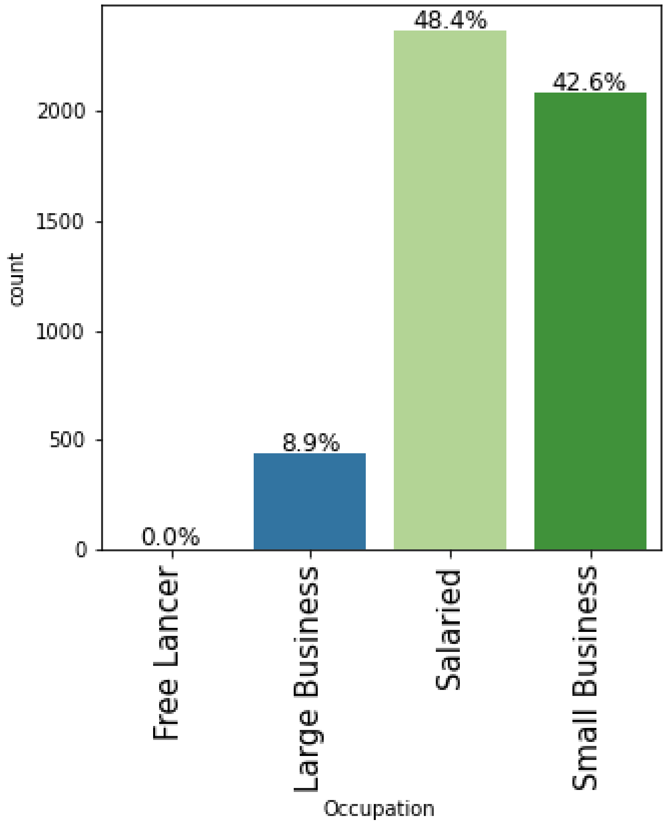
In [45]:
```
def labeled_barplot(data, feature, perc=False, n=None):
    """
    Barplot with percentage at the top

    data: dataframe
    feature: dataframe column
    perc: whether to display percentages instead of count (default is False)
    n: displays the top n category levels (default is None, i.e., display all le
    """

    total = len(data[feature])
    count = data[feature].nunique()
    if n is None:
        plt.figure(figsize=(count + 1, 5))
    else:
        plt.figure(figsize=(n + 1, 5))

    plt.xticks(rotation=90, fontsize=15)
    ax = sns.countplot(
        data=data,
        x=feature,
        palette="Paired",
        order=data[feature].value_counts().index[:n].sort_values(),
    )

    for p in ax.patches:
        if perc == True:
```

```
            label = "{:.1f}%".format(100 * p.get_height() / total)
        else:
            label = p.get_height()

        x = p.get_x() + p.get_width() / 2
        y = p.get_height()

        ax.annotate(
            label,
            (x, y),
            ha="center",
            va="center",
            size=12,
            xytext=(0, 5),
            textcoords="offset points",
        )

    plt.show()
```

In [46]:
```
labeled_barplot(data, "TypeofContact", perc=True)
```

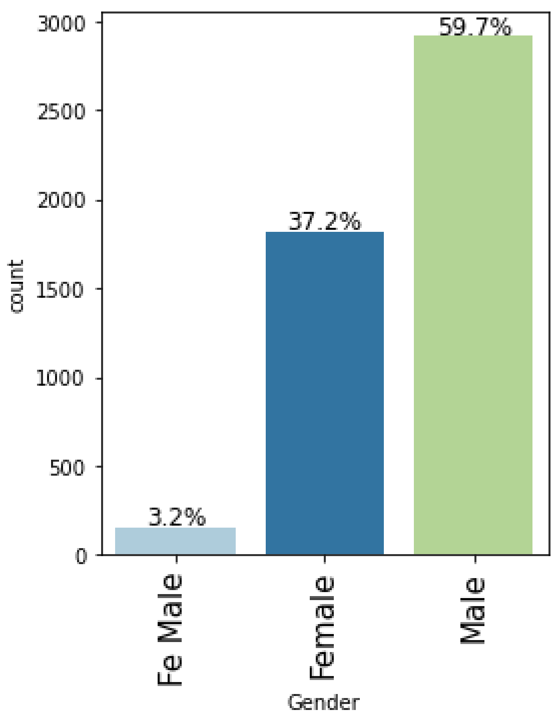


In [47]:
```
labeled_barplot(data, "Occupation", perc=True)
```

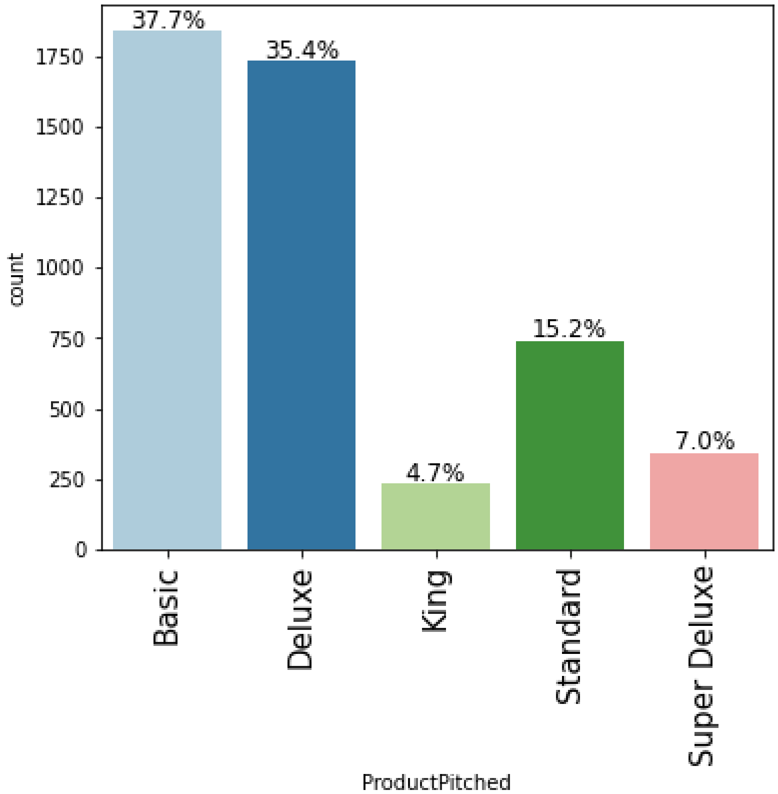


- Most people are Salaried or Small Business.

In [48]:
```
labeled_barplot(data, "Gender", perc=True)
```

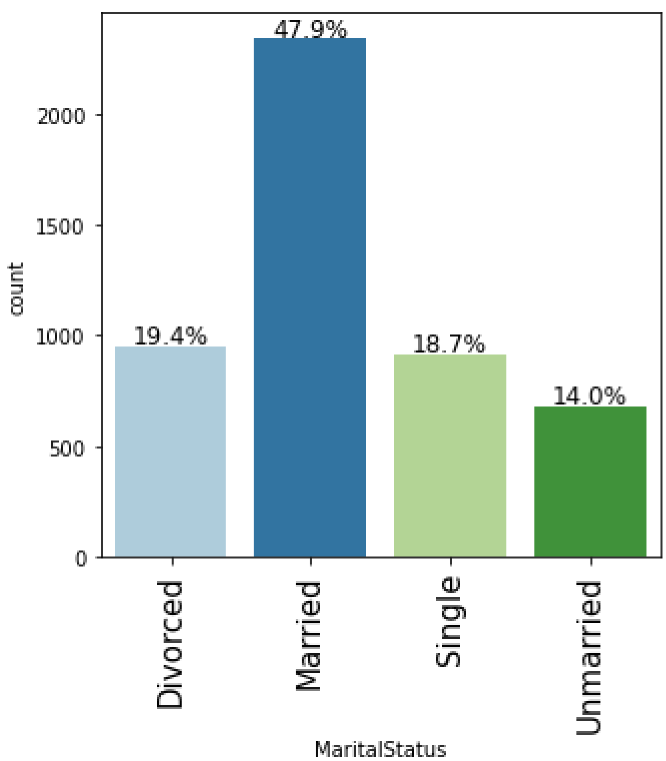


In [49]:
```
labeled_barplot(data, "ProductPitched", perc=True)
```



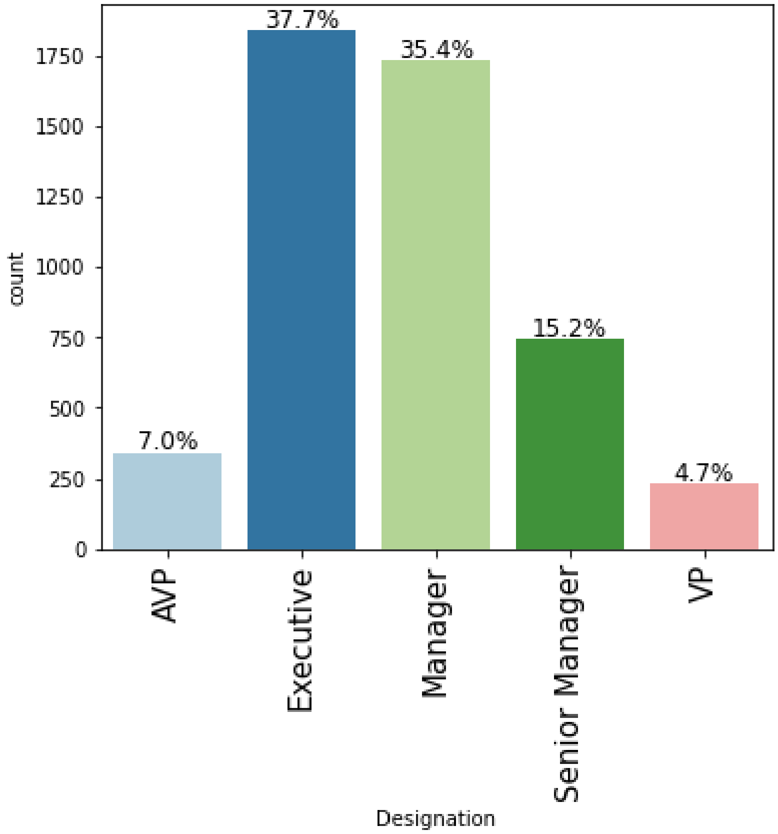
- The Basic and Deluxe package were pitched the most.

In [50]:
```
labeled_barplot(data, "MaritalStatus", perc=True)
```



- There are more Married individuals.

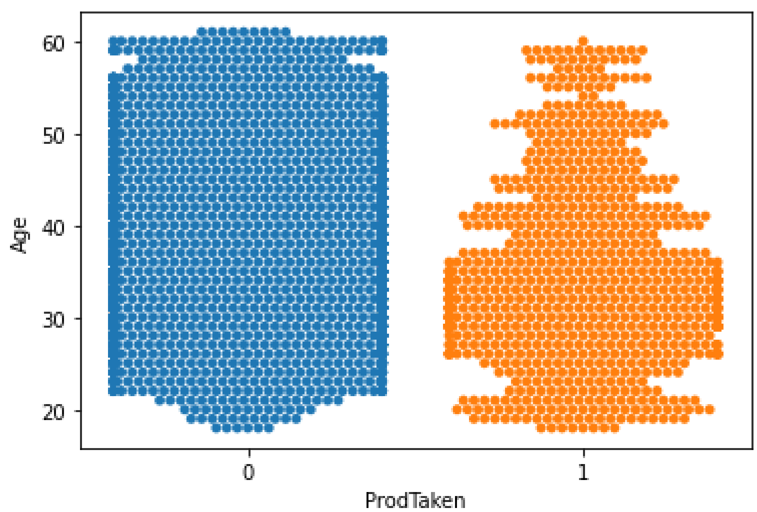
In [51]:
```
labeled_barplot(data, "Designation", perc=True)
```



- Executive and Manager are the most common positions.

In [52]:
```
sns.swarmplot(df["ProdTaken"], df["Age"])
```

Out[52]: <AxesSubplot:xlabel='ProdTaken', ylabel='Age'>



- Looks like people of age 26-36 are more likely to purchase a travel package.

ProductPitched

In [53]:
```
gk = df.groupby("ProductPitched")
gk.get_group("Basic")
```

Out[53]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
2	200002	1	37.0	Self Enquiry	1	8.0	Free Lancer	Ma
3	200003	0	33.0	Company Invited	1	9.0	Salaried	Fema
4	200004	0	NaN	Self Enquiry	1	8.0	Small Business	Ma
5	200005	0	32.0	Company Invited	1	8.0	Salaried	Ma
6	200006	0	59.0	Self Enquiry	1	9.0	Small Business	Fema
...	...	...	...	...	...	...	...	.
4881	204881	1	41.0	Self Enquiry	2	25.0	Salaried	Ma
4882	204882	1	37.0	Self Enquiry	2	20.0	Salaried	Ma
4884	204884	1	28.0	Company Invited	1	31.0	Salaried	Ma
4886	204886	1	19.0	Self Enquiry	3	16.0	Small Business	Ma
4887	204887	1	36.0	Self Enquiry	1	14.0	Salaried	Ma

1842 rows × 20 columns

In [54]:
```
gk = df.groupby("ProductPitched")
gk.get_group("Standard")
```

Out[54]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
8	200008	0	38.0	Company Invited	1	29.0	Salaried	Ma
15	200015	0	29.0	Self Enquiry	1	27.0	Salaried	Fema
22	200022	0	34.0	Self Enquiry	1	13.0	Salaried	Fe Ma
28	200028	0	44.0	Self Enquiry	1	13.0	Small Business	Fema
43	200043	0	27.0	Company Invited	3	14.0	Salaried	Ma
...	...	...	...	...	...	...	...	.
4852	204852	1	59.0	Self Enquiry	1	9.0	Large Business	Fema
4856	204856	1	37.0	Self Enquiry	3	17.0	Small Business	Ma
4870	204870	1	57.0	Self Enquiry	3	23.0	Salaried	Fema

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
4871	204871	1	41.0	Self Enquiry	3	23.0	Small Business	Ma
4885	204885	1	52.0	Self Enquiry	3	17.0	Salaried	Fema

742 rows × 20 columns

In [55]:
```python
gk = df.groupby("ProductPitched")
gk.get_group("Deluxe")
```

Out[55]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
0	200000	1	41.0	Self Enquiry	3	6.0	Salaried	Fema
1	200001	0	49.0	Company Invited	1	14.0	Salaried	Ma
9	200009	0	36.0	Self Enquiry	1	33.0	Small Business	Ma
11	200011	0	NaN	Self Enquiry	1	21.0	Salaried	Fema
20	200020	0	NaN	Company Invited	1	17.0	Salaried	Fema
...	...	...	...	...	...	...	...	.
4876	204876	1	52.0	Self Enquiry	3	34.0	Salaried	Ma
4877	204877	1	39.0	Company Invited	1	16.0	Salaried	Ma
4878	204878	1	35.0	Self Enquiry	1	17.0	Small Business	Ma
4880	204880	1	59.0	Self Enquiry	1	28.0	Small Business	Fema
4883	204883	1	49.0	Self Enquiry	3	9.0	Small Business	Ma

1732 rows × 20 columns

In [56]:
```python
gk = df.groupby("ProductPitched")
gk.get_group("Super Deluxe")
```

Out[56]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
18	200018	0	53.0	Self Enquiry	3	8.0	Salaried	Fema
65	200065	0	55.0	Self Enquiry	1	14.0	Small Business	Fema
90	200090	0	40.0	Company Invited	1	6.0	Salaried	Ma
98	200098	0	58.0	Self Enquiry	3	16.0	Small Business	Ma

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
112	200112	0	54.0	Company Invited	2	32.0	Salaried	Fema
...	...	...	...	...	...	...	...	.
4775	204775	0	47.0	Self Enquiry	3	9.0	Small Business	Fema
4781	204781	0	51.0	Company Invited	1	9.0	Small Business	Fema
4808	204808	0	55.0	Self Enquiry	1	10.0	Salaried	Ma
4827	204827	1	46.0	Self Enquiry	3	20.0	Small Business	Ma
4865	204865	1	42.0	Company Invited	3	16.0	Salaried	Ma

342 rows × 20 columns

In [57]:
```
gk = df.groupby("ProductPitched")
gk.get_group("King")
```

Out[57]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
25	200025	0	53.0	Self Enquiry	1	11.0	Salaried	Femal
29	200029	0	46.0	Self Enquiry	3	8.0	Small Business	Femal
45	200045	1	41.0	Self Enquiry	1	18.0	Large Business	Femal
62	200062	0	50.0	Self Enquiry	1	13.0	Small Business	Femal
105	200105	0	59.0	Company Invited	2	8.0	Salaried	Femal
...	...	...	...	...	...	...	...	.
4772	204772	0	54.0	Self Enquiry	1	14.0	Small Business	Femal
4783	204783	0	47.0	Self Enquiry	1	22.0	Salaried	Mal
4812	204812	0	44.0	Self Enquiry	1	10.0	Salaried	Mal
4813	204813	0	50.0	Self Enquiry	1	11.0	Small Business	Mal
4816	204816	1	28.0	Self Enquiry	3	9.0	Small Business	Femal

230 rows × 20 columns

Designation

In [58]:
```
gk = df.groupby("Designation")
gk.get_group("Senior Manager")
```

Out[58]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
8	200008	0	38.0	Company Invited	1	29.0	Salaried	Ma
15	200015	0	29.0	Self Enquiry	1	27.0	Salaried	Fema
22	200022	0	34.0	Self Enquiry	1	13.0	Salaried	Fe Ma
28	200028	0	44.0	Self Enquiry	1	13.0	Small Business	Fema
43	200043	0	27.0	Company Invited	3	14.0	Salaried	Ma
...	...	...	...	...	...	...	...	.
4852	204852	1	59.0	Self Enquiry	1	9.0	Large Business	Fema
4856	204856	1	37.0	Self Enquiry	3	17.0	Small Business	Ma
4870	204870	1	57.0	Self Enquiry	3	23.0	Salaried	Fema
4871	204871	1	41.0	Self Enquiry	3	23.0	Small Business	Ma
4885	204885	1	52.0	Self Enquiry	3	17.0	Salaried	Fema

742 rows × 20 columns

In [59]:
```
gk = df.groupby("Designation")
gk.get_group("VP")
```

Out[59]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
25	200025	0	53.0	Self Enquiry	1	11.0	Salaried	Femal
29	200029	0	46.0	Self Enquiry	3	8.0	Small Business	Femal
45	200045	1	41.0	Self Enquiry	1	18.0	Large Business	Femal
62	200062	0	50.0	Self Enquiry	1	13.0	Small Business	Femal
105	200105	0	59.0	Company Invited	2	8.0	Salaried	Femal
...	...	...	...	...	...	...	...	.
4772	204772	0	54.0	Self Enquiry	1	14.0	Small Business	Femal
4783	204783	0	47.0	Self Enquiry	1	22.0	Salaried	Mal

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
4812	204812	0	44.0	Self Enquiry	1	10.0	Salaried	Mal
4813	204813	0	50.0	Self Enquiry	1	11.0	Small Business	Mal
4816	204816	1	28.0	Self Enquiry	3	9.0	Small Business	Femal

230 rows × 20 columns

In [60]:

```
gk = df.groupby("Designation")
gk.get_group("AVP")
```

Out[60]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
18	200018	0	53.0	Self Enquiry	3	8.0	Salaried	Fema
65	200065	0	55.0	Self Enquiry	1	14.0	Small Business	Fema
90	200090	0	40.0	Company Invited	1	6.0	Salaried	Ma
98	200098	0	58.0	Self Enquiry	3	16.0	Small Business	Ma
112	200112	0	54.0	Company Invited	2	32.0	Salaried	Fema
...	...	...	...	...	...	...	...	.
4775	204775	0	47.0	Self Enquiry	3	9.0	Small Business	Fema
4781	204781	0	51.0	Company Invited	1	9.0	Small Business	Fema
4808	204808	0	55.0	Self Enquiry	1	10.0	Salaried	Ma
4827	204827	1	46.0	Self Enquiry	3	20.0	Small Business	Ma
4865	204865	1	42.0	Company Invited	3	16.0	Salaried	Ma

342 rows × 20 columns

In [61]:

```
gk = df.groupby("Designation")
gk.get_group("Executive")
```

Out[61]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
2	200002	1	37.0	Self Enquiry	1	8.0	Free Lancer	Ma
3	200003	0	33.0	Company Invited	1	9.0	Salaried	Fema
4	200004	0	NaN	Self Enquiry	1	8.0	Small Business	Ma

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
5	200005	0	32.0	Company Invited	1	8.0	Salaried	Ma
6	200006	0	59.0	Self Enquiry	1	9.0	Small Business	Fema
...	...	...	...	...	...	...	...	.
4881	204881	1	41.0	Self Enquiry	2	25.0	Salaried	Ma
4882	204882	1	37.0	Self Enquiry	2	20.0	Salaried	Ma
4884	204884	1	28.0	Company Invited	1	31.0	Salaried	Ma
4886	204886	1	19.0	Self Enquiry	3	16.0	Small Business	Ma
4887	204887	1	36.0	Self Enquiry	1	14.0	Salaried	Ma

1842 rows × 20 columns

Basic = Executive

Standard = Senior Manager

Deluxe = Manager

Super Deluxe = AVP

King = VP

Data Pre-processing

Outliers

NumberOfTrips

In [62]:

```
gk = df.groupby("NumberOfTrips")
gk.get_group(19.0)
```

Out[62]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender
385	200385	1	30.0	Company Invited	1	10.0	Large Business	Male

In [63]:

```
gk = df.groupby("NumberOfTrips")
gk.get_group(20.0)
```

Out[63]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
2829	202829	1	31.0	Company Invited	1	11.0	Large Business	Mal

In [64]:
```
gk = df.groupby("NumberOfTrips")
gk.get_group(21.0)
```

Out[64]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender
816	200816	0	39.0	Company Invited	1	15.0	Salaried	Male

In [65]:
```
gk = df.groupby("NumberOfTrips")
gk.get_group(22.0)
```

Out[65]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
3260	203260	0	40.0	Company Invited	1	16.0	Salaried	Mal

- 19+ trips a year doesn't seem reasonable, so I will change each value to the median value of "NumberOfTrips" given their "Designation" position.

In [66]:
```
gk = df.groupby("Designation")
gk.get_group("Executive").median()
```

Out[66]:
```
CustomerID                  202444.5
ProdTaken                        0.0
Age                             32.0
CityTier                         1.0
DurationOfPitch                 13.0
NumberOfPersonVisiting           3.0
NumberOfFollowups                4.0
PreferredPropertyStar            3.0
NumberOfTrips                    3.0
Passport                         0.0
PitchSatisfactionScore           3.0
OwnCar                           1.0
NumberOfChildrenVisiting         1.0
MonthlyIncome                20689.0
dtype: float64
```

In [67]:
```
df.iloc[[385, 2829], [13]] = 3.0
```

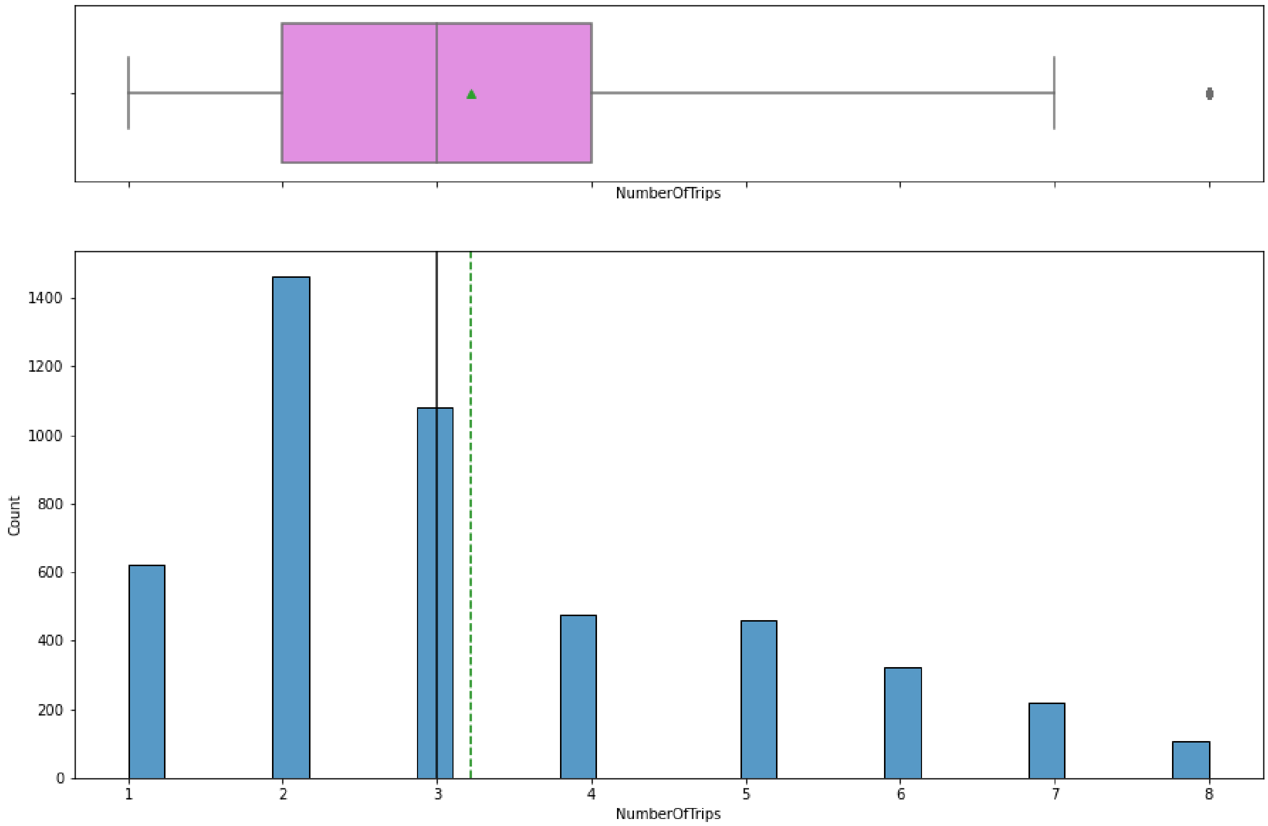
In [68]:
```
gk = df.groupby("Designation")
gk.get_group("Manager").median()
```

Out[68]:
```
CustomerID                  202441.5
ProdTaken                        0.0
Age                             36.0
CityTier                         1.0
DurationOfPitch                 14.0
NumberOfPersonVisiting           3.0
NumberOfFollowups                4.0
PreferredPropertyStar            3.0
NumberOfTrips                    3.0
Passport                         0.0
PitchSatisfactionScore           3.0
```

```
OwnCar                         1.0
NumberOfChildrenVisiting       1.0
MonthlyIncome              22922.0
dtype: float64
```

In [69]:
```
df.iloc[[816, 3260], [13]] = 3.0
```

In [70]:
```
histogram_boxplot(df, "NumberOfTrips")
```



MonthlyIncome

In [71]:
```
gk = df.groupby("MonthlyIncome")
gk.get_group(95000.0)
```

Out[71]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender
38	200038	0	36.0	Self Enquiry	1	11.0	Salaried	Female

In [72]:
```
gk = df.groupby("MonthlyIncome")
gk.get_group(98678.0)
```

Out[72]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
2482	202482	0	37.0	Self Enquiry	1	12.0	Salaried	Femal

- These two were pitched "Basic" and has "Executive" position, so will change MonthlyIncome to the median salary of the "Executive" position.

In [73]:
```
gk = df.groupby("MonthlyIncome")
gk.get_group(1000.0)
```

Out[73]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender
142	200142	0	38.0	Self Enquiry	1	9.0	Large Business	Female

In [74]:
```
gk = df.groupby("MonthlyIncome")
gk.get_group(4678.0)
```

Out[74]:

	CustomerID	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gende
2586	202586	0	39.0	Self Enquiry	1	10.0	Large Business	Femal

- These two were pitched "Deluxe" and has "Manager" position, so will change MonthlyIncome to the median salary of the "Manager" position.

In [75]:
```
gk = df.groupby("Designation")
gk.get_group("Executive").median()
```

Out[75]:
```
CustomerID                 202444.5
ProdTaken                       0.0
Age                            32.0
CityTier                        1.0
DurationOfPitch                13.0
NumberOfPersonVisiting          3.0
NumberOfFollowups               4.0
PreferredPropertyStar           3.0
NumberOfTrips                   3.0
Passport                        0.0
PitchSatisfactionScore          3.0
OwnCar                          1.0
NumberOfChildrenVisiting        1.0
MonthlyIncome               20689.0
dtype: float64
```

In [76]:
```
df.iloc[[38, 2482], [19]] = 20689.0
```

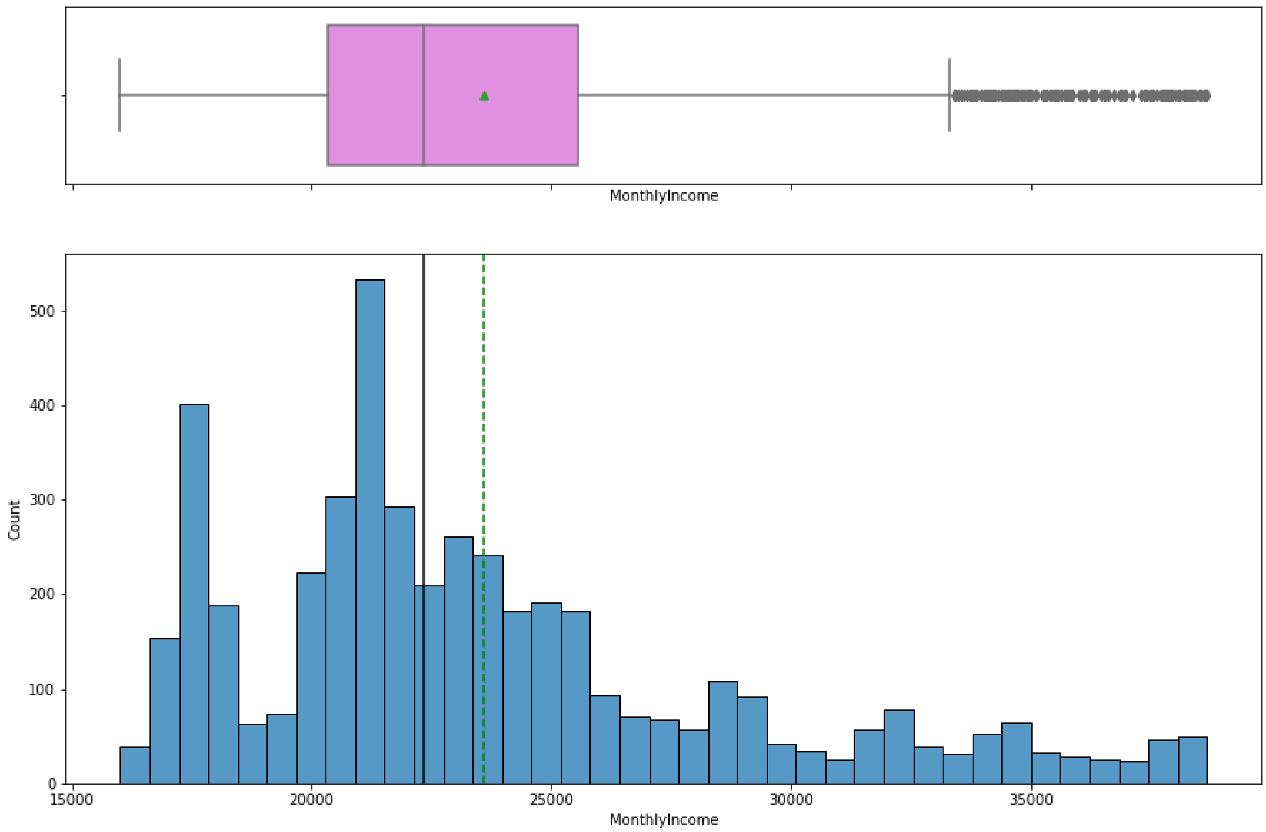
In [77]:
```
gk = df.groupby("Designation")
gk.get_group("Manager").median()
```

Out[77]:
```
CustomerID                 202441.5
ProdTaken                       0.0
Age                            36.0
CityTier                        1.0
DurationOfPitch                14.0
NumberOfPersonVisiting          3.0
NumberOfFollowups               4.0
PreferredPropertyStar           3.0
```

```
NumberOfTrips                  3.0
Passport                       0.0
PitchSatisfactionScore         3.0
OwnCar                         1.0
NumberOfChildrenVisiting       1.0
MonthlyIncome              22922.0
dtype: float64
```

In [78]:
```
df.iloc[[142, 2586], [19]] = 22922.0
```

In [79]:
```
histogram_boxplot(df, "MonthlyIncome")
```



Missing Value Treatment

In [80]:
```
df.isnull().sum()
```

Out[80]:
```
CustomerID                  0
ProdTaken                   0
Age                       226
TypeofContact              25
CityTier                    0
DurationOfPitch           251
Occupation                  0
Gender                      0
NumberOfPersonVisiting      0
NumberOfFollowups          45
ProductPitched              0
PreferredPropertyStar      26
MaritalStatus               0
NumberOfTrips             140
Passport                    0
```

```
PitchSatisfactionScore        0
OwnCar                        0
NumberOfChildrenVisiting     66
Designation                   0
MonthlyIncome               233
dtype: int64
```

In [81]:
```
df["Age"].fillna(df["Age"].median(), inplace=True)
```

In [82]:
```
df["TypeofContact"].fillna("Self Enquiry", inplace=True)
```

In [83]:
```
df["DurationOfPitch"].fillna(df["DurationOfPitch"].median(), inplace=True)
```

In [84]:
```
df["NumberOfFollowups"].fillna(df["NumberOfFollowups"].median(), inplace=True)
```

In [85]:
```
df["PreferredPropertyStar"].fillna(df["PreferredPropertyStar"].median(), inplace
```

In [86]:
```
df["NumberOfTrips"].fillna(df["NumberOfTrips"].median(), inplace=True)
```

In [87]:
```
df["NumberOfChildrenVisiting"].fillna(1.0, inplace=True)
```

In [88]:
```
df["MonthlyIncome"].fillna(df["MonthlyIncome"].median(), inplace=True)
```

In [89]:
```
df.isnull().sum()
```

Out[89]:
```
CustomerID                  0
ProdTaken                   0
Age                         0
TypeofContact               0
CityTier                    0
DurationOfPitch             0
Occupation                  0
Gender                      0
NumberOfPersonVisiting      0
NumberOfFollowups           0
ProductPitched              0
PreferredPropertyStar       0
MaritalStatus               0
NumberOfTrips               0
Passport                    0
PitchSatisfactionScore      0
OwnCar                      0
NumberOfChildrenVisiting    0
Designation                 0
MonthlyIncome               0
dtype: int64
```

Data Cleaning

Gender typo

In [90]:
```
my_tab = pd.crosstab(index=df["Gender"], columns="count")
my_tab
```

Out[90]:

col_0	count
Gender	
Fe Male	155
Female	1817
Male	2916

In [91]:
```
def changer(x):
    if x == "Fe Male":
        return "Female"
    elif x == "Male":
        return "Male"
    else:
        return "Female"


df["Gender"] = df.Gender.apply(changer)
```

In [92]:
```
my_tab = pd.crosstab(index=df["Gender"], columns="count")
my_tab
```

Out[92]:

col_0	count
Gender	
Female	1972
Male	2916

Preparing data for modeling

Dummy Variables

In [93]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4888 entries, 0 to 4887
Data columns (total 20 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   CustomerID                4888 non-null   int64
 1   ProdTaken                 4888 non-null   int64
 2   Age                       4888 non-null   float64
 3   TypeofContact             4888 non-null   object
 4   CityTier                  4888 non-null   int64
 5   DurationOfPitch           4888 non-null   float64
 6   Occupation                4888 non-null   object
 7   Gender                    4888 non-null   object
```

```
 8   NumberOfPersonVisiting    4888 non-null   int64  
 9   NumberOfFollowups         4888 non-null   float64
 10  ProductPitched            4888 non-null   object 
 11  PreferredPropertyStar     4888 non-null   float64
 12  MaritalStatus             4888 non-null   object 
 13  NumberOfTrips             4888 non-null   float64
 14  Passport                  4888 non-null   int64  
 15  PitchSatisfactionScore    4888 non-null   int64  
 16  OwnCar                    4888 non-null   int64  
 17  NumberOfChildrenVisiting  4888 non-null   float64
 18  Designation               4888 non-null   object 
 19  MonthlyIncome             4888 non-null   float64
dtypes: float64(7), int64(7), object(6)
memory usage: 763.9+ KB
```

In [94]:
```
df1 = pd.get_dummies(
    df,
    columns=[
        "TypeofContact",
        "Occupation",
        "Gender",
        "ProductPitched",
        "MaritalStatus",
        "Designation",
    ],
    drop_first=True,
)
```

In [95]:
```
df1.drop(["CustomerID"], inplace=True, axis=1)
```

- Dropped "CustomerID" as it is usless.

In [96]:
```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4888 entries, 0 to 4887
Data columns (total 29 columns):
 #   Column                       Non-Null Count  Dtype  
---  ------                       --------------  -----  
 0   ProdTaken                    4888 non-null   int64  
 1   Age                          4888 non-null   float64
 2   CityTier                     4888 non-null   int64  
 3   DurationOfPitch              4888 non-null   float64
 4   NumberOfPersonVisiting       4888 non-null   int64  
 5   NumberOfFollowups            4888 non-null   float64
 6   PreferredPropertyStar        4888 non-null   float64
 7   NumberOfTrips                4888 non-null   float64
 8   Passport                     4888 non-null   int64  
 9   PitchSatisfactionScore       4888 non-null   int64  
 10  OwnCar                       4888 non-null   int64  
 11  NumberOfChildrenVisiting     4888 non-null   float64
 12  MonthlyIncome                4888 non-null   float64
 13  TypeofContact_Self Enquiry   4888 non-null   uint8  
 14  Occupation_Large Business    4888 non-null   uint8  
 15  Occupation_Salaried          4888 non-null   uint8  
 16  Occupation_Small Business    4888 non-null   uint8  
 17  Gender_Male                  4888 non-null   uint8  
 18  ProductPitched_Deluxe        4888 non-null   uint8  
 19  ProductPitched_King          4888 non-null   uint8  
```

```
 20  ProductPitched_Standard      4888 non-null   uint8
 21  ProductPitched_Super Deluxe  4888 non-null   uint8
 22  MaritalStatus_Married        4888 non-null   uint8
 23  MaritalStatus_Single         4888 non-null   uint8
 24  MaritalStatus_Unmarried      4888 non-null   uint8
 25  Designation_Executive        4888 non-null   uint8
 26  Designation_Manager          4888 non-null   uint8
 27  Designation_Senior Manager   4888 non-null   uint8
 28  Designation_VP               4888 non-null   uint8
dtypes: float64(7), int64(6), uint8(16)
memory usage: 572.9 KB
```

In [97]:

```
df1.sample(10)
```

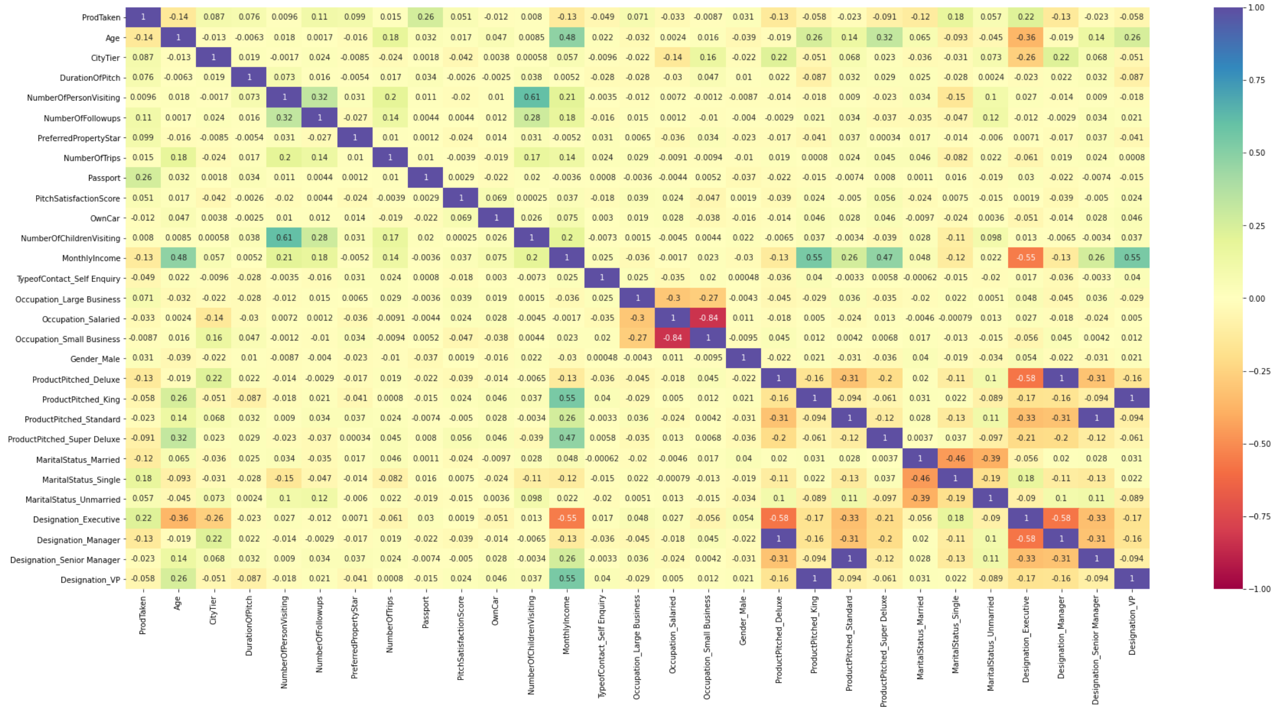
Out[97]:

	ProdTaken	Age	CityTier	DurationOfPitch	NumberOfPersonVisiting	NumberOfFollowups	P
2448	0	28.0	1	9.0	3	4.0	
4233	0	33.0	3	15.0	4	5.0	
2726	0	30.0	3	9.0	3	4.0	
39	0	33.0	3	6.0	2	2.0	
1347	0	36.0	2	8.0	3	4.0	
1924	0	29.0	1	13.0	2	3.0	
4701	0	56.0	1	9.0	4	4.0	
2693	0	46.0	1	14.0	4	3.0	
1211	0	37.0	3	6.0	2	5.0	
1333	1	46.0	3	16.0	3	3.0	

10 rows × 29 columns

In [98]:

```
plt.figure(figsize=(30, 14))
sns.heatmap(df1.corr(), annot=True, vmin=-1, vmax=1, cmap="Spectral")
plt.show()
```

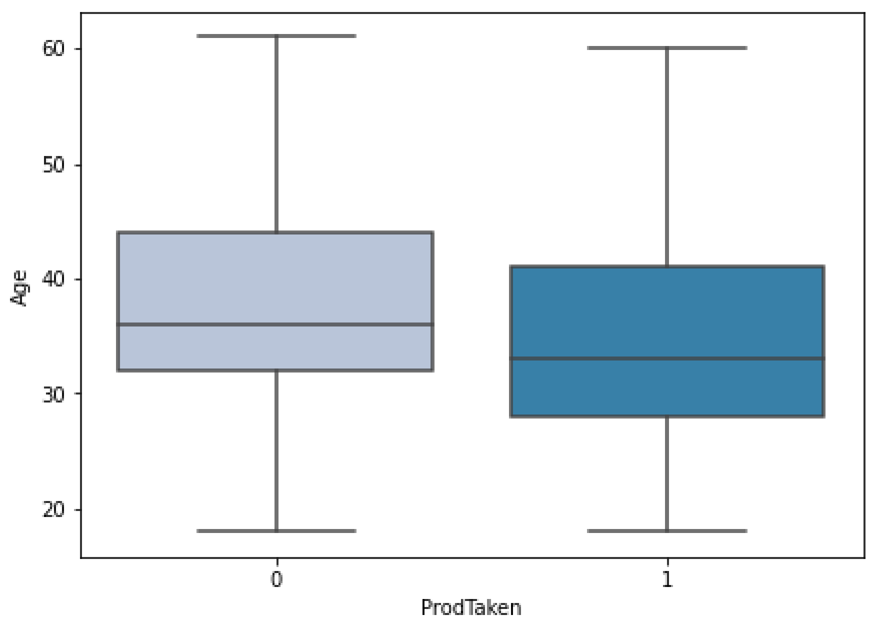


In [99]:

```python
def boxplot(x):
    plt.figure(figsize=(7, 5))
    sns.boxplot(df1["ProdTaken"], x, palette="PuBu")
    plt.show()
```

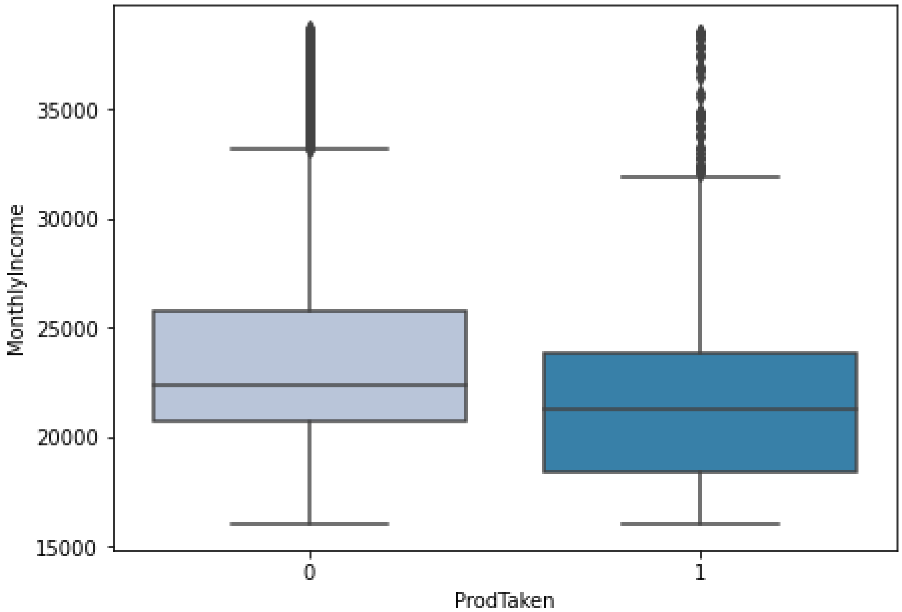
In [100…

```python
boxplot(df1["Age"])
```



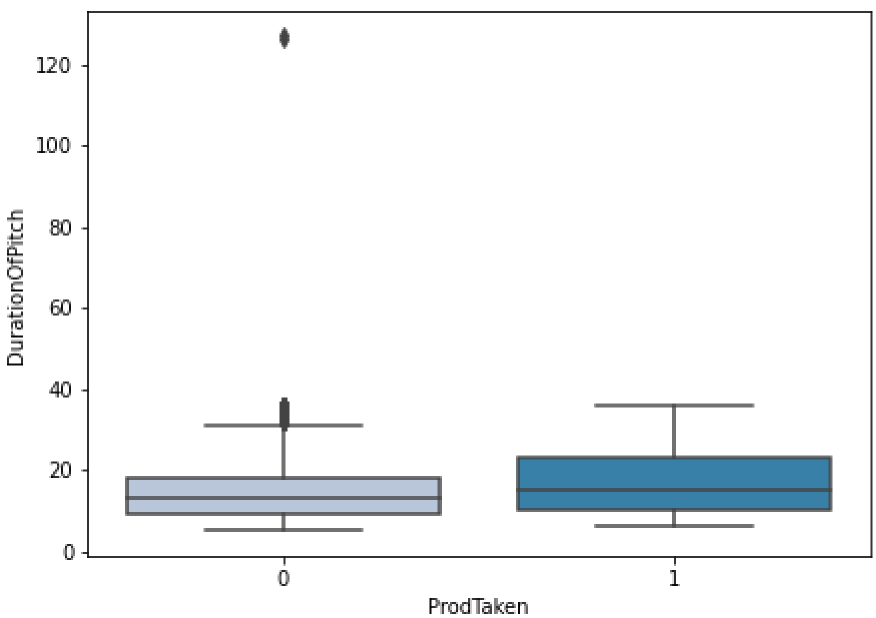
In [101…

```python
boxplot(df1["MonthlyIncome"])
```



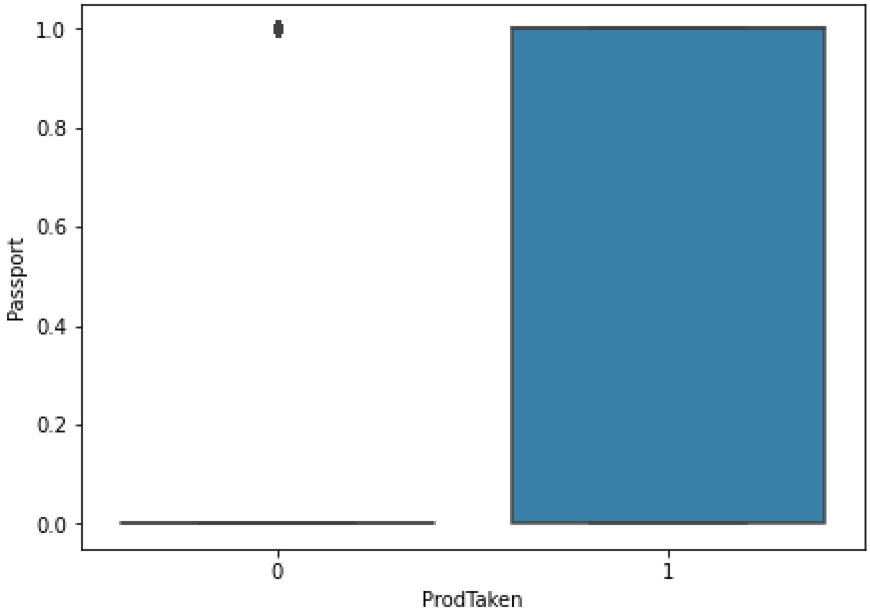
In [102…

```
boxplot(df1["DurationOfPitch"])
```



In [103…

```
boxplot(df1["Passport"])
```



Split Data

In [104…
```
X = df1.drop("ProdTaken", axis=1)
y = df1["ProdTaken"]
```

In [105…
```
# Splitting data into training and test set:
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y
)
print(X_train.shape, X_test.shape)
```

```
(3421, 28) (1467, 28)
```

In [106…
```
print("Number of rows in train data =", X_train.shape[0])
print("Number of rows in test data =", X_test.shape[0])
```

```
Number of rows in train data = 3421
Number of rows in test data = 1467
```

In [107…
```
print("Percentage of classes in training set:")
print(y_train.value_counts(normalize=True))
print("Percentage of classes in test set:")
print(y_test.value_counts(normalize=True))
```

```
Percentage of classes in training set:
0    0.811751
1    0.188249
Name: ProdTaken, dtype: float64
Percentage of classes in test set:
0    0.811861
1    0.188139
Name: ProdTaken, dtype: float64
```

In [108…
```
y.value_counts(1)
```

Out[108…

```
0    0.811784
1    0.188216
Name: ProdTaken, dtype: float64
```

Model Evaluation

Model Prediction Errors

1. Predicting someone bought a travel package, but didn't (FP)
2. Predicting someone did not buy a travel package, but did. (FN)

Which is more important?

1. If the model predicted someone bought a travel package, but didn't, the model is giving a false sense of potential revenue and the marketing team will go after the wrong person.
2. If the model predicted someone not buying a travel package, but did, the model is also not reflective of the potential revenue. However, the marketing team will not spend money on this person.
   - If you advertise to the wrong people, you lose money.
   - If you don't advertise to the right people, you lose potential money.

Which metric to optimize?

I don't know how much anything costs, but to have a successful business, you have to make more money than you lose. Given the circumstance, I will be focusing mainly on the F1-score followed by precision. I think it is slightly more important to minimize false positives, so you can lower needless spending.

Functions for Model Evaluation

In [109…

```
# defining a function to compute different metrics to check performance of a cla
def model_performance_classification_sklearn(model, predictors, target):
    """
    Function to compute different metrics to check classification model performa

    model: classifier
    predictors: independent variables
    target: dependent variable
    """

    # predicting using the independent variables
    pred = model.predict(predictors)

    acc = accuracy_score(target, pred)  # to compute Accuracy
    recall = recall_score(target, pred)  # to compute Recall
    precision = precision_score(target, pred)  # to compute Precision
    f1 = f1_score(target, pred)  # to compute F1-score

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
```

```
        {
            "Accuracy": acc,
            "Recall": recall,
            "Precision": precision,
            "F1": f1,
        },
        index=[0],
    )

    return df_perf
```

In [110…

```
def confusion_matrix_sklearn(model, predictors, target):
    """
    To plot the confusion_matrix with percentages

    model: classifier
    predictors: independent variables
    target: dependent variable
    """
    y_pred = model.predict(predictors)
    cm = confusion_matrix(target, y_pred)
    labels = np.asarray(
        [
            ["{0:0.0f}".format(item) + "\n{0:.2%}".format(item / cm.flatten().su
            for item in cm.flatten()
        ]
    ).reshape(2, 2)

    plt.figure(figsize=(6, 4))
    sns.heatmap(cm, annot=labels, fmt="")
    plt.ylabel("True label")
    plt.xlabel("Predicted label")
```

Decision Tree, Random Forest, Bagging, and Tuned Models

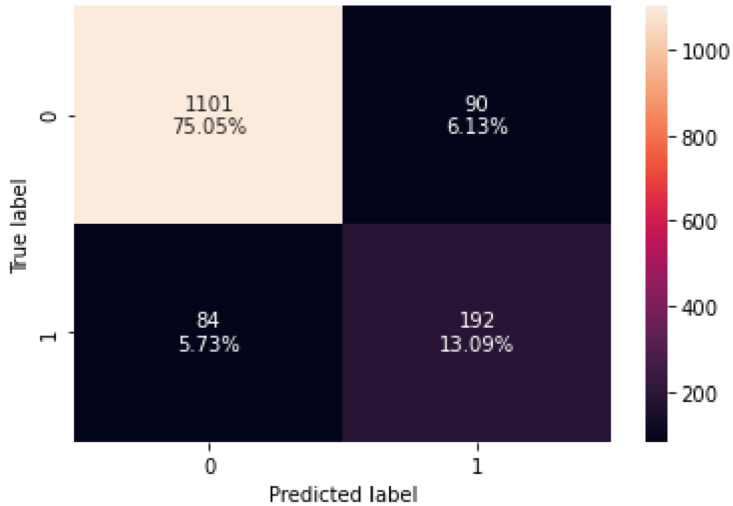
Decision Tree

In [111…

```
# Fitting the model
d_tree = DecisionTreeClassifier(random_state=1)
d_tree.fit(X_train, y_train)

# Calculating different metrics
dtree_model_train_perf = model_performance_classification_sklearn(
    d_tree, X_train, y_train
)
print("Training performance:\n", dtree_model_train_perf)
dtree_model_test_perf = model_performance_classification_sklearn(d_tree, X_test,
print("Testing performance:\n", dtree_model_test_perf)
# Creating confusion matrix
confusion_matrix_sklearn(d_tree, X_test, y_test)
```

```
Training performance:
    Accuracy   Recall   Precision    F1
```

```
0        1.0      1.0         1.0  1.0
Testing performance:
    Accuracy    Recall  Precision        F1
0   0.881391  0.695652   0.680851  0.688172
```



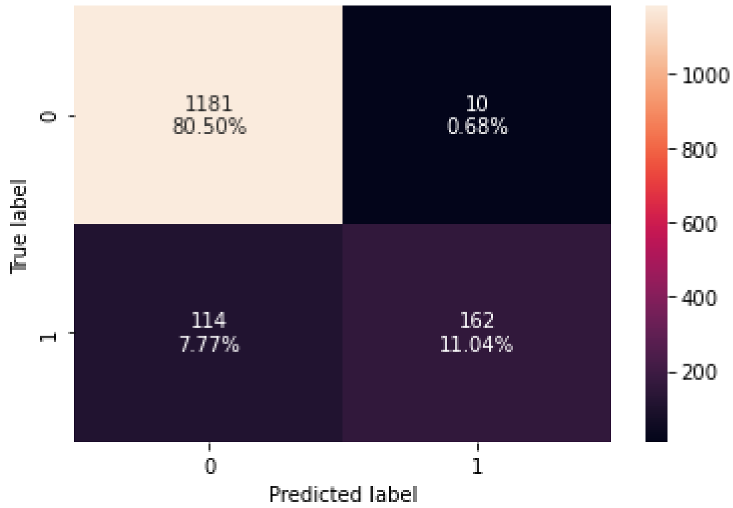
- The Decision Tree is overfitting.

Random Forest

In [112…

```python
# Fitting the model
rf_estimator = RandomForestClassifier(random_state=1)
rf_estimator.fit(X_train, y_train)

# Calculating different metrics
rf_estimator_model_train_perf = model_performance_classification_sklearn(
    rf_estimator, X_train, y_train
)
print("Training performance:\n", rf_estimator_model_train_perf)
rf_estimator_model_test_perf = model_performance_classification_sklearn(
    rf_estimator, X_test, y_test
)
print("Testing performance:\n", rf_estimator_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(rf_estimator, X_test, y_test)
```

```
Training performance:
    Accuracy  Recall  Precision   F1
0        1.0     1.0        1.0  1.0
Testing performance:
    Accuracy    Recall  Precision        F1
0   0.915474  0.586957    0.94186  0.723214
```



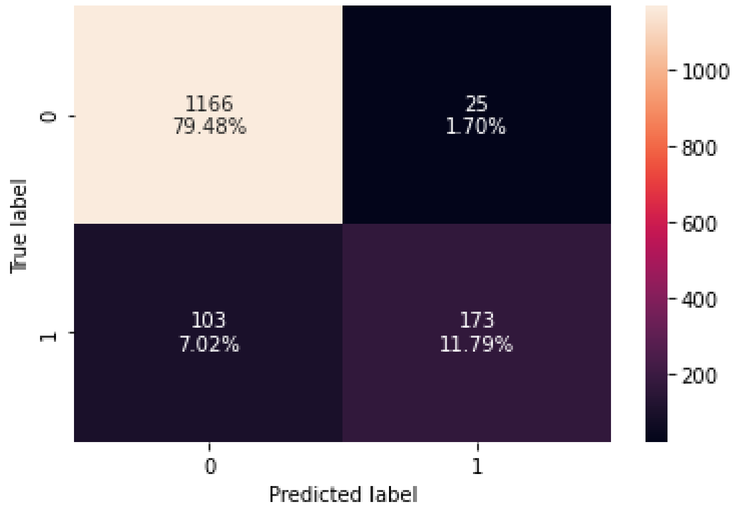
- Random Forest is also overfitting
- Has a lot better precision than decision tree.

Bagging Classifier

In [113…

```
# Fitting the model
bagging_classifier = BaggingClassifier(random_state=1)
bagging_classifier.fit(X_train, y_train)

# Calculating different metrics
bagging_classifier_model_train_perf = model_performance_classification_sklearn(
    bagging_classifier, X_train, y_train
)
print("Training performance:\n", bagging_classifier_model_train_perf)
bagging_classifier_model_test_perf = model_performance_classification_sklearn(
    bagging_classifier, X_test, y_test
)
print("Testing performance:\n", bagging_classifier_model_test_perf)
# Creating confusion matrix
confusion_matrix_sklearn(bagging_classifier, X_test, y_test)
```

```
Training performance:
    Accuracy    Recall  Precision        F1
0  0.994154  0.97205   0.996815  0.984277
Testing performance:
    Accuracy    Recall  Precision        F1
0  0.912747  0.626812   0.873737  0.729958
```



- Train and Test are closer, but still overfitting.

Bagging Classifier with weighted Decision Tree

In [114…

```
bagging_wt = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(
        criterion="gini", class_weight={0: 0.19, 1: 0.81}, random_state=1
    ),
    random_state=1,
)
bagging_wt.fit(X_train, y_train)
```

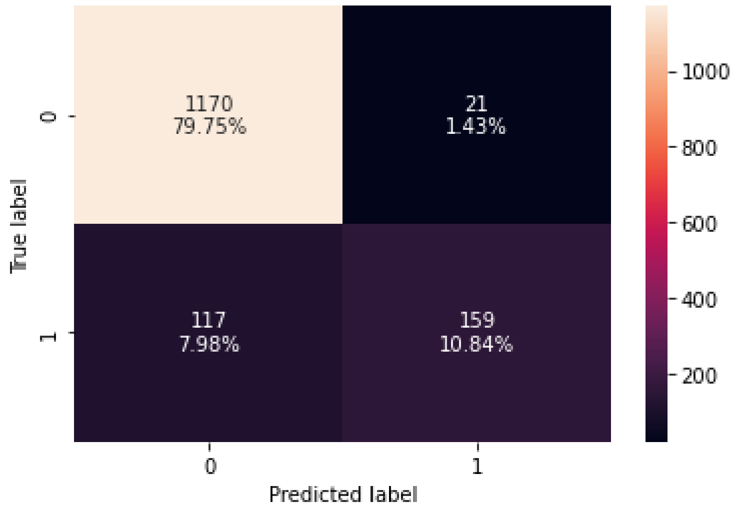
Out[114…

```
BaggingClassifier(base_estimator=DecisionTreeClassifier(class_weight={0: 0.19,
                                                                       1: 0.81},
                                                        random_state=1),
                  random_state=1)
```

In [115…

```
# Calculating different metrics
bagging_wt_classifier_model_train_perf = model_performance_classification_sklear
    bagging_wt, X_train, y_train
)
print("Training performance:\n", bagging_wt_classifier_model_train_perf)
bagging_wt_classifier_model_test_perf = model_performance_classification_sklearn
    bagging_wt, X_test, y_test
)
print("Testing performance:\n", bagging_wt_classifier_model_test_perf)
# Creating confusion matrix
confusion_matrix_sklearn(bagging_wt, X_test, y_test)
```

```
Training performance:
    Accuracy    Recall  Precision        F1
0  0.992985  0.967391   0.995208  0.981102
Testing performance:
    Accuracy    Recall  Precision        F1
0   0.90593  0.576087   0.883333  0.697368
```



Tuned Decision Tree

In [116…

```
# Choose the type of classifier.
dtree_estimator = DecisionTreeClassifier(
    class_weight={0: 0.19, 1: 0.81}, random_state=1
)

# Grid of parameters to choose from
parameters = {
    "max_depth": np.arange(2, 10),
    "min_samples_leaf": [5, 7, 10, 15],
    "max_leaf_nodes": [2, 3, 5, 10, 15],
    "min_impurity_decrease": [0.0001, 0.001, 0.01, 0.1],
}

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.f1_score)

# Run the grid search
grid_obj = GridSearchCV(dtree_estimator, parameters, scoring=scorer, n_jobs=-1)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
dtree_estimator = grid_obj.best_estimator_

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

Out[116…

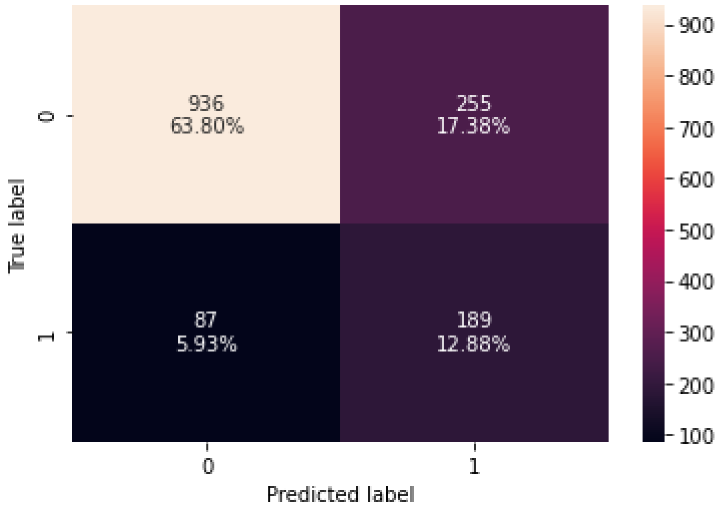
```
DecisionTreeClassifier(class_weight={0: 0.19, 1: 0.81}, max_depth=6,
                       max_leaf_nodes=15, min_impurity_decrease=0.0001,
                       min_samples_leaf=5, random_state=1)
```

In [117…

```
# Calculating different metrics
dtree_estimator_model_train_perf = model_performance_classification_sklearn(
    dtree_estimator, X_train, y_train
)
print("Training performance:\n", dtree_estimator_model_train_perf)
dtree_estimator_model_test_perf = model_performance_classification_sklearn(
    dtree_estimator, X_test, y_test
)
```

```
print("Testing performance:\n", dtree_estimator_model_test_perf)
# Creating confusion matrix
confusion_matrix_sklearn(dtree_estimator, X_test, y_test)
```

```
Training performance:
    Accuracy    Recall  Precision        F1
0  0.754458  0.690994   0.409761  0.514451
Testing performance:
    Accuracy    Recall  Precision     F1
0  0.766871  0.684783   0.425676  0.525
```



- Model isn't overfitting and actually performing a little better on the test data.
- However, the model is very weak.

Tuned Random Forest

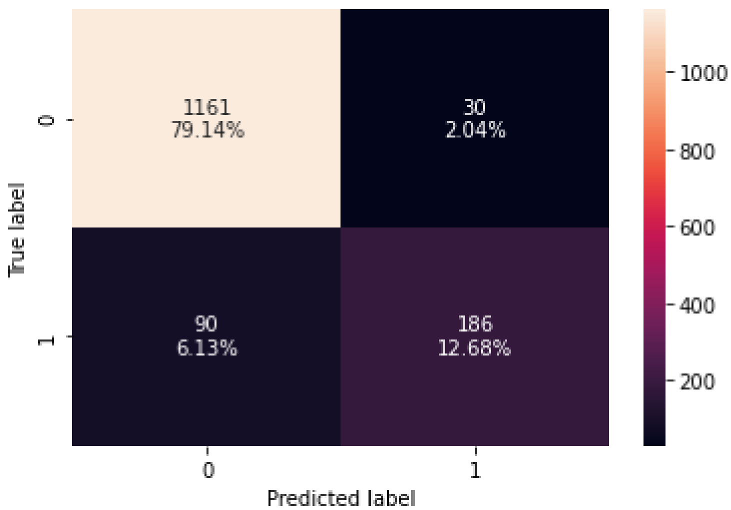
In [118…

```
# Choose the type of classifier.
rf_tuned = RandomForestClassifier(
    class_weight={0: 0.19, 1: 0.81}, random_state=1, oob_score=True, bootstrap=T
)

parameters = {
    "max_depth": list(np.arange(5, 30, 5)) + [None],
    "max_features": ["sqrt", "log2", None],
    "min_samples_leaf": np.arange(1, 15, 5),
    "min_samples_split": np.arange(2, 20, 5),
    "n_estimators": np.arange(10, 110, 10),
}


# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.f1_score)

# Run the grid search
grid_obj = GridSearchCV(rf_tuned, parameters, scoring=scorer, cv=5, n_jobs=-1)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
rf_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
rf_tuned.fit(X_train, y_train)
```

Out[118…
```
RandomForestClassifier(class_weight={0: 0.19, 1: 0.81}, max_depth=20,
                       max_features=None, min_samples_split=7, n_estimators=40,
                       oob_score=True, random_state=1)
```

In [119…
```python
# Calculating different metrics
rf_tuned_model_train_perf = model_performance_classification_sklearn(
    rf_tuned, X_train, y_train
)
print("Training performance:\n", rf_tuned_model_train_perf)
rf_tuned_model_test_perf = model_performance_classification_sklearn(
    rf_tuned, X_test, y_test
)
print("Testing performance:\n", rf_tuned_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(rf_tuned, X_test, y_test)
```

```
Training performance:
    Accuracy  Recall  Precision        F1
0  0.995031     1.0   0.974281  0.986973
Testing performance:
    Accuracy    Recall  Precision        F1
0    0.9182  0.673913   0.861111  0.756098
```



- Tuned random forest is overfitting.

Tuned Bagging Classifier

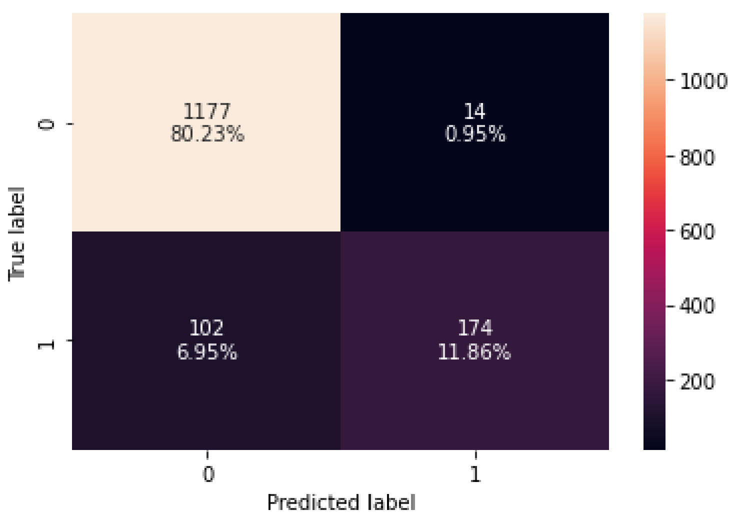
In [120…
```python
# Choose the type of classifier.
bagging_estimator_tuned = BaggingClassifier(random_state=1)

# Grid of parameters to choose from
parameters = {
    "max_samples": [0.7, 0.8, 0.9, 1],
    "max_features": [0.7, 0.8, 0.9, 1],
    "n_estimators": [10, 20, 30, 40, 50],
}

# Type of scoring used to compare parameter combinations
acc_scorer = metrics.make_scorer(metrics.recall_score)
```

```
# Run the grid search
grid_obj = GridSearchCV(bagging_estimator_tuned, parameters, scoring=acc_scorer,
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
bagging_estimator_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
bagging_estimator_tuned.fit(X_train, y_train)
```

Out[120…
```
BaggingClassifier(max_features=0.9, max_samples=0.9, n_estimators=50,
                  random_state=1)
```

In [121…
```
# Calculating different metrics
bagging_estimator_tuned_model_train_perf = model_performance_classification_skle
    bagging_estimator_tuned, X_train, y_train
)
print("Training performance:\n", bagging_estimator_tuned_model_train_perf)
bagging_estimator_tuned_model_test_perf = model_performance_classification_sklea
    bagging_estimator_tuned, X_test, y_test
)
print("Testing performance:\n", bagging_estimator_tuned_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(bagging_estimator_tuned, X_test, y_test)
```

```
Training performance:
    Accuracy  Recall  Precision   F1
0       1.0     1.0        1.0  1.0
Testing performance:
    Accuracy    Recall  Precision    F1
0  0.920927  0.630435   0.925532  0.75
```



- The model is overfitting, but does okay in accuracy and precision.

Comparing Models

In [122…
```
# training performance comparison

models_train_comp_df = pd.concat(
```

```
    [
        dtree_model_train_perf.T,
        dtree_estimator_model_train_perf.T,
        rf_estimator_model_train_perf.T,
        rf_tuned_model_train_perf.T,
        bagging_classifier_model_train_perf.T,
        bagging_estimator_tuned_model_train_perf.T,
        bagging_wt_classifier_model_train_perf.T,
    ],
    axis=1,
)
models_train_comp_df.columns = [
    "Decision Tree",
    "Decision Tree Estimator",
    "Random Forest Estimator",
    "Random Forest Tuned",
    "Bagging Classifier",
    "Bagging Estimator Tuned",
    "Bagging WT",
]
print("Training performance comparison:")
models_train_comp_df
```

```
Training performance comparison:
```

Out[122…

	Decision Tree	Decision Tree Estimator	Random Forest Estimator	Random Forest Tuned	Bagging Classifier	Bagging Estimator Tuned	Bagging WT
Accuracy	1.0	0.754458	1.0	0.995031	0.994154	1.0	0.992985
Recall	1.0	0.690994	1.0	1.000000	0.972050	1.0	0.967391
Precision	1.0	0.409761	1.0	0.974281	0.996815	1.0	0.995208
F1	1.0	0.514451	1.0	0.986973	0.984277	1.0	0.981102

In [123…

```
# testing performance comparison

models_test_comp_df = pd.concat(
    [
        dtree_model_test_perf.T,
        dtree_estimator_model_test_perf.T,
        rf_estimator_model_test_perf.T,
        rf_tuned_model_test_perf.T,
        bagging_classifier_model_test_perf.T,
        bagging_estimator_tuned_model_test_perf.T,
        bagging_wt_classifier_model_test_perf.T,
    ],
    axis=1,
)
models_test_comp_df.columns = [
    "Decision Tree",
    "Decision Tree Estimator",
    "Random Forest Estimator",
    "Random Forest Tuned",
    "Bagging Classifier",
    "Bagging Estimator Tuned",
    "Bagging WT",
]
```

```python
print("Testing performance comparison:")
models_test_comp_df
```

Testing performance comparison:

Out[123…

	Decision Tree	Decision Tree Estimator	Random Forest Estimator	Random Forest Tuned	Bagging Classifier	Bagging Estimator Tuned	Bagging WT
Accuracy	0.881391	0.766871	0.915474	0.918200	0.912747	0.920927	0.905930
Recall	0.695652	0.684783	0.586957	0.673913	0.626812	0.630435	0.576087
Precision	0.680851	0.425676	0.941860	0.861111	0.873737	0.925532	0.883333
F1	0.688172	0.525000	0.723214	0.756098	0.729958	0.750000	0.697368

All models are overfitting, but I'd probably go with the Tuned Bagging Estimator because it has the best F1 and precision score with a decent recall. The Tuned Random Forest is a close second.

Feature Importance of Tuned Random Forest

In [124…

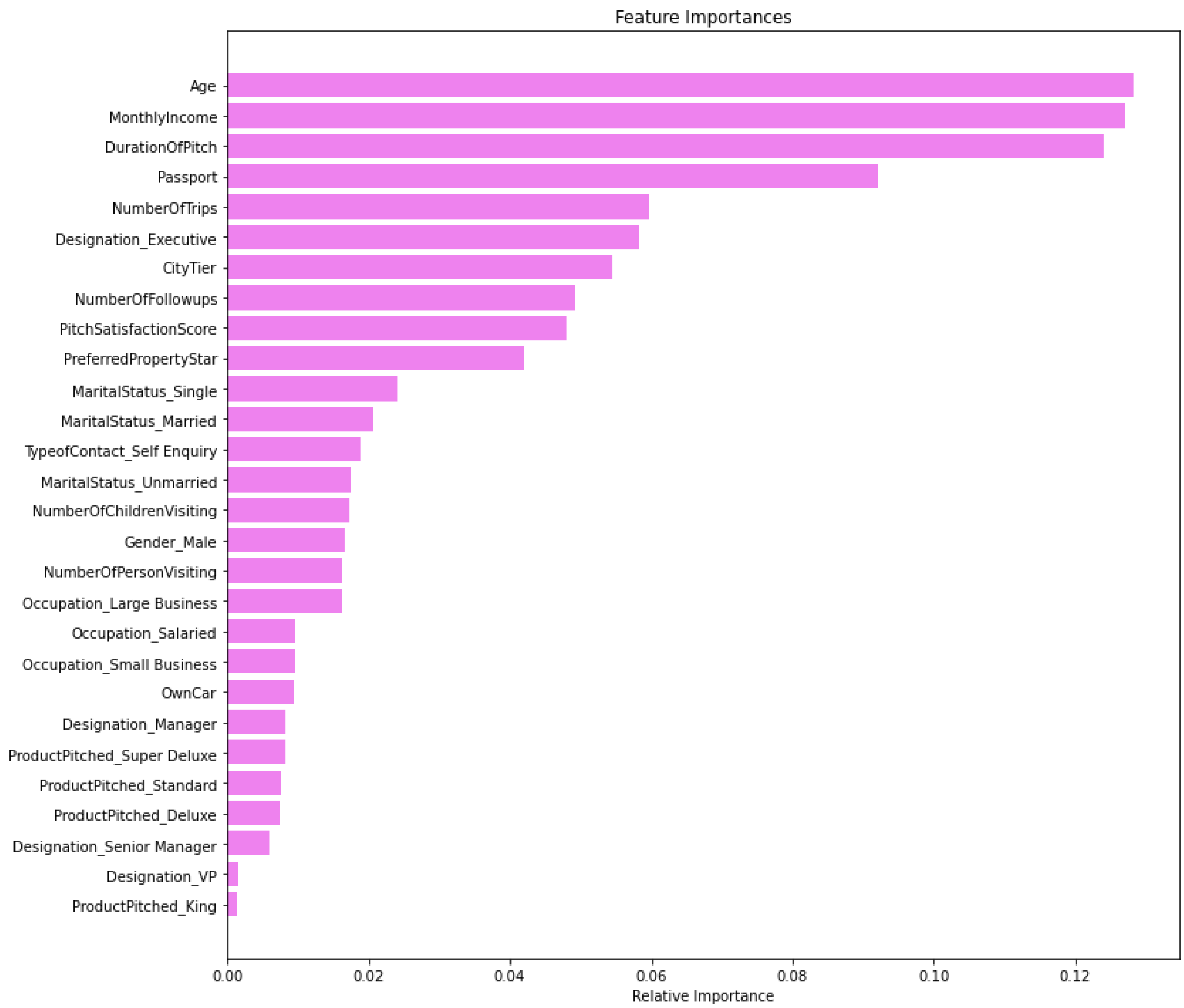
```python
print(
    pd.DataFrame(
        rf_tuned.feature_importances_, columns=["Imp"], index=X_train.columns
    ).sort_values(by="Imp", ascending=False)
)
```

```
                                Imp
Age                        0.128213
MonthlyIncome              0.126999
DurationOfPitch            0.124031
Passport                   0.092030
NumberOfTrips              0.059737
Designation_Executive      0.058289
CityTier                   0.054476
NumberOfFollowups          0.049232
PitchSatisfactionScore     0.047970
PreferredPropertyStar      0.042039
MaritalStatus_Single       0.024014
MaritalStatus_Married      0.020730
TypeofContact_Self Enquiry 0.018915
MaritalStatus_Unmarried    0.017503
NumberOfChildrenVisiting   0.017334
Gender_Male                0.016597
NumberOfPersonVisiting     0.016295
Occupation_Large Business  0.016172
Occupation_Salaried        0.009717
Occupation_Small Business  0.009707
OwnCar                     0.009458
Designation_Manager        0.008301
ProductPitched_Super Deluxe 0.008248
ProductPitched_Standard    0.007614
ProductPitched_Deluxe      0.007396
Designation_Senior Manager 0.006039
Designation_VP             0.001632
ProductPitched_King        0.001312
```

In [125…

```python
feature_names = X_train.columns
importances = rf_tuned.feature_importances_
indices = np.argsort(importances)
```

```
plt.figure(figsize=(12, 12))
plt.title("Feature Importances")
plt.barh(range(len(indices)), importances[indices], color="violet", align="cente
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel("Relative Importance")
plt.show()
```



- Age, MonthlyIncome, DurationOfPitch, and Passport are the four most important features when it comes to purchasing a travel package.

Boosting: AdaBoost, Gradient Boost, XGBoost, Tuned Versions, and Stacking

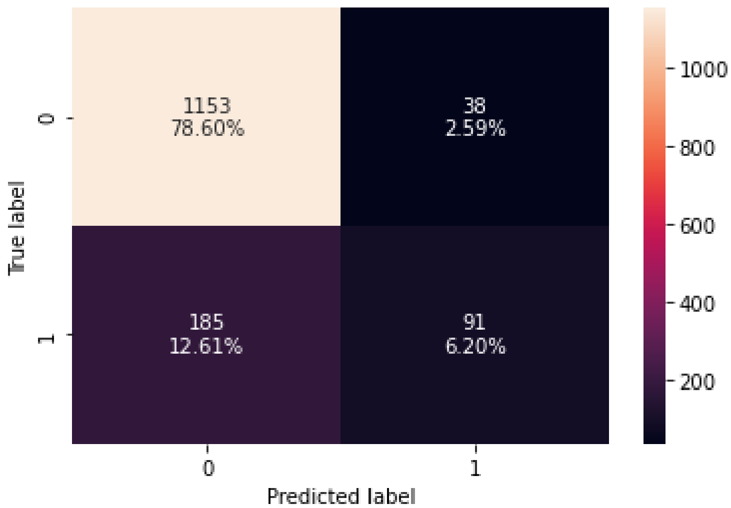
AdaBoost

In [126…

```
# Fitting the model
ab_classifier = AdaBoostClassifier(random_state=1)
ab_classifier.fit(X_train, y_train)
```

```
# Calculating different metrics
ab_classifier_model_train_perf = model_performance_classification_sklearn(
    ab_classifier, X_train, y_train
)
print(ab_classifier_model_train_perf)
ab_classifier_model_test_perf = model_performance_classification_sklearn(
    ab_classifier, X_test, y_test
)
print(ab_classifier_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(ab_classifier, X_test, y_test)
```

```
   Accuracy   Recall  Precision        F1
0  0.845367  0.32764   0.687296  0.443743
   Accuracy   Recall  Precision        F1
0  0.847989  0.32971   0.705426  0.449383
```



- Model is not overfitting, but very weak in terms of recall and F1.

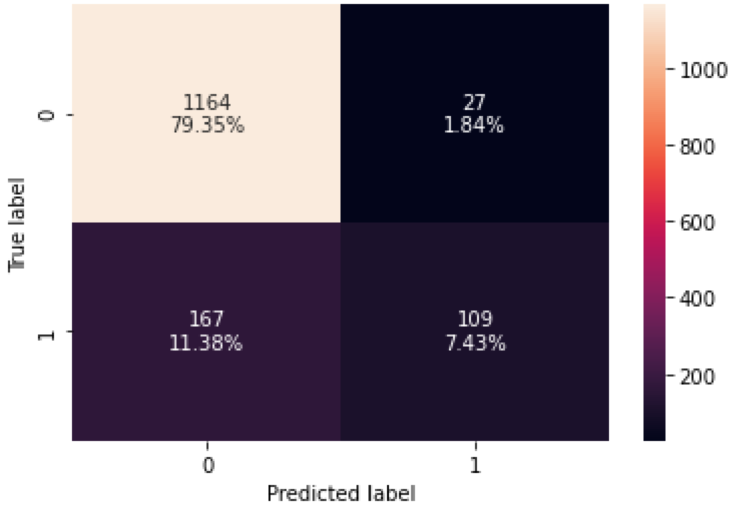
Gradient Boost

In [127…

```
# Fitting the model
gb_classifier = GradientBoostingClassifier(random_state=1)
gb_classifier.fit(X_train, y_train)

# Calculating different metrics
gb_classifier_model_train_perf = model_performance_classification_sklearn(
    gb_classifier, X_train, y_train
)
print("Training performance:\n", gb_classifier_model_train_perf)
gb_classifier_model_test_perf = model_performance_classification_sklearn(
    gb_classifier, X_test, y_test
)
print("Testing performance:\n", gb_classifier_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(gb_classifier, X_test, y_test)
```

```
Training performance:
    Accuracy    Recall  Precision        F1
0  0.883952  0.444099       0.88  0.590299
```

```
Testing performance:
    Accuracy    Recall  Precision        F1
0  0.867757  0.394928   0.801471  0.529126
```



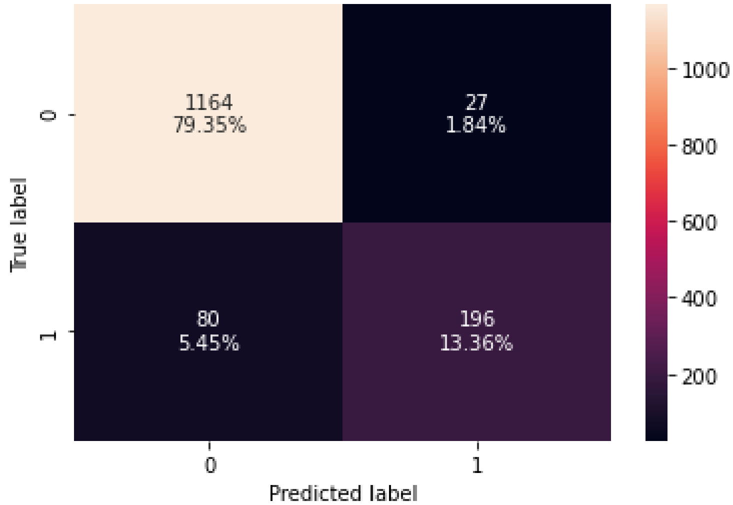
- Model is not really overfitting.
- Recall, precision, and F1 are better.

XGBoost

In [128…

```
# Fitting the model
xgb_classifier = XGBClassifier(random_state=1, eval_metric="logloss")
xgb_classifier.fit(X_train, y_train)

# Calculating different metrics
xgb_classifier_model_train_perf = model_performance_classification_sklearn(
    xgb_classifier, X_train, y_train
)
print("Training performance:\n", xgb_classifier_model_train_perf)
xgb_classifier_model_test_perf = model_performance_classification_sklearn(
    xgb_classifier, X_test, y_test
)
print("Testing performance:\n", xgb_classifier_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(xgb_classifier, X_test, y_test)
```

```
Training performance:
    Accuracy    Recall  Precision        F1
0  0.999415  0.996894        1.0  0.998445
Testing performance:
    Accuracy    Recall  Precision        F1
0  0.927062  0.710145   0.878924  0.785571
```



- Model looks to be overfitting, but test performance is better than previous models.

Tuned AdaBoost

In [129…

```
# Choose the type of classifier.
abc_tuned = AdaBoostClassifier(random_state=1)

# Grid of parameters to choose from
parameters = {
    # Let's try different max_depth for base_estimator
    "base_estimator": [
        DecisionTreeClassifier(max_depth=1),
        DecisionTreeClassifier(max_depth=2),
        DecisionTreeClassifier(max_depth=3),
    ],
    "n_estimators": np.arange(10, 110, 10),
    "learning_rate": np.arange(0.1, 2, 0.1),
}

# Type of scoring used to compare parameter  combinations
scorer = metrics.make_scorer(metrics.f1_score)

# Run the grid search
grid_obj = GridSearchCV(abc_tuned, parameters, scoring=scorer, cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
abc_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
abc_tuned.fit(X_train, y_train)
```

Out[129…

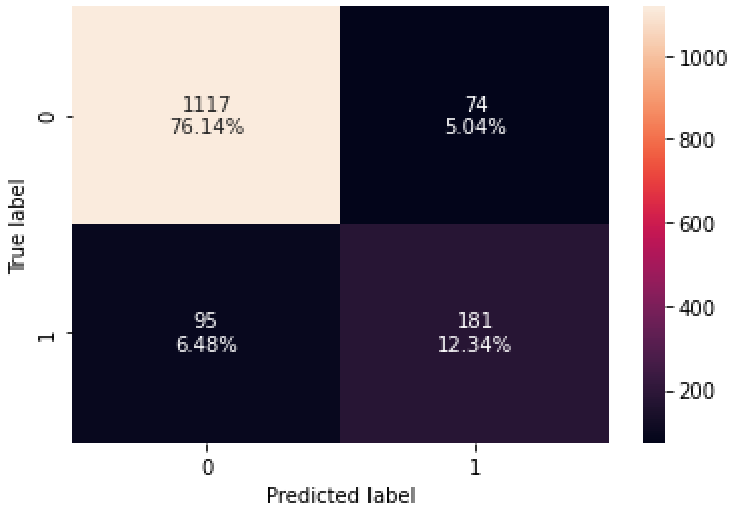
```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=3),
                   learning_rate=1.1, n_estimators=100, random_state=1)
```

In [130…

```
# Calculating different metrics
abc_tuned_model_train_perf = model_performance_classification_sklearn(
    abc_tuned, X_train, y_train
)
print(abc_tuned_model_train_perf)
```

```
abc_tuned_model_test_perf = model_performance_classification_sklearn(
    abc_tuned, X_test, y_test
)
print(abc_tuned_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(abc_tuned, X_test, y_test)
```

```
   Accuracy    Recall  Precision        F1
0  0.989184  0.951863   0.990307  0.970705
   Accuracy    Recall  Precision        F1
0  0.884799  0.655797   0.709804  0.681733
```



- Model is overfitting, but is stronger than the untuned AdaBoost.

Tuned Gradient Boost

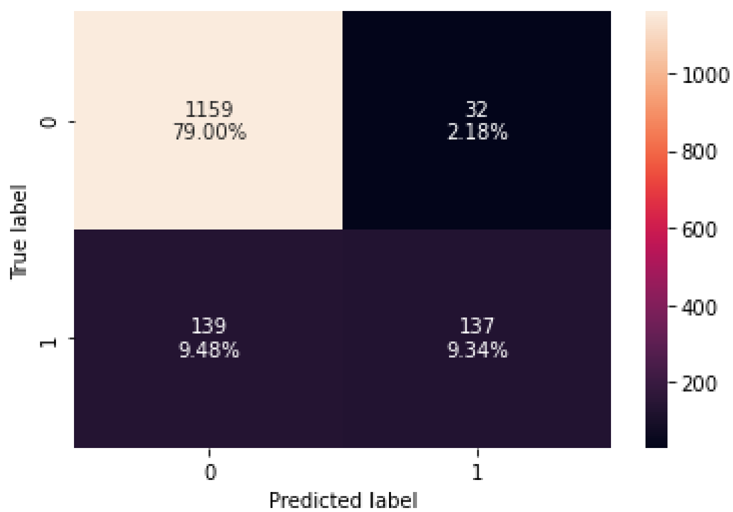
In [131…

```
# Choose the type of classifier.
gbc_tuned = GradientBoostingClassifier(
    init=AdaBoostClassifier(random_state=1), random_state=1
)

# Grid of parameters to choose from
parameters = {
    "n_estimators": [100, 150, 200, 250],
    "subsample": [0.8, 0.9, 1],
    "max_features": [0.7, 0.8, 0.9, 1],
}

# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.f1_score)

# Run the grid search
grid_obj = GridSearchCV(gbc_tuned, parameters, scoring=scorer, cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
gbc_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
gbc_tuned.fit(X_train, y_train)
```

Out[131…
```
GradientBoostingClassifier(init=AdaBoostClassifier(random_state=1),
                           max_features=0.8, n_estimators=250, random_state=1,
                           subsample=0.9)
```

In [132…
```
# Calculating different metrics
gbc_tuned_model_train_perf = model_performance_classification_sklearn(
    gbc_tuned, X_train, y_train
)
print("Training performance:\n", gbc_tuned_model_train_perf)
gbc_tuned_model_test_perf = model_performance_classification_sklearn(
    gbc_tuned, X_test, y_test
)
print("Testing performance:\n", gbc_tuned_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(gbc_tuned, X_test, y_test)
```

```
Training performance:
    Accuracy    Recall  Precision        F1
0  0.920491  0.613354   0.944976  0.743879
Testing performance:
    Accuracy    Recall  Precision       F1
0  0.883436  0.496377   0.810651  0.61573
```



- Model is overfitting, but stronger than the untuned gradient boost.

Tuned XGBoost

In [133…
```
# Choose the type of classifier.
xgb_tuned = XGBClassifier(random_state=1, eval_metric="logloss")

# Grid of parameters to choose from
parameters = {
    "n_estimators": [10, 30, 50],
    "scale_pos_weight": [1, 2, 5],
    "subsample": [0.7, 0.9, 1],
    "learning_rate": [0.05, 0.1, 0.2],
    "colsample_bytree": [0.7, 0.9, 1],
    "colsample_bylevel": [0.5, 0.7, 1],
    "max_depth": [3, 6, 9],
}
```

```
# Type of scoring used to compare parameter combinations
scorer = metrics.make_scorer(metrics.f1_score)

# Run the grid search
grid_obj = GridSearchCV(xgb_tuned, parameters, scoring=scorer, cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
xgb_tuned = grid_obj.best_estimator_

# Fit the best algorithm to the data.
xgb_tuned.fit(X_train, y_train)
```

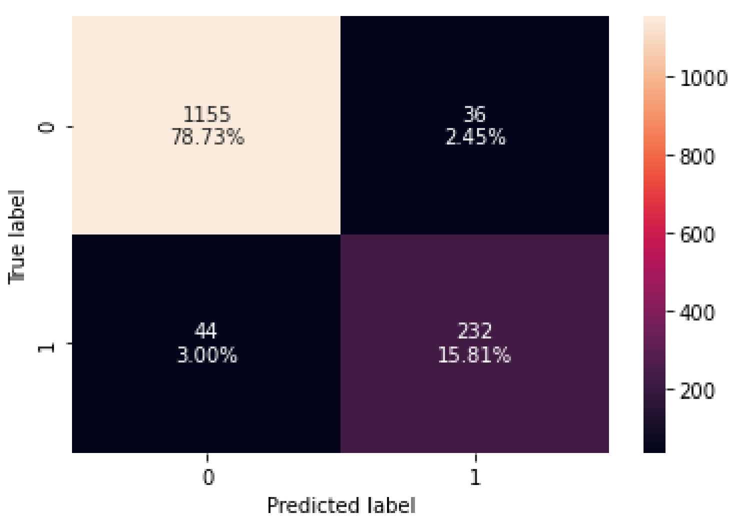
Out[133…

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=0.7,
              colsample_bynode=1, colsample_bytree=0.7, eval_metric='logloss',
              gamma=0, gpu_id=-1, importance_type='gain',
              interaction_constraints='', learning_rate=0.2, max_delta_step=0,
              max_depth=9, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=50, n_jobs=12,
              num_parallel_tree=1, random_state=1, reg_alpha=0, reg_lambda=1,
              scale_pos_weight=5, subsample=1, tree_method='exact',
              validate_parameters=1, verbosity=None)
```

In [134…

```
# Calculating different metrics
xgb_tuned_model_train_perf = model_performance_classification_sklearn(
    xgb_tuned, X_train, y_train
)
print("Training performance:\n", xgb_tuned_model_train_perf)
xgb_tuned_model_test_perf = model_performance_classification_sklearn(
    xgb_tuned, X_test, y_test
)
print("Testing performance:\n", xgb_tuned_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(xgb_tuned, X_test, y_test)
```

```
Training performance:
    Accuracy  Recall  Precision        F1
0  0.999708     1.0    0.99845  0.999224
Testing performance:
    Accuracy   Recall  Precision        F1
0  0.945467  0.84058   0.865672  0.852941
```



- Model is overfitting a little bit, but is better than the untuned XGBoost.

Stacking

In [135…

```
estimators = [
    ("Random Forest Tuned", rf_tuned),
    ("Bagging Estimator Tuned", bagging_estimator_tuned),
    ("XGBoost Classifier", xgb_classifier),
]

final_estimator = xgb_tuned

stacking_classifier = StackingClassifier(
    estimators=estimators, final_estimator=final_estimator
)

stacking_classifier.fit(X_train, y_train)
```

Out[135…

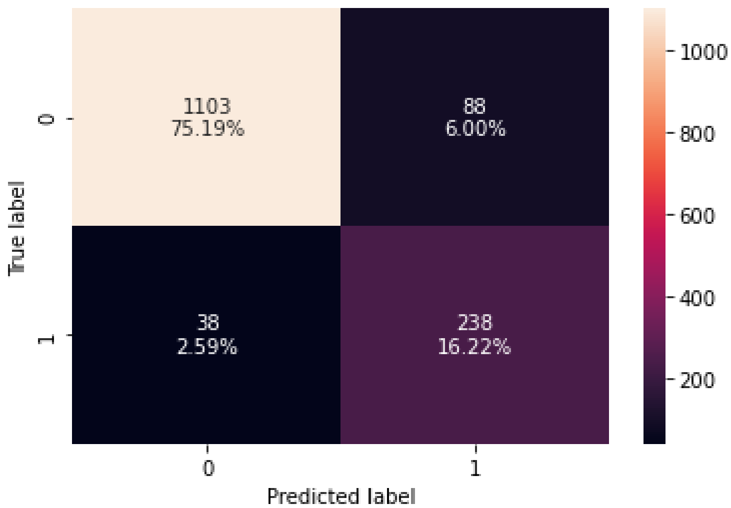
```
StackingClassifier(estimators=[('Random Forest Tuned',
                                RandomForestClassifier(class_weight={0: 0.19,
                                                                     1: 0.81},
                                                       max_depth=20,
                                                       max_features=None,
                                                       min_samples_split=7,
                                                       n_estimators=40,
                                                       oob_score=True,
                                                       random_state=1)),
                               ('Bagging Estimator Tuned',
                                BaggingClassifier(max_features=0.9,
                                                  max_samples=0.9,
                                                  n_estimators=50,
                                                  random_state=1)),
                               ('XGBoost Classifier',
                                XGBClassifier(bas...
                                              eval_metric='logloss', gamma=0,
                                              gpu_id=-1,
                                              importance_type='gain',
                                              interaction_constraints='',
                                              learning_rate=0.2,
                                              max_delta_step=0, max_depth=9,
                                              min_child_weight=1,
                                              missing=nan,
                                              monotone_constraints='()',
                                              n_estimators=50, n_jobs=12,
                                              num_parallel_tree=1,
                                              random_state=1, reg_alpha=0,
                                              reg_lambda=1,
                                              scale_pos_weight=5,
                                              subsample=1,
                                              tree_method='exact',
                                              validate_parameters=1,
                                              verbosity=None))
```

In [136…

```
# Calculating different metrics
stacking_classifier_model_train_perf = model_performance_classification_sklearn(
    stacking_classifier, X_train, y_train
)
print("Training performance:\n", stacking_classifier_model_train_perf)
stacking_classifier_model_test_perf = model_performance_classification_sklearn(
```

```
    stacking_classifier, X_test, y_test
)
print("Testing performance:\n", stacking_classifier_model_test_perf)

# Creating confusion matrix
confusion_matrix_sklearn(stacking_classifier, X_test, y_test)
```

```
Training performance:
   Accuracy  Recall  Precision        F1
0  0.994446     1.0   0.971342  0.985463
Testing performance:
   Accuracy    Recall  Precision        F1
0   0.91411  0.862319   0.730061  0.790698
```



- Model is overfitting and has poor precision performance.

Comparing Models

In [137…

```
# training performance comparison

models_train_comp_df = pd.concat(
    [
        ab_classifier_model_train_perf.T,
        abc_tuned_model_train_perf.T,
        gb_classifier_model_train_perf.T,
        gbc_tuned_model_train_perf.T,
        xgb_classifier_model_train_perf.T,
        xgb_tuned_model_train_perf.T,
        stacking_classifier_model_train_perf.T,
    ],
    axis=1,
)
models_train_comp_df.columns = [
    "Adaboost Classifier",
    "Adabosst Classifier Tuned",
    "Gradient Boost Classifier",
    "Gradient Boost Classifier Tuned",
    "XGBoost Classifier",
    "XGBoost Classifier Tuned",
    "Stacking Classifier",
]
```

```
print("Training performance comparison:")
models_train_comp_df
```

Training performance comparison:

Out[137…

	Adaboost Classifier	Adabosst Classifier Tuned	Gradient Boost Classifier	Gradient Boost Classifier Tuned	XGBoost Classifier	XGBoost Classifier Tuned	Stacking Classifier
Accuracy	0.845367	0.989184	0.883952	0.920491	0.999415	0.999708	0.994446
Recall	0.327640	0.951863	0.444099	0.613354	0.996894	1.000000	1.000000
Precision	0.687296	0.990307	0.880000	0.944976	1.000000	0.998450	0.971342
F1	0.443743	0.970705	0.590299	0.743879	0.998445	0.999224	0.985463

In [138…

```
# test performance comparison

models_test_comp_df = pd.concat(
    [
        ab_classifier_model_test_perf.T,
        abc_tuned_model_test_perf.T,
        gb_classifier_model_test_perf.T,
        gbc_tuned_model_test_perf.T,
        xgb_classifier_model_test_perf.T,
        xgb_tuned_model_test_perf.T,
        stacking_classifier_model_test_perf.T,
    ],
    axis=1,
)
models_test_comp_df.columns = [
    "Adaboost Classifier",
    "Adabosst Classifier Tuned",
    "Gradient Boost Classifier",
    "Gradient Boost Classifier Tuned",
    "XGBoost Classifier",
    "XGBoost Classifier Tuned",
    "Stacking Classifier",
]
print("Testing performance comparison:")
models_test_comp_df
```

Testing performance comparison:

Out[138…

	Adaboost Classifier	Adabosst Classifier Tuned	Gradient Boost Classifier	Gradient Boost Classifier Tuned	XGBoost Classifier	XGBoost Classifier Tuned	Stacking Classifier
Accuracy	0.847989	0.884799	0.867757	0.883436	0.927062	0.945467	0.914110
Recall	0.329710	0.655797	0.394928	0.496377	0.710145	0.840580	0.862319
Precision	0.705426	0.709804	0.801471	0.810651	0.878924	0.865672	0.730061
F1	0.449383	0.681733	0.529126	0.615730	0.785571	0.852941	0.790698

All models are overfitting, but I'd probably go with the Tuned XGBoost Claddifier if I had to pick from these as it has the best F1 and good precision and recall.

Feature Importance of Tuned XGBoost Model

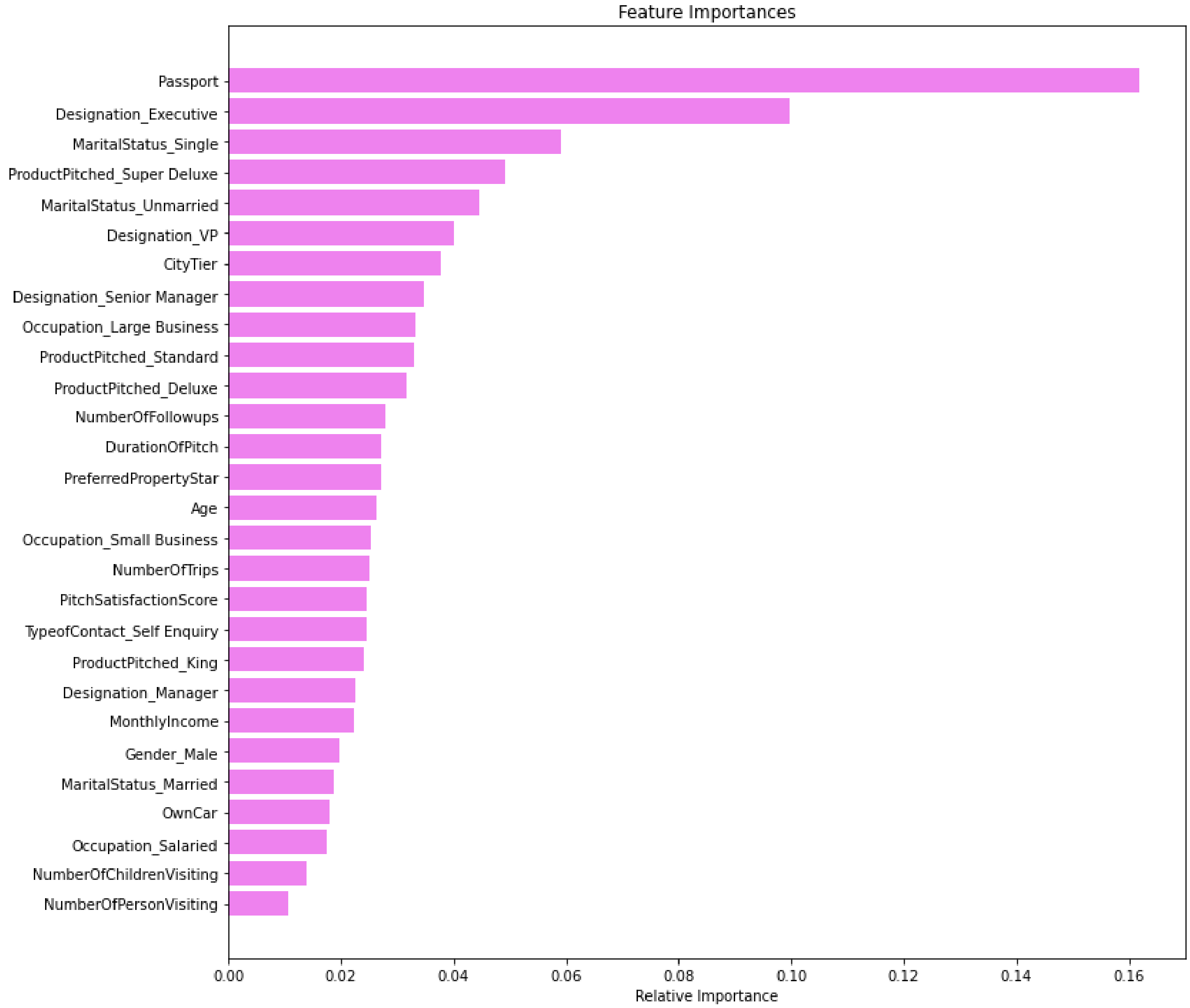
In [139…

```
print(
    pd.DataFrame(
        xgb_tuned.feature_importances_, columns=["Imp"], index=X_train.columns
    ).sort_values(by="Imp", ascending=False)
)
```

```
                                  Imp
Passport                     0.161820
Designation_Executive        0.099646
MaritalStatus_Single         0.058996
ProductPitched_Super Deluxe  0.049183
MaritalStatus_Unmarried      0.044578
Designation_VP               0.040049
CityTier                     0.037761
Designation_Senior Manager   0.034663
Occupation_Large Business    0.033176
ProductPitched_Standard      0.033013
ProductPitched_Deluxe        0.031652
NumberOfFollowups            0.027789
DurationOfPitch              0.027125
PreferredPropertyStar        0.027058
Age                          0.026332
Occupation_Small Business    0.025358
NumberOfTrips                0.025134
PitchSatisfactionScore       0.024557
TypeofContact_Self Enquiry   0.024534
ProductPitched_King          0.024169
Designation_Manager          0.022650
MonthlyIncome                0.022258
Gender_Male                  0.019705
MaritalStatus_Married        0.018686
OwnCar                       0.017951
Occupation_Salaried          0.017611
NumberOfChildrenVisiting     0.013943
NumberOfPersonVisiting       0.010603
```

In [140…

```
feature_names = X_train.columns
importances = xgb_tuned.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(12, 12))
plt.title("Feature Importances")
plt.barh(range(len(indices)), importances[indices], color="violet", align="cente
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel("Relative Importance")
plt.show()
```



- Passport, Designation_Executive, and MaritalStatus_Single are the three most importance features when it comes to purchasing a travel package.

Actionable Insights & Recommendations

The data show that people ~26-40, people with a MonthlyIncome of ~18,000-24,000, and people with a passport have a higher likelihood of buying a travel package.

It may be a good idea to pitch the new Wellness Tourism Package to young people in their late 20s/early 30s, as that age group are likely trying to maintain or even enhance their healthy lifestyle if they have one.

Another age group to pitch the new Wellness Tourism Package is people in their mid-40s/50s. They tend to become more conscious of their health and may want to get into a healthy lifestyle.

For this new Wellness Tourism Package, it may be benificial to collect data on the travel goals of individuals or their active/health level.