Cross selling Vehicle Insurance to Health Insurance Customers- A Machine Learning Approach

3/15/2021

Poojitha Bhat

Introduction:

Cross selling is one of the business strategies used by a lot of companies to improve their profits. With an access to the existing customer base, cross selling can significantly improve a company's performance without having to acquire new customer base. In order to achieve the same, the companies usually target all the existing customers for a new/different product and expect a conversion. However with right metrics it is possible to frame a strategy to identify the correct customers and the channels to approach them.

The aim of this project is to use Machine Learning to predict if a customer is likely to buy vehicle insurance from the company provided he already owns a health insurance from the same company.

An ML model in this scenario will be able to identify the right set of customers to target. It will also reduce the resources in usage by the company and contribute significantly to the profits if used to strategize correctly.

Python and Tableau are the main tools used for the development and analysis of the models in this exercise.

Data Cleansing

The data was sourced from Kaggle(https://www.kaggle.com/anmolkumar/health-insurance-cross-sell-prediction). The CSV imported into the kernel for further analysis.

```
#Importing the labeled dataset available for training the algorithm
# The dataset contains details of the Vehicle insurance holders and response to the cross sales targeting
health_insurance = pd.read_csv('train.csv')
```

```
type(health_insurance)
```

```
pandas.core.frame.DataFrame
```

First five rows obtained to understand how the data is structured.

```
health_insurance.head()
```

	id	Gender	Age	Driving_License	Region_Code	Previously_Insured	Vehicle_Age	Vehicle_Damage	Annual_Premium	Policy_Sales_Channel	Vintage
0	1	Male	44	1	28.0	0	> 2 Years	Yes	40454.0	26.0	217
1	2	Male	76	1	3.0	0	1-2 Year	No	33536.0	26.0	183
2	3	Male	47	1	28.0	0	> 2 Years	Yes	38294.0	26.0	27
3	4	Male	21	1	11.0	1	< 1 Year	No	28619.0	152.0	203
4	5	Female	29	1	41.0	1	< 1 Year	No	27496.0	152.0	39

Policy_Sales_Channel	Vintage	Response
26.0	217	1
26.0	183	0
26.0	27	1
152.0	203	0
152.0	39	0

It can be seen that we have information about Gender and Age of the health insurance holder, the region to which the person belongs to, the age of the insurance holder, if the vehicle has been previously insured or not, the annual premium paid by the customer, the sales channel through which he was reached out to, the number of days for which the customer

has been on books and finally the Response to a cross sale lead. The variable response is going to be our target variable. The rest will be our independent features.

```
health_insurance.shape
```

```
(381109, 12)
```

The data contains 381109 rows and 12 columns.

```
health_insurance.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 381109 entries, 0 to 381108
Data columns (total 12 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   id                    381109 non-null  int64  
 1   Gender                381109 non-null  object 
 2   Age                   381109 non-null  int64  
 3   Driving_License       381109 non-null  int64  
 4   Region_Code           381109 non-null  float64
 5   Previously_Insured    381109 non-null  int64  
 6   Vehicle_Age           381109 non-null  object 
 7   Vehicle_Damage        381109 non-null  object 
 8   Annual_Premium        381109 non-null  float64
 9   Policy_Sales_Channel  381109 non-null  float64
 10  Vintage               381109 non-null  int64  
 11  Response              381109 non-null  int64  
dtypes: float64(3), int64(6), object(3)
memory usage: 34.9+ MB
```

There are 9 columns of numeric datatype and 3 of object type seen.

Of the numeric datatype, id,Age, Driving_license,Prevously Insured, Vintage and Response are seen to be integers.Region_Code, Annual Premium and Policy Sales Channel are seen to be float.

Gender, Vehicle Age and Vehicle Damage are seen to be of object type.

Integers seem to be either IDs or categorical variables. While objects are purely categorical, Floats are purely amounts.

Further analysis needs to be done to understand these assumptions.

```
health_insurance.describe()
```

	id	Age	Driving_License	Region_Code	Previously_Insured	Annual_Premium	Policy_Sales_Channel	Vintage	Respor
count	381109.000000	381109.000000	381109.000000	381109.000000	381109.000000	381109.000000	381109.000000	381109.000000	381109.000(
mean	190555.000000	38.822584	0.997869	26.388807	0.458210	30564.389581	112.034295	154.347397	0.122
std	110016.836208	15.511611	0.046110	13.229888	0.498251	17213.155057	54.203995	83.671304	0.327
min	1.000000	20.000000	0.000000	0.000000	0.000000	2630.000000	1.000000	10.000000	0.000(
25%	95278.000000	25.000000	1.000000	15.000000	0.000000	24405.000000	29.000000	82.000000	0.000(
50%	190555.000000	36.000000	1.000000	28.000000	0.000000	31669.000000	133.000000	154.000000	0.000(
75%	285832.000000	49.000000	1.000000	35.000000	1.000000	39400.000000	152.000000	227.000000	0.000(
max	381109.000000	85.000000	1.000000	52.000000	1.000000	540165.000000	163.000000	299.000000	1.000(

Below are the findings from the data:

- The ids seem to be unique integer identifiers of the customers. Their uniqueness to be determined.

- The minimum age of the customer opting for Vehicle insurance is 20 while maximum age is 85. On an average the age of a customer opting for vehicle insurance is 38-39.

- Driving License as per the assumption is a categorical variable containing only 0s and 1s.99.78% of the customers opting for the insurance have a driving license.

- Region_code as per assumption is also a categorical variable representing the region from which the customer is opting for the insurance. There seem to be 53 regions with codes ranging from 0 to 52.

- Previously insured is also a categorical variable with 1 standing for 'Yes' and 0 standing for 'No'. 45.82% of the customers are seen to hold insurance previously.

- Annual Premium represents the amount to be paid annually for the insurance. It is seen to range from 2630 to as high as 540,165 dollars. On an average a customer pays a premium of 30,564 for annual premium

- Policy sales channel is a categorical variable ranging from 1 to 163. There seem to be 163 channels of reaching out to customers.

- Vintage representing the number of days a customer has been with the company. The most recent customer has been with the company for 10 days and the oldest one for 299 days. On an average a customer has been for 154 days.

- The response can be either 1 for 'Yes' or a 0 for 'No'. 12% of the health insurance holders seem to have responded positively at opting for a vehicle insurance

```
health_insurance['id'].nunique()
```

```
381109
```

The IDs seem to be unique for all the rows and hence are unique identifiers of each customer.

Let’s find out if there are any missing values that do not fit in.

```
nullcounts = health_insurance.isnull().sum().sort_values(ascending=False)
```

```
nullcounts
```

```
Response                0
Vintage                 0
Policy_Sales_Channel    0
Annual_Premium          0
Vehicle_Damage          0
Vehicle_Age             0
Previously_Insured      0
Region_Code             0
Driving_License         0
Age                     0
Gender                  0
id                      0
dtype: int64
```

There are no missing values found in any of the variables.

Let’s check if there are any values that do not fit into the definition of the variables they’re representing.

```
health_insurance['Gender'].value_counts()
```

```
Male      206089
Female    175020
```

```
health_insurance['Gender'] = health_insurance['Gender'].astype(str)
health_insurance['Gender'].replace({'Male':'M', 'Female': 'F'}, inplace =True)
```

```
health_insurance['Gender'].unique()
```

```
array(['M', 'F'], dtype=object)
```

There are no missing values seen. The datatype of the field is of integer and hence changing that as object. Replacing Male and Female with M and F respectively for further usage.

Age:

```
health_insurance['Age'].value_counts()
```

```
24    25960
23    24256
22    20964
25    20636
21    16457
      ...
81       56
82       29
83       22
84       11
85       11
Name: Age, Length: 66, dtype: int64
```

No Outliers or missing values seen in Age. The range of age is also practically placed from 24 to 85.

Driving License:

```
health_insurance['Driving_License'].value_counts()
```

```
1    380297
0       812
Name: Driving_License, dtype: int64
```

The customers either have a driving license or they do not. The values make sense however there is an imbalance seen here.

Previously Insured:

```
health_insurance['Previously_Insured'].value_counts()
```

```
0    206481
1    174628
```

The customers may (1) or may not (0) be insured previously. The representation is also seen to be balanced.

Region Code:

```
health_insurance['Region_Code'].value_counts()
```

```
28.0    106415
8.0      33877
46.0     19749
41.0     18263
15.0     13308
30.0     12191
29.0     11065
50.0     10243
```

```
  43.0      2639
  17.0      2617
  26.0      2587
  25.0      2503
  24.0      2415
  38.0      2026
  0.0       2021
  16.0      2007
  31.0      1960
  23.0      1960
  20.0      1935
  49.0      1832
  4.0       1801
  34.0      1664
  19.0      1535
  22.0      1309
  40.0      1295
  5.0       1279
  1.0       1008
  44.0       808
  42.0       591
  52.0       267
  51.0       183
  Name: Region_Code, dtype: int64
```

There are 53 region codes seen. No missing values found.

The precision will be set to 0 in order to remove the decimal points.

```
pd.set_option('precision', 0)
```

```
health_insurance['Region_Code'].value_counts()
```

```
28    106415
8      33877
46     19749
41     18263
15     13308
30     12191
29     11065
50     10243
3       9251
```

The decimal points have now been removed. The values represent the region codes accurately.

Annual Premium:

```
health_insurance['Annual_Premium'].value_counts()
```

```
2630     64877
69856      140
39008       41
38287       38
45179       38
           ...
62326        1
59733        1
55934        1
75387        1
53346        1
```

There are outliers in annual premiums towards the higher range the remaining values seem to fall in a practical range. The premiums in higher range will be dealt with in the preprocessing stage by using appropriate bins.

Policy Sales Channel:

```
health_insurance['Policy_Sales_Channel'].value_counts()
```

```
152    134545
26      79476
124     73835
160     21754
156     10653
           ...
84          1
123         1
144         1
43          1
143         1
```

The channels have been masked using integer codes and seem to be following a pattern. There are certain channels with fewer responses seen. These will be dealt with in the pre processing stage.

Vintage:

```
health_insurance['Vintage'].unique()
```

```
array([217, 183,  27, 203,  39, 176, 249,  72,  28,  80,  46, 289, 221,
        15,  58, 147, 256, 299, 158, 102, 116, 177, 232,  60, 180,  49,
        57, 223, 136, 222, 149, 169,  88, 253, 107, 264, 233,  45, 184,
       251, 153, 186,  71,  34,  83,  12, 246, 141, 216, 130, 282,  73,
       171, 283, 295, 165,  30, 218,  22,  36,  79,  81, 100,  63, 242,
       277,  61, 111, 167,  74, 235, 131, 243, 248, 114, 281,  62, 189,
       139, 138, 209, 254, 291,  68,  92,  52,  78, 156, 247, 275,  77,
       181, 229, 166,  16,  23,  31, 293, 219,  50, 155,  66, 260,  19,
       258, 117, 193, 204, 212, 144, 234, 206, 228, 125,  29,  18,  84,
       230,  54, 123, 101,  86,  13, 237,  85,  98,  67, 128,  95,  89,
        99, 208, 134, 135, 268, 284, 119, 226, 105, 142, 207, 272, 263,
        64,  40, 245, 163,  24, 265, 202, 259,  91, 106, 190, 162,  33,
       194, 287, 292,  69, 239, 132, 255, 152, 121, 150, 143, 198, 103,
       127, 285, 214, 151, 199,  56,  59, 215, 104, 238, 120,  21,  32,
       270, 211, 200, 197,  11, 213,  93, 113, 178,  10, 290,  94, 231,
       296,  47, 122, 271, 278, 276,  96, 240, 172, 257, 224, 173, 220,
       185,  90,  51, 205,  70, 160, 137, 168,  87, 118, 288, 126, 241,
        82, 227, 115, 164, 236, 286, 244, 108, 274, 201,  97,  25, 174,
       182, 154,  48,  20,  53,  17, 261,  41, 266,  35, 140, 269, 146,
       145,  65, 298, 133, 195,  55, 188,  75,  38,  43, 110,  37, 129,
       170, 109, 267, 279, 112, 280,  76, 191,  26, 161, 179, 175, 252,
        42, 124, 187, 148, 294,  44, 157, 192, 262, 159, 210, 250,  14,
       273, 297, 225, 196], dtype=int64)
```

Vintage represents the number of days for which the customers have been on books. We see that the values are under 365. This means that the data has been collected for latest customers only.

Vehicle Age:

```
health_insurance['Vehicle_Age'].value_counts()
```

```
1-2 Year     200316
< 1 Year     164786
> 2 Years     16007
```

The vehicles are seen to be less than 1 year, 1-2 years or more than 2 years old.

Vehicle Damage:

```
health_insurance['Vehicle_Damage'].value_counts()
```

```
Yes    192413
No     188696
```

The vehicles are either damaged in the past or not damaged. We have a balanced view of the same.

Response:

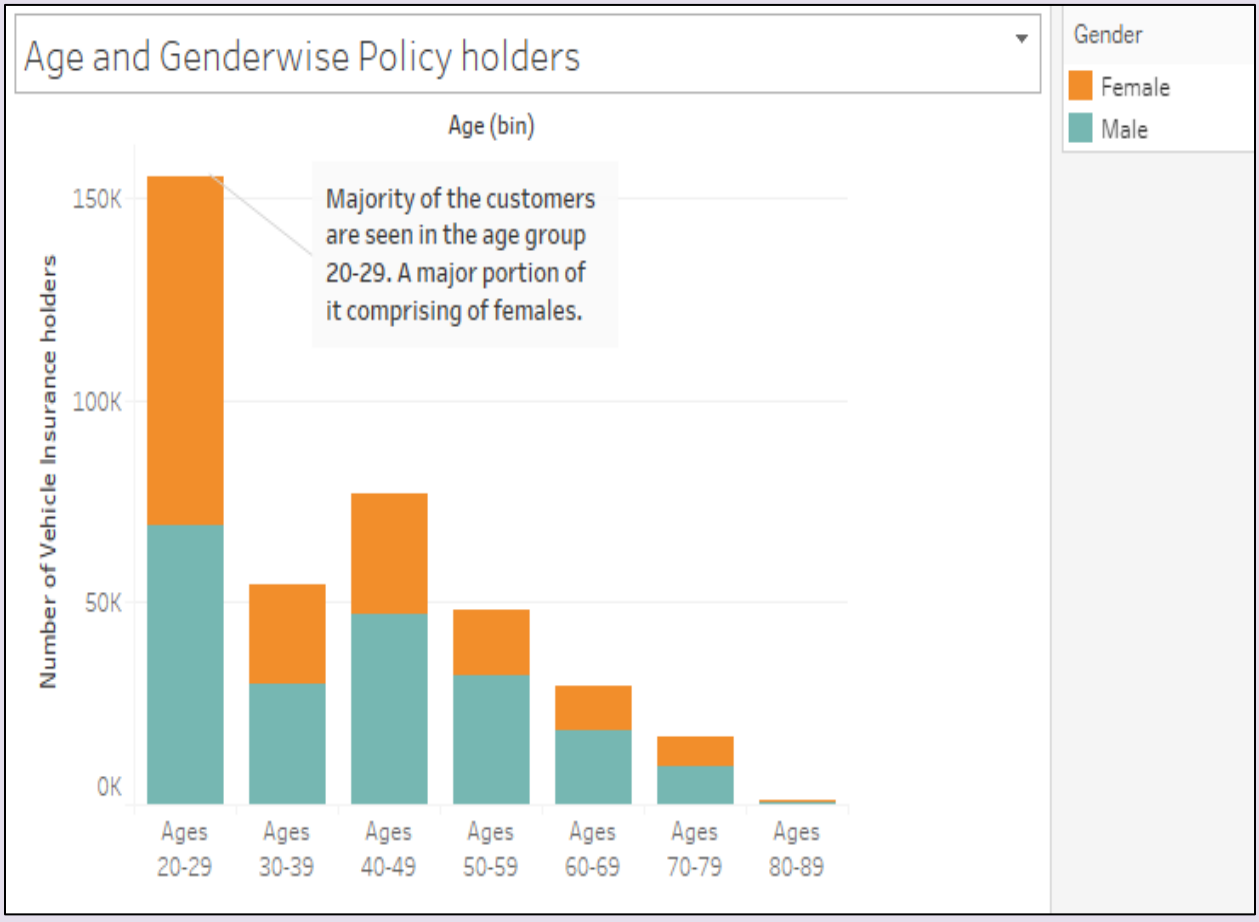
```
health_insurance['Response'].value_counts()
```

```
0    333744
1     46587
```

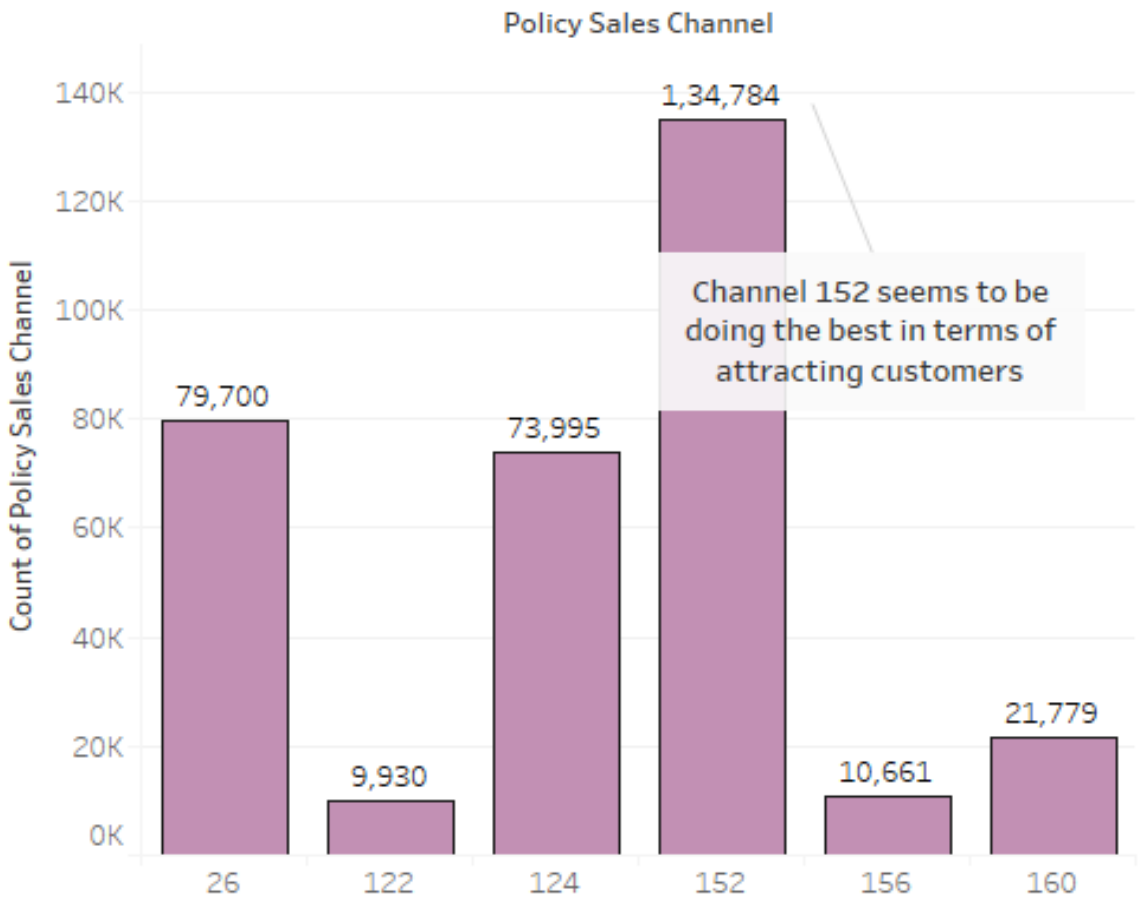
46587 responses have been recorded as Yes and remaining 333744 as No. There is an imbalance seen in the responses.

Exploratory Data Analysis

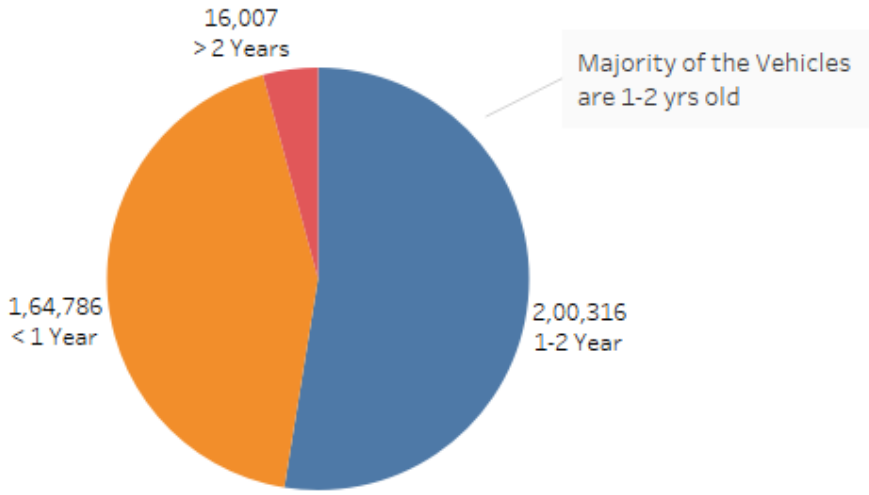
Let's now explore how each of these variables are distributed and how they relate to the responses received.



Channelwise policies

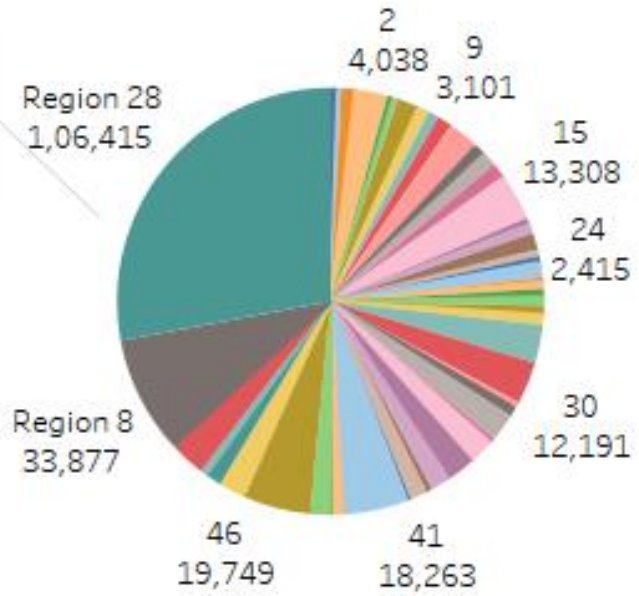


Policy holding based on ages of vehicles

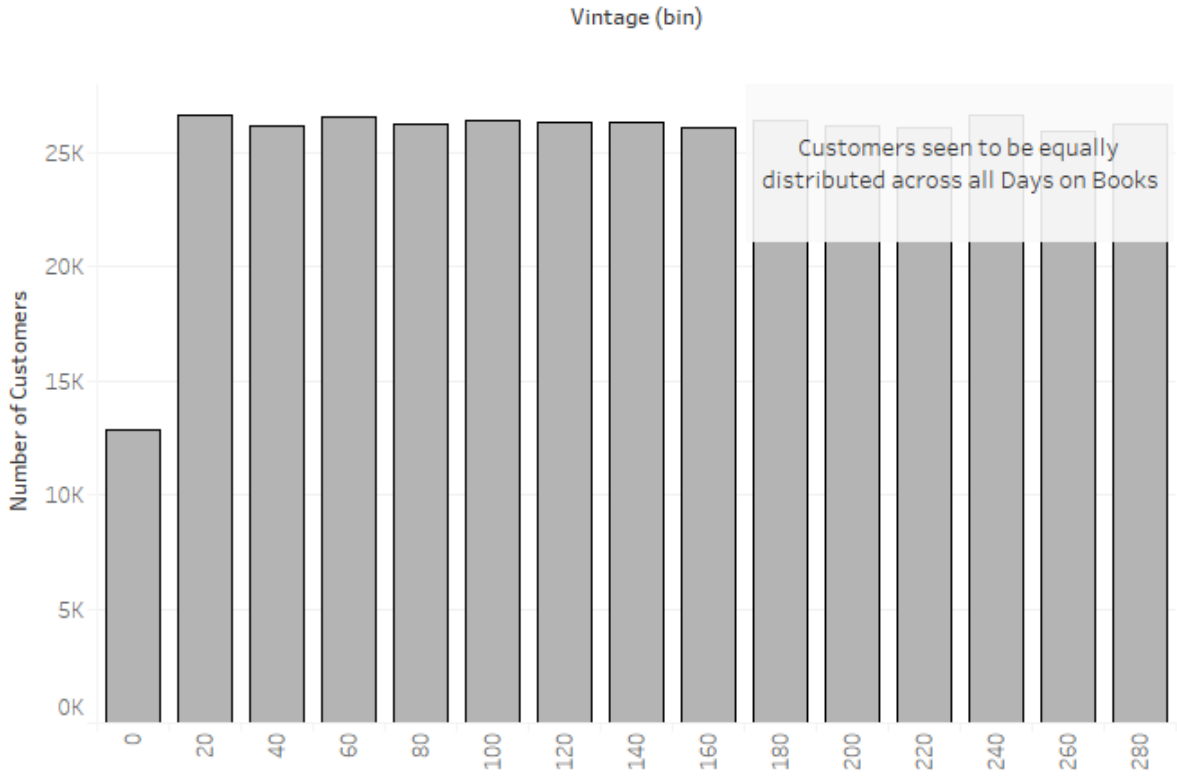


Regionwise policies

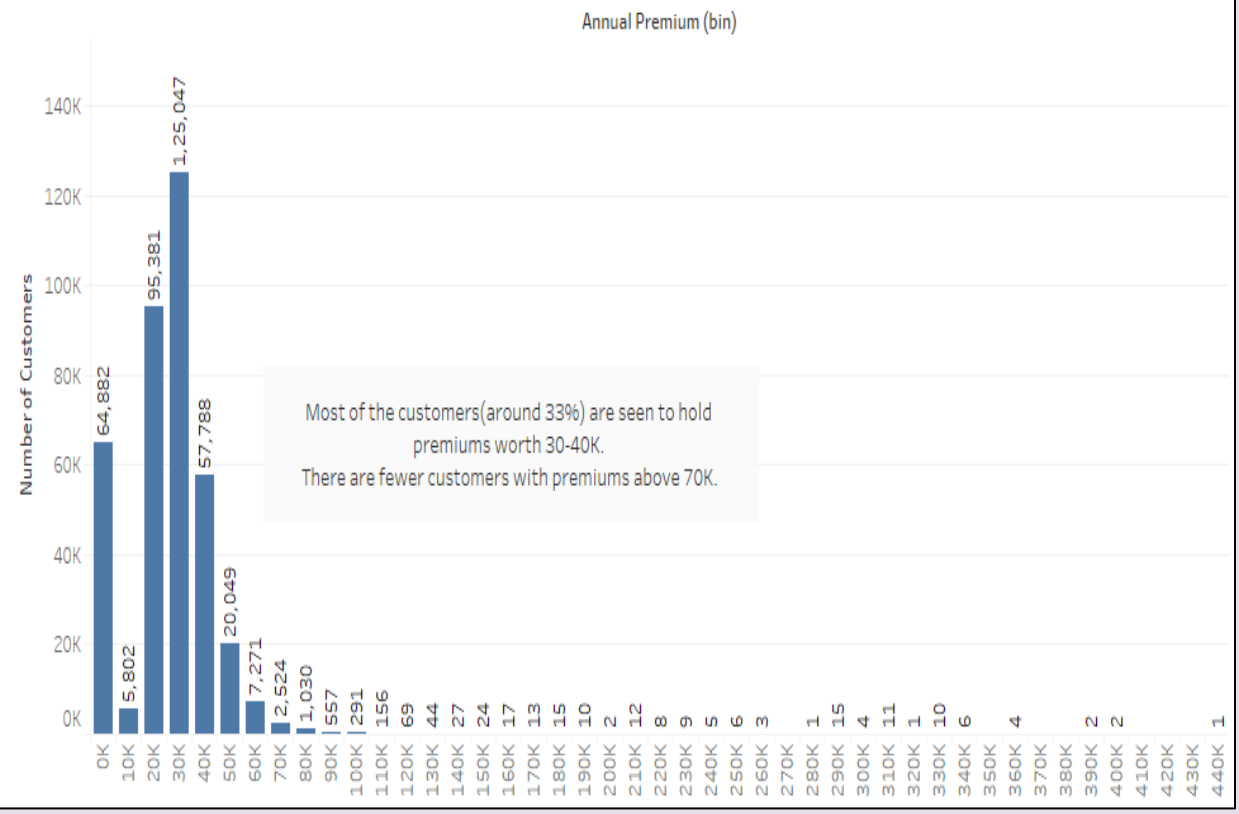
Region 28 with 27.92% policies has the highest number of policy holders



Customers based on Days on Books

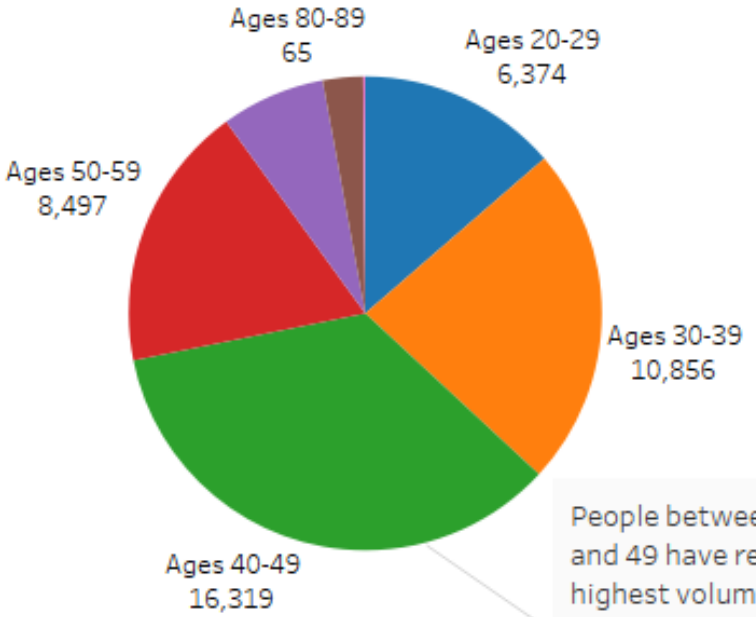


Annual premium distribution

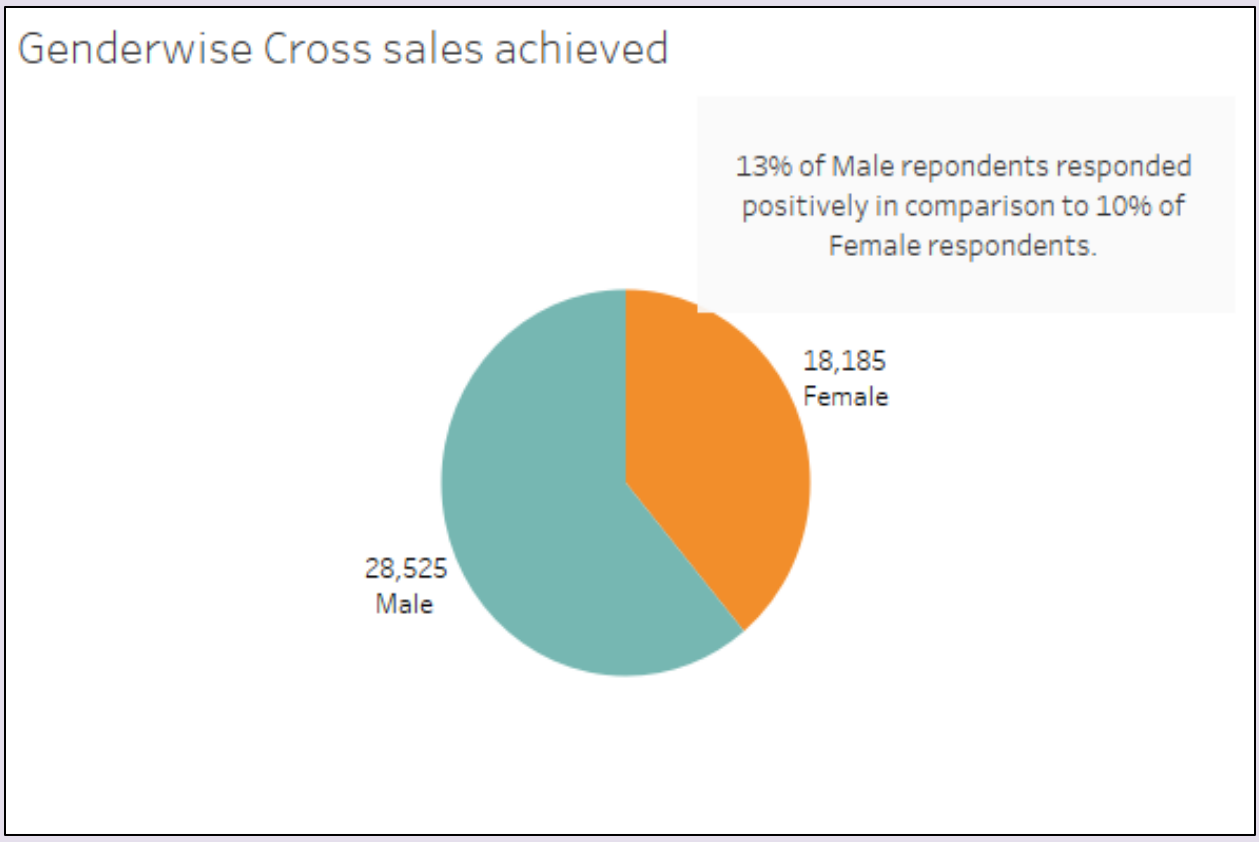


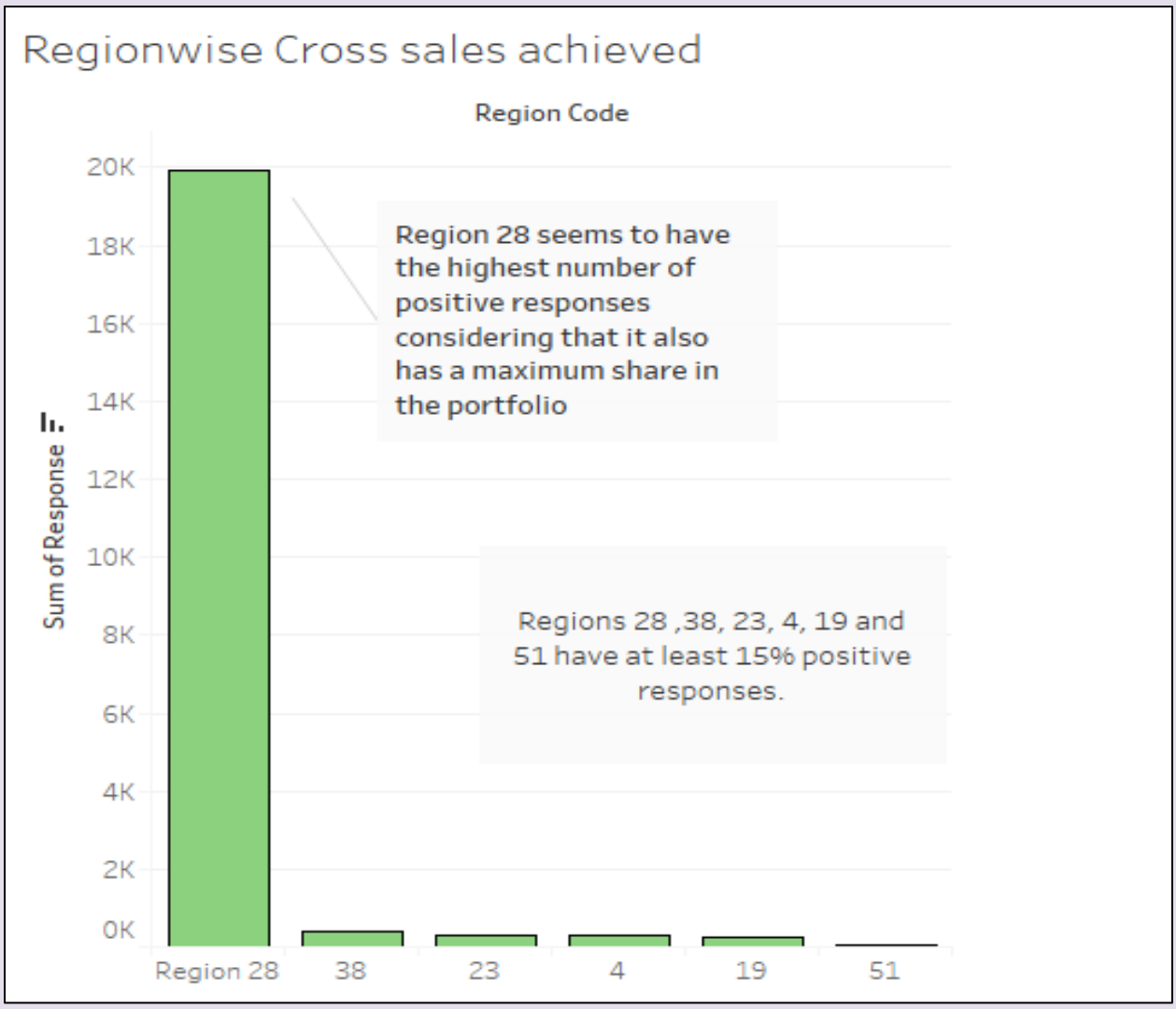
EDA on the responses received:

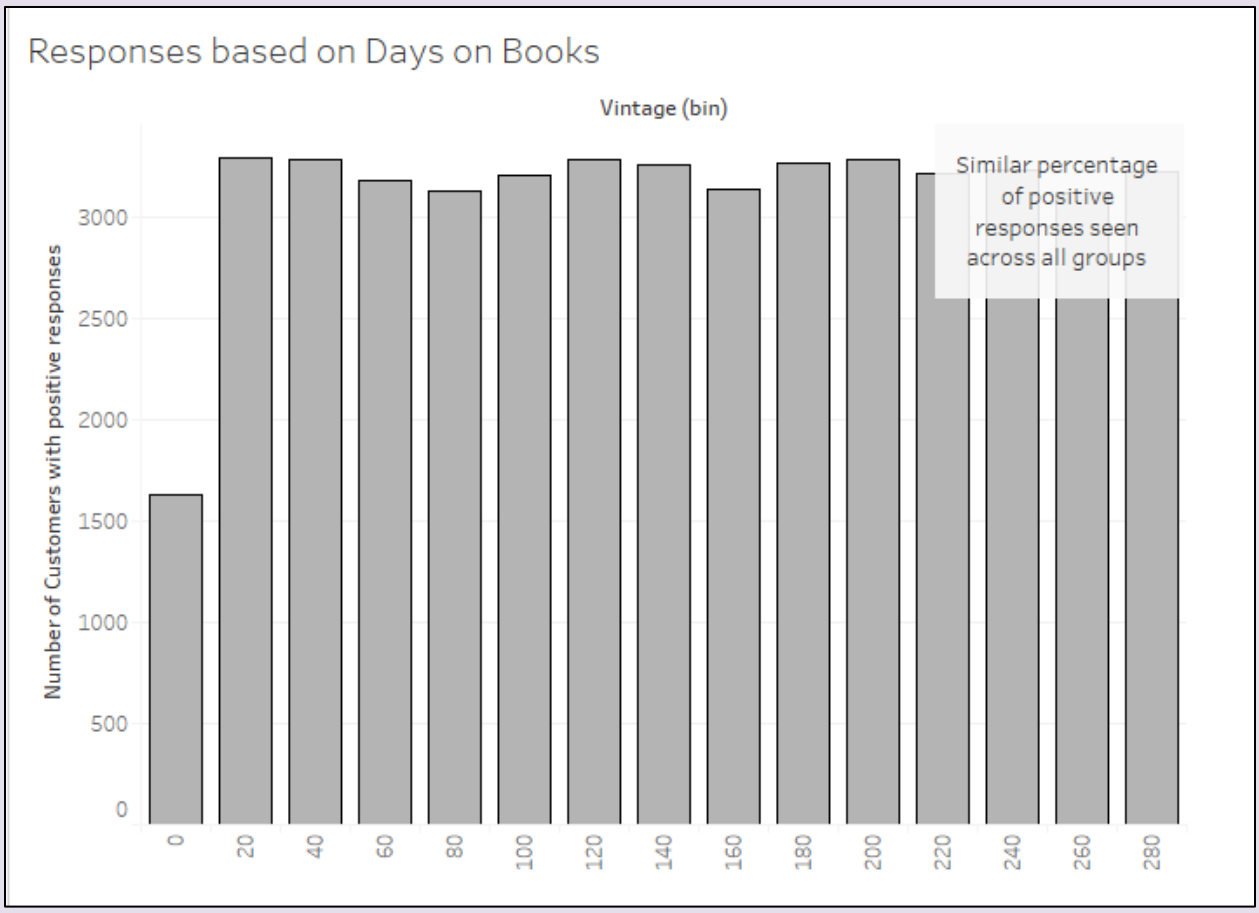
Cross sales achieved by age groups

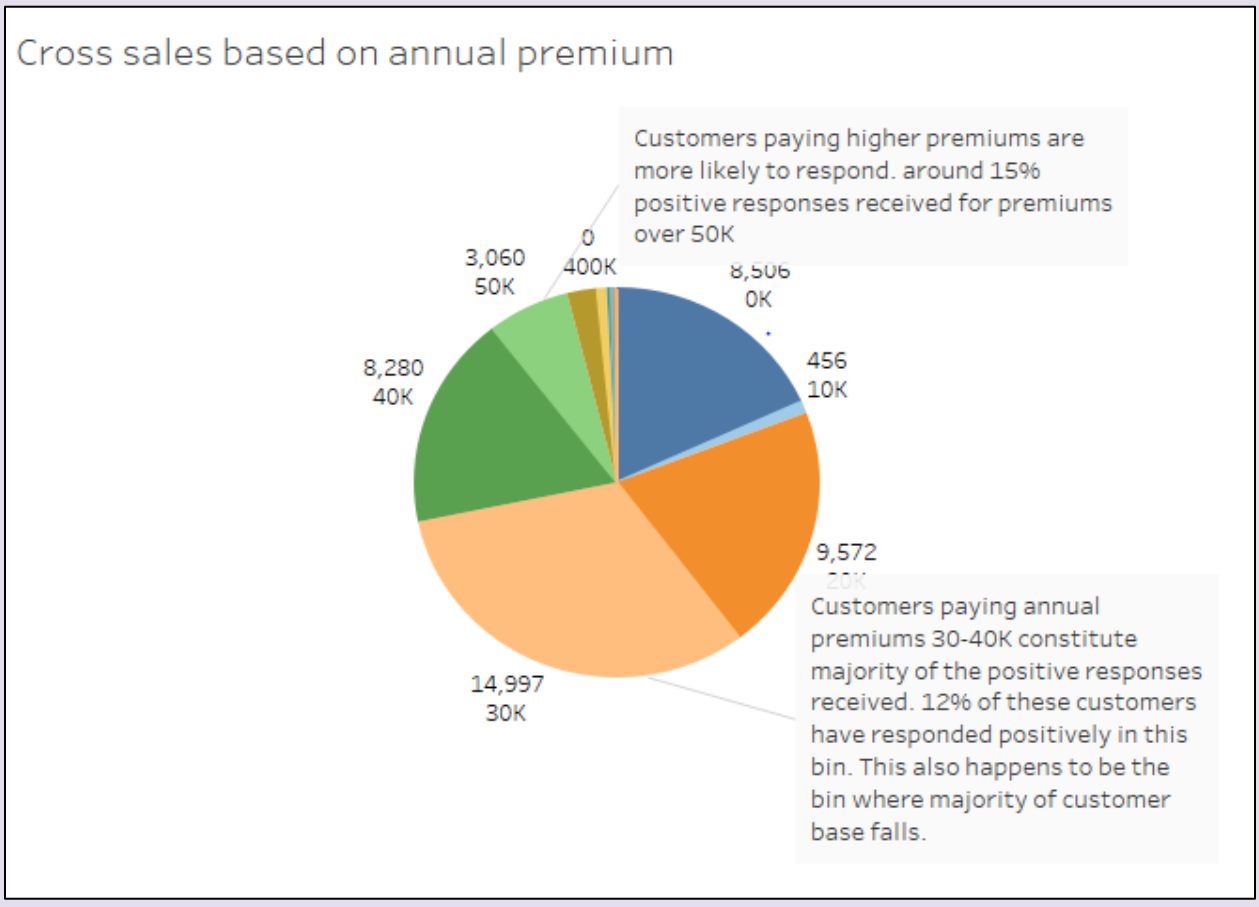


People between ages 40 and 49 have responded in highest volume. 20% of the responses were positive making it the highest positively respondent group.


Genderwise Cross sales achieved
13% of Male repondents responded positively in comparison to 10% of Female respondents.
18,185
Female
28,525
Male


Regionwise Cross sales achieved
Region Code
Sum of Response
20K
18K
16K
14K
12K
10K
8K
6K
4K
2K
0K
Region 28
38
23
4
19
51
Region 28 seems to have the highest number of positive responses considering that it also has a maximum share in the portfolio
Regions 28 ,38, 23, 4, 19 and 51 have at least 15% positive responses.





Findings from EDA:

- Majority of the policy holders are seen to be of age groups 20-29. A major portion of it was also seen to be females.
- Channels 152, 124, 122, 26, 156, 160 are doing the best in terms of interaction with the customers.
- Regions 28, 8, 46, 41, 15 and 30 have the highest percentage of customer base.
- Vehicles of majority of insurance holders are 1-2 years old.
- Customers have interacted with the company from 0-299 days.
- Most of the customers are paying Annual premiums less than 70K.
- 20% of the Health Insurance holders aged 40-49 bought Vehicle Insurance as well.
- 13% of Male respondents bought the Vehicle insurance. While only 10% of Female health insurance did the same.
- Regions 28, 8, 46,41, 15 and 30 are the most responsive for cross sales in comparison to the rest.
- Customers seem to be responsive regardless of the number of days they have been on books.
- People paying higher premiums are more likely to respond to cross sales request compared to the ones paying lower premiums.

Pre-Processing

The data will now be processed to make it fit to train a classification model. We will be looking at features that require one hot encoding, binning , dropping and renaming.

Let's take a look at the data that got exported from the previous step.

```
insurance.head()
```

	Unnamed: 0	id	Gender	Age	Driving_License	Region_Code	Previously_Insured	Vehicle_Age	Vehicle_Damage	Annual_Premium
0	0	1	M	44	1	28.0	0	> 2 Years	Yes	40454.0
1	1	2	M	76	1	3.0	0	1-2 Year	No	33536.0
2	2	3	M	47	1	28.0	0	> 2 Years	Yes	38294.0
3	3	4	M	21	1	11.0	1	< 1 Year	No	28619.0
4	4	5	F	29	1	41.0	1	< 1 Year	No	27496.0

The index field has got exported twice and needs to be dropped.

```
insurance.drop(['Unnamed: 0'], axis =1, inplace =True)
```

```
insurance.columns
```

```
Index(['id', 'Gender', 'Age', 'Driving_License', 'Region_Code',
       'Previously_Insured', 'Vehicle_Age', 'Vehicle_Damage', 'Annual_Premium',
       'Policy_Sales_Channel', 'Vintage', 'Response'],
      dtype='object')
```

We have now dropped the index field.

Binning of appropriate columns:

Age:

```
print('Minimum age:', insurance.Age.min())
print('Maximum age:', insurance.Age.max())
```

```
Minimum age: 20
Maximum age: 85
```

Age can be divided into 20s, 30s, 40s, 50s, 60s and above 70. There are fewer customers who have crossed 70 as per the EDA hence we can group all ages above 70 into one bin.

```
def age_bin(x):
    if x['Age'] <30:
        Age_segment = '20s'
    elif x['Age'] <40:
        Age_segment = '30s'
    elif x['Age'] <50:
        Age_segment = '40s'
    elif x['Age'] <60:
        Age_segment = '50s'
    elif x['Age'] <70:
        Age_segment = '60s'
    else:
        Age_segment = '70plus'
    return Age_segment

insurance['Age_Segment'] = insurance.apply(lambda x: age_bin(x), axis=1)
```

Let’s check if it has worked.

```
insurance['Age_Segment'].value_counts()
```

```
20s       154941
40s        76696
30s        54160
50s        47906
60s        28946
70plus     17682
```

```
insurance[['Age','Age_Segment']].head(10)
```

	Age	Age_Segment
0	44	40s
1	76	70plus
2	47	40s
3	21	20s
4	29	20s
5	24	20s
6	23	20s
7	56	50s
8	24	20s
9	32	30s

Annual Premium:

Annual Premium also requires a similar binning as age.

```
print('Minimum premium:', insurance.Annual_Premium.min())
print('Maximum premium:', insurance.Annual_Premium.max())
```

```
Minimum premium: 2630.0
Maximum premium: 99999.0
```

```
def premium_bin(x):
    if x['Annual_Premium'] <10000:
        Premium_segment = '0-10k'
    elif x['Annual_Premium'] <20000:
        Premium_segment = '10-20k'
    elif x['Annual_Premium'] <30000:
        Premium_segment = '20-30k'
    elif x['Annual_Premium'] <40000:
        Premium_segment = '30-40k'
    elif x['Annual_Premium'] <50000:
        Premium_segment = '40-50k'
    elif x['Annual_Premium'] <60000:
        Premium_segment = '50-60k'
    elif x['Annual_Premium'] <70000:
        Premium_segment = '60-70k'
    else:
        Premium_segment = '70plus'
    return Premium_segment

insurance['Premium_Segment'] = insurance.apply(lambda x: premium_bin(x), axis=1)
```

Let's check if it has worked.

```
insurance['Premium_Segment'].value_counts()
```

```
30-40k    125047
20-30k     95381
0-10k      64882
40-50k     57788
50-60k     20049
60-70k      7271
10-20k      5802
70plus      4111
```

```
insurance[['Premium_Segment','Annual_Premium']].head(10)
```

	Premium_Segment	Annual_Premium
0	40-50k	40454.0
1	30-40k	33536.0
2	30-40k	38294.0
3	20-30k	28619.0
4	20-30k	27496.0
5	0-10k	2630.0
6	20-30k	23367.0
7	30-40k	32031.0
8	20-30k	27619.0
9	20-30k	28771.0

All the premiums seem to be falling under appropriate bins.

Vintage:

The number of days a customer has been on books can also be a key indicator.

```
insurance['Vintage'].describe()
```

```
count    380331.000000
mean        154.344056
std          83.668499
min          10.000000
25%          82.000000
50%         154.000000
75%         227.000000
max         299.000000
```

This KPI however seems to be distributed from 10 to 299 days. It is important to divide days into months.

```
insurance['Months_On_Books'] = (round(insurance['Vintage']/30,0)).astype(str) + ' months'
```

Let's see if it has worked.

```
insurance[['Months_On_Books', 'Vintage']].head(10)
```

	Months_On_Books	Vintage
0	7.0 months	217
1	6.0 months	183
2	1.0 months	27
3	7.0 months	203
4	1.0 months	39
5	6.0 months	176
6	8.0 months	249
7	2.0 months	72
8	1.0 months	28
9	3.0 months	80

```
insurance['Months_On_Books'].value_counts()
```

```
4.0 months     40888
8.0 months     40830
2.0 months     40664
6.0 months     40561
1.0 months     38238
3.0 months     38160
5.0 months     38002
9.0 months     37930
7.0 months     37815
10.0 months    19465
0.0 months      7778
```

The distribution seems to be spread out evenly across all months on books. However 0 months doesn't seem to be a meaningful criteria. It can be included as a part of 1 month bin.

```
insurance['Months_On_Books'][insurance['Months_On_Books']=='0.0 months']= '1.0 months'
```

```
insurance['Months_On_Books'].value_counts()
```

```
1.0 months     46016
4.0 months     40888
8.0 months     40830
2.0 months     40664
6.0 months     40561
3.0 months     38160
5.0 months     38002
9.0 months     37930
7.0 months     37815
10.0 months    19465
```

We have successfully included less than 1 month under 1 month bin.

Channels:

The customers are contacted through various channels. From EDA it was seen that channels 152, 26, 124, 160 and 156 were commonly used. These can be mapped to their actual channels. The remaining will be categorized under ‘Others’.

```
def policy(x):
    if x['Policy_Sales_Channel'] == 152:
        channel = 'Internet'
    elif x['Policy_Sales_Channel'] == 26:
        channel = 'Direct Response'
    elif x['Policy_Sales_Channel'] == 124.0:
        channel = 'Independent Agencies'
    elif x['Policy_Sales_Channel'] == 160.0:
        channel = 'Affinity Group'
    elif x['Policy_Sales_Channel'] == 156.0:
        channel = 'Exclusive/Captive Agents'
    else:
        channel = 'others'
    return channel
```

```
insurance['Policy_channel_type'] = insurance.apply((lambda x : policy(x)) , axis=1)
```

```
insurance['Policy_channel_type'].value_counts()
```

```
Internet                  134545
Direct Response            79476
Independent Agencies       73835
others                     60068
Affinity Group             21754
Exclusive/Captive Agents   10653
```

The Channel codes have been designated to their appropriate Channels.

Regions:

As per the EDA, 28,8,46, 41 and 15 are the regions with maximum customers. Their actual names can be assigned and the remaining categorized as others.

```
def region(x):
    if x['Region_Code'] == 28.0:
        region = 'West Bengal'
    elif x['Region_Code'] == 8.0:
        region = 'Haryana'
    elif x['Region_Code'] == 46.0:
        region = 'Goa'
    elif x['Region_Code'] == 41.0:
        region = 'Andhra Pradesh'
    elif x['Region_Code'] == 15.0:
        region = 'Maharashtra'
    else:
        region = 'Others'
    return region
```

```
insurance['Region_name'] = insurance.apply((lambda x: region(x)), axis =1)
```

```
insurance['Region_name'].value_counts()
```

```
Others            189373
West Bengal       105928
Haryana            33776
Goa                19744
Andhra Pradesh     18211
Maharashtra        13299
```

The regions have properly been assigned to their names.

```
insurance[['Region_Code', 'Region_name']].head(10)
```

	Region_Code	Region_name
0	28.0	West Bengal
1	3.0	Others
2	28.0	West Bengal
3	11.0	Others
4	41.0	Andhra Pradesh
5	33.0	Others
6	11.0	Others
7	28.0	West Bengal
8	3.0	Others
9	6.0	Others

Now, the parent fields need to be dropped. The newer fields will be retained.

```
insurance_fields_dropped = insurance.drop(['Region_Code',
                                           'Policy_Sales_Channel',
                                           'Vintage',
                                           'Age',
                                           'Annual_Premium'], axis=1)
```

```
insurance_fields_dropped.head(10)
```

	id	Gender	Driving_License	Previously_Insured	Vehicle_Age	Vehicle_Damage	Response	Age_Segment	Premium_Segment	Months_On_Books
0	1	M	1	0	> 2 Years	Yes	Yes	40s	40-50k	7.0 months
1	2	M	1	0	1-2 Year	No	No	70plus	30-40k	6.0 months
2	3	M	1	0	> 2 Years	Yes	Yes	40s	30-40k	1.0 months
3	4	M	1	1	< 1 Year	No	No	20s	20-30k	7.0 months
4	5	F	1	1	< 1 Year	No	No	20s	20-30k	1.0 months
5	6	F	1	0	< 1 Year	Yes	No	20s	0-10k	6.0 months
6	7	M	1	0	< 1 Year	Yes	No	20s	20-30k	8.0 months
7	8	F	1	0	1-2 Year	Yes	Yes	50s	30-40k	2.0 months
8	9	F	1	1	< 1 Year	No	No	20s	20-30k	1.0 months
9	10	F	1	1	< 1 Year	No	No	30s	20-30k	3.0 months

There are certain fields that need to be converted into Booleans.

```
Bool = {'Yes':1, 'No':0}
```

```
insurance_fields_dropped.replace(Bool,inplace=True)
```

```
insurance_fields_dropped.head(10)
```

	id	Gender	Driving_License	Previously_Insured	Vehicle_Age	Vehicle_Damage	Response
0	1	M	1	0	> 2 Years	1	1
1	2	M	1	0	1-2 Year	0	0
2	3	M	1	0	> 2 Years	1	1
3	4	M	1	1	< 1 Year	0	0
4	5	F	1	1	< 1 Year	0	0
5	6	F	1	0	< 1 Year	1	0
6	7	M	1	0	< 1 Year	1	0
7	8	F	1	0	1-2 Year	1	1
8	9	F	1	1	< 1 Year	0	0
9	10	F	1	1	< 1 Year	0	0

The Yes and No have now been replaced with 1s and 0s.

One Hot Encoding:

One Hot encoding will be applied on all the categorical values. Let's start with Vehicle_age

```
Vehicle_Age = pd.get_dummies(insurance_fields_dropped['Vehicle_Age'], prefix = 'Vehicle_Age')
```

```
Vehicle_Age.head()
```

	Vehicle_Age_1-2 Year	Vehicle_Age_< 1 Year	Vehicle_Age_> 2 Years
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	1	0

Vehicle age has been successfully converted to the required columns.

```
Age_Segment = pd.get_dummies(insurance_fields_dropped['Age_Segment'], prefix = 'Age_Segment')
Premium_Segment = pd.get_dummies(insurance_fields_dropped['Premium_Segment'], prefix = 'Premium_Segment')
Months_On_Books = pd.get_dummies(insurance_fields_dropped['Months_On_Books'], prefix = 'Months_On_Books')
Policy_channel_type = pd.get_dummies(insurance_fields_dropped['Policy_channel_type'], prefix = 'Policy_channel_type')
Region_name = pd.get_dummies(insurance_fields_dropped['Region_name'], prefix = 'Region')
Gender = pd.get_dummies(insurance_fields_dropped['Gender'], prefix = 'Gender')
```

```
Age_Segment.head(10)
```

	Age_Segment_20s	Age_Segment_30s	Age_Segment_40s	Age_Segment_50s	Age_Segment_60s	Age_Segment_70plus
0	0	0	1	0	0	0
1	0	0	0	0	0	1
2	0	0	1	0	0	0
3	1	0	0	0	0	0
4	1	0	0	0	0	0
5	1	0	0	0	0	0
6	1	0	0	0	0	0
7	0	0	0	1	0	0
8	1	0	0	0	0	0
9	0	1	0	0	0	0

```
Premium_Segment.head(10)
```

	Premium_Segment_0-10k	Premium_Segment_10-20k	Premium_Segment_20-30k	Premium_Segment_30-40k	Pre
0	0	0	0	0	
1	0	0	0	1	
2	0	0	0	1	
3	0	0	1	0	
4	0	0	1	0	
5	1	0	0	0	
6	0	0	1	0	
7	0	0	0	1	
8	0	0	1	0	
9	0	0	1	0	

```
Months_On_Books.head()
```

	Months_On_Books_1.0 months	Months_On_Books_10.0 months	Months_On_Books_2.0 months	Months_On_Books_3.0 months	Months_On_Books_4.0 months	M
0	0	0	0	0	0	
1	0	0	0	0	0	
2	1	0	0	0	0	
3	0	0	0	0	0	
4	1	0	0	0	0	

```
Policy_channel_type.head()
```

	Policy_channel_type_Affinity Group	Policy_channel_type_Direct Response	Policy_channel_type_Exclusive/Captive Agents	Policy_channel_type_Independent Agencies
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0
3	0	0	0	0
4	0	0	0	0

```
Region_name.head()
```

	Region_Andhra Pradesh	Region_Goa	Region_Haryana	Region_Maharashtra	Region_Others	Region_West Bengal
0	0	0	0	0	0	1
1	0	0	0	0	1	0
2	0	0	0	0	0	1
3	0	0	0	0	1	0
4	1	0	0	0	0	0

```
Gender.head()
```

	Gender_F	Gender_M
0	0	1
1	0	1
2	0	1
3	0	1
4	1	0

All the fields seem to be encoded to their respective columns.

Let’s now concat all these tables into one table.

```
insurance_fields_encoded = pd.concat([insurance_fields_dropped.drop(['Vehicle_Age',
                                                                     'Age_Segment',
                                                                     'Premium_Segment',
                                                                     'Months_On_Books',
                                                                     'Policy_channel_type',
                                                                     'Region_name','Gender'],axis=1) ,
                                      Age_Segment,
                                      Premium_Segment,
                                      Months_On_Books,
                                      Policy_channel_type,
                                      Region_name,
                                      Gender], axis=1)
```

```
insurance_fields_encoded.head()
```

	id	Driving_License	Previously_Insured	Vehicle_Damage	Response	Age_Segment_20s	Age_Segment_30s	Age_Segment_40s	Age_Segment_50s
0	1	1	0	1	1	0	0	1	0
1	2	1	0	0	0	0	0	0	0
2	3	1	0	1	1	0	0	1	0
3	4	1	1	0	0	1	0	0	0
4	5	1	1	0	0	1	0	0	0

5 rows × 43 columns

We now have a complete table with all the categorical features encoded.

Handling Imbalanced data:

As seen from the initial stages, the responses seem to be highly imbalanced. We have fewer positive responses in comparison to negative responses.

Hence it is crucial to handle this imbalance. We will import imblearn and oversample the under- represented population.

```
from imblearn import over_sampling
Dependent = insurance_fields_encoded['Response']
Independent = insurance_fields_encoded.drop(['Response'], axis=1)
oversample = over_sampling.SMOTE()
Indep,Dep = oversample.fit_resample(Independent, Dependent)
```

Concatenating the independent and dependent datasets:

```
balanced_dataset = pd.concat([Indep,Dep], axis=1)
```

```
balanced_dataset.columns
```

```
Index(['id', 'Driving_License', 'Previously_Insured', 'Vehicle_Damage',
       'Age_Segment_20s', 'Age_Segment_30s', 'Age_Segment_40s',
       'Age_Segment_50s', 'Age_Segment_60s', 'Age_Segment_70plus',
       'Premium_Segment_0-10k', 'Premium_Segment_10-20k',
       'Premium_Segment_20-30k', 'Premium_Segment_30-40k',
       'Premium_Segment_40-50k', 'Premium_Segment_50-60k',
       'Premium_Segment_60-70k', 'Premium_Segment_70plus',
       'Months_On_Books_1.0 months', 'Months_On_Books_10.0 months',
       'Months_On_Books_2.0 months', 'Months_On_Books_3.0 months',
       'Months_On_Books_4.0 months', 'Months_On_Books_5.0 months',
       'Months_On_Books_6.0 months', 'Months_On_Books_7.0 months',
       'Months_On_Books_8.0 months', 'Months_On_Books_9.0 months',
       'Policy_channel_type_Affinity Group',
       'Policy_channel_type_Direct Response',
       'Policy_channel_type_Exclusive/Captive Agents',
       'Policy_channel_type_Independent Agencies',
       'Policy_channel_type_Internet', 'Policy_channel_type_others',
       'Region_Andhra Pradesh', 'Region_Goa', 'Region_Haryana',
       'Region_Maharashtra', 'Region_Others', 'Region_West Bengal', 'Gender_F',
       'Gender_M', 'Response'],
      dtype='object')
```

```
balanced_dataset['Response'].value_counts()
```

```
1    333744
0    333744
```

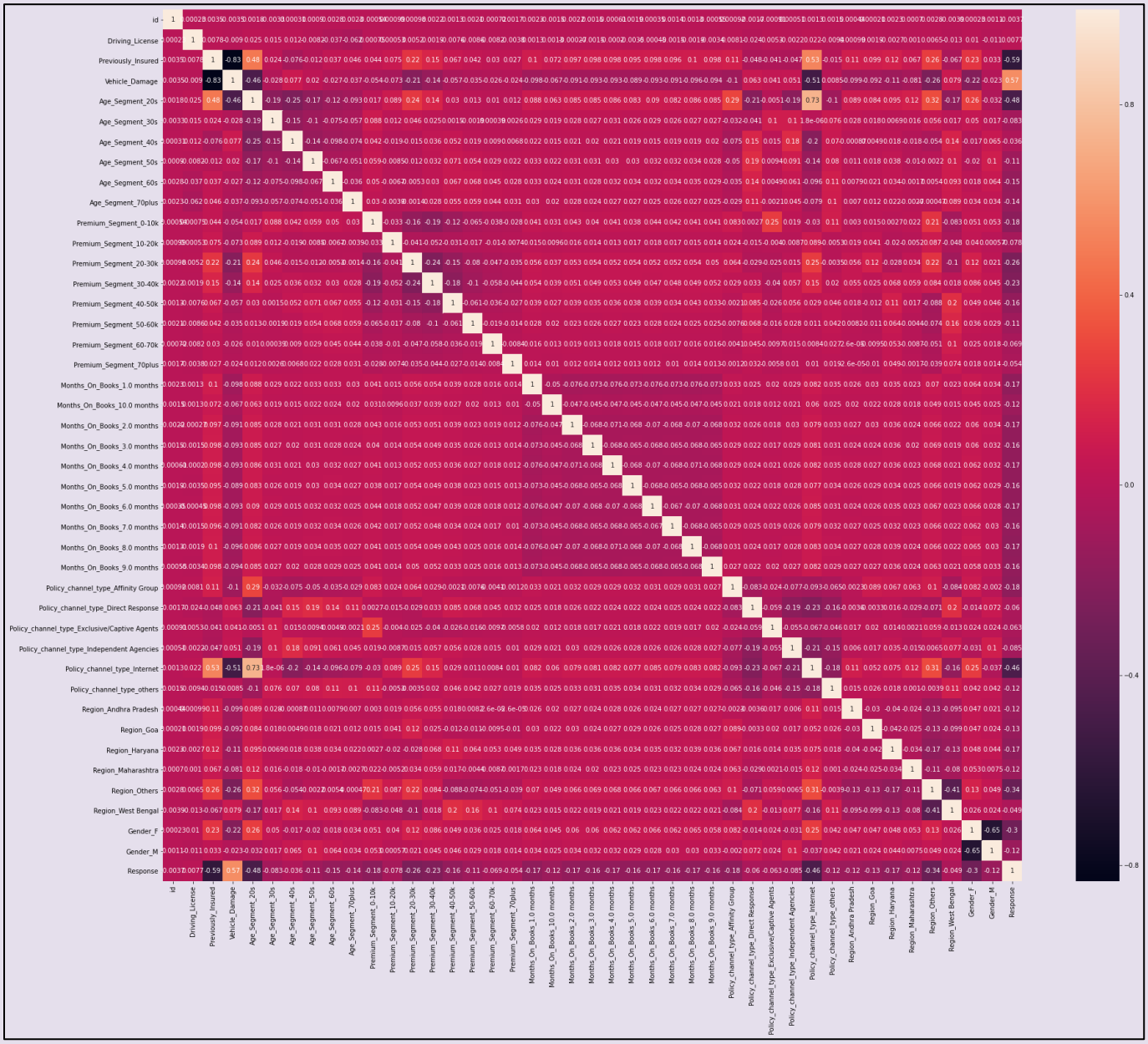
We now have a balanced dataset with equal representation of the positive and negative responses.

Correlation Analysis:

Let’s try to analyze how the variables are correlated to each other.

```
corr_data= balanced_dataset.corr()
```

```
plt.figure(figsize = (30,25))
sns.heatmap(
    corr_data,
    annot = True)
```



People who have been previously insured or have claimed expenses due to vehicle damage are seen to be less likely to respond to the cross sales.

People are also less likely to respond when contacted over internet.

As expected, the age group has a correlation with the channel through which the customers are reached.

Modeling

By training the ML models we aim to increasingly target the right set of customers who are more likely to buy health insurance from us. Currently we are reaching out to the entire customer base and trying to achieve a conversion. However reducing the customer base that is being reached out to, will reduce the resources invested in by the company.

So we aim at increasing the number of True Positives (Customers who are more likely to convert) and reduction in the False Positives(Customers less likely to convert). The performance metric we will be focusing upon will be the ROC_AUC_score.

Let's start by training a logistic regression model, the simplest of all algorithms to train.

Logistic Regression:

```
logreg = LogisticRegression()
logreg_model = logreg.fit(X_train, y_train)
```

Assigning the predicted values to y_predicted:

```
y_predicted =logreg_model.predict(X_test)
```

Let's understand how the model is performing.

```
print('Logistic Regression Performance')
print('Accuracy: ' ,accuracy_score(y_predicted,y_test))
print( 'Recall: ' ,recall_score(y_predicted,y_test))
print('Precision: ' ,precision_score(y_predicted,y_test))
print('f1 score: ' ,f1_score(y_predicted,y_test))
print('roc_auc_score: ' ,roc_auc_score(y_predicted,y_test))
```

```
Logistic Regression Performance
Accuracy:  0.8604024030322551
Recall:  0.8481166581418466
Precision:  0.8779583803710401
f1 score:  0.8627795558522648
roc_auc_score:  0.8608468375777742
```

We are getting a fairly good accuracy of 86%, precision of 87% and roc_auc_score of 0.86.

We have used default parameters for training. Now, we will optimize the model further and check for any improvement in performance.

```
# Listing out a range of Regularization parameters Cs to fit the Logistic regression model
C =[0.01, 0.1, 1, 10, 100, 1000]
penalties =['l2']
```

Performing GridSearchCV on the data using these hyperparameters:

```
gridsearch_logreg= GridSearchCV(logreg, param_grid = {'C': C, 'penalty': penalties}, cv =5, scoring= 'roc_auc')
```

```
model_logreg = gridsearch_logreg.fit(X_train,y_train)
```

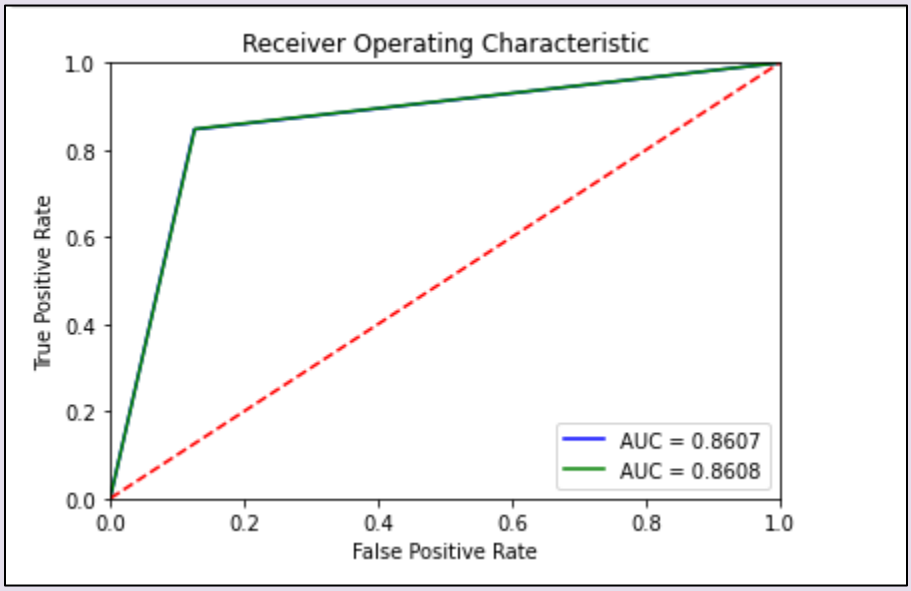
```
print(model_logreg.best_params_,model_logreg.best_score_)
```

```
{'C': 1000, 'penalty': 'l2'} 0.861106024094514
```

With a penalty of 1000, we get an roc_auc of 0.8611 which is a marginal improvement from the default model.

Let's plot an roc_auc_curve for these two models and compare them.

```
import matplotlib
from matplotlib import pyplot as plt
logreg =LogisticRegression(C=0.01, penalty ='l2')
best_logreg_model = logreg.fit(X_train, y_train)
fpr_logreg_best,tpr_logreg_best,threshold_logreg_best = roc_curve(best_logreg_model.predict(X_test),y_test)
roc_auc_logreg_best = auc(fpr_logreg_best,tpr_logreg_best)
plt.title('Receiver Operating Characteristic')
plt.plot(fpr_logreg_best,tpr_logreg_best, 'b',label = 'AUC = %0.4f' % roc_auc_logreg_best)
plt.plot(fpr_logreg,tpr_logreg, 'g',label = 'AUC = %0.4f' % roc_auc_logreg)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



The models are giving equally good performance.

Decision Trees classifier:

Let's try to understand how decision tree fits on the data.

```
DT =DecisionTreeClassifier()
DT_model = DT.fit(X_train , y_train)
```

Let's analyze its performance.

```
print('Decision Trees- Entropy Model performance')
print('Recall: ' ,recall_score(DT_model.predict(X_test), y_test))
print('Accuracy: ' ,accuracy_score(DT_model.predict(X_test), y_test))
print('Precision: ',precision_score(DT_model.predict(X_test), y_test))
print('f1 score: ',f1_score(DT_model.predict(X_test), y_test))
print('roc_auc: ',roc_auc_score(DT_model.predict(X_test), y_test))
```

```
Decision Trees- Entropy Model performance
Recall:  0.8942570376529279
Accuracy:  0.8979510304773605
Precision:  0.9025745027323496
f1 score:  0.8983965195773772
roc_auc:  0.8979847056455208
```

We are getting an roc_auc of 0.89 which is better than the optimized logistic regression model.

The above model was fitted on entropy criterion. Let's fit the model based on gini criterion as well.

```
DT_gini =DecisionTreeClassifier(criterion='gini')
DT_gini_model = DT_gini.fit(X_train , y_train)
```

```
print('Decision Trees- Gini Impurity Model performance')
print('Recall: ' ,recall_score(DT_gini_model.predict(X_test), y_test))
print('Accuracy: ' ,accuracy_score(DT_gini_model.predict(X_test), y_test))
print('Precision: ',precision_score(DT_gini_model.predict(X_test), y_test))
print('f1 score: ',f1_score(DT_gini_model.predict(X_test), y_test))
print('roc_auc: ',roc_auc_score(DT_gini_model.predict(X_test), y_test))
```

```
Decision Trees- Gini Impurity Model performance
Recall:  0.8945623013948147
Accuracy:  0.8982157036060465
Precision:  0.9027842992297471
f1 score:  0.8986544944658255
roc_auc:  0.8982486000574799
```

Gini model's performance is at par with the entropy model.

Let's try to optimize the features and check for any overfitting happening.

```
depth = [3,5,10,15,20]
```

```
gscv =GridSearchCV(DT_gini, param_grid = {'max_depth': depth}, cv =5, scoring= 'roc_auc')
```

```
gini_model_cv = gscv.fit(X_train, y_train)
```

```
print(gini_model_cv.best_params_)
print(gini_model_cv.best_score_)
```

```
{'max_depth': 10}
0.9725156679257461
```

It's interesting to note that we are getting the best score at a depth of 10. There is a possibility that the model is overfitting. Let's try to tune the hyperparameters and then test for overfitting.

```
best_gini_model = DecisionTreeClassifier(criterion = 'gini',max_depth =10)
best_gini_model_fitted = best_gini_model.fit(X_train,y_train)
print('Recall_Score: ' , recall_score(y_test,best_gini_model_fitted.predict(X_test) ))
print('Accuracy: ' ,accuracy_score(y_test,best_gini_model_fitted.predict(X_test)))
print('Precision: ',precision_score(y_test,best_gini_model_fitted.predict(X_test)))
print('f1 score: ',f1_score(y_test,best_gini_model_fitted.predict(X_test)))
print('roc_auc: ',roc_auc_score(y_test,best_gini_model_fitted.predict(X_test)))
```

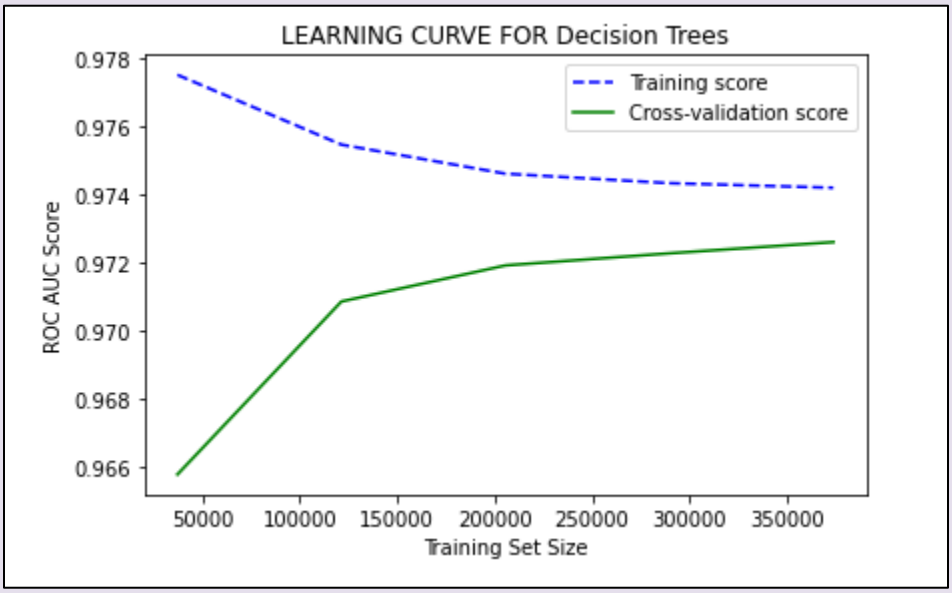
```
Recall_Score:  0.8576480813610797
Accuracy:  0.9058762428400925
Precision:  0.9491525423728814
f1 score:  0.9010832143756823
roc_auc:  0.9058634815192818
```

The model is giving an amazing accuracy of ~90% with test set and the AUC is 0.906. This seems to be a good model to work with. However we also see a reduction in accuracy from 97% on training set to 90% on test set. Let's test for any overfitting by plotting learning curve.

```
from sklearn.model_selection import learning_curve
train_sizes, train_scores, valid_scores = learning_curve(best_gini_model,
                                                         X_train, y_train,
                                                         train_sizes=np.linspace(0.1, 1.0, 5),
                                                         cv=5,
                                                         scoring='roc_auc',
                                                         n_jobs =-1)
```

```
mean_training = np.mean(train_scores, axis=1)
mean_testing = np.mean(valid_scores, axis=1)
standard_dev_training = np.std(train_scores, axis=1)
standard_dev_testing = np.std(valid_scores, axis=1)
```

```
import matplotlib
from matplotlib import pyplot as plt
plt.plot(train_sizes, mean_training, '--', color="b",  label="Training score")
plt.plot(train_sizes, mean_testing, color="g", label="Cross-validation score")

# Drawing plot
plt.title("LEARNING CURVE FOR Decision Trees")
plt.xlabel("Training Set Size"), plt.ylabel("ROC AUC Score"), plt.legend(loc="best")
plt.tight_layout()
plt.show()
```



The learning curves for training set and validation set are seen to converge as more data gets added. The model is seen to have a low bias(considering the training scores) and a moderate variance(considering how the curves are converging)

Now let's try to train the entropy model with best hyperparameters.

```
gscv_entropy =GridSearchCV(DT_model, param_grid = {'max_depth': depth}, cv =5, scoring= 'roc_auc')
```

```
entropy_model_cv = gscv_entropy.fit(X_train,y_train)
```

```
print(entropy_model_cv.best_params_)
print(entropy_model_cv.best_score_)
```

```
{'max_depth': 10}
0.972235849472869
```

The entropy model is also performing well with a depth of 10 and the roc_auc_score is around 0.97 similar to gini model. Let's check how this fitted model performs on test set.

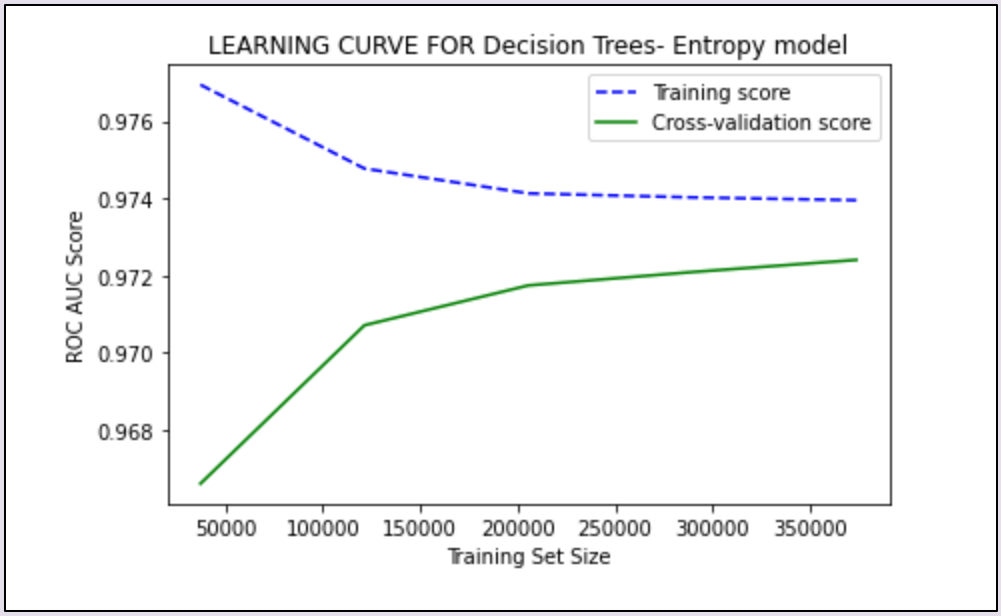
```
entropy_model = DecisionTreeClassifier(criterion = 'entropy',max_depth =10)
entropy_model_fitted = entropy_model.fit(X_train, y_train)
print('Recall: ', recall_score(y_test,entropy_model_fitted.predict(X_test) ))
print('Accuracy: ' ,accuracy_score(y_test,entropy_model_fitted.predict(X_test)))
print('Precision: ',precision_score(y_test,entropy_model_fitted.predict(X_test)))
print('f1 score: ',f1_score(y_test,entropy_model_fitted.predict(X_test)))
print('roc_auc: ',roc_auc_score(y_test,entropy_model_fitted.predict(X_test)))
```

```
Recall:  0.8598678431685698
Accuracy:  0.9066452930630671
Precision:  0.9487334801762115
f1 score:  0.902117454865329
roc_auc:  0.9066735756022537
```

The performance is similar to the gini model. We again see a reduction in the performance between the train and test sets. So let's plot learning curves to confirm on overfitting.

```
sizes, train_scores, valid_scores =
learning_curve(entropy_model,
               X_train, y_train,
               train_sizes=np.linspace(0.1, 1.0, 5),
               cv=5, scoring='roc_auc',  n_jobs =-1)
```

```
mean_training = np.mean(train_scores, axis=1)
mean_testing = np.mean(valid_scores, axis=1)

plt.plot(train_sizes, mean_training, '--', color="b",  label="Training score")
plt.plot(train_sizes, mean_testing, color="g", label="Cross-validation score")

# Drawing plot
plt.title("LEARNING CURVE FOR Decision Trees- Entropy model")
plt.xlabel("Training Set Size"), plt.ylabel("ROC AUC Score"), plt.legend(loc="best")
plt.tight_layout()
plt.show()
```



We see the training and cross validation scores converging towards the end. Hence we see a small amount of bias with a moderate variance.

Random Forest Classifier:

Let's fit a random forest model and find out the impact on the roc_auc.

```
rfc = RandomForestClassifier(n_estimators=10)
```

```
rfc_model = rfc.fit(X_train, y_train)
```

```
print('Random Forest Classifier')
print('Recall: ' , recall_score(y_test, rfc_model.predict(X_test) ))
print('Accuracy: ' , accuracy_score(y_test, rfc_model.predict(X_test) ))
print('Precision: ' , precision_score(y_test, rfc_model.predict(X_test) ))
print('f1 score: ' , f1_score(y_test, rfc_model.predict(X_test) ))
print('roc score: ' , roc_auc_score(y_test, rfc_model.predict(X_test) ))
```

```
Random Forest Classifier
Recall:  0.8927332881572129
Accuracy:  0.9081434428480826
Precision:  0.9215133622729237
f1 score:  0.9068950506676384
roc score:  0.9081780736224457
```

The roc_auc score for the model is 0.908 which is at par with most of the models fitted so far.

Let's optimize the model using GridSearchCV.

The estimators will be provided between 25 and 150 and we will use 10 to 40 features for training.

```
rfc = RandomForestClassifier()
estimators = [25,50,100,150]
features =[10,20,30,40]
gridsearch_rf= GridSearchCV(rfc, param_grid = {'n_estimators': estimators, 'max_features':features},
                            cv =5,
                            n_jobs =-1,
                            scoring= 'roc_auc')
```

```
rf_gs= gridsearch_rf.fit(X_train,y_train)
```

```
print(rf_gs.best_params_,rf_gs.best_score_)
```

```
{'max_features': 40, 'n_estimators': 150} 0.9741215006578428
```

With 40 features and 150 trees we are getting an roc_auc of 0.97.

Let's see how the model performs in terms of other metrics.

```
rfc_best= RandomForestClassifier(n_estimators= 150, max_features= 40)
rfc_best_fitted = rfc_best.fit(X_train, y_train)
print('Random Forest Classifier- Performance')
print('Recall: ', recall_score(y_test,rfc_best_fitted.predict(X_test)))
print('Accuracy: ' ,accuracy_score(y_test,rfc_best_fitted.predict(X_test)))
print('Precision: ',precision_score(y_test,rfc_best_fitted.predict(X_test)))
print('f1 score: ',f1_score(y_test,rfc_best_fitted.predict(X_test)))
print('roc_auc: ',roc_auc_score(y_test,rfc_best_fitted.predict(X_test)))
```

```
Random Forest Classifier- Performance
Recall:  0.8892338097899865
Accuracy:  0.9145455362627155
Precision:  0.9367521187777333
f1 score:  0.9123746709953607
roc_auc:  0.9145608402157411
```

The roc_auc is similar to other models we have trained previously. The recall is seen to be at 88% meaning that we are letting a few potential customers slip through.

Gradient Boosting Classifier:

With Gradient Boosting classifier let's check what learning rates and number of trees are required to get best roc_auc.

```
[ ] learning_rates = [0.05, 0.1, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]
    estimators = [10,25,50,100,150]
    max_roc_score =0
    for learning_rate in learning_rates:
      for estimator in estimators:
        gb = GradientBoostingClassifier(n_estimators=estimator, learning_rate = learning_rate, max_features=2, random_state =
        gb_model = gb.fit(X_train, y_train)
        score = roc_auc_score(gb_model.predict(X_test), y_test)
        if score > max_roc_score:
          best_learning_rate = learning_rate
          best_estimator = estimator
          max_roc_score = score


    print('best_learning_rate is:', best_learning_rate)
    print('best_estimator is:', best_estimator )
```

```
best_learning_rate is: 0.5
best_estimator is: 150
```

```
max_roc_score

0.9362567417366444
```

The roc_auc_score is the best among all the models we have trained so far. Let's check for other parameters.

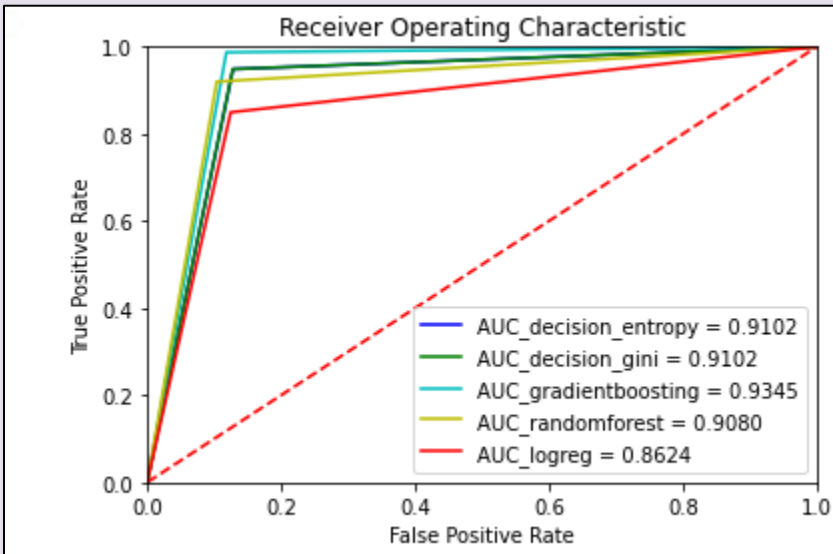
```
gb_y = gb_model.predict(X_test)

print("Accuracy score",accuracy_score(gb_y, y_test))
print("Precision score: ", precision_score(gb_y, y_test))
print("Recall score: ", recall_score(gb_y, y_test))
print("Roc score: ", roc_auc_score(gb_y, y_test))
```

```
Accuracy score 0.9275245072335665
Precision score:  0.8655436140964758
Recall score:  0.9878594249201278
Roc score:  0.9342131761878961
```

The model seems to be performing well in terms of roc score for a learning rate of 0.5 and 150 estimators. The recall is high at 0.98 and the precision low at 0.86 which is an indicator of the model leaning towards more false positives. However in our case we are bothered about maximizing our conversion rate and hence this can be considered to be the best model trained so far.

Let's compare all the models in terms of ROC_AUC_score and other metrics.

```
plt.title('Receiver Operating Characteristic')
plt.plot(fpr_entropy_opt,tpr_entropy_opt, 'b',label = 'AUC_decision_entropy = %0.4f' % roc_auc_entropy_opt)
plt.plot(fpr_gini_opt,tpr_gini_opt, 'g',label = 'AUC_decision_gini = %0.4f' % roc_auc_gini_opt)
plt.plot(fpr_gb_best,tpr_gb_best, 'c',label = 'AUC_gradientboosting = %0.4f' % roc_auc_gb_best)
plt.plot(fpr_rfc_best,tpr_rfc_best, 'y',label = 'AUC_randomforest = %0.4f' % roc_auc_rfc_best)
plt.plot(fpr_logreg_best,tpr_logreg_best, 'r',label = 'AUC_logreg = %0.4f' % roc_auc_logreg_best)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



	Precision	Recall	Accuracy	Roc_Auc_score
Logistic reg	0.8797	0.8496	0.8610	0.8620
Decision Trees-Gini	0.9480	0.8600	0.9067	0.9068
Decision Trees-Entropy	0.9480	0.8598	0.9066	0.9066
Random Forest	0.9360	0.8890	0.9140	0.9140
Gradient Boosting	0.8650	0.9870	0.9270	0.9342

Gradient Boosting is seen to have the highest area under the curve but is having more false positives for a higher number of True positives.

Random Forest is seen to be having a slightly lower AUC. It is classifying True positives and a smaller rate of False positives.

In case of Insurance leads, a False positive only leads to a customer who is less likely to respond to a cross sale lead. Hence we are OK with a higher True positive rate at the cost of a small False negative rate.

Observations and Conclusion:

The objective of this project was to identify as many potential customers for cross sales of insurance as possible. We modeled the training data on Logistic Regression, Decision Trees, Random Forest and Gradient Boosting keeping the same criterion in mind. The performance metric chosen to identify the best model was the roc_auc_score. The models were evaluated in such a manner that we obtained maximum True positives (potential conversions) with tolerable False positives(non converting leads). This would maximize the profits while minimizing the expenses.

When the models were fitted, it was seen that all the algorithms gave a fairly good performance. Out of these, Gradient Boosting will be the model considered for further testing as we obtained the highest roc_auc_score with the same. The recall was also the maximum in case of Gradient Boosting which means that we would be identifying as many potential customers as possible.