EAI6000 – FUNDAMENTALS OF ARTIFICAL INTELLIGENCE– 72170

FINAL REPORT

SUBMITTED BY:

JAYESH KAMAT (001061996)

KASHIKA TYAGI (001028896)

ROHIT KHATU (001028819)

RUTURAJ HARAL (001018953)

SUBMITTED TO: PROF. KASUN SAMARASINGHE

In this project, we were working on "**Predicting Health Insurance Owners' who will be interested in Vehicle Insurance"**. We have used the health insurance cross-sell prediction dataset from **Kaggle - Dataset**. The insurance company who had provided the health insurance to the customers now wanted to predict who will be interested in buying the vehicle insurance. We **choose this data** - to understand how the cross-sell policy can be useful for the organization and how analytics can prove to be one of the essentials in doing this by analyzing the data and targeting the right people to buy the insurance policy.

ANALYSIS:

As the data we received was huge in the amount we used spark to read the data and later we converted it to pandas. We then checked for some missing values in the dataset and we found that there were no missing values which can be seen in fig-1, if we would have found any missing values, we would have found some correlation to impute those values with mean, median, or mode whichever would have been appropriate to the situation. Then we wanted to find out the datatypes for each variable which can be seen in fig-2.

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

Fig-1

```
id                    int32
Gender               object
Age                   int32
Driving_License       int32
Region_Code           int32
Previously_Insured    int32
Vehicle_Age          object
Vehicle_Damage       object
Annual_Premium        int32
Policy_Sales_Channel  int32
Vintage               int32
Response              int32
```

Fig-2

We then visualized the data to get some insights like we wanted to understand the count of male and female in the dataset from the fig-3 we can see the number of male and female count. Also, we then checked the ratio of another categorical variable which is vehicle age, and plotted it using a bar graph in which the vehicle between age 1-2 remains the highest, followed by less than 1 year in Fig-4.

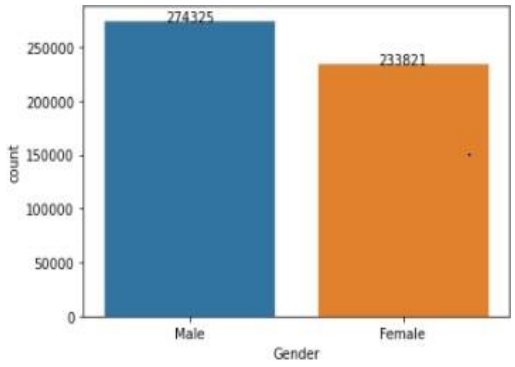


Fig-3

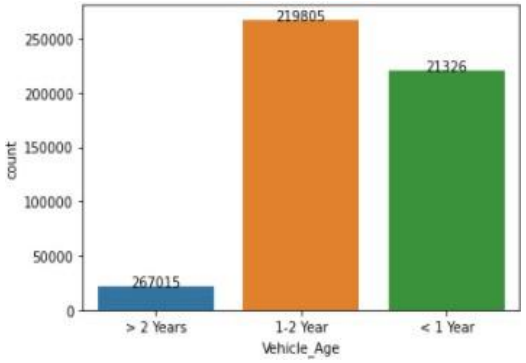


Fig-4

We then used a Distribution plot from the seaborn library to plot the age column to understand which age group is the most in the dataset and we can see that the age group between 21-28 is the highest from fig-5 and we also used it on another column which is the annual premium to check the highest amount of premium and we can see that 10-20k Indian rupees is the highest from fig-6.

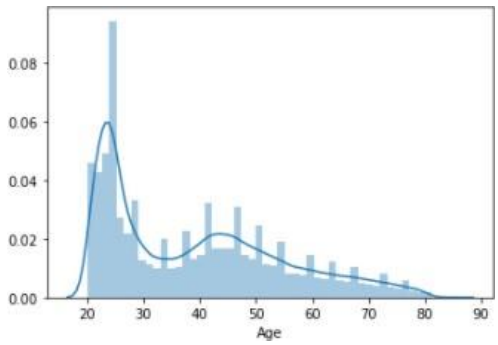


Fig-5

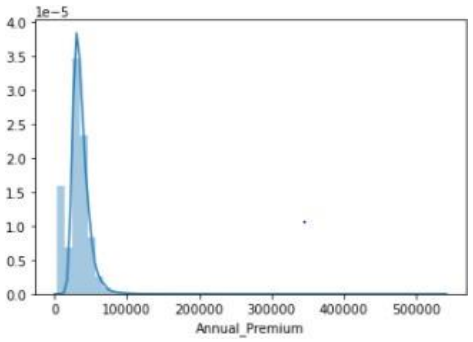


Fig-6

There are 3 categorical variables as we can see in the dataset namely **Gender, Vehicle_Damage, Vehicle_Age**. Since the values in the variables Gender and vehicle_damage is 2, so we used Boolean values like 0,1 to convert it. The column vehicle_age had 3 unique values as shown in the below figure:

```
In [16]: result_df.Vehicle_Age.unique()
Out[16]: array(['> 2 Years', '1-2 Year', '< 1 Year'], dtype=object)
```

So, we decided to use the label encoder to convert it to rank wise values, like 1-2 years was imputed as '0', < 1 year as '1', and > 2years as '2' as shown in the below figure.

]:

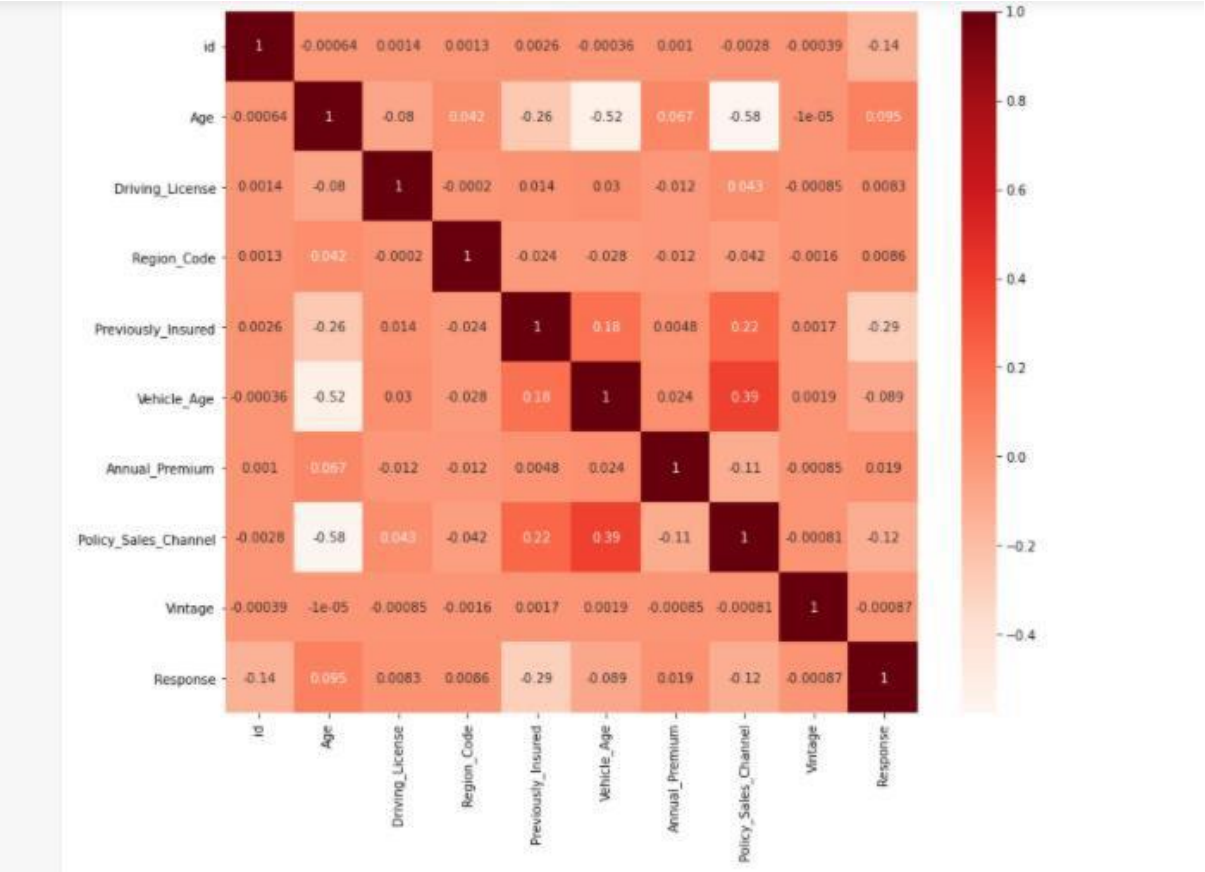
	id	Gender	Age	Driving_License	Region_Code	Previously_Insured	Vehicle_Age	Vehicle_Damage	Annual_Premium	Policy_Sales_Channel	Vintage	Resp
0	1	Male	44	1	28	0	> 2 Years	Yes	40454	26	217	
1	2	Male	76	1	3	0	1-2 Year	No	33536	26	183	
2	3	Male	47	1	28	0	> 2 Years	Yes	38294	26	27	
3	4	Male	21	1	11	1	< 1 Year	No	28619	152	203	
4	5	Female	29	1	41	1	< 1 Year	No	27496	152	39	

:

	id	Gender	Age	Driving_License	Region_Code	Previously_Insured	Vehicle_Age	Vehicle_Damage	Annual_Premium	Policy_Sales_Channel	Vintage	Resp
0	1	1	44	1	28	0	2	1	40454	26	217	
1	2	1	76	1	3	0	0	0	33536	26	183	
2	3	1	47	1	28	0	2	1	38294	26	27	
3	4	1	21	1	11	1	1	0	28619	152	203	
4	5	0	29	1	41	1	1	0	27496	152	39	

Overall, we found that the **vehicle age and vehicle damage matter the most** which decides the annual premium based on that. The low age of the vehicle and less damage can help you get a very reasonable premium. This is one of the **important aspects and it does matter** because it tells us a lot about how vehicle insurance is decided and there are certain measures that you need to take to avoid hefty annual premium rate on vehicles.

From the below figure we can see the correlation plot with respect to response as the output variable, it gives information that which variables have a positive and which have a negative impact on the target or the output variable. From the below figure we can see that Age, driving_license, region_code, previously_insured, vehicle_Age, Annual_premium has a positive impact on the target variable, and Policy_sales_channel, vintage has a negative impact on the target variable.



In the **first week,** we had performed some of the operations on the dataset such as data **cleaning** where we checked whether the dataset has some missing values or null values and how we could impute those values but when we check for the full values there weren't any.

Then we **visualized** some of the categorical variables to get some of the insights, like checking the male and female count in the dataset, checking vehicle age. We also used dist.

plot to check the age group of the people and found that 21-28 is the highest in the dataset and also visualized the annual premium rates.

We converted the categorical variables using the **label encoding** technique. And computed the **correlation matrix** to find out which variables have a positive impact and negative impact on the target variable.

In the next step we have show stepwise analysis of how we implemented the algorithms and their outcomes related to our dataset.

LINEAR REGRESSION:

In this business problem, we are focusing on Annual premium which is basically the amount paid by the customer to the insurance company to get his vehicle insured for a certain period of time.

We used linear regression to predict the annual premium based on the trained data. Firstly, we divided the data into training and test set with a ratio of around 70% and 30%.

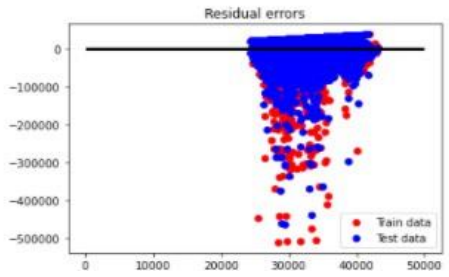
```
In [26]: from sklearn.model_selection import train_test_split
         train_features, test_features, train_labels, test_labels = train_test_split(x, y, test_size = 0.3, random_state = 0)
```

Then we applied the regression model to predict the annual premium and received a Root mean square error (RMSE) of around 1685 and r-square of around 2%. It is indeed not good accuracy but we can improve it using some other models like using regularization technique, decision tree, random forest, and gradient boosting which can help us to yield better accuracy.

```
In [30]: from sklearn.linear_model import LinearRegression
         model = LinearRegression()
         model.fit(train_features, train_labels)
         y_pred = model.predict(test_features)
         print(model.coef_)
         print(model.intercept_)

[-2.20296565e+02  5.68341300e+01 -2.40386630e+03 -2.20994504e+01
  1.84695901e+03  2.87446966e+03  7.89872640e+02 -4.05438669e+01
 -2.30672795e-01  9.38180788e+02]
33231.52629203656

In [31]: from sklearn.metrics import mean_squared_error
         mse = mean_squared_error(test_labels, y_pred)
         rmse = np.sqrt(mse)
         print("RMSE value: {:.4f}".format(rmse))

RMSE value: 16885.9328

In [32]: # Calculate and print r2_score

         from sklearn.metrics import r2_score
         print ("R2 Score value: {:.4f}".format(r2_score(test_labels, y_pred)))

R2 Score value: 0.0202
```

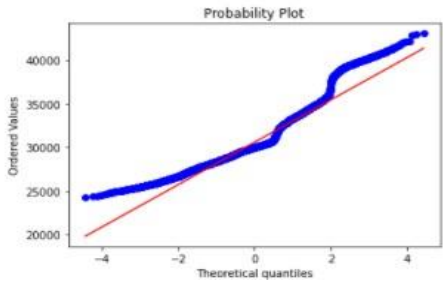
Then we went on to plot the residual vs fitted plot (we named it as residual errors) and we can see that there is a lot of distortion in the data commonly known as noise.

```
In [33]: # Plotting residual errors
         plt.scatter(model.predict(train_features), model.predict(train_features) - train_labels, color = 'red', label = 'Train data')
         plt.scatter(model.predict(test_features), model.predict(test_features) - test_labels, color = 'blue', label = 'Test data')
         plt.hlines(xmin = 0, xmax = 50000, y = 0, linewidth = 3)
         plt.title('Residual errors')
         plt.legend(loc = 4)
         plt.show()
```



A Q-Q plot is a scatterplot created by plotting two sets of quantiles against one another. If both sets of quantiles came from the same distribution, we should see the points forming a line that's roughly straight. The main purpose of the QQ plot is to check whether the 2 sets of data come from the same distribution.

In [34]:
```
import statsmodels.api as sm
import pylab
import scipy.stats as stats
stats.probplot(y_pred.reshape(-1), dist="norm", plot=pylab)
pylab.show()
```



Regularized regression is a type of regression where the coefficient estimates are constrained to zero. The magnitude (size) of coefficients, as well as the magnitude of the error term, are penalized. We Used regularization techniques like lasso and ridge to constraint the coefficients to 0.

We can see from below figure for lasso regression where after applying the model there is a slight change in the accuracy.

In [37]:
```
lasso = Lasso()
lasso.fit(train_features,train_labels)
train_score=lasso.score(train_features,train_labels)
test_score=lasso.score(test_features,test_labels)
coeff_used = np.sum(lasso.coef_!=0)
print("training score:", train_score)
print("test score: ", test_score)
print("number of features used: ", coeff_used)
lasso001 = Lasso(alpha=0.01, max_iter=10e5)
lasso001.fit(train_features,train_labels)
train_score001=lasso001.score(train_features,train_labels)
test_score001=lasso001.score(test_features,test_labels)
coeff_used001 = np.sum(lasso001.coef_!=0)
print("training score for alpha=0.01:", train_score001)
print("test score for alpha =0.01: ", test_score001)
print("number of features used: for alpha =0.01:", coeff_used001)
lasso00001 = Lasso(alpha=0.0001, max_iter=10e5)
lasso00001.fit(train_features,train_labels)
train_score00001=lasso00001.score(train_features,train_labels)
test_score00001=lasso00001.score(test_features,test_labels)
coeff_used00001 = np.sum(lasso00001.coef_!=0)
print("training score for alpha=0.0001:", train_score00001)
print("test score for alpha =0.0001: ", test_score00001)
print("number of features used: for alpha =0.0001:", coeff_used00001)
```

```
training score: 0.021558759070389066
test score:  0.020205815285361806
number of features used:  10
training score for alpha=0.01: 0.02156062130858072
test score for alpha =0.01:  0.02020252115769039
number of features used: for alpha =0.01: 10
training score for alpha=0.0001: 0.02156062149479454
test score for alpha =0.0001:  0.020202470244255255
number of features used: for alpha =0.0001: 10
```

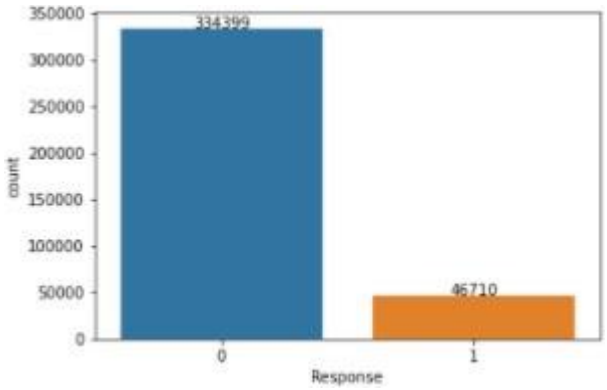
Also we applied the ridge regression and we cannot seen much change in the accuracy what we received from the linear regression.

```
In [38]: from sklearn.linear_model import Ridge
ridgereg = Ridge(alpha=0, normalize=True)
ridgereg.fit(train_features, train_labels)
y_pred = ridgereg.predict(test_features)
from sklearn import metrics
print("R-Square Value",r2_score(test_labels,y_pred))
print ("mean_absolute_error :",metrics.mean_absolute_error(test_labels, y_pred))
print ("mean_squared_error : ",metrics.mean_squared_error(test_labels, y_pred))
print ("root_mean_squared_error : ",np.sqrt(metrics.mean_squared_error(test_labels, y_pred)))
ridgereg = Ridge(alpha=0.1, normalize=True)
ridgereg.fit(train_features, train_labels)
y_pred = ridgereg.predict(test_features)
print("R-Square Value",r2_score(test_labels,y_pred))
print ("mean_absolute_error :",metrics.mean_absolute_error(test_labels, y_pred))
print ("mean_squared_error : ",metrics.mean_squared_error(test_labels, y_pred))
print ("root_mean_squared_error : ",np.sqrt(metrics.mean_squared_error(test_labels, y_pred)))
print(ridgereg.coef_)

R-Square Value 0.02020246972837836
mean_absolute_error : 11950.144186561989
mean_squared_error :  285134727.5654296
root_mean_squared_error :  16885.932830774545
R-Square Value 0.020080346450857656
mean_absolute_error : 11939.191499945779
mean_squared_error :  285170267.1400758
root_mean_squared_error :  16886.985140636436
[-1.83744566e+02  5.12760030e+01 -2.35563745e+03 -1.96128702e+01
  1.30913268e+03  2.40050122e+03  3.91052677e+02 -3.56696578e+01
 -2.03058346e-01  8.41517289e+02]
```

LOGISTIC REGRESSION:

In the 2nd type of business question, by analyzing the data we are mainly focusing on who is going to buy the vehicle insurance. We are using logistic regression since we need to predict binary outcomes for the variable response.

Before using the logistic regression, we had a check on the proportion of the people who bought vehicle insurance and people who didn't buy the insurance. From the below figure we can see that it clearly looks like an imbalanced dataset.



To deal with this type of issue we used an imbalance dataset handling technique which is under-sampling. In this form, data is randomly selected from the normal category to match

the number of data with a 50:50 percent ratio. For this, we used Nearmiss library from the imblearn package.

```
In [38]: from imblearn.under_sampling import NearMiss
         nr = NearMiss()
         train_features, train_labels = nr.fit_resample(train_features, train_labels)
```

Then applied logistic regression and checked the accuracy. Since it is an imbalance dataset our major Focus remains on the Precision and recall values. We receive an accuracy of around 35%, recall of around 82%, and precision of around 10% as shown in the figure below:

```
In [40]: from sklearn.linear_model import LogisticRegression

         logreg = LogisticRegression(max_iter=1000)

         logreg.fit(train_features, train_labels)

         y_pred=logreg.predict(test_features)

         print(classification_report(test_labels, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.30      0.46    138423
           1       0.11      0.82      0.19     14021

    accuracy                           0.35    152444
   macro avg       0.53      0.56      0.32    152444
weighted avg       0.87      0.35      0.43    152444
```

```
In [41]: from sklearn import metrics
         print("Accuracy:",metrics.accuracy_score(test_labels, y_pred)*100)
```

```
Accuracy: 35.086982760882684
```

```
In [42]: print("Accuracy:",metrics.accuracy_score(test_labels, y_pred))
         print("Precision:",metrics.precision_score(test_labels, y_pred))
         print("Recall:",metrics.recall_score(test_labels, y_pred))
```

```
Accuracy: 0.35086982760882685
Precision: 0.1067787664700599
Recall: 0.8224805648669853
```

Also computed the confusion matrix to check the true positive and true negative values which can be seen from the below figure:

```
In [43]: from sklearn import metrics
         cnf_matrix = metrics.confusion_matrix(test_labels, y_pred)
         cnf_matrix
```

```
Out[43]: array([[41956, 96467],
                [ 2489, 11532]], dtype=int64)
```

We then used RFE for improving the accuracy of our model. The Recursive Feature Elimination (RFE) method is a feature selection approach. It works by recursively removing attributes and building a model on those attributes that remain.

```
In [58]: from sklearn.feature_selection import RFE
         rfe = RFE(logreg, n_features_to_select=8)
         fit = rfe.fit(train_features, train_labels)
         print("Num Features: %d" % fit.n_features_)
         print("Selected Features: %s" % fit.support_)
         print("Feature Ranking: %s" % fit.ranking_)
         print("Features:", train_features.columns)

Num Features: 8
Selected Features: [False  True  True  True  True  True  True False  True False]
Feature Ranking: [4 1 1 1 1 1 1 3 1 2]
Features: Index(['id', 'Gender', 'Age', 'Driving_License', 'Region_Code',
       'Previously_Insured', 'Vehicle_Age', 'Vehicle_Damage', 'Annual_Premium',
       'Policy_Sales_Channel', 'Vintage'],
      dtype='object')
```

```
In [64]: logreg_imp = LogisticRegression(max_iter=1000)
         train_important = train_features.drop(['id','Annual_Premium','Vintage'], axis=1)
         test_important = test_features.drop(['id','Annual_Premium','Vintage'], axis=1)
         logreg_imp.fit(train_important, train_labels)
         predictions = logreg_imp.predict(test_important)
         errors = abs(predictions - test_labels)
         print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
         print(classification_report(test_labels, y_pred))

Mean Absolute Error: 0.38 degrees.
              precision    recall  f1-score   support

           0       0.94      0.30      0.46    138423
           1       0.11      0.82      0.19     14021

    accuracy                           0.35    152444
   macro avg       0.53      0.56      0.32    152444
weighted avg       0.87      0.35      0.43    152444
```

After using RFE we can see that the accuracy of the model looks more satisfying with around 62%, the precision of around 18%, and a recall value of 96%.

```
In [66]: from sklearn import metrics
         print("Accuracy:",metrics.accuracy_score(test_labels, predictions)*100)

Accuracy: 61.53341554931647
```

```
In [67]: print("Accuracy:",metrics.accuracy_score(test_labels, predictions))
         print("Precision:",metrics.precision_score(test_labels, predictions))
         print("Recall:",metrics.recall_score(test_labels, predictions))

Accuracy: 0.6153341554931647
Precision: 0.1887842644904792
Recall: 0.9651950645460381
```

Then after computing again the confusion matrix we can see that false positive and false negative rates have declined.

```
In [65]: from sklearn import metrics
         cnf_matrix = metrics.confusion_matrix(test_labels, predictions)
         cnf_matrix

Out[65]: array([[80271, 58152],
                [  488, 13533]], dtype=int64)
```

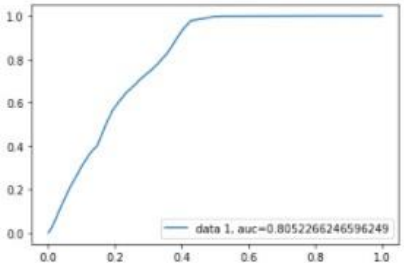
AUC ranges in value from 0 to 1. A model whose predictions are 100% wrong has an AUC of 0.0; one whose predictions are 100% correct has an AUC of 1.0.

AUC is desirable for the following two reasons:

- AUC is scale-invariant. It measures how well predictions are ranked, rather than their absolute values.
- AUC is a classification-threshold-invariant. It measures the quality of the model's predictions irrespective of what classification threshold is chosen.

From the below figure we can see that we have received an AUC score of around 80.52%.

```
In [68]: y_pred_proba = logreg_imp.predict_proba(test_important)[::,1]
         fpr, tpr, _ = metrics.roc_curve(test_labels,  y_pred_proba)
         auc = metrics.roc_auc_score(test_labels, y_pred_proba)
         plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
         plt.legend(loc=4)
         plt.show()
```



DECISION TREE:

A decision tree is a type of flowchart that demonstrates a clear pathway to a decision. It is a type of algorithm that has condition control statements that are used to classify the data. The decision tree consists of a tree-like structure that has decision nodes and leaf nodes in it. Decision nodes represent a decision and leaf nodes represent a decision that is taken after computing all attributes. Decision trees are mainly used so that we can breakdown of our

complex data into manageable parts.

```
In [ ]: x=result_df.drop('Response', axis = 1)
        y=result_df['Response']
        from sklearn.model_selection import train_test_split
        train_features, test_features, train_labels, test_labels = train_test_split(x, y, test_size = 0.3, random_state = 0)

In [93]: from sklearn.tree import DecisionTreeClassifier
         dt = DecisionTreeClassifier()
         dt.fit(train_features, train_labels)

Out[93]: DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                                max_depth=None, max_features=None, max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, presort='deprecated',
                                random_state=None, splitter='best')

In [94]: dt_pred = dt.predict(test_features)

In [95]: errors = abs(dt_pred - test_labels)
         print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
         print("Accuracy:",metrics.accuracy_score(test_labels, dt_pred)*100)
         conf = confusion_matrix(test_labels,predictions)
         print(conf)

         Mean Absolute Error: 0.13 degrees.
         Accuracy: 86.7210254257301
         [[55086 83337]
          [ 1820 12201]]

In [96]: print("Accuracy:",metrics.accuracy_score(test_labels, dt_pred))
         print("Precision:",metrics.precision_score(test_labels, dt_pred))
         print("Recall:",metrics.recall_score(test_labels, dt_pred))

         Accuracy: 0.867210254257301
         Precision: 0.29282099094299413
         Recall: 0.3136010270308823
```

We didn’t apply SMOTE (imbalance data handling technique) to decision tree since it can by default handle the imbalance dataset.

We receive accuracy as 86.72%, Recall about 32.36% and Precision of around 29.28% as shown in the figure. Also computed the confusion matrix to check the true positive and true negative values.

```
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier()
dt.fit(train_features, train_labels)
dt_pred = dt.predict(test_features)
stop = timeit.default_timer()
print('Time: ', stop - start)
```

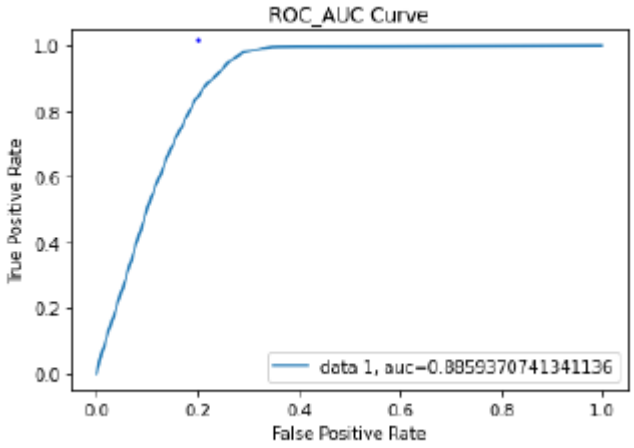
```
Time:  5.57106930000009
```

Run time for decision tree was comparatively lower than other models which was around 5.57 seconds which can be seen in above figure.

```
In [97]: importances = list(dt.feature_importances_)
         feature_importances = [(feature, round(importance, 2)) for feature, importance in zip(x.columns, importances)]
         feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse = True)
         [print('Variable: {:20} Importance: {}'.format(*pair)) for pair in feature_importances];

Variable: id                   Importance: 0.29
Variable: Vintage              Importance: 0.19
Variable: Annual_Premium       Importance: 0.18
Variable: Age                  Importance: 0.1
Variable: Vehicle_Damage       Importance: 0.09
Variable: Region_Code          Importance: 0.08
Variable: Policy_Sales_Channel Importance: 0.04
Variable: Gender               Importance: 0.02
Variable: Previously_Insured   Importance: 0.01
Variable: Vehicle_Age          Importance: 0.01
Variable: Driving_License      Importance: 0.0
```

In the above image we calculated the feature importance for our data. Basically, the higher the value, the more important is the feature. In this we can see that id has the highest importance followed by Vintage and Annual Premium.

```
In [103]: y_pred_proba = dt.predict_proba(test_features)[::,1]
          fpr, tpr, _ = metrics.roc_curve(test_labels,  y_pred_proba)
          auc = metrics.roc_auc_score(test_labels, y_pred_proba)
          print(auc)
          plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
          plt.legend(loc=4)
          plt.title("ROC_AUC Curve")
          plt.xlabel("False Positive Rate")
          plt.ylabel("True Positive Rate")
          plt.show()

0.618443448576811
```



ROC curve are mainly used to check or visualize the performance of multi classification problem. ROC (Receiver Operating Characteristic) – AUC (Area under curve) tells us how much model is capable of distinguishing between the classes. The higher is the AUC better is the performance of the model. Here we got the AUC score as 61.84%

The training process of the decision tree is a recursive process, the algorithm will continue to choose the optimal partitioning properties, generation of branches and nodes, the current node

contains the samples all belong to the same category, do not need to divide. So if minority data are all in one area of the feature space, then all samples will be will be obtained.

RANDOM FOREST:

Random forest is a supervised learning algorithm that uses both classification and regression. Random forest algorithm creates multiple decision trees on data samples and then further gets the prediction from each of them and finally chooses the best solution. Random forest reduces the overfitting problem that is mainly caused by the decision tree by averaging the result of them.

We didn't apply SMOTE (imbalance data handling technique) for random forest as well since it can by default handle the imbalance dataset.

In the above image we can see that we have used random forest classifier and got the accuracy as 90.47%, Precision as 39.9% and Recall as 7.0%. Also computed the confusion matrix to check the true positive and true negative values.

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators = 1000, random_state=42)
rf.fit(train_features, train_labels)
predictions = rf.predict(test_features)
stop = timeit.default_timer()
print('Time: ', stop - start)
```

```
Time:  1013.0330710999988
```

```
In [105]: predictions = rf.predict(test_features)
          errors = abs(predictions - test_labels)
          print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')

          Mean Absolute Error: 0.1 degrees.

In [110]: print("Accuracy:",metrics.accuracy_score(test_labels, predictions))
          print("Precision:",metrics.precision_score(test_labels, predictions))
          print("Recall:",metrics.recall_score(test_labels, predictions))
          conf = confusion_matrix(test_labels,predictions)
          print(conf)

          Accuracy: 0.9047781480412479
          Precision: 0.3998381222172563
          Recall: 0.07046572997646387
          [[136940   1483]
           [ 13033    988]]
```

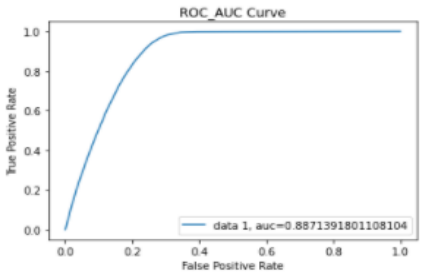
Run time for Random forest was higher than decision tree which was around 1013 seconds

which can be seen from above figure.

```
In [107]: importances = list(rf.feature_importances_)
          feature_importances = [(feature, round(importance, 2)) for feature, importance in zip(x.columns, importances)]
          feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse = True)
          [print('Variable: {:20} Importance: {}'.format(*pair)) for pair in feature_importances];

Variable: id                   Importance: 0.26
Variable: Vintage              Importance: 0.19
Variable: Annual_Premium       Importance: 0.17
Variable: Age                  Importance: 0.12
Variable: Region_Code          Importance: 0.08
Variable: Vehicle_Damage       Importance: 0.06
Variable: Policy_Sales_Channel Importance: 0.05
Variable: Previously_Insured   Importance: 0.04
Variable: Gender               Importance: 0.01
Variable: Vehicle_Age          Importance: 0.01
Variable: Driving_License      Importance: 0.0
```

We have used feature importance to see which feature has higher importance among all features. We got the same features as we received in the Decision tree feature importance.

```
In [111]: y_pred_proba = rf.predict_proba(test_features)[::,1]
          fpr, tpr, _ = metrics.roc_curve(test_labels,  y_pred_proba)
          auc = metrics.roc_auc_score(test_labels, y_pred_proba)
          print(auc)
          plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
          plt.legend(loc=4)
          plt.title("ROC_AUC Curve")
          plt.xlabel("False Positive Rate")
          plt.ylabel("True Positive Rate")
          plt.show()

0.8871391801108104
```



From the image above we can see that we got the AUC score as 88.71%.

GRADIENT BOOSTING:

Gradient boosting is a technique mainly used for prediction speed, accuracy when it comes to large and complex data. The main objective of any supervised learning algorithm is to define loss and implement methods so that loss can be minimized. It deals with the best possible next model when it is combined with previous models to minimize the overall prediction error. The key idea is to set the target outcomes for the next model in order to minimize the error. The errors received previously is highlighted and combined with the next learner so that the errors are reduced significantly.

In Gradient Boosting, we got the accuracy as 83.90%, Precision as 31.69% and Recall as

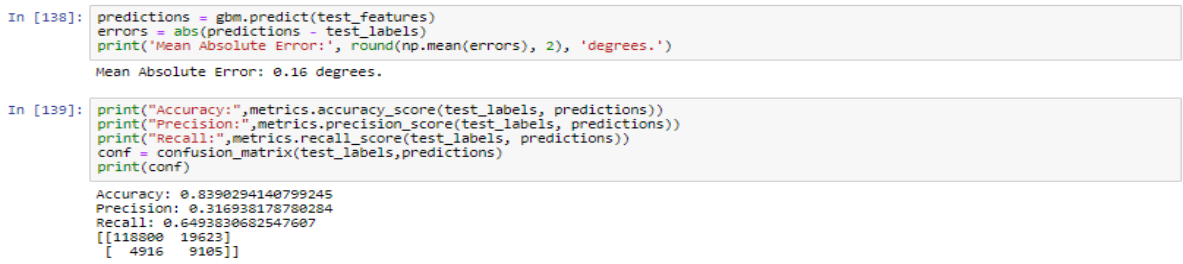
64.93%. Also computed the confusion matrix to check the true positive and true negative values.

```
param ={
        "learning_rate" : [1, 0.5, 0.25, 0.1, 0.05, 0.01],
        "n_estimators" : [100, 200]}

grid_search = GridSearchCV(estimator = gbm, param_grid = param,
                           cv = 3, n_jobs = -1, verbose = 2)
grid_search.fit(train_features, train_labels)
grid_search.best_params_
```

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  36 out of  36 | elapsed: 18.5min finished
{'learning_rate': 0.5, 'n_estimators': 200}
```

```
from sklearn.ensemble import GradientBoostingClassifier
gbm = GradientBoostingClassifier()
gbm.fit(train_features,train_labels)
predictions = gbm.predict(test_features)
stop = timeit.default_timer()
print('Time: ', stop - start)
```

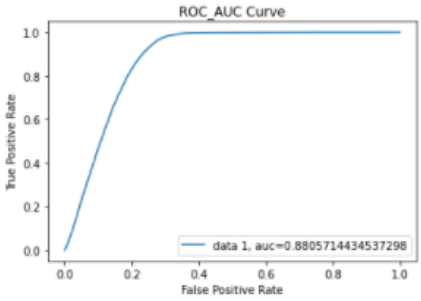
```
Time:  207.11286189999373
```



Run time for Gradient Boosting Method was higher than decision tree which was around 207 seconds but lower than Random forest model which can be seen from above figure.

```
In [140]: y_pred_proba = gbm.predict_proba(test_features)[::,1]
fpr, tpr, _ = metrics.roc_curve(test_labels,  y_pred_proba)
auc = metrics.roc_auc_score(test_labels, y_pred_proba)
print(auc)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.title("ROC_AUC Curve")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.show()
```

```
0.8805714434537298
```



For Gradient boosting we got AUC score as 88.05% as seen in the image above.

XGBOOST:

XGboost stands for extreme gradient boosting. Gradient boosting is mainly used to accurately predict a target variable by combining the estimates of a set of weaker and simpler models. XGboost provides a parallel tree boosting that solves a number of data problems in a fast and accurate way. XGBoost can be used to solve regression and classification problems.

For XGBoost, we got accuracy as 79.96%, Precision as 28.83% and Recall as 80.29%. Also computed the confusion matrix to check the true positive and true negative values.

```
from sklearn.model_selection import GridSearchCV
parameters = {
     "eta"    : [0.05, 0.10, 0.15, 0.20, 0.25, 0.30 ] ,
     "max_depth"        : [ 3, 4, 5, 6, 8, 10, 12, 15]}

grid = GridSearchCV(model_xgb,
                    parameters, n_jobs=-1,
                    scoring="neg_log_loss",
                    cv=3, verbose=2)
grid.fit(train_features, train_labels)
```

```
Fitting 3 folds for each of 48 candidates, totalling 144 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:  8.6min
[Parallel(n_jobs=-1)]: Done 144 out of 144 | elapsed: 48.2min finished
```

```
GridSearchCV(cv=3,
             estimator=XGBClassifier(base_score=0.5, booster='gbtree',
                                     colsample_bylevel=1, colsample_bynode=1,
                                     colsample_bytree=1, 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=0,
                                     num_parallel_tree=1, random_state=0,
                                     reg_alpha=0, reg_lambda=1,
                                     scale_pos_weight=1, subsample=1,
                                     tree_method='exact', validate_parameters=1
                                     verbosity=None),
             n_jobs=-1,
             param_grid={'eta': [0.05, 0.1, 0.15, 0.2, 0.25, 0.3],
                         'max_depth': [3, 4, 5, 6, 8, 10, 12, 15]},
             scoring='neg_log_loss', verbose=2)
```

```
from xgboost import XGBClassifier as xgb
model_xgb = xgb()
model_xgb.fit(train_features,train_labels)
best_preds = model_xgb.predict(test_features)
stop = timeit.default_timer()
print('Time: ', stop - start)
```

```
Time:  40.503347099991515
```

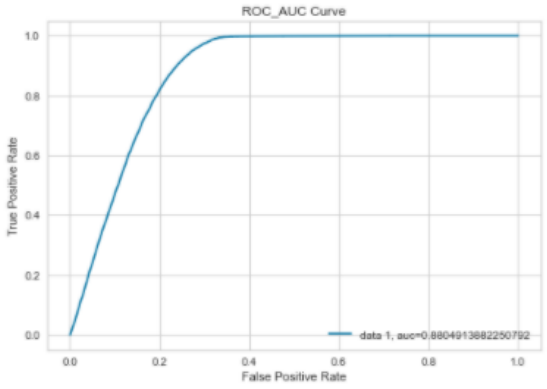
In [158]:
```
pred_xgb = np.asarray([np.argmax(line) for line in best_preds])
errors = abs(pred_xgb - test_labels)
print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
print("Accuracy:",metrics.accuracy_score(test_labels, pred_xgb))
print("Precision:",metrics.precision_score(test_labels, pred_xgb))
print("Recall:",metrics.recall_score(test_labels, pred_xgb))
conf = confusion_matrix(test_labels,pred_xgb)
print(conf)
```

```
Mean Absolute Error: 0.2 degrees.
Accuracy: 0.7996247802471728
Precision: 0.28836351527880943
Recall: 0.8029384494686541
[[110640  27783]
 [  2763  11258]]
```

Run time for Gradient Boosting Method was higher than decision tree which was around 40 seconds but lower than Random forest model and Gradient boosting method which can be seen from above figure.

In [205]:
```
y_pred_proba = model_xgb.predict_proba(test_features)[::,1]
fpr, tpr, _ = metrics.roc_curve(test_labels,  y_pred_proba)
auc = metrics.roc_auc_score(test_labels, y_pred_proba)
print(auc)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.title("ROC_AUC Curve")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.show()
```

```
0.8804913882250792
```



For XG boost we got AUC score as 88.04% as seen in the image above.

NEURAL NETWORKS:

Since the dataset is imbalance, we used Smote package which is used for oversampling. Oversampling is a technique in which we add values in the same dimensions to make the negative value balance with positive value in the dataset.

```
In [176]: from sklearn.model_selection import train_test_split
          from imblearn.over_sampling import SMOTE
          train_features, test_features, train_labels, test_labels = train_test_split(x, y, test_size = 0.3)
          nr = SMOTE()
          train_features, train_labels = nr.fit_sample(train_features, train_labels)

In [177]: print('Training Features Shape:', train_features.shape)
          print('Training Labels Shape:', train_labels.shape)
          print('Testing Features Shape:', test_features.shape)
          print('Testing Labels Shape:', test_labels.shape)

          Training Features Shape: (645986, 11)
          Training Labels Shape: (645986,)
          Testing Features Shape: (152444, 11)
          Testing Labels Shape: (152444,)
```

Sequential is the easiest way to build a model in Keras. It allows you to build a model layer by layer. Each layer has weights that correspond to the layer that follows it. We use the 'add()' function to add layers to our model. We will add two layers and an output layer. Which can be seen in below figure.

The main advantage of using the RELU function over other activation functions is that it does not activate all the neurons at the same time. This means that the neurons will only be deactivated if the output of the linear transformation is less than 0

```
In [178]: def build_model():
              classifier = Sequential()
              classifier.add(Dense(6, activation='relu', kernel_initializer='random_normal', input_dim=11))
              classifier.add(Dense(6, activation='relu', kernel_initializer='random_normal'))
              classifier.add(Dense(1, activation='sigmoid', kernel_initializer='random_normal'))
              classifier.compile(optimizer ='adam',loss='binary_crossentropy', metrics =['accuracy'])
              return classifier

In [179]: keras_model = build_model()
          keras_model.fit(train_features,train_labels, batch_size=64, epochs=100)
          Epoch 92/100
          10094/10094 [==============================] - 18s 2ms/step - loss: 0.3478 - accuracy: 0.8484
          Epoch 93/100
          10094/10094 [==============================] - 18s 2ms/step - loss: 0.3478 - accuracy: 0.8484
          Epoch 94/100
          10094/10094 [==============================] - 18s 2ms/step - loss: 0.3477 - accuracy: 0.8483
          Epoch 95/100
          10094/10094 [==============================] - 18s 2ms/step - loss: 0.3477 - accuracy: 0.8485
          Epoch 96/100
          10094/10094 [==============================] - 18s 2ms/step - loss: 0.3478 - accuracy: 0.8484
          Epoch 97/100
          10094/10094 [==============================] - 18s 2ms/step - loss: 0.3477 - accuracy: 0.8483
          Epoch 98/100
          10094/10094 [==============================] - 18s 2ms/step - loss: 0.3478 - accuracy: 0.8483
          Epoch 99/100
          10094/10094 [==============================] - 17s 2ms/step - loss: 0.3478 - accuracy: 0.8483
          Epoch 100/100
          10094/10094 [==============================] - 17s 2ms/step - loss: 0.3477 - accuracy: 0.8484

Out[179]: <tensorflow.python.keras.callbacks.History at 0x22535d4b970>
```

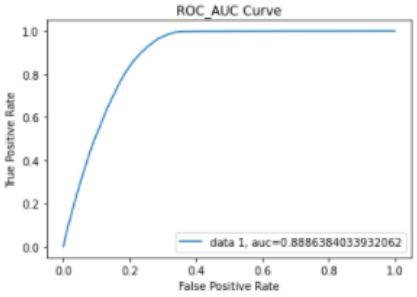
We can see from the above figure you can see that we have used epochs=100. Epochs is basically number of times all of the training vectors are used once to update the weights. We can see that after 100 iterations we received an accuracy rate of around 84.84% and loss (Error over the training set since we are using classification it is log loss) of around 34.77% on our training dataset.

After using sequential model, we got accuracy as 75.17%, Precision as 26.27% and Recall as 94.30%. Also computed the confusion matrix to check the true positive and true negative values which can be seen in the figure below.

```
In [182]: print("Accuracy:",metrics.accuracy_score(test_labels, y_pred))
          print("Precision:",metrics.precision_score(test_labels, y_pred))
          print("Recall:",metrics.recall_score(test_labels, y_pred))
          cm = confusion_matrix(test_labels, y_pred)
          print(cm)

          Accuracy: 0.7517514628322531
          Precision: 0.2627609400807944
          Recall: 0.943075494607528
          [[101396  37047]
           [   797  13204]]
```

From the below figure we can see that we have received an AUC score of around 88.86%.

```
In [183]: from keras.wrappers.scikit_learn import KerasClassifier
          from sklearn.metrics import auc
          from sklearn.metrics import roc_curve
          y_pred_keras = keras_model.predict(test_features).ravel()
          fpr_keras, tpr_keras, thresholds_keras = roc_curve(test_labels, y_pred_keras)
          auc_keras = auc(fpr_keras, tpr_keras)
          plt.plot(fpr_keras,tpr_keras,label="data 1, auc="+str(auc_keras))
          plt.legend(loc=4)
          plt.title("ROC_AUC Curve")
          plt.xlabel("False Positive Rate")
          plt.ylabel("True Positive Rate")
          plt.show()
```



CONCLUSION:

- We applied Linear and Logistic Regression models to solve our business questions of predicting an annual premium customer should be paying for their Vehicle Insurance using Linear Regression and to identify customers interested in buying Vehicle Insurance using the Logistic Regression model.

- After applying Logistic Regression, we received an accuracy of around 35% and in order to improve the accuracy of our model, we applied the feature selection technique using the Recursive Feature Elimination (RFE) method to eliminate the less important features. RFE method improved the accuracy of our model from 35% to 62% with a precision of around 18% and a recall value of 96%. In the further step, we used decision tree , random forest classifier, gradient boosting and XGboost and obtained an accuracy of 90%, 90.47% ,84%,84.71% respectively. From this we can see that random forest yield the best accuracy for predicting which customers are going to buy the vehicle insurance again.
- In the final stage we used neural networks model to solve our problem of predicting whether the customers owning Health Insurance are going to buy Vehicle Insurance or not and we got the accuracy as 75.17%.