Tourism Project / Yeoman Yoon

In [1]:
```
import pandas as pd
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
#model
from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
#Bagging
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
import warnings
warnings.filterwarnings("ignore")
#Boosting
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.ensemble import StackingClassifier
```

Import Data

This file is not CSV. Call "ExcelFile .xlsx" and import with correct sheet to use as the data.

In [2]:
```
excel = pd.ExcelFile('Tourism.xlsx')
data = pd.read_excel(excel, 'Tourism')
```

In [3]:
```
data.head(5)
```

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]:
```
data.shape
```

Out[4]:
```
(4888, 20)
```

Data Dictionary

Customer details:

1. CustomerID: Unique customer ID
2. ProdTaken: Whether the customer has purchased a package or not (0: No, 1: Yes)
3. Age: Age of customer
4. TypeofContact: How customer was contacted (Company Invited or Self Inquiry)
5. 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
6. Occupation: Occupation of customer
7. Gender: Gender of customer
8. NumberOfPersonVisiting: Total number of persons planning to take the trip with the customer
9. PreferredPropertyStar: Preferred hotel property rating by customer
10. MaritalStatus: Marital status of customer
11. NumberOfTrips: Average number of trips in a year by customer
12. Passport: The customer has a passport or not (0: No, 1: Yes)
13. OwnCar: Whether the customers own a car or not (0: No, 1: Yes)
14. NumberOfChildrenVisiting: Total number of children with age less than 5 planning to take the trip with the customer
15. Designation: Designation of the customer in the current organization
16. MonthlyIncome: Gross monthly income of the customer

Customer interaction data:

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

In [5]:
```
data.drop(["CustomerID"], axis=1, inplace = True) # Drop customer ID, we are n
ot using this
```

In [6]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4888 entries, 0 to 4887
Data columns (total 19 columns):
ProdTaken                   4888 non-null int64
Age                         4662 non-null float64
TypeofContact               4863 non-null object
CityTier                    4888 non-null int64
DurationOfPitch             4637 non-null float64
Occupation                  4888 non-null object
Gender                      4888 non-null object
NumberOfPersonVisiting      4888 non-null int64
NumberOfFollowups           4843 non-null float64
ProductPitched              4888 non-null object
PreferredPropertyStar       4862 non-null float64
MaritalStatus               4888 non-null object
NumberOfTrips               4748 non-null float64
Passport                    4888 non-null int64
PitchSatisfactionScore      4888 non-null int64
OwnCar                      4888 non-null int64
NumberOfChildrenVisiting    4822 non-null float64
Designation                 4888 non-null object
MonthlyIncome               4655 non-null float64
dtypes: float64(7), int64(6), object(6)
memory usage: 725.7+ KB
```

Datatype Conversion

For the memory purpose, Convert object to category.

In [7]:
```
data.nunique().sort_values(ascending=False)
```

Out[7]:
```
MonthlyIncome               2475
Age                           44
DurationOfPitch               34
NumberOfTrips                 12
NumberOfFollowups              6
Designation                    5
NumberOfPersonVisiting         5
ProductPitched                 5
PitchSatisfactionScore         5
MaritalStatus                  4
Occupation                     4
NumberOfChildrenVisiting       4
PreferredPropertyStar          3
Gender                         3
CityTier                       3
Passport                       2
OwnCar                         2
TypeofContact                  2
ProdTaken                      2
dtype: int64
```

In [8]:
```
to_category = ['TypeofContact','Occupation','Gender','ProductPitched','Marital
Status','Designation']

for col in to_category:
    data[col]=data[col].astype('category')
```

Null Check

Because we have 8 columns with null problems. Check for the amount of Nulls.
TypeofContact(Only Category) has mode of "Self Enquiry". Fill the rest in with median value.

In [9]:
```
data.isnull().sum().sort_values(ascending=False)
```

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

In [10]:
```
data['TypeofContact'].fillna('Self Enquiry', inplace=True) # Fill with Mode
```

In [11]:
```
data.fillna(data.median(), inplace=True) # Fill with median
```

In [12]:
```
data.isnull().sum().sort_values(ascending=False)
```

Out[12]:
```
MonthlyIncome             0
NumberOfFollowups         0
Age                       0
TypeofContact             0
CityTier                  0
DurationOfPitch           0
Occupation                0
Gender                    0
NumberOfPersonVisiting    0
ProductPitched            0
Designation               0
PreferredPropertyStar     0
MaritalStatus             0
NumberOfTrips             0
Passport                  0
PitchSatisfactionScore    0
OwnCar                    0
NumberOfChildrenVisiting  0
ProdTaken                 0
dtype: int64
```

Error and Outlier Treatment

After soft EDA, couple of errors/outliers were detected.

1. 155 'Fe Male' in Gender Column
2. Only 2 Occupation == 'Free Lancer'
3. Only 4 16000 < MonthlyIncome > 39000 (Considered to be signifcant outliers)
4. Only 2 DurationOfPitch > 36 (Considered to be signifcant outliers)

Fix 'Fe Male', change 'Free Lancer' to 'Small business', and drop 6 outliers above

In [13]:
```
data[data['Gender']=='Fe Male']
```

Out[13]:

	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender	NumberOfF
22	0	34.0	Self Enquiry	1	13.0	Salaried	Fe Male	
55	0	33.0	Company Invited	1	6.0	Salaried	Fe Male	
116	0	34.0	Self Enquiry	1	11.0	Small Business	Fe Male	
131	0	50.0	Company Invited	3	18.0	Small Business	Fe Male	
154	0	32.0	Company Invited	3	14.0	Small Business	Fe Male	
...	...	...	...	...	...	...	...	
4795	0	33.0	Company Invited	1	9.0	Salaried	Fe Male	
4810	0	32.0	Self Enquiry	1	31.0	Small Business	Fe Male	
4811	0	60.0	Self Enquiry	3	10.0	Salaried	Fe Male	
4817	1	30.0	Company Invited	1	17.0	Salaried	Fe Male	
4849	1	33.0	Company Invited	1	10.0	Salaried	Fe Male	

155 rows × 19 columns

In [14]:
```
data[data['Occupation']=='Free Lancer']
```

Out[14]:

	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender	NumberOfF
2	1	37.0	Self Enquiry	1	8.0	Free Lancer	Male	
2446	1	38.0	Self Enquiry	1	9.0	Free Lancer	Male	

In [15]:
```
data[data['MonthlyIncome']<16000]
```

Out[15]:

	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender	NumberOfF
142	0	38.0	Self Enquiry	1	9.0	Large Business	Female	
2586	0	39.0	Self Enquiry	1	10.0	Large Business	Female	

In [16]:
```
data[data['MonthlyIncome']>39000]
```

Out[16]:

	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender	NumberOfF
38	0	36.0	Self Enquiry	1	11.0	Salaried	Female	
2482	0	37.0	Self Enquiry	1	12.0	Salaried	Female	

In [17]:
```
data[data['DurationOfPitch']>36]
```

Out[17]:

	ProdTaken	Age	TypeofContact	CityTier	DurationOfPitch	Occupation	Gender	NumberOfF
1434	0	36.0	Company Invited	3	126.0	Salaried	Male	
3878	0	53.0	Company Invited	3	127.0	Salaried	Male	

In [18]:
```
data['Gender'] = data['Gender'].replace('Fe Male','Female')
data['Occupation'] = data['Occupation'].replace('Free Lancer','Small Busines
s')
data=data.drop([38,142,1434,2482,2586,3878])
```

In [19]:
```
data['Gender'] = data['Gender'].cat.remove_categories('Fe Male')
data['Occupation'] = data['Occupation'].cat.remove_categories('Free Lancer')
```

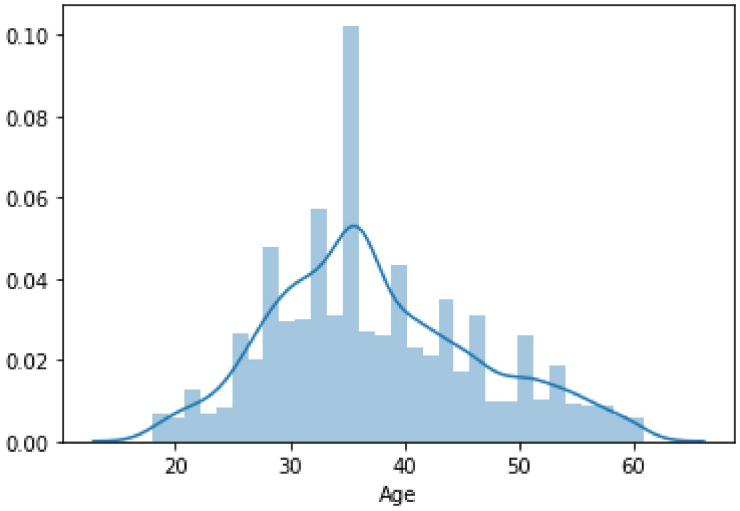
Univariate EDA

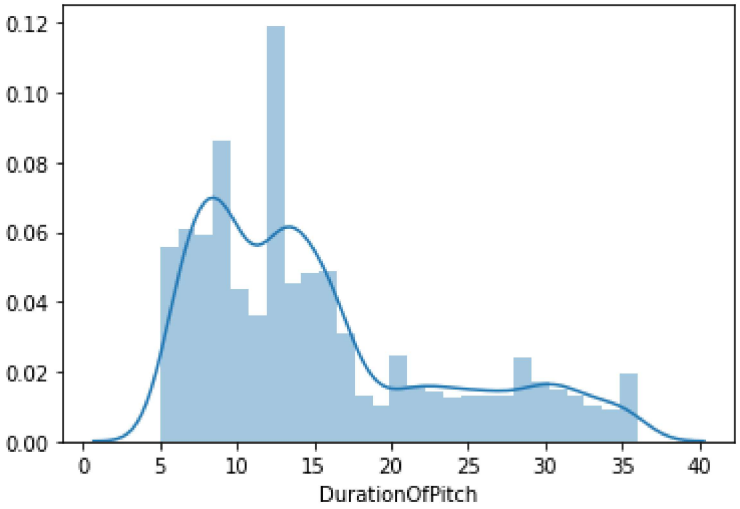
In [20]:
```
data.describe().T
```

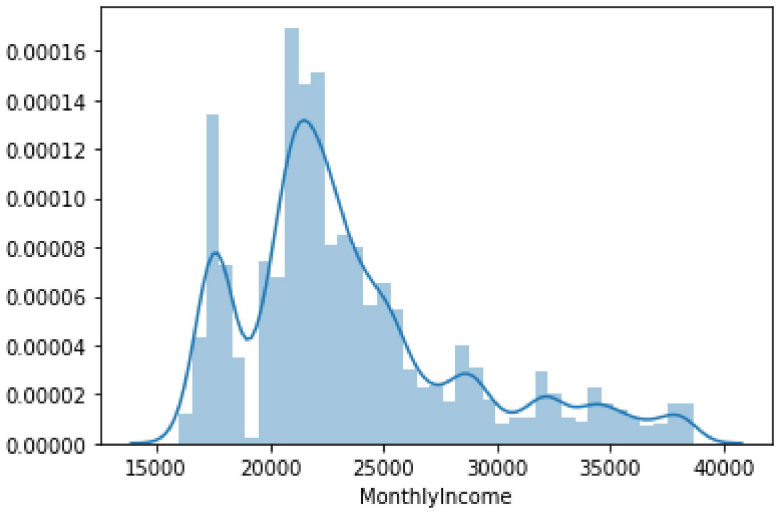
Out[20]:

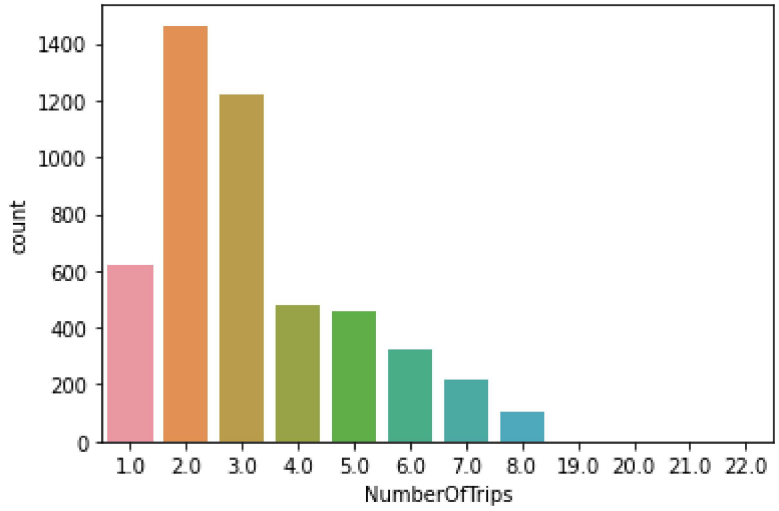
	count	mean	std	min	25%	50%	75%
ProdTaken	4882.0	0.188447	0.391109	0.0	0.00	0.0	0.00
Age	4882.0	37.544449	9.107621	18.0	31.00	36.0	43.00
CityTier	4882.0	1.654240	0.916551	1.0	1.00	1.0	3.00
DurationOfPitch	4882.0	15.321385	8.010017	5.0	9.00	13.0	19.00
NumberOfPersonVisiting	4882.0	2.905571	0.724985	1.0	2.00	3.0	3.00
NumberOfFollowups	4882.0	3.710979	0.998584	1.0	3.00	4.0	4.00
PreferredPropertyStar	4882.0	3.578247	0.797020	3.0	3.00	3.0	4.00
NumberOfTrips	4882.0	3.229824	1.823279	1.0	2.00	3.0	4.00
Passport	4882.0	0.290455	0.454019	0.0	0.00	0.0	1.00
PitchSatisfactionScore	4882.0	3.078656	1.365255	1.0	2.00	3.0	4.00
OwnCar	4882.0	0.620238	0.485377	0.0	0.00	1.0	1.00
NumberOfChildrenVisiting	4882.0	1.185170	0.852417	0.0	1.00	1.0	2.00
MonthlyIncome	4882.0	23538.973986	5029.312592	16009.0	20486.25	22347.0	25419.75

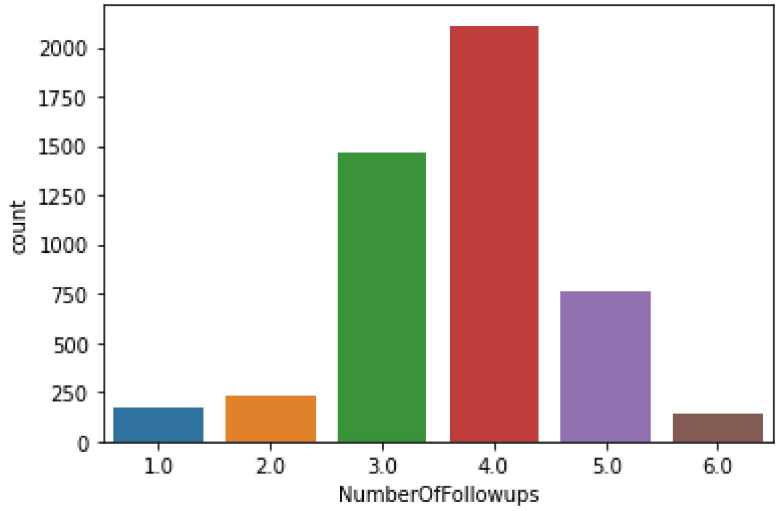
In [21]:
```
distPlotList = ['Age', 'DurationOfPitch','MonthlyIncome']
countPlotList = ['NumberOfTrips','NumberOfFollowups','Designation','NumberOfPersonVisiting',
                 'ProductPitched','PitchSatisfactionScore','MaritalStatus','Occupation','NumberOfChildrenVisiting',
                 'PreferredPropertyStar','Gender','CityTier','Passport','OwnCar','TypeofContact','ProdTaken']
for i in distPlotList:
    sns.distplot(data[i])
    plt.show()

for i in countPlotList:
    sns.countplot(data[i])
    plt.show()
```

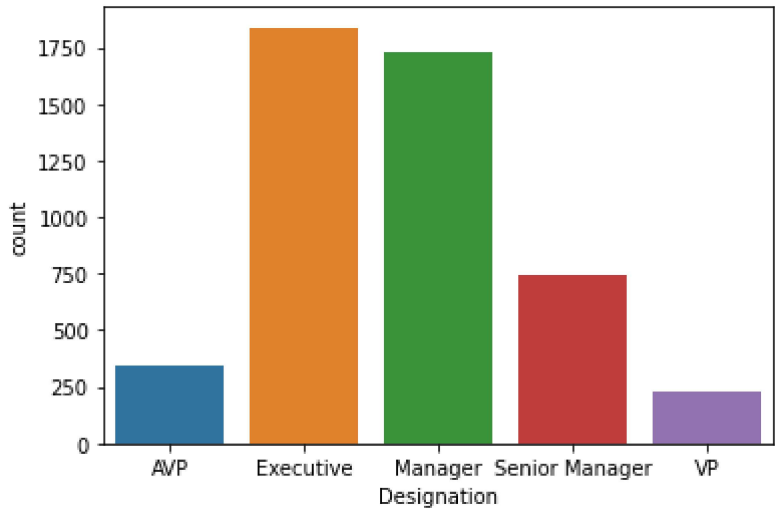

0.10
0.08
0.06
0.04
0.02
0.00
20
30
40
50
60
Age

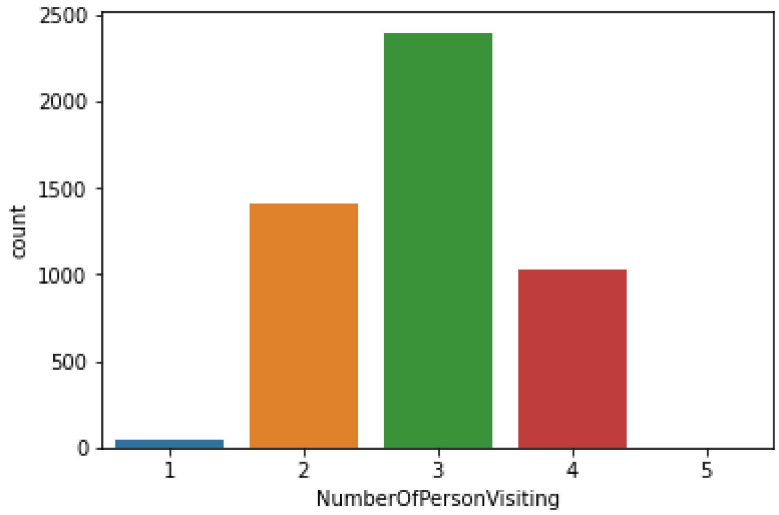

0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
5
10
15
20
25
30
35
40
DurationOfPitch

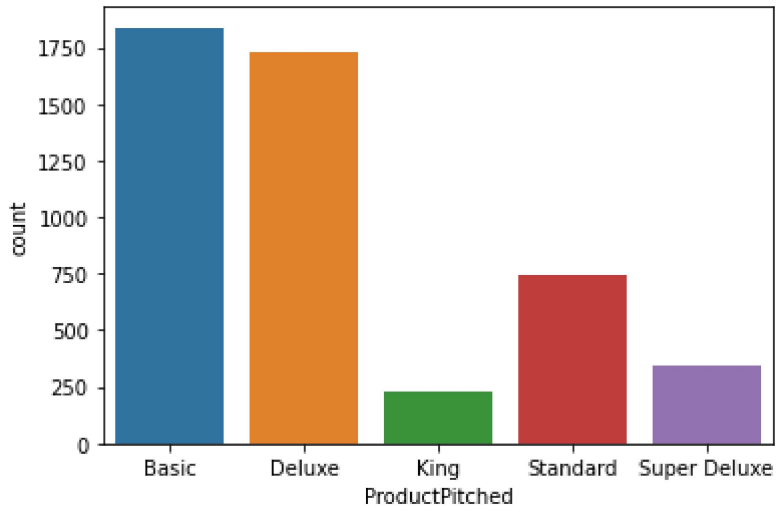

0.00016
0.00014
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
15000
20000
25000
30000
35000
40000
MonthlyIncome

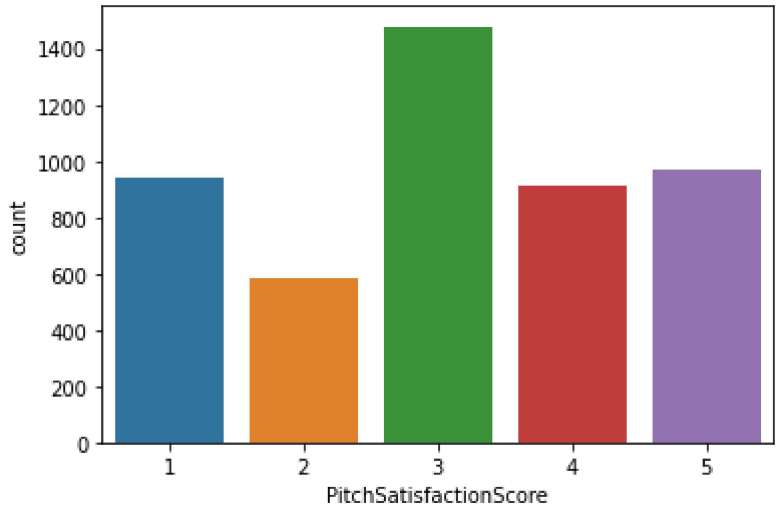

count
1400
1200
1000
800
600
400
200
0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
19.0
20.0
21.0
22.0
NumberOfTrips

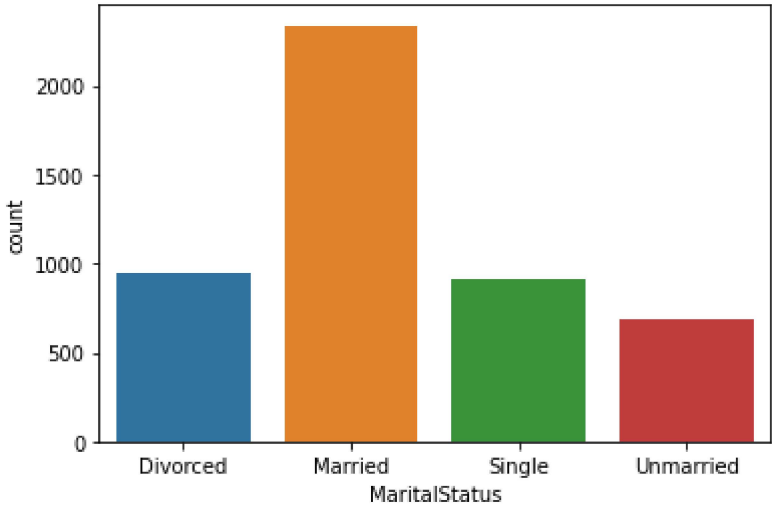

count
2000
1750
1500
1250
1000
750
500
250
0
1.0
2.0
3.0
4.0
5.0
6.0
NumberOfFollowups

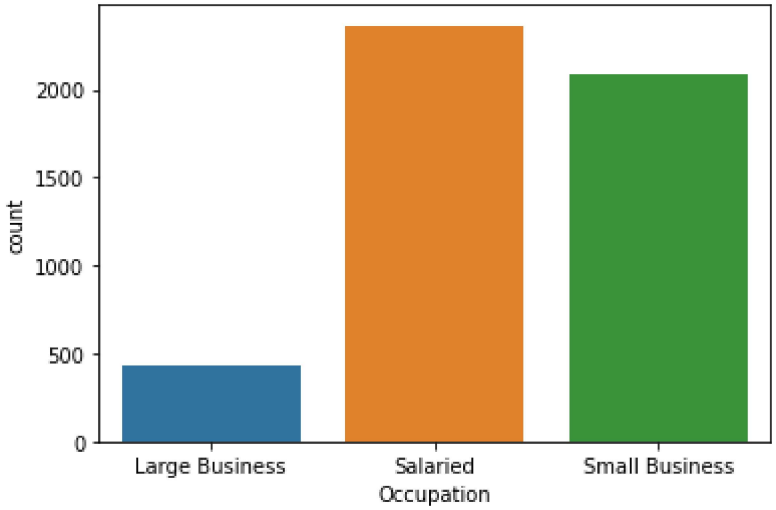

count
1750
1500
1250
1000
750
500
250
0
AVP
Executive
Manager
Senior Manager
VP
Designation

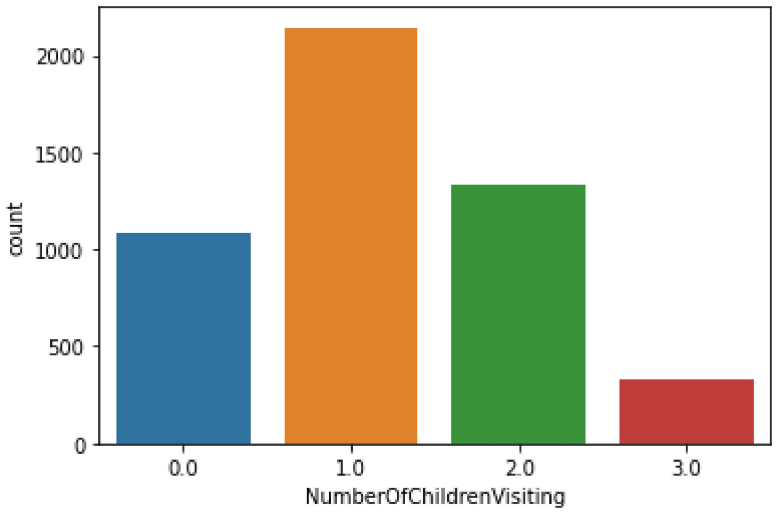

2500
2000
1500
1000
500
0
count
1
2
3
4
5
NumberOfPersonVisiting

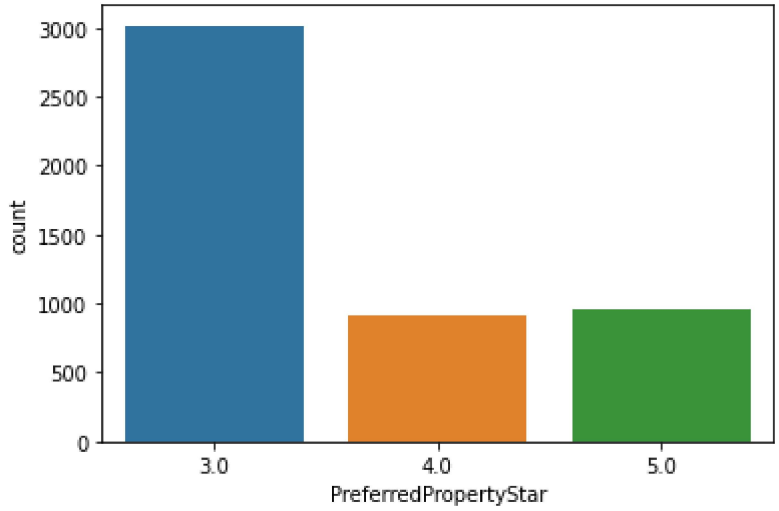

1750
1500
1250
1000
750
500
250
0
count
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

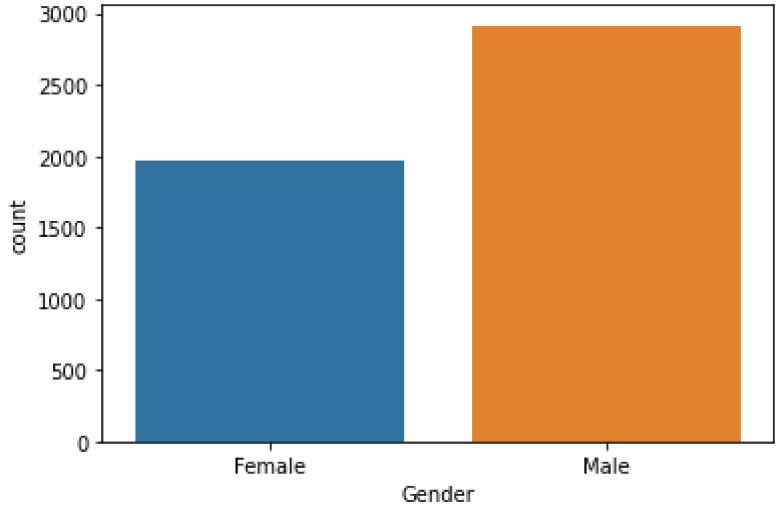

1400
1200
1000
800
600
400
200
0
count
1
2
3
4
5
PitchSatisfactionScore

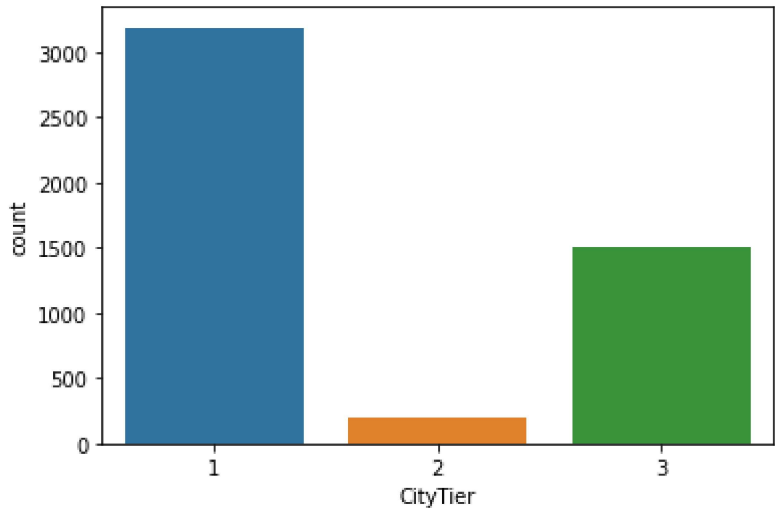

2000
1500
1000
500
0
count
Divorced
Married
Single
Unmarried
MaritalStatus

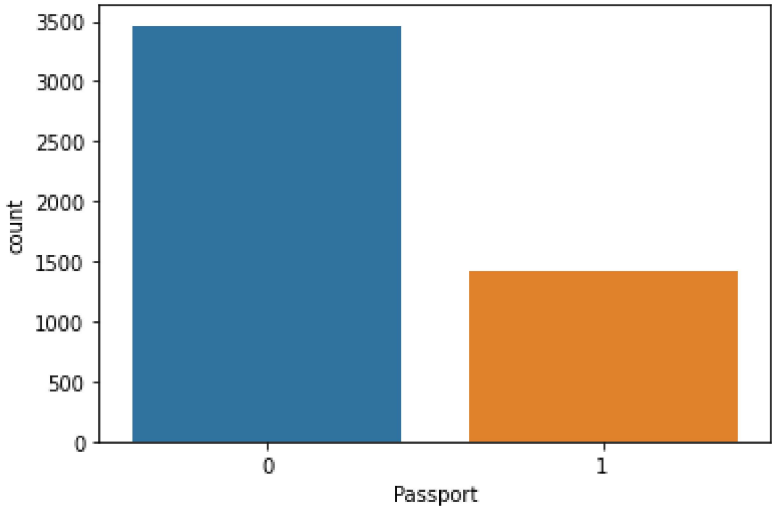

2000
1500
1000
500
0
count
Large Business
Salaried
Small Business
Occupation

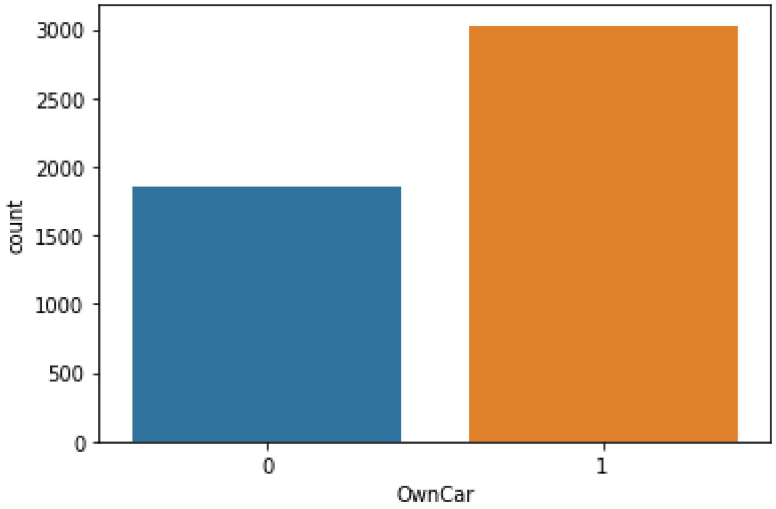

2000
1500
1000
500
0
count
0.0
1.0
2.0
3.0
NumberOfChildrenVisiting

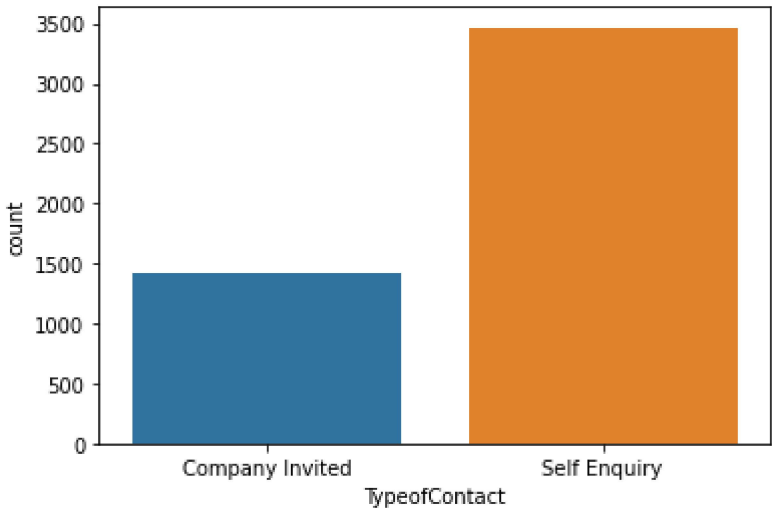

3000
2500
2000
1500
1000
500
0
count
3.0
4.0
5.0
PreferredPropertyStar

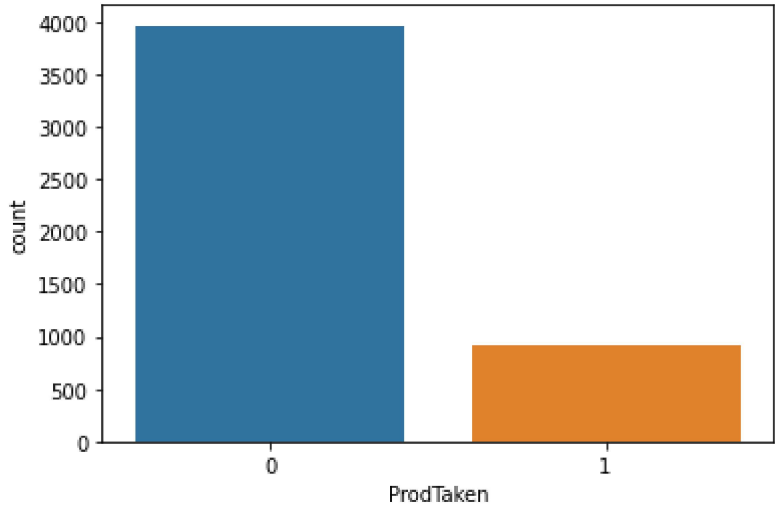

3000
2500
2000
1500
1000
500
0
count
Female
Male
Gender


3000
2500
2000
1500
1000
500
0
count
1
2
3
CityTier


3500
3000
2500
2000
1500
1000
500
0
count
0
1
Passport


3000
2500
2000
1500
1000
500
0
count
0
1
OwnCar


3500
3000
2500
2000
1500
1000
500
0
count
Company Invited
Self Enquiry
TypeofContact



Univariate Conclusion:

1. Average age is 37. It is important to implement tour package that fits the age target.
2. Average monthly income is 23500. It is important to know the customers' budget. (According to Bank of America, typical customers use 5% of their annual income on trip.) reference: https://www.washingtonpost.com/lifestyle/travel/how-to-set-and-stick-to-a-vacation-budget/2020/06/24/a3980212-b49b-11ea-a8da-693df3d7674a_story.html (https://www.washingtonpost.com/lifestyle/travel/how-to-set-and-stick-to-a-vacation-budget/2020/06/24/a3980212-b49b-11ea-a8da-693df3d7674a_story.html)
3. People generally go to 2-3 trips with 2-4 people. It is important to implement tour package that fits this.
4. About 30% of people has passport and about 60% of people has vehicle. It is possible to plan trip abroad or road trips.
5. people either live in tier 1 or tier 3 city. Consider the expectation of hotel quality since they want 3 stars or higher.

Bivariate EDA

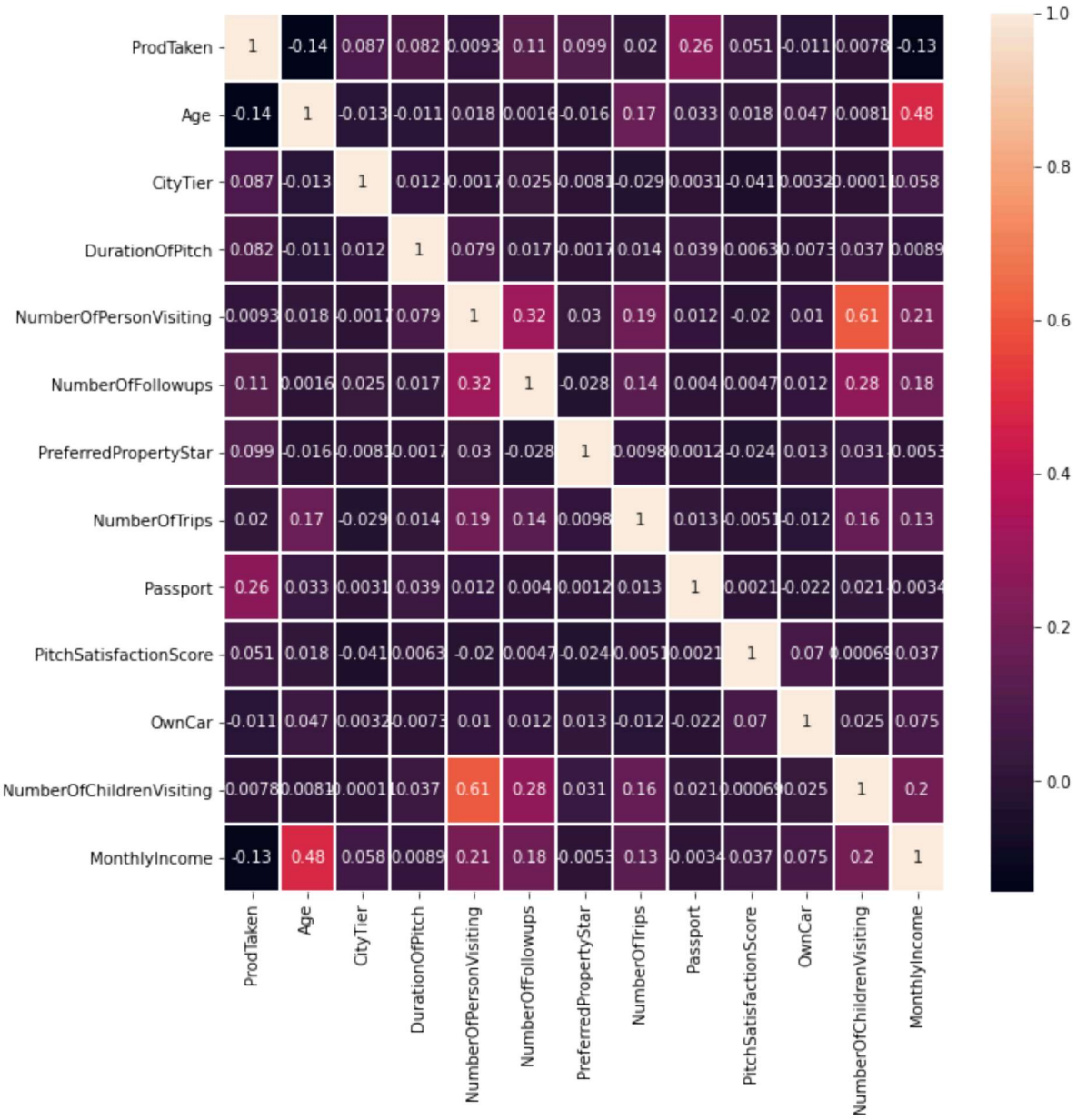
In [22]:
```
data.corr()
```

Out[22]:

	ProdTaken	Age	CityTier	DurationOfPitch	NumberOfPersonVisiting
ProdTaken	1.000000	-0.143669	0.086929	0.081964	0.009302
Age	-0.143669	1.000000	-0.013228	-0.010866	0.017748
CityTier	0.086929	-0.013228	1.000000	0.011511	-0.001664
DurationOfPitch	0.081964	-0.010866	0.011511	1.000000	0.078892
NumberOfPersonVisiting	0.009302	0.017748	-0.001664	0.078892	1.000000
NumberOfFollowups	0.111682	0.001582	0.024653	0.017378	0.324810
PreferredPropertyStar	0.098595	-0.015692	-0.008111	-0.001677	0.029986
NumberOfTrips	0.019698	0.174567	-0.028940	0.013866	0.190012
Passport	0.261654	0.032650	0.003096	0.039282	0.011764
PitchSatisfactionScore	0.051275	0.017794	-0.041133	0.006268	-0.019817
OwnCar	-0.011460	0.047245	0.003206	-0.007280	0.010439
NumberOfChildrenVisiting	0.007770	0.008123	-0.000114	0.037071	0.606140
MonthlyIncome	-0.130329	0.483594	0.057656	0.008890	0.209633

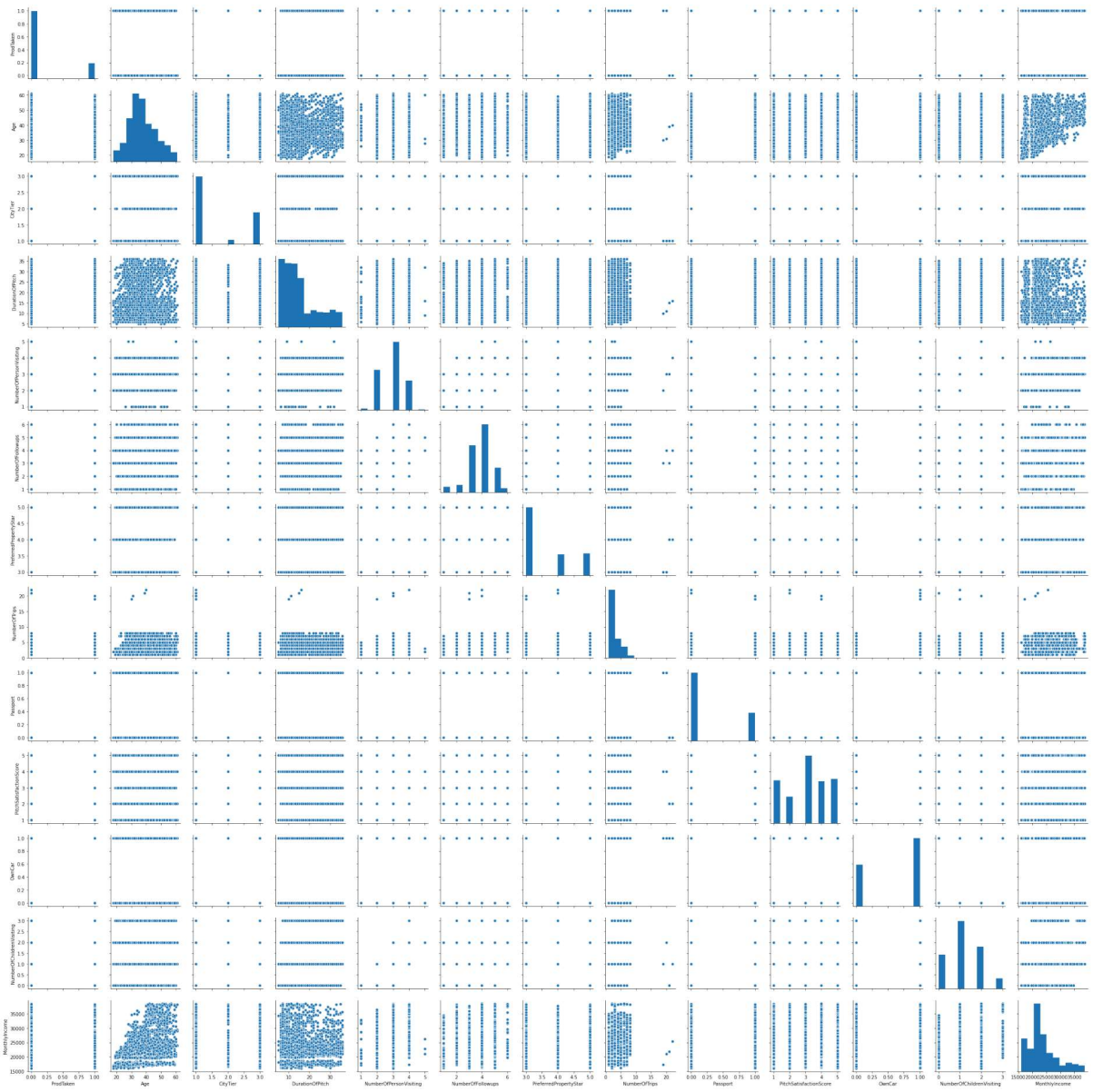
In [23]:
```
plt.subplots(figsize=(10,10))
sns.heatmap(data.corr(), annot =True, linewidth=1)
```

Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x247b8bde948>

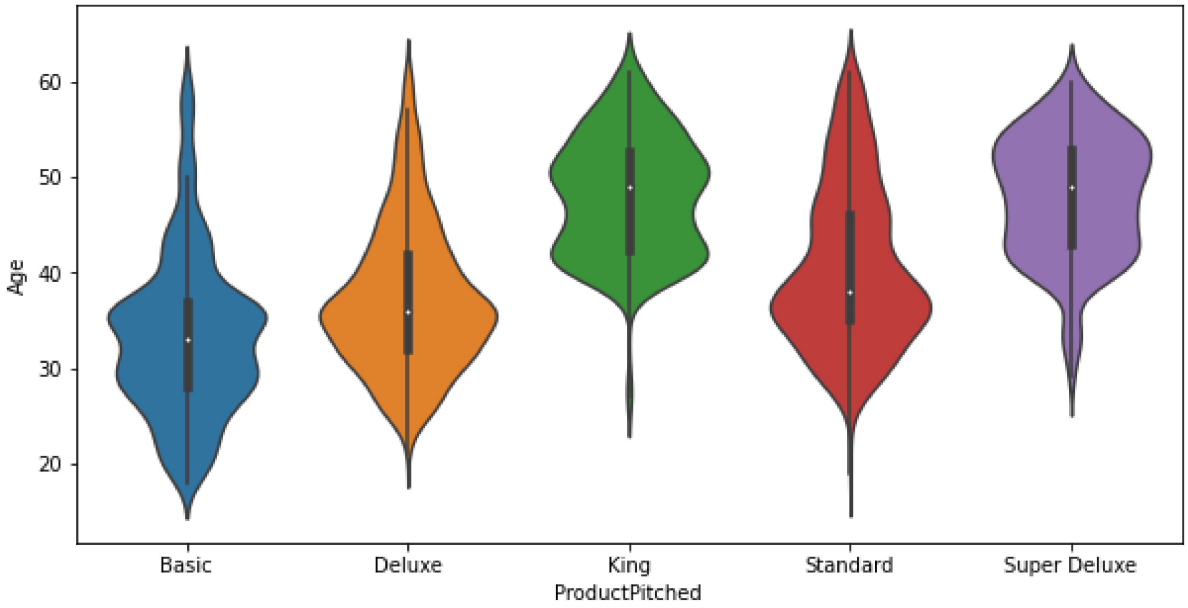


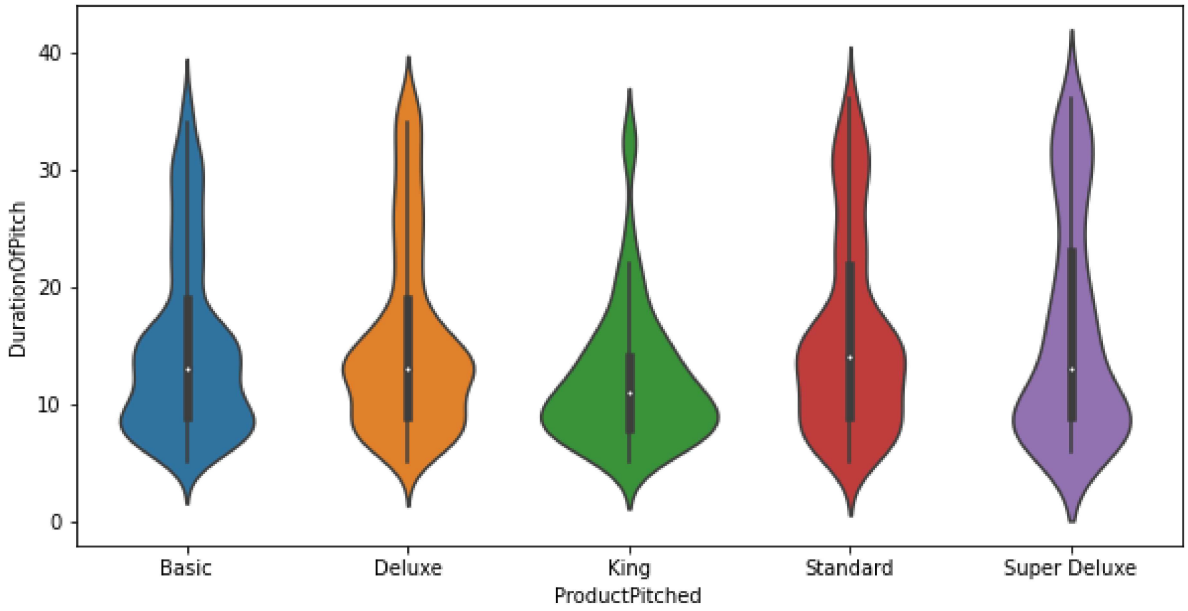
In [24]:
```
sns.pairplot(data)
```

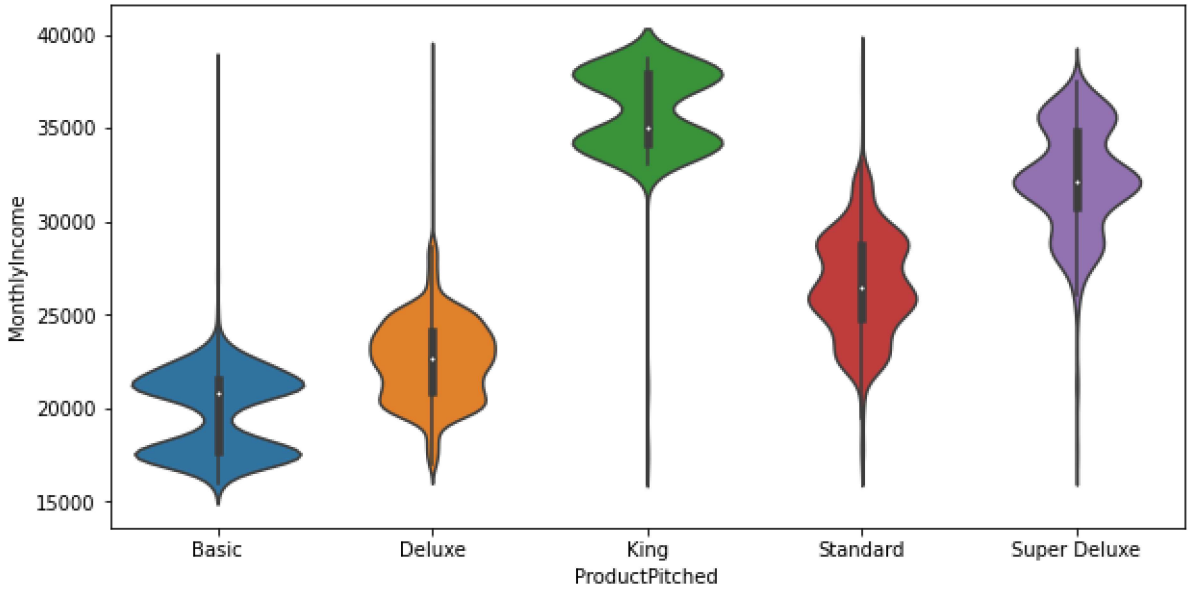
Out[24]:
```
<seaborn.axisgrid.PairGrid at 0x247b9d733c8>
```



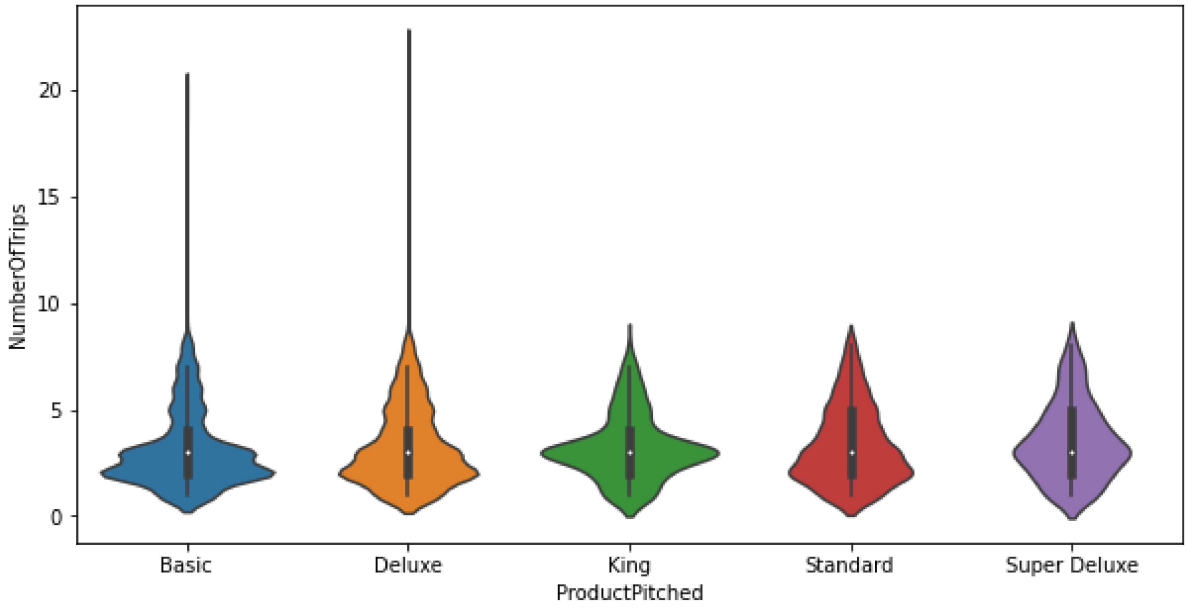
In [25]:
```
for i in distPlotList:
    plt.figure(figsize=(10,5))
    sns.violinplot(x=data['ProductPitched'], y=data[i])
    plt.show()
```

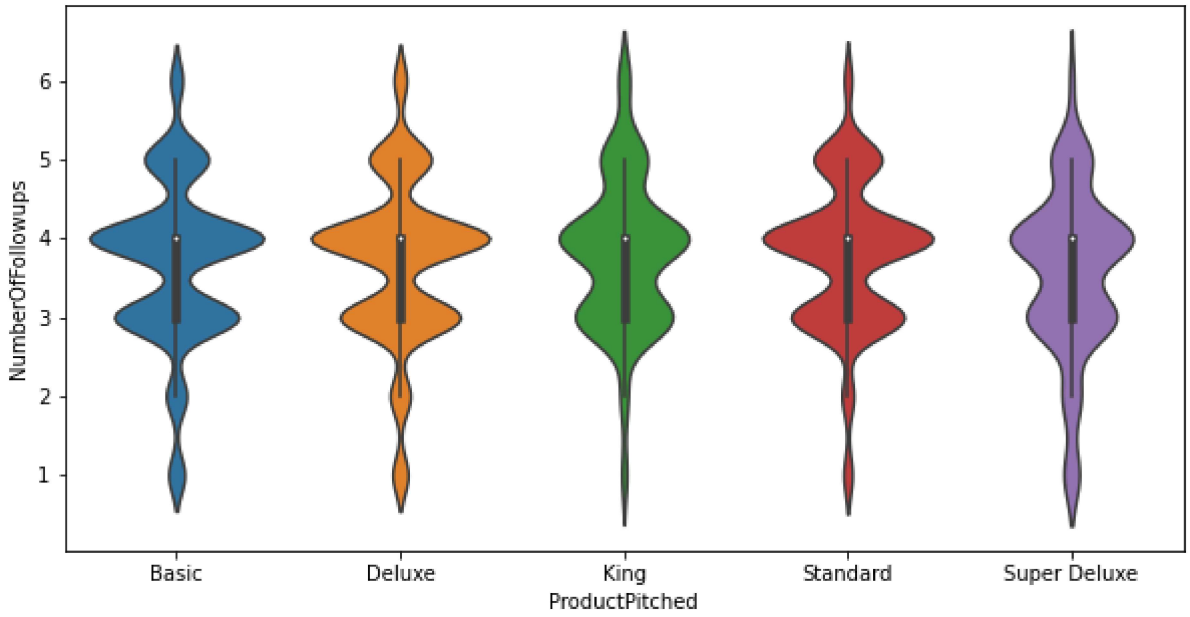

Age
60
50
40
30
20
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

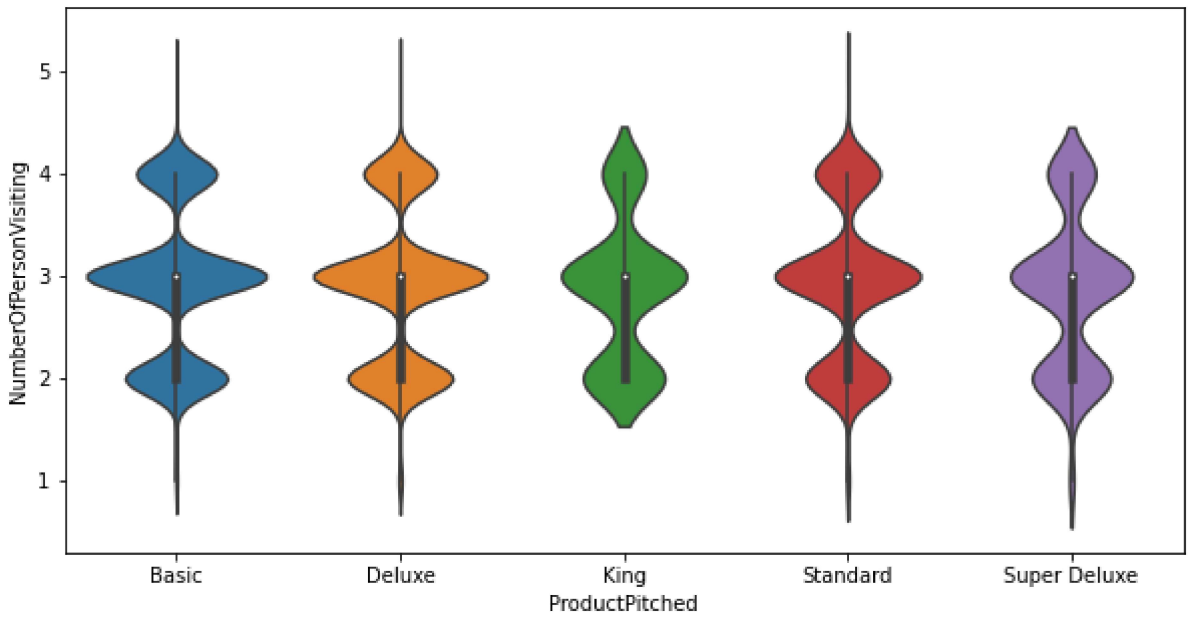

DurationOfPitch
40
30
20
10
0
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

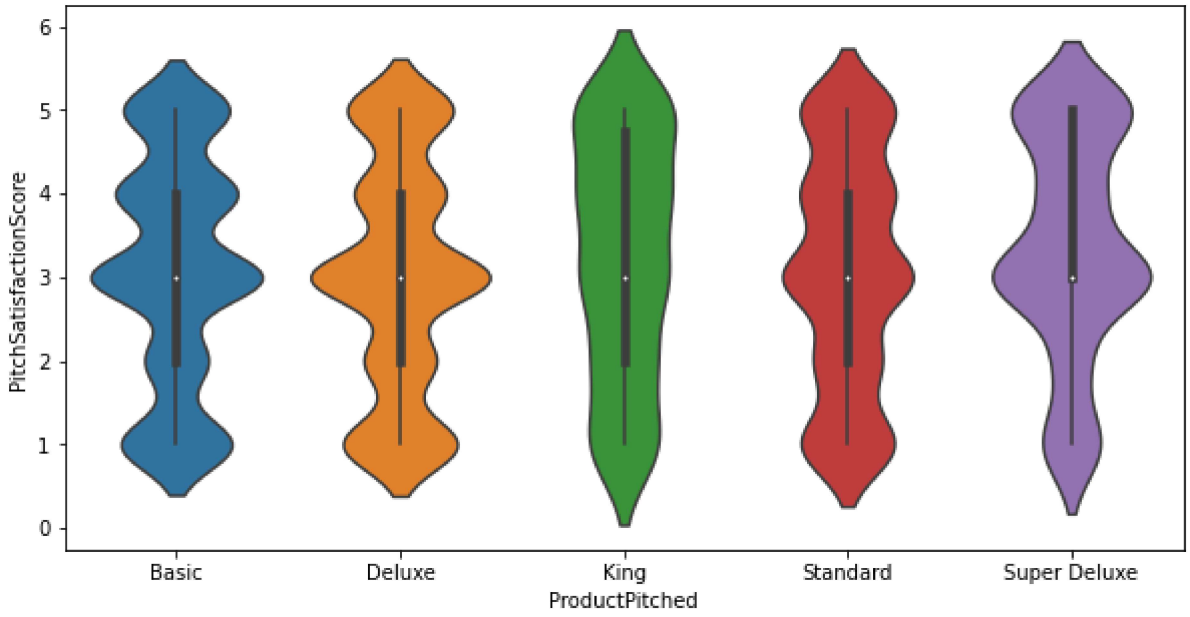

MonthlyIncome
40000
35000
30000
25000
20000
15000
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

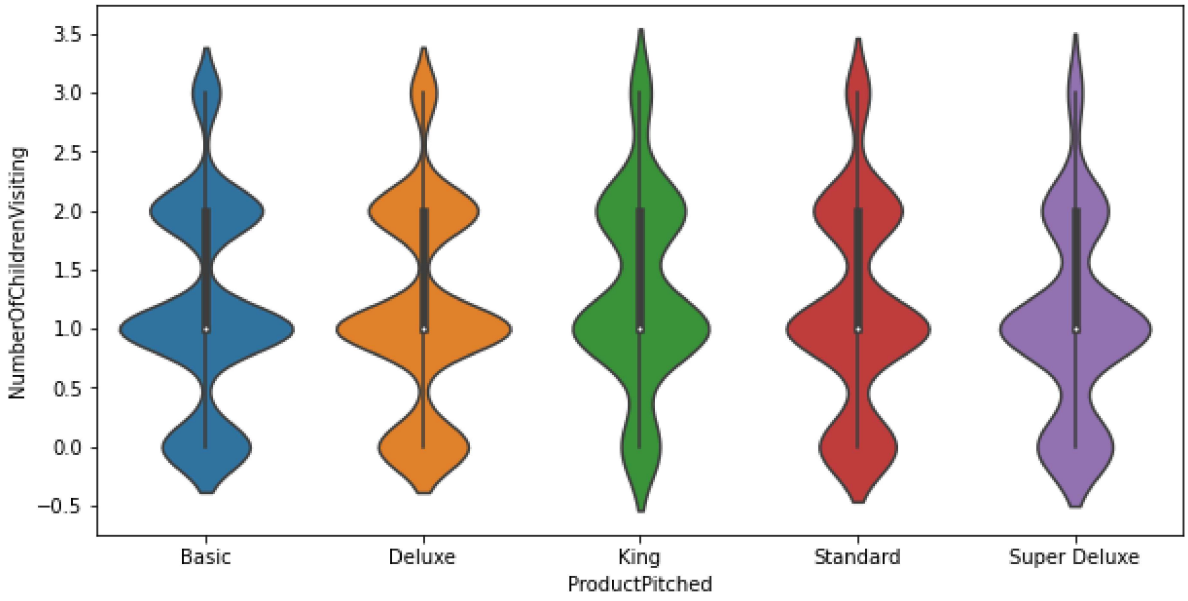
In [26]:
```
numericalPlotList = ['NumberOfTrips','NumberOfFollowups','NumberOfPersonVisiti
ng',
                'PitchSatisfactionScore','NumberOfChildrenVisiting',
                'PreferredPropertyStar','CityTier','Passport','OwnCar','ProdTa
ken']

for i in numericalPlotList:
    plt.figure(figsize=(10,5))
    sns.violinplot(x=data['ProductPitched'], y=data[i])
    plt.show()
```

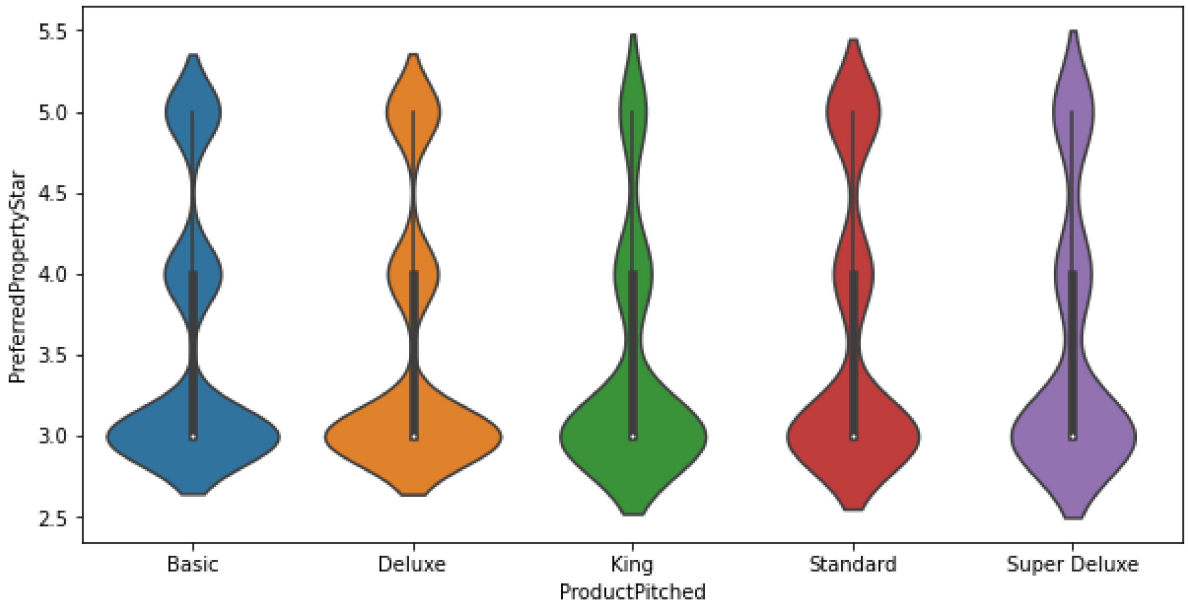

NumberOfTrips
20
15
10
5
0
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

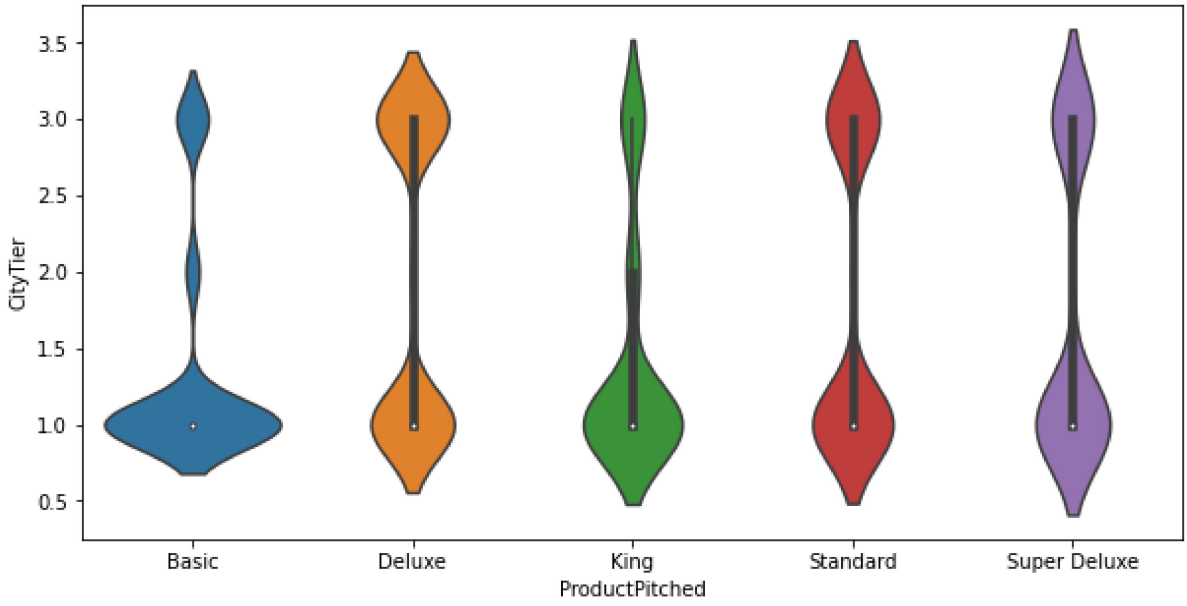

NumberOfFollowups
6
5
4
3
2
1
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

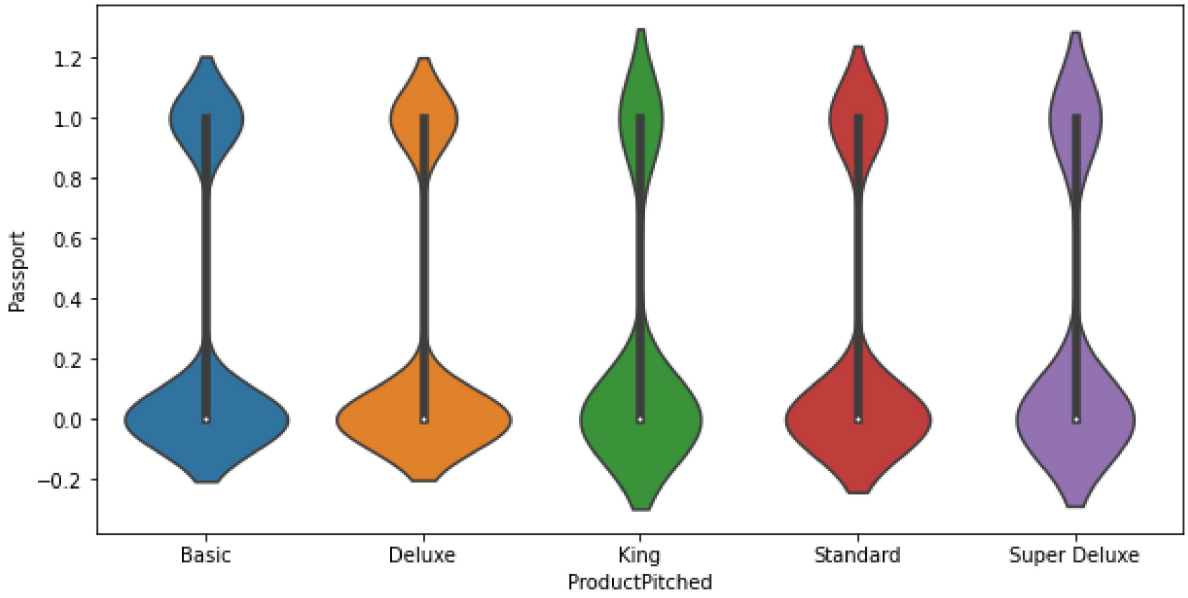

NumberOfPersonVisiting
5
4
3
2
1
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

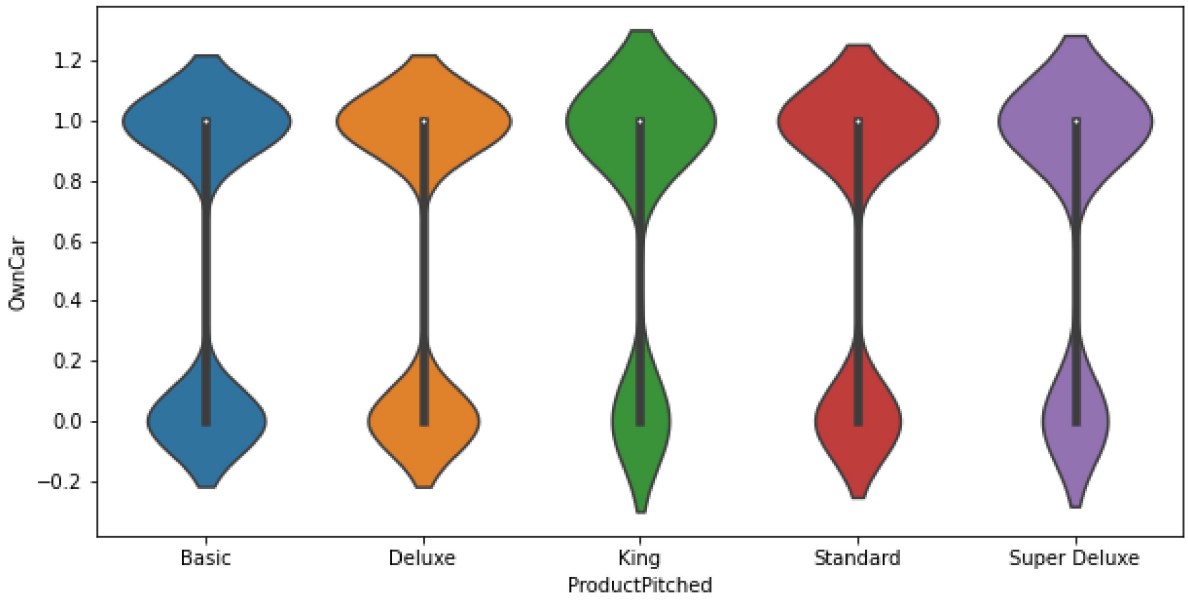

PitchSatisfactionScore
0
1
2
3
4
5
6
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

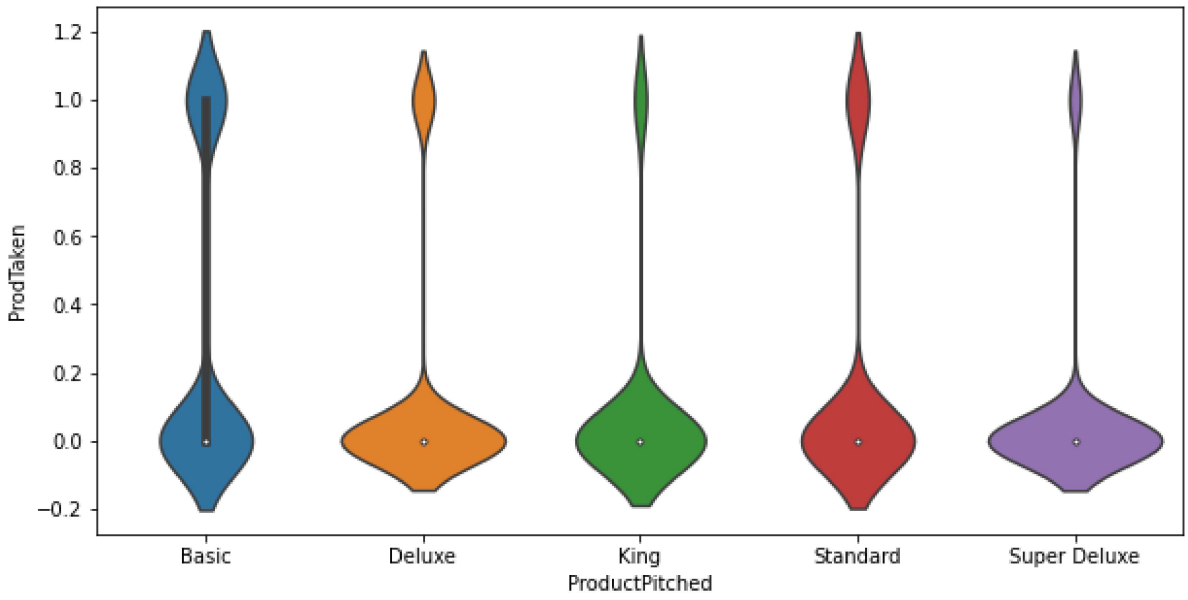

NumberOfChildrenVisiting
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched


PreferredPropertyStar
2.5
3.0
3.5
4.0
4.5
5.0
5.5
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

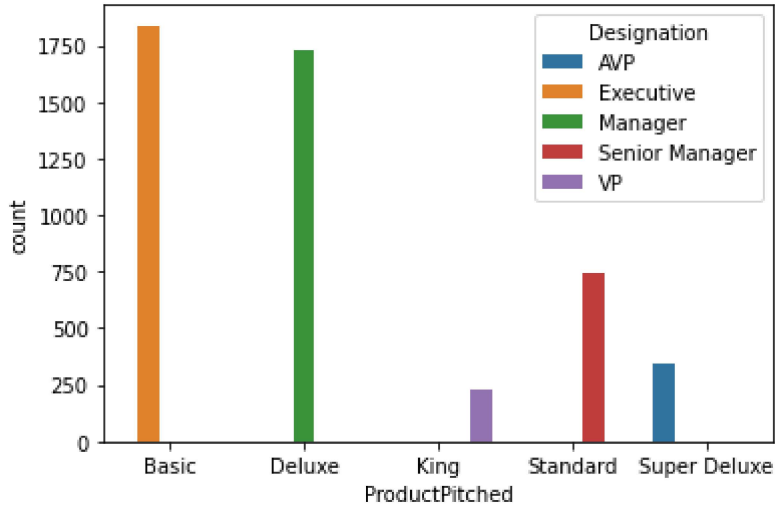

CityTier
3.5
3.0
2.5
2.0
1.5
1.0
0.5
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

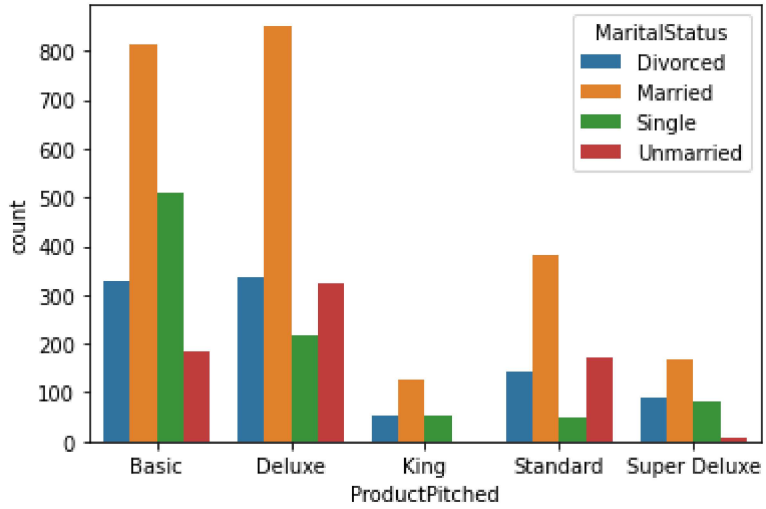

Passport
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

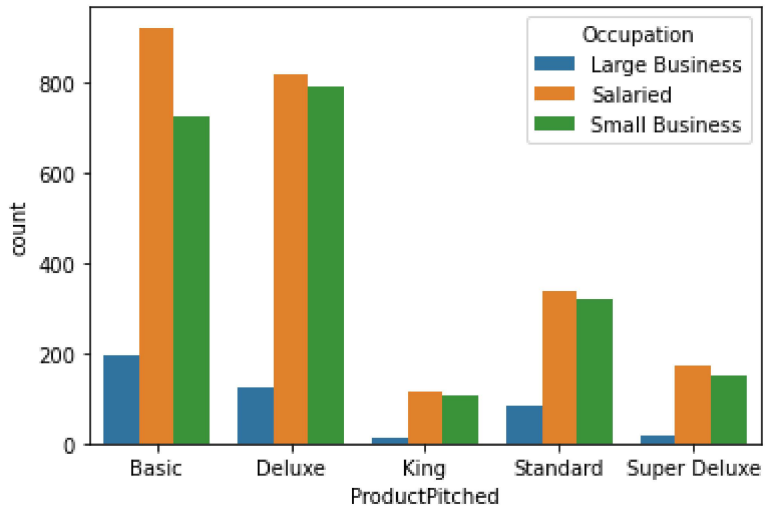

OwnCar
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

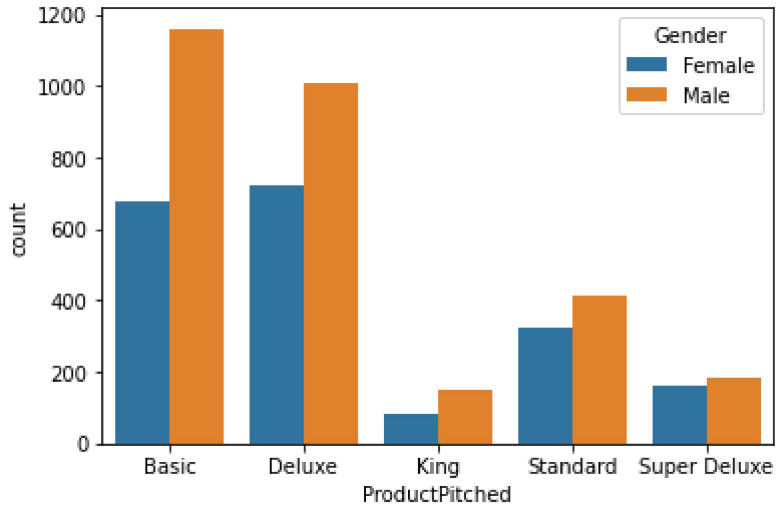

1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
ProdTaken
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched

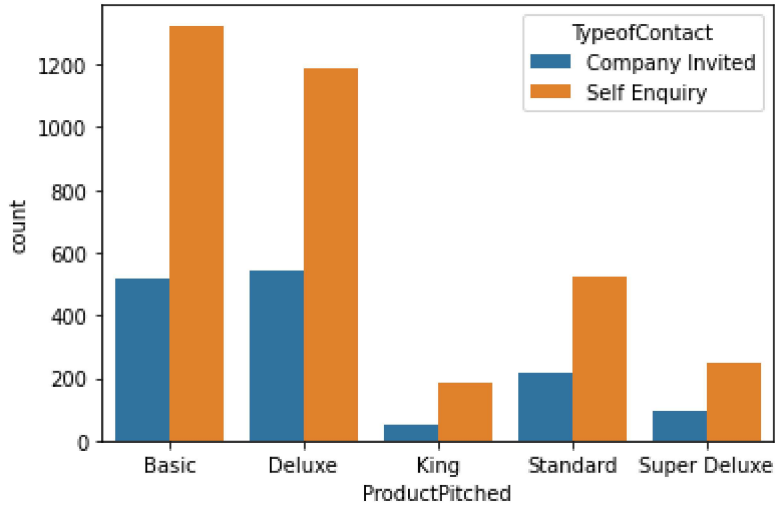
In [27]:
```
categoricalPlotList = ['Designation', 'MaritalStatus', 'Occupation','Gender',
'TypeofContact']
for i in categoricalPlotList:
    sns.countplot(data['ProductPitched'], hue = data[i])
    plt.show()
```


Designation
AVP
Executive
Manager
Senior Manager
VP
count
1750
1500
1250
1000
750
500
250
0
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched


MaritalStatus
Divorced
Married
Single
Unmarried
count
800
700
600
500
400
300
200
100
0
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched


Occupation
Large Business
Salaried
Small Business
count
800
600
400
200
0
Basic
Deluxe
King
Standard
Super Deluxe
ProductPitched





In [28]:
```
data[data['Designation']=='AVP']['ProductPitched'].unique()
```

Out[28]:
```
[Super Deluxe]
Categories (1, object): [Super Deluxe]
```

Bivariate Conclusion:

1. The most important fact is that the Designation determines Product Pitched. This can be a strategy of marketing or serious preference of people by designation.
2. King, Super Deluxe, Standard, Deluxe, Basic is the order from higher monthly income to lower monthly income.
3. People who visited more than 9 times tend to pick deluxe or Basic. People may find that it's not worth it to go somewhere too fancy when travelling often.
4. There is no people taking King Pitch if it is their first visit.

Before going into model making, it is more important to secure recall rate in this particular case. For the sake of project, assume missing potential cutomer is 4 times more costly than marketing costs. Hence, recall is 4 times more valuable than precision.

Data Split

In [29]:
```
X = data.drop(['ProdTaken'],axis=1)
X = pd.get_dummies(X,drop_first=True)
y = data['ProdTaken']
```

In [30]:
```
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)
```

```
(3417, 27) (1465, 27)
```

In [31]:
```
print("Class distritution: ")
print(y.value_counts(1))
print("")
print("Test Class distritution: ") # good distribution
print(y_test.value_counts(1))
```

```
Class distritution:
0    0.811553
1    0.188447
Name: ProdTaken, dtype: float64

Test Class distritution:
0    0.811604
1    0.188396
Name: ProdTaken, dtype: float64
```

Function Implementaion

In [32]:
```
def get_score_acc_rec_prec(model):
    '''
    model : Classifier to predict
    [0:train_acc,1:test_acc,2:train_recall,3:test_recall,4:train_precision,5test_precision]
    '''

    list = [] # defining an empty list to store all scores

    pred_train = model.predict(x_train)
    pred_test = model.predict(x_test)

    #Create Scores
    train_acc = model.score(x_train,y_train)
    test_acc = model.score(x_test,y_test)
    train_recall = metrics.recall_score(y_train,pred_train)
    test_recall = metrics.recall_score(y_test,pred_test)
    train_precision = metrics.precision_score(y_train,pred_train)
    test_precision = metrics.precision_score(y_test,pred_test)

    # Add accuracy in the list
    list.append(train_acc)
    list.append(test_acc)
    list.append(train_recall)
    list.append(test_recall)
    list.append(train_precision)
    list.append(test_precision)

    return list # returning the list with train and test scores
```

In [33]:
```
def print_score(list,what="all"):
    ''''''
    if what == "acc":
        print("Accuracy on training set : ", list[0])
        print("Accuracy on test set : ", list[1])
    elif what == "rec":
        print("Recall on training set : ", list[2])
        print("Recall on test set : ", list[3])
    elif what == "prec":
        print("Precision on training set : ", list[4])
        print("Precision on test set : ", list[5])
    else:
        print("Accuracy on training set : ", list[0])
        print("Accuracy on test set : ", list[1])
        print("Recall on training set : ", list[2])
        print("Recall on test set : ", list[3])
        print("Precision on training set : ", list[4])
        print("Precision on test set : ", list[5])
```

In [34]:
```
def make_cm(model,y_actual,labels=[1, 0]):
    '''
    model : classifier to predict
    y_actual : ground truth (y_test)

    '''

    y_predict = model.predict(x_test)
    cm=metrics.confusion_matrix( y_actual, y_predict, labels=[1, 0])
    df_cm = pd.DataFrame(cm, index = [i for i in ["1","0"]],
                  columns = [i for i in ["Predict 1","Predict 0"]])

#     annotation=[cm]
# [['1116\n76.18%' '73\n4.98%']
#  ['91\n6.21%' '185\n12.63%']]
    sns.heatmap(df_cm, annot=True, fmt='')
```

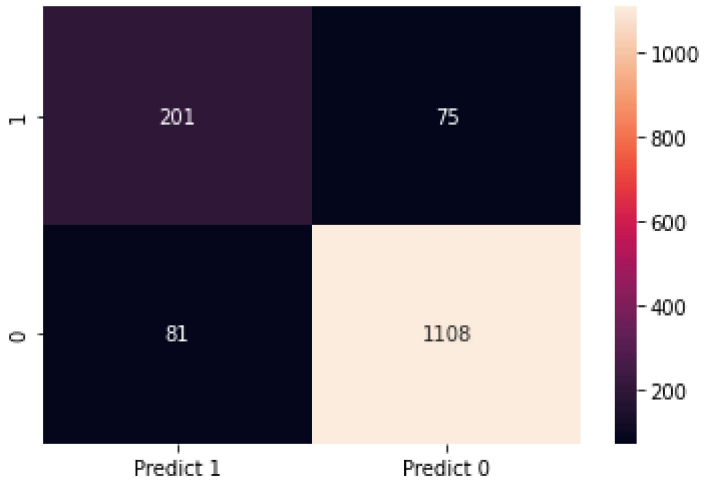
Bagging:

Decision Tree

In [35]:
```
dTree = DecisionTreeClassifier(criterion='gini',random_state=1)
dTree.fit(x_train, y_train)
```

Out[35]:
```
DecisionTreeClassifier(random_state=1)
```

In [36]:
```
make_cm(dTree,y_test)
```



In [37]:
```
dTree_score = get_score_acc_rec_prec(dTree)
print_score(dTree_score,'all')
```

```
Accuracy on training set :  1.0
Accuracy on test set :  0.8935153583617748
Recall on training set :  1.0
Recall on test set :  0.7282608695652174
Precision on training set :  1.0
Precision on test set :  0.7127659574468085
```

Random Forest

In [38]:
```
rForest = RandomForestClassifier(random_state=1)
rForest.fit(x_train,y_train)
```

Out[38]:
```
RandomForestClassifier(random_state=1)
```

In [39]:
```
make_cm(rForest,y_test)
```

In [40]:
```
rForest_score = get_score_acc_rec_prec(rForest)
print_score(rForest_score,'all')
```

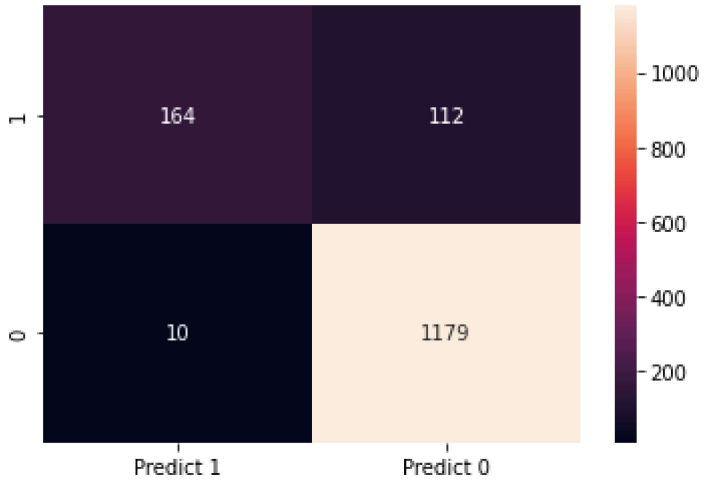
```
Accuracy on training set :  1.0
Accuracy on test set :  0.9228668941979522
Recall on training set :  1.0
Recall on test set :  0.6231884057971014
Precision on training set :  1.0
Precision on test set :  0.9502762430939227
```

Random Forest with Class_Weight

In [41]:
```
rForestPlus = RandomForestClassifier(random_state=1, class_weight={0:0.25,1:0.
75})
rForestPlus.fit(x_train,y_train)
```

Out[41]:
```
RandomForestClassifier(class_weight={0: 0.25, 1: 0.75}, random_state=1)
```

In [42]:
```
make_cm(rForestPlus,y_test)
```



In [43]:
```
rForestPlus_score = get_score_acc_rec_prec(rForestPlus)
print_score(rForestPlus_score,'all')
```

```
Accuracy on training set :  1.0
Accuracy on test set :  0.9167235494880546
Recall on training set :  1.0
Recall on test set :  0.5942028985507246
Precision on training set :  1.0
Precision on test set :  0.9425287356321839
```

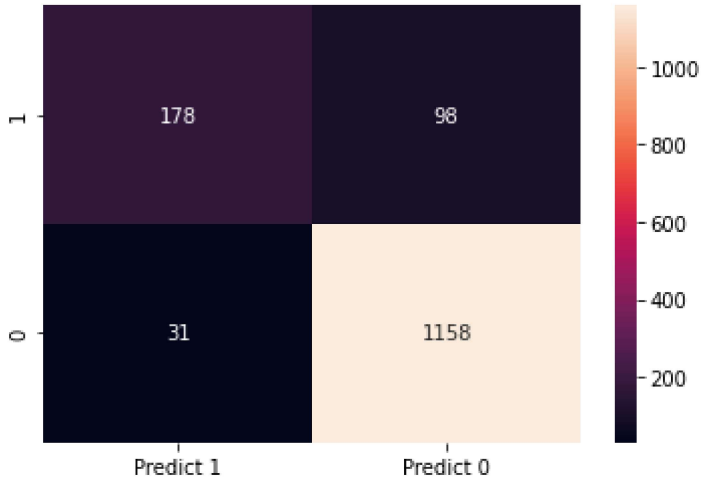
The Class_weight did not help the model

Bagging Classifier

In [44]:
```
bClassifier = BaggingClassifier(random_state=1)
bClassifier.fit(x_train,y_train)
```

Out[44]:
```
BaggingClassifier(random_state=1)
```

In [45]:
```
make_cm(bClassifier,y_test)
```



In [46]:
```
bClassifier_score = get_score_acc_rec_prec(bClassifier)
print_score(bClassifier_score,'all')
```

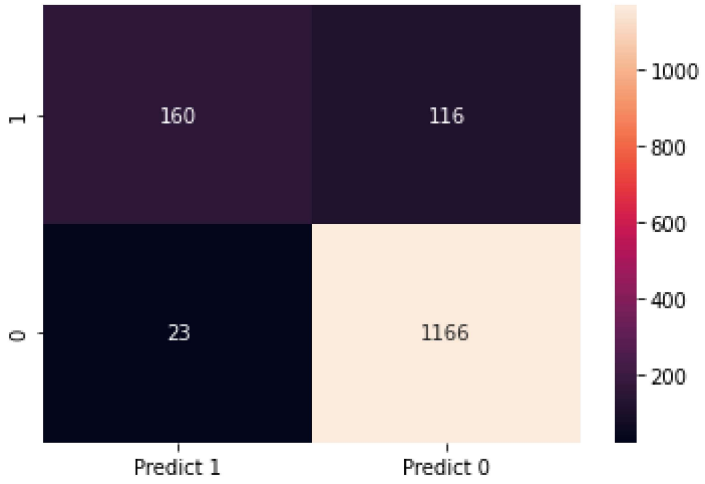
```
Accuracy on training set :  0.9926836406204272
Accuracy on test set :  0.9119453924914676
Recall on training set :  0.9611801242236024
Recall on test set :  0.644927536231884
Precision on training set :  1.0
Precision on test set :  0.8516746411483254
```

Bagging Classifier with Class_Weight (Didn't help)

In [47]:
```
bClassifierPlus = BaggingClassifier(base_estimator=DecisionTreeClassifier(crit
erion='gini',class_weight={0:0.25,1:0.75},random_state=1),random_state=1)
bClassifierPlus.fit(x_train,y_train)
```

Out[47]:
```
BaggingClassifier(base_estimator=DecisionTreeClassifier(class_weight={0: 0.2
5,
                                                                       1: 0.7
5},
                                                        random_state=1),
                  random_state=1)
```

In [48]:
```
make_cm(bClassifierPlus,y_test)
```



In [49]:
```
bClassifierPlus_score = get_score_acc_rec_prec(bClassifierPlus)
print_score(bClassifierPlus_score,'all')
```

```
Accuracy on training set :  0.9906350599941469
Accuracy on test set :  0.9051194539249147
Recall on training set :  0.9565217391304348
Recall on test set :  0.5797101449275363
Precision on training set :  0.9935483870967742
Precision on test set :  0.8743169398907104
```

The Class_weight helped increasing precision but not recall (made it worse)

Improving Bagging Models

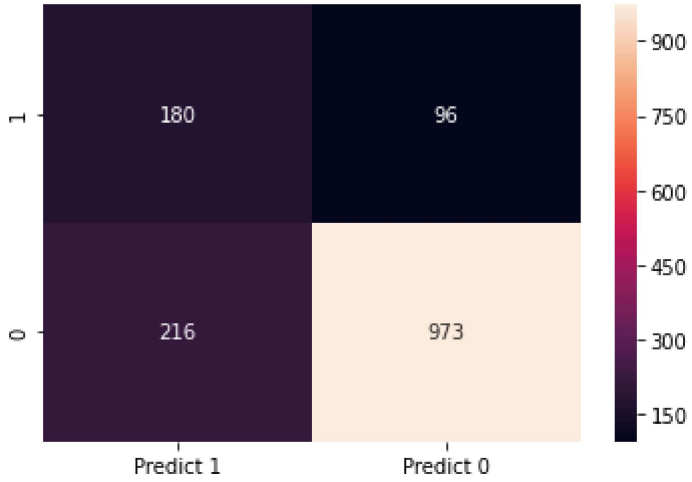
Decision Tree

In [50]:
```
dTreeOptimum = DecisionTreeClassifier(class_weight={0:0.25,1:0.75},random_stat
e=1)

# parameters to find optimum
parameters_dTree = {'max_depth': np.arange(2,30),
              'min_samples_leaf': [1, 2, 5, 7, 9],
              'max_leaf_nodes' : [2, 3, 5, 10, 14],
              'min_impurity_decrease': [0.0001,0.001,0.01,0.1]
             }

scorer = metrics.make_scorer(metrics.recall_score)

# Run the grid search
grid_dTree = GridSearchCV(dTreeOptimum, parameters_dTree, scoring=scorer, cv =
5)
grid_dTree = grid_dTree.fit(x_train, y_train)
```

```
# Set the best extimator.
dTreeOptimum = grid_dTree.best_estimator_

# Fit the best algorithm to the data.
dTreeOptimum.fit(x_train, y_train)
```

Out[50]:
```
DecisionTreeClassifier(class_weight={0: 0.25, 1: 0.75}, max_depth=6,
                       max_leaf_nodes=14, min_impurity_decrease=0.0001,
                       random_state=1)
```

In [51]:
```
make_cm(dTreeOptimum,y_test)
```



In [52]:
```
dTreeOptimum_score = get_score_acc_rec_prec(dTreeOptimum)
print_score(dTreeOptimum_score,'all')
```

```
Accuracy on training set :  0.800702370500439
Accuracy on test set :  0.7870307167235495
Recall on training set :  0.6413043478260869
Recall on test set :  0.6521739130434783
Precision on training set :  0.47856315179606024
Precision on test set :  0.45454545454545453
```

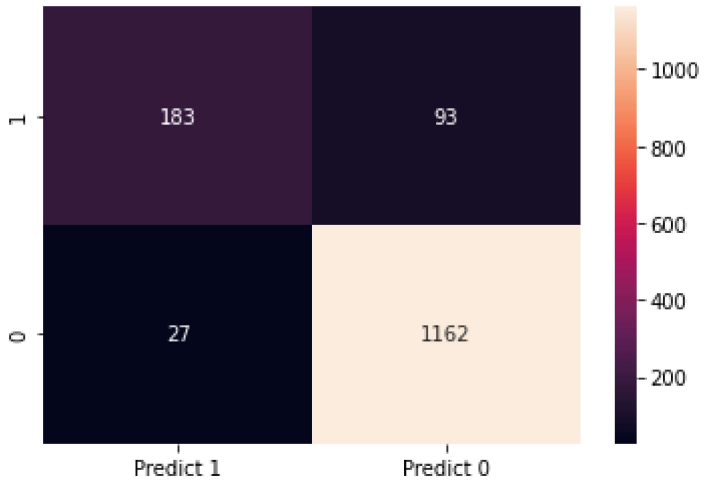
Overfitting is resolved, but the score is worse

Bagging Classifier

In [53]:
```
dTreeImp = DecisionTreeClassifier(class_weight={0:0.25,1:0.75},random_state=1)
parameters_bagging= {'base_estimator':[dTreeImp],
                'n_estimators':[5,8,15,50,100],
                'max_features': [0.7,0.8,0.9,1]
               }

grid_bClass = GridSearchCV(BaggingClassifier(random_state=1,bootstrap=True),
                           param_grid=parameters_bagging, scoring = scorer, cv
= 5)
grid_bClass.fit(x_train, y_train)

bClassifierOptimum  = grid_bClass.best_estimator_
bClassifierOptimum.fit(x_train,y_train)
```

Out[53]:
```
BaggingClassifier(base_estimator=DecisionTreeClassifier(class_weight={0: 0.2
5,
                                                                       1: 0.7
5},
                                                        random_state=1),
                  max_features=0.9, n_estimators=15, random_state=1)
```

In [54]:
```
make_cm(bClassifierOptimum,y_test)
```



In [55]:
```
bClassifierOptimum_score = get_score_acc_rec_prec(bClassifierOptimum)
print_score(bClassifierOptimum_score,'all')
```

```
Accuracy on training set :  0.9964881474978051
Accuracy on test set :  0.9180887372013652
Recall on training set :  0.984472049689441
Recall on test set :  0.6630434782608695
Precision on training set :  0.9968553459119497
Precision on test set :  0.8714285714285714
```

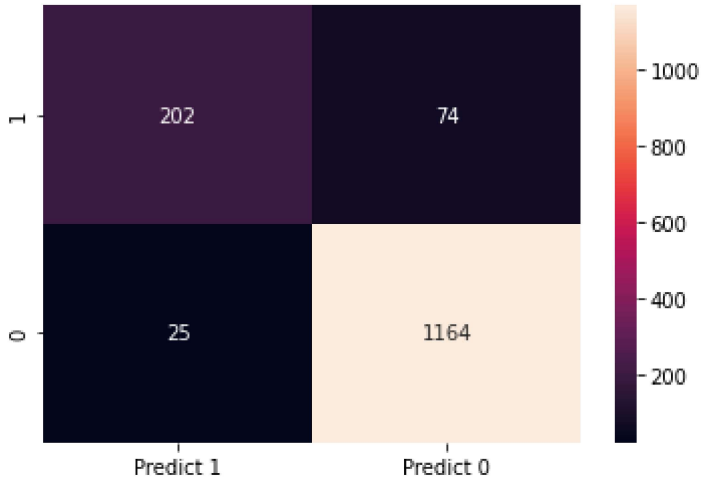
Still suspect for Overfitting. Recall is not satisfying

Random Forest

In [56]:
```
rForestOptimum = RandomForestClassifier(random_state=1)

# parameters to find optimum
parameters_rForest = {
        "n_estimators": [501], # more tree is mostly better.
        "min_samples_leaf": np.arange(1,6),
        "max_features": [0.7,0.9,'log2','auto']
}

grid_rForest = GridSearchCV(rForestOptimum, parameters_rForest, scoring='recal
l',cv=5)
grid_rForest = grid_rForest.fit(x_train, y_train)

rForestOptimum = grid_rForest.best_estimator_
rForestOptimum.fit(x_train, y_train)
```

Out[56]:
```
RandomForestClassifier(max_features=0.7, n_estimators=501, random_state=1)
```

In [57]:
```
make_cm(rForestOptimum,y_test)
```



In [58]:
```
rForestOptimum_score = get_score_acc_rec_prec(rForestOptimum)
print_score(rForestOptimum_score,'all')
```

```
Accuracy on training set :  1.0
Accuracy on test set :  0.9324232081911262
Recall on training set :  1.0
Recall on test set :  0.7318840579710145
Precision on training set :  1.0
Precision on test set :  0.8898678414096917
```

Overfitting is present, but the accuracy along with recall increased significantly.

Compare Bagging Models

In [59]:
```
# defining list of models
modelList = [dTree,dTreeOptimum,
          bClassifier,bClassifierPlus,bClassifierOptimum,
          rForest,rForestPlus,rForestOptimum]

# Empty lists to add results
acc_train = []
acc_test = []
rec_train = []
rec_test = []
prec_train = []
prec_test = []

# loop score
for model in modelList:
    score = get_score_acc_rec_prec(model)
    acc_train.append(score[0])
    acc_test.append(score[1])
    rec_train.append(score[2])
    rec_test.append(score[3])
    prec_train.append(score[4])
    prec_test.append(score[5])
```

In [60]:
```
compare_frame = pd.DataFrame({'Model':['Decision Tree','Optimum Decision Tre
e','Bagging Classifier',
                                       'Weighted Bagging Classifier','Optim
um Bagging Classifier',
                                       'Random Forest','Weighted Random For
est','Optimum Random Forest'],
                                       'TrainAccuracy': acc_train,
                                       'TestAccuracy': acc_test,
                                       'TrainRecall': rec_train,
                                       'TestRecall': rec_test,
                                       'TrainPrecision': prec_train,
                                       'TestPrecision': prec_test})
compare_frame
```

Out[60]:

	Model	TrainAccuracy	TestAccuracy	TrainRecall	TestRecall	TrainPrecision	TestPrecision
0	Decision Tree	1.000000	0.893515	1.000000	0.728261	1.000000	0.712766
1	Optimum Decision Tree	0.800702	0.787031	0.641304	0.652174	0.478563	0.454545
2	Bagging Classifier	0.992684	0.911945	0.961180	0.644928	1.000000	0.851675
3	Weighted Bagging Classifier	0.990635	0.905119	0.956522	0.579710	0.993548	0.874317
4	Optimum Bagging Classifier	0.996488	0.918089	0.984472	0.663043	0.996855	0.871429
5	Random Forest	1.000000	0.922867	1.000000	0.623188	1.000000	0.950276
6	Weighted Random Forest	1.000000	0.916724	1.000000	0.594203	1.000000	0.942529
7	Optimum Random Forest	1.000000	0.932423	1.000000	0.731884	1.000000	0.889868

Bagging Conclusion:

Among 8 Bagging classifiers, Optimum Random Forest Classifier has the best Accuracy and Recall.

Boosting:

AdaBoost Classifier

In [61]:
```
adaBoost = AdaBoostClassifier(random_state=1)
```

```
adaBoost.fit(x_train,y_train)
```

Out[61]:
```
AdaBoostClassifier(random_state=1)
```

In [62]:
```
make_cm(adaBoost,y_test)
```

In [63]:
```
adaBoost_score = get_score_acc_rec_prec(adaBoost)
print_score(adaBoost_score,'all')
```

```
Accuracy on training set :  0.8484050336552531
Accuracy on test set :  0.8552901023890785
Recall on training set :  0.3136645962732919
Recall on test set :  0.391304347826087
Precision on training set :  0.7266187050359713
Precision on test set :  0.7105263157894737
```

No overfitting detected but poor recall.

Gradient Boosting Classifier

In [64]:
```
gBoost = GradientBoostingClassifier(random_state=1)
gBoost.fit(x_train,y_train)
```

Out[64]:
```
GradientBoostingClassifier(random_state=1)
```

In [65]:
```
make_cm(gBoost,y_test)
```

In [66]:
```
gBoost_score = get_score_acc_rec_prec(gBoost)
print_score(gBoost_score,'all')
```

```
Accuracy on training set :  0.8884986830553117
Accuracy on test set :  0.868259385665529
Recall on training set :  0.4798136645962733
Recall on test set :  0.42391304347826086
Precision on training set :  0.8704225352112676
Precision on test set :  0.7748344370860927
```

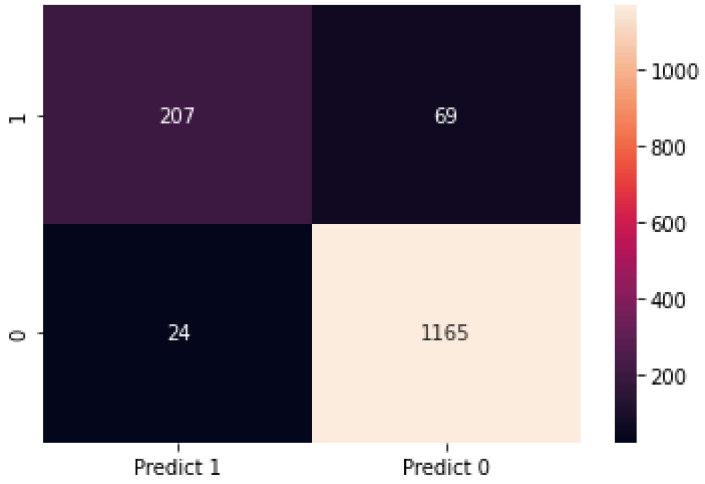
No overfitting detected but poor recall.

XGBoost Classifier

In [67]:
```
xgBoost = XGBClassifier(random_state=1, eval_metric='logloss')
xgBoost.fit(x_train,y_train)
```

Out[67]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, eval_metric='logloss',
              gamma=0, gpu_id=-1, importance_type='gain',
              interaction_constraints='', learning_rate=0.300000012,
              max_delta_step=0, max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=100, n_jobs=8,
              num_parallel_tree=1, random_state=1, reg_alpha=0, reg_lambda=1,
              scale_pos_weight=1, subsample=1, tree_method='exact',
              validate_parameters=1, verbosity=None)
```

In [68]:
```
make_cm(xgBoost,y_test)
```



In [69]:
```
xgBoost_score = get_score_acc_rec_prec(xgBoost)
print_score(xgBoost_score,'all')
```

```
Accuracy on training set :  0.9997073456248171
Accuracy on test set :  0.936518771331058
Recall on training set :  0.9984472049689441
Recall on test set :  0.75
Precision on training set :  1.0
Precision on test set :  0.8961038961038961
```

Overfitting is present, but pretty good recall.

Improving Models

AdaBoost

In [70]:
```
adaBoostOptimum = AdaBoostClassifier(random_state=1)

parameters_adaBoost = {
    "base_estimator":[DecisionTreeClassifier(max_depth=1),DecisionTreeClassifi
er(max_depth=2),
                      DecisionTreeClassifier(max_depth=3)],
    "n_estimators": np.arange(10,110,10),
    "learning_rate":np.arange(0.1,2,0.2)
}

grid_adaBoost = GridSearchCV(adaBoostOptimum, parameters_adaBoost, scoring="re
call",cv=5)
grid_adaBoost = grid_adaBoost.fit(x_train, y_train)

adaBoostOptimum = grid_adaBoost.best_estimator_
adaBoostOptimum.fit(x_train, y_train)
```

Out[70]:
```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=3),
                   learning_rate=1.7000000000000004, n_estimators=90,
                   random_state=1)
```

In [71]:
```
make_cm(adaBoostOptimum,y_test)
```

In [72]:
```
adaBoostOptimum_score = get_score_acc_rec_prec(adaBoostOptimum)
print_score(adaBoostOptimum_score,'all')
```

```
Accuracy on training set :  0.9760023412350015
Accuracy on test set :  0.8703071672354948
Recall on training set :  0.9130434782608695
Recall on test set :  0.5905797101449275
Precision on training set :  0.9576547231270358
Precision on test set :  0.6791666666666667
```

A lot better than not optimized adaBoost, but still not satisfying recall.

Gradient

In [73]:
```
gBoostOptimum = GradientBoostingClassifier(init=AdaBoostClassifier(random_stat
e=1),random_state=1)

parameters_gBoost = {
    "n_estimators": [100,180,250],
    "subsample":[0.8,0.9,1],
    "max_features":[0.7,0.9,1]
}

grid_gBoost = GridSearchCV(gBoostOptimum, parameters_gBoost, scoring="recall",
cv=5)
grid_gBoost = grid_gBoost.fit(x_train, y_train)

gBoostOptimum = grid_gBoost.best_estimator_
gBoostOptimum.fit(x_train, y_train)
```

Out[73]:
```
GradientBoostingClassifier(init=AdaBoostClassifier(random_state=1),
                           max_features=0.9, n_estimators=250, random_state=
1,
                           subsample=0.8)
```

In [74]:
```
make_cm(gBoostOptimum,y_test)
```

In [75]:
```
gBoostOptimum_score = get_score_acc_rec_prec(gBoostOptimum)
print_score(gBoostOptimum_score,'all')
```

```
Accuracy on training set :  0.9209833187006146
Accuracy on test set :  0.889419795221843
Recall on training set :  0.6211180124223602
Recall on test set :  0.5181159420289855
Precision on training set :  0.9389671361502347
Precision on test set :  0.8313953488372093
```

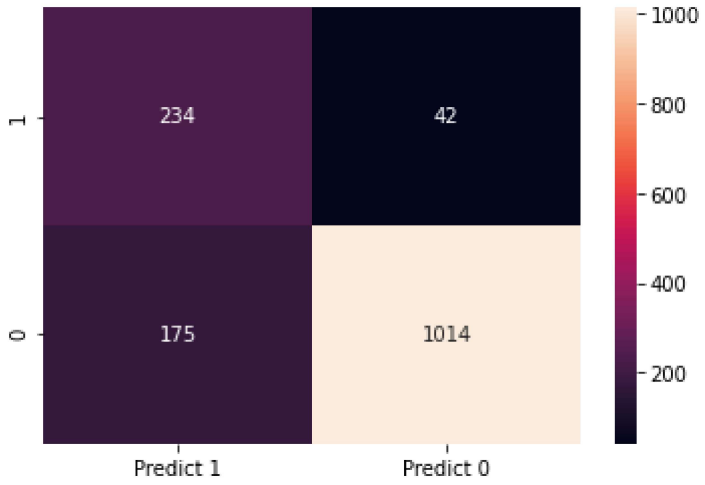
A lot better than not optimized gradientBoost, but still not satisfying recall.

XGBoost Classifier

In [76]:
```
xgBoostOptimum= XGBClassifier(random_state=1, eval_metric='logloss')

parameters_xgBoost = {
    "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]
}

grid_xgBoosting = GridSearchCV(xgBoostOptimum, parameters_xgBoost,scoring="recall",cv=5)
grid_xgBoosting = grid_xgBoosting.fit(x_train, y_train)

xgBoostOptimum = grid_xgBoosting.best_estimator_
xgBoostOptimum.fit(x_train, y_train)
```

Out[76]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=0.5,
              colsample_bynode=1, colsample_bytree=0.7, eval_metric='logloss',
              gamma=0, gpu_id=-1, importance_type='gain',
              interaction_constraints='', learning_rate=0.1, max_delta_step=0,
              max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=50, n_jobs=8,
              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 [77]:
```
make_cm(xgBoostOptimum,y_test)
```



In [78]:
```
xgBoostOptimum_score = get_score_acc_rec_prec(xgBoostOptimum)
print_score(xgBoostOptimum_score,'all')
```

```
Accuracy on training set :  0.9201053555750659
Accuracy on test set :  0.851877133105802
Recall on training set :  0.9503105590062112
Recall on test set :  0.8478260869565217
Precision on training set :  0.7174677608440797
Precision on test set :  0.5721271393643031
```

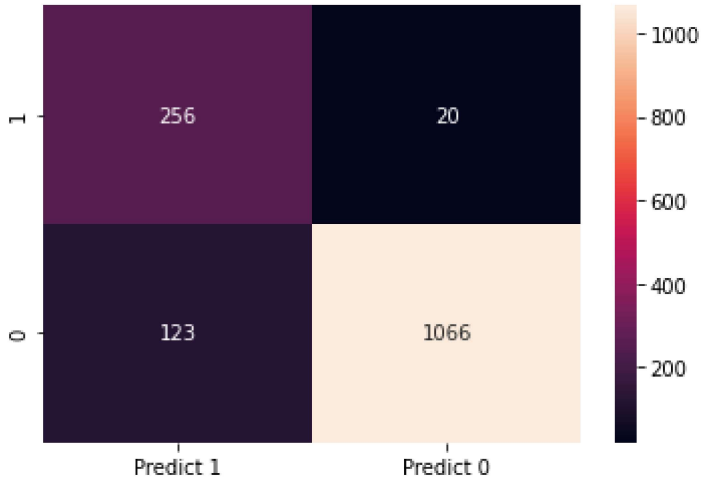
Much better than not optimized xgBoost. Decent recall. need to check if better model can be found.

Stacking Classifier

In [79]:
```
estimators = [('Random Forest',rForestOptimum), ('Gradient Boosting',gBoostOpt
imum), ('Decision Tree',dTreeOptimum)]
final_estimator = xgBoostOptimum

stacking = StackingClassifier(estimators=estimators,final_estimator=final_esti
mator)
stacking.fit(x_train,y_train)
```

Out[79]:
```
StackingClassifier(estimators=[('Random Forest',
                                RandomForestClassifier(max_features=0.7,
                                                       n_estimators=501,
                                                       random_state=1)),
                               ('Gradient Boosting',
                                GradientBoostingClassifier(init=AdaBoostClass
ifier(random_state=1),
                                                           max_features=0.9,
                                                           n_estimators=250,
                                                           random_state=1,
                                                           subsample=0.8)),
                               ('Decision Tree',
                                DecisionTreeClassifier(class_weight={0: 0.25,
                                                                     1: 0.7
5},
                                                       max_depth...
                                              eval_metric='logloss', gamma
=0,
                                              gpu_id=-1,
                                              importance_type='gain',
                                              interaction_constraints='',
                                              learning_rate=0.1,
                                              max_delta_step=0, max_depth=
6,
                                              min_child_weight=1,
                                              missing=nan,
                                              monotone_constraints='()',
                                              n_estimators=50, n_jobs=8,
                                              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 [80]:
```
make_cm(stacking,y_test)
```



In [81]:
```
stacking_score = get_score_acc_rec_prec(stacking)
print_score(stacking_score,'all')
```

```
Accuracy on training set :  0.9961954931226222
Accuracy on test set :  0.9023890784982935
Recall on training set :  1.0
Recall on test set :  0.927536231884058
Precision on training set :  0.9802130898021308
Precision on test set :  0.6754617414248021
```

Compare All Models

In [82]:
```
#list of all the models
modelList = [dTree,dTreeOptimum,bClassifier,bClassifierPlus,bClassifierOptimu
m,rForest,rForestPlus,rForestOptimum,
             adaBoost,gBoost,xgBoost,adaBoostOptimum,gBoostOptimum,xgBoostOpti
mum,stacking]

acc_train = []
acc_test = []
rec_train = []
rec_test = []
prec_train = []
prec_test = []

for model in modelList:
    score = get_score_acc_rec_prec(model)
    acc_train.append(score[0])
    acc_test.append(score[1])
    rec_train.append(score[2])
    rec_test.append(score[3])
    prec_train.append(score[4])
    prec_test.append(score[5])
```

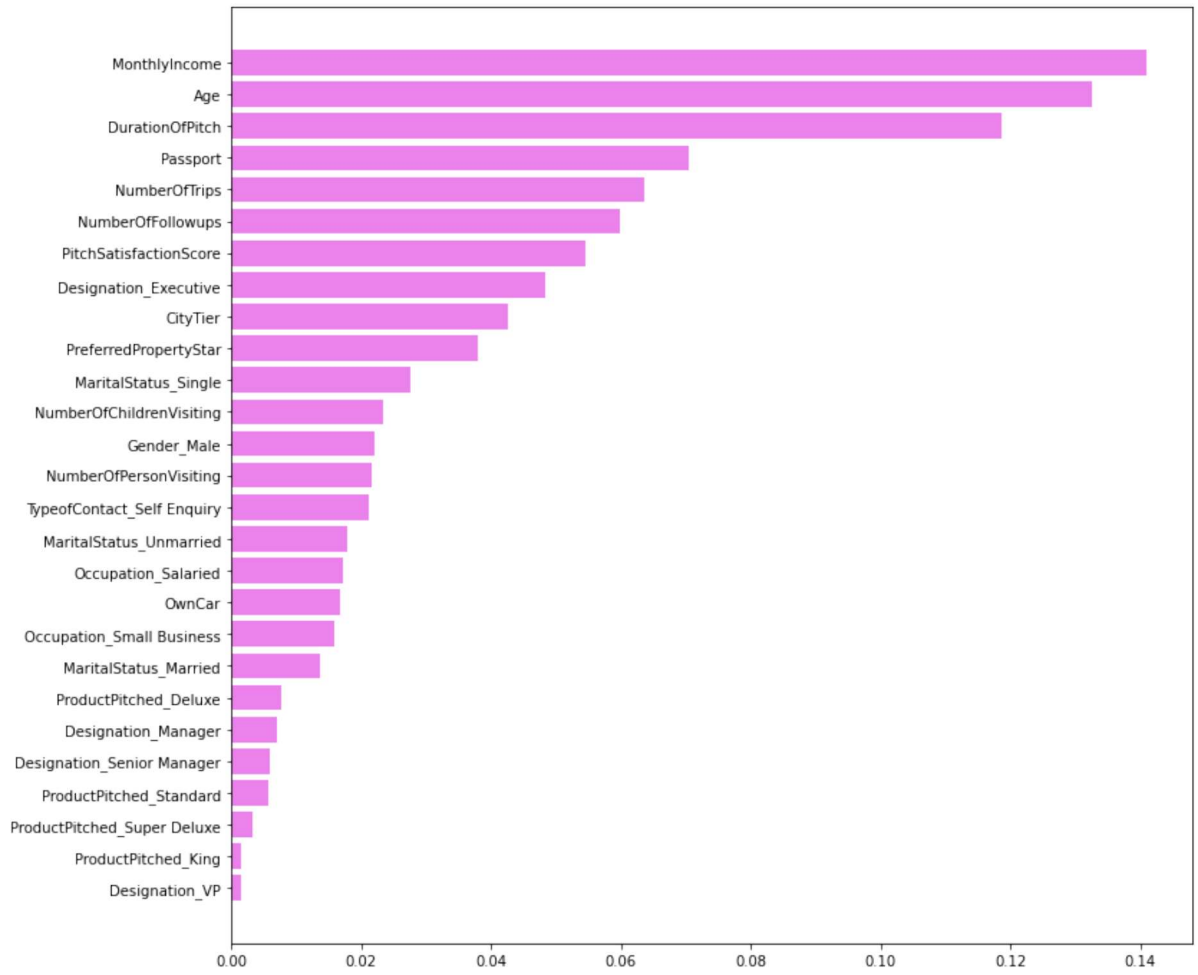
In [83]:
```
comparison_frame = pd.DataFrame({'Model':['Decision Tree','Optimum Decision Tr
ee','Bagging Classifier',
                                          'Weighted Bagging Classifier','Optim
um Bagging Classifier',
                                          'Random Forest','Weighted Random For
est','Optimum Random Forest',
                                          'AdaBoost Classifier','Gradient Boos
ting Classifier','XGBoost Classifier',
                                          'Optimum AdaBoost Classifier','Optim
um Gradient Boosting Classifier',
                                          'Optimum XGBoost Classifier','Stacki
ng Classifier'],
                                          'TrainAccuracy': acc_train,'TestAccu
racy': acc_test,
                                          'TrainRecall':rec_train,'TestRecal
l':rec_test,
                                          'TrainPrecision':prec_train,'TestPre
cision':prec_test})


comparison_frame
```

Out[83]:

	Model	TrainAccuracy	TestAccuracy	TrainRecall	TestRecall	TrainPrecision	TestPrecision
0	Decision Tree	1.000000	0.893515	1.000000	0.728261	1.000000	0.712766
1	Optimum Decision Tree	0.800702	0.787031	0.641304	0.652174	0.478563	0.454545
2	Bagging Classifier	0.992684	0.911945	0.961180	0.644928	1.000000	0.851675
3	Weighted Bagging Classifier	0.990635	0.905119	0.956522	0.579710	0.993548	0.874317
4	Optimum Bagging Classifier	0.996488	0.918089	0.984472	0.663043	0.996855	0.871429
5	Random Forest	1.000000	0.922867	1.000000	0.623188	1.000000	0.950276
6	Weighted Random Forest	1.000000	0.916724	1.000000	0.594203	1.000000	0.942529
7	Optimum Random Forest	1.000000	0.932423	1.000000	0.731884	1.000000	0.889868
8	AdaBoost Classifier	0.848405	0.855290	0.313665	0.391304	0.726619	0.710526
9	Gradient Boosting Classifier	0.888499	0.868259	0.479814	0.423913	0.870423	0.774834
10	XGBoost Classifier	0.999707	0.936519	0.998447	0.750000	1.000000	0.896104
11	Optimum AdaBoost Classifier	0.976002	0.870307	0.913043	0.590580	0.957655	0.679167
12	Optimum Gradient Boosting Classifier	0.920983	0.889420	0.621118	0.518116	0.938967	0.831395
13	Optimum XGBoost Classifier	0.920105	0.851877	0.950311	0.847826	0.717468	0.572127
14	Stacking Classifier	0.996195	0.902389	1.000000	0.927536	0.980213	0.675462

Feature Importance (Measured with Random Forest)

In [84]:
```
feature_names = x_train.columns
importance = rForestOptimum.feature_importances_
indices = np.argsort(importance)

plt.figure(figsize=(12,12))
plt.barh(range(len(indices)), importance[indices], color='violet', align='cent
er')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.show()
```



Conclusion & Insights:

Since the new project will focus on wellness, it is also important to check the customer interest towards health. However, knowing that older people tend to be more concerned about health, the age target will be high and the product is flexible(okay to be expensive) with the price.

After EDA and Model Building, it became more clear that:

- The monthly income of the customer will mostly determine which tour package will be chosen.
- Age is also closely related with which package will be chosen. Note age is highly correlated with monthly income.
- Duration of the pitch, passport, and number of trips are ranked below monthly income and age.
- It seems that designation of customer limits the exposure of various packages. It should be seriously considered to advertise customers with multiple packages at a time. Or, data is not as accurate which products were advertised and which package was taken.
- Executives take tours more than any other designation. Consider setting marketing priorities.
- Consider developing abroad or driving tour since people with passport are likely to have tour.

In [ ]: