car_insurance_prediction (1)

January 19, 2025

1 Insurance Claim Prediction Project

1.1 Project Overview

The objective of this project is to predict the likelihood of an insurance claim based on various features of the policyholder and the insured vehicle. By leveraging machine learning algorithms, we aim to build a model that can classify whether an insurance claim will be filed for a particular policy, helping insurance companies assess risk and make data-driven decisions.

1.2 Data Description

The dataset contains various features related to the insurance policies and insured cars. The key features are:

- **policy_id**: The unique identifier for each insurance policy.
- **policy_tenure**: The length of time (in years) that the policy has been active.
- **age_of_car**: The age of the insured car (in years) at the time the policy was taken.
- **age_of_policyholder**: The age of the policyholder (in years) at the time the policy was taken.

- **area_cluster**: A categorical variable representing the cluster or category to which the area of residence belongs.
- **population_density**: A measure of the population density of the area where the policyholder resides.
- **Make**: The make or manufacturer of the insured car.
- **segment**: The segment or category to which the insured car belongs (e.g., compact, sedan, SUV).
- **fuel_type**: The type of fuel used by the insured car (e.g., petrol, diesel, electric).
- **max_torque**: The maximum torque output of the car's engine.
- **max_power**: The maximum power output of the car's engine.
- **engine_type**: The type of engine used in the insured car (e.g., inline, V-type).
- **airbags**: The number of airbags installed in the car.
- **is_esc**: A binary variable indicating whether the car has an electronic stability control (ESC) system.
- **is_adjustable_steering**: A binary variable indicating whether the car has adjustable steering.
- **is_tpms**: A binary variable indicating whether the car has a tire pressure monitoring system (TPMS).
- **is_parking_sensors**: A binary variable indicating whether the car has parking sensors.
- **is_parking_camera**: A binary variable indicating whether the car has a parking camera.
- **ncap_rating**: The safety rating of the car according to the New Car Assessment Program (NCAP).
- **is_claim** (Target Variable): A binary variable indicating whether an insurance claim has been filed for the car policy.

1.3 Problem Statement

The main goal of this project is to predict whether an insurance claim will be filed for a given policy based on the features provided in the dataset. The target variable is **is_claim**, which is binary (0 for no claim, 1 for claim).

1.4 Objective

- **Risk Assessment**: Identify which factors increase the likelihood of an insurance claim.
- **Predictive Model**: Build a predictive model that helps insurance companies assess the risk of claims for future customers and make informed decisions.
- **Feature Importance**: Understand which features are most predictive of an insurance claim.

1.5 Approach

1. **Data Preprocessing**: Clean the data by handling missing values, encoding categorical features, and scaling numerical variables.
2. **Exploratory Data Analysis (EDA)**: Analyze the data to identify trends, correlations, and patterns related to insurance claims.
3. **Model Building**: Train machine learning models (e.g., Logistic Regression, Decision Trees, Random Forests) to predict the likelihood of an insurance claim.
4. **Evaluation**: Evaluate the models using accuracy, precision, recall, F1-score, and ROC-AUC to determine the best performing model.

5. **Insights**: Provide insights based on the model's performance and the importance of various features in predicting insurance claims.

1.6 Dataset

The dataset used for this project is provided by Learnbay and contains various columns, including both numerical and categorical features that describe the insurance policies and the cars covered under those policies.

1.7 Tools & Libraries Used

- Python
- Jupyter Notebook
- pandas
- numpy
- scikit-learn
- matplotlib
- seaborn
- plotly
- SHAP

[3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import plotly.offline as py
pd.options.plotting.backend = "plotly"
import warnings
warnings.filterwarnings('ignore')
```

[5]:
```
pd.set_option("display.max_columns",None)
```

[7]:
```
df = pd.read_csv("car_insurance_prediction_data.csv")
```

[7]:
```
df.head()
```

[7]:

	policy_id	policy_tenure	age_of_car	age_of_policyholder	area_cluster	\
0	ID00001	0.515874	0.05	0.644231	C1	
1	ID00002	0.672619	0.02	0.375000	C2	
2	ID00003	0.841110	0.02	0.384615	C3	
3	ID00004	0.900277	0.11	0.432692	C4	
4	ID00005	0.596403	0.11	0.634615	C5	

	population_density	make	segment	model	fuel_type	max_torque	\

```
0                4990     1       A    M1       CNG   60Nm@3500rpm
1               27003     1       A    M1       CNG   60Nm@3500rpm
2                4076     1       A    M1       CNG   60Nm@3500rpm
3               21622     1      C1    M2    Petrol  113Nm@4400rpm
4               34738     2       A    M3    Petrol   91Nm@4250rpm

          max_power          engine_type  airbags is_esc  \
0  40.36bhp@6000rpm   F8D Petrol Engine        2     No
1  40.36bhp@6000rpm   F8D Petrol Engine        2     No
2  40.36bhp@6000rpm   F8D Petrol Engine        2     No
3  88.50bhp@6000rpm  1.2 L K12N Dualjet        2    Yes
4  67.06bhp@5500rpm             1.0 SCe        2     No

  is_adjustable_steering is_tpms is_parking_sensors is_parking_camera  \
0                     No      No                Yes                No
1                     No      No                Yes                No
2                     No      No                Yes                No
3                    Yes      No                Yes               Yes
4                     No      No                 No               Yes

  rear_brakes_type  displacement  cylinder transmission_type  gear_box  \
0             Drum           796         3            Manual         5
1             Drum           796         3            Manual         5
2             Drum           796         3            Manual         5
3             Drum          1197         4         Automatic         5
4             Drum           999         3         Automatic         5

  steering_type  turning_radius  length  width  height  gross_weight  \
0         Power             4.6    3445   1515    1475          1185
1         Power             4.6    3445   1515    1475          1185
2         Power             4.6    3445   1515    1475          1185
3      Electric             4.8    3995   1735    1515          1335
4      Electric             5.0    3731   1579    1490          1155

  is_front_fog_lights is_rear_window_wiper is_rear_window_washer  \
0                  No                   No                   No
1                  No                   No                   No
2                  No                   No                   No
3                 Yes                   No                   No
4                  No                   No                   No

  is_rear_window_defogger is_brake_assist is_power_door_locks  \
0                      No             No                  No
1                      No             No                  No
2                      No             No                  No
3                     Yes            Yes                 Yes
4                      No             No                 Yes
```

```
  is_central_locking is_power_steering is_driver_seat_height_adjustable  \
0                 No               Yes                                No
1                 No               Yes                                No
2                 No               Yes                                No
3                Yes               Yes                               Yes
4                Yes               Yes                                No

  is_day_night_rear_view_mirror is_ecw is_speed_alert  ncap_rating  is_claim
0                            No     No            Yes            0         0
1                            No     No            Yes            0         0
2                            No     No            Yes            0         0
3                           Yes    Yes            Yes            2         0
4                           Yes    Yes            Yes            2         0
```

[9]:
```
def missing_plot(dataset):
    null_feat = pd.DataFrame(dataset.isnull().sum(),columns=['Count'])
    null_percentage = pd.DataFrame(dataset.isnull().sum()/
 ↪len(dataset),columns=['Count'])

    trace = go.Bar(x= null_feat.index, y = null_feat['Count'],opacity=0.8,
                   text = null_percentage['Count'],textposition='auto',
                   marker=dict(color = '#D84E5F',
                               line = dict(color = '#000000',width = 1.5)))
    layout = dict(height=600,width = 1000,title = 'Missing values analysis by␣
 ↪Barplot')

    fig = dict(data=[trace], layout= layout)

    py.iplot(fig)

def check(df_):
    print('SHAPE'.center(60,'*'))
    print('OBSERVATIONS -------->{}'.format(df_.shape[0]))
    print('FEATURES -------->{}'.format(df_.shape[1]))
    print('TYPES OF FETAURES'.center(60,'*'))
    print(df_.dtypes,'\n')
    print('Duplicate Values Analysis'.center(60,'*'))
    print('\n',df_.duplicated().sum(),'\n')
    print(''.center(60,'*'))
```

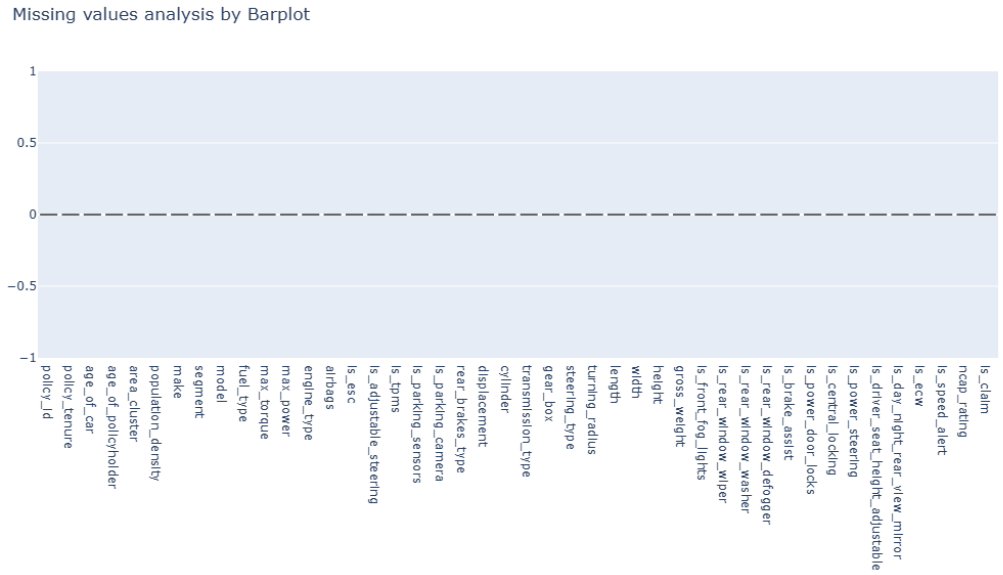
[11]:
```
check(df)
missing_plot(df)
```

```
**************************SHAPE**************************
OBSERVATIONS -------->58592
FEATURES -------->44
********************TYPES OF FETAURES*********************
```

```
policy_id                            object
policy_tenure                       float64
age_of_car                          float64
age_of_policyholder                 float64
area_cluster                         object
population_density                    int64
make                                  int64
segment                              object
model                                object
fuel_type                            object
max_torque                           object
max_power                            object
engine_type                          object
airbags                               int64
is_esc                               object
is_adjustable_steering               object
is_tpms                              object
is_parking_sensors                   object
is_parking_camera                    object
rear_brakes_type                     object
displacement                          int64
cylinder                              int64
transmission_type                    object
gear_box                              int64
steering_type                        object
turning_radius                      float64
length                                int64
width                                 int64
height                                int64
gross_weight                          int64
is_front_fog_lights                  object
is_rear_window_wiper                 object
is_rear_window_washer                object
is_rear_window_defogger              object
is_brake_assist                      object
is_power_door_locks                  object
is_central_locking                   object
is_power_steering                    object
is_driver_seat_height_adjustable     object
is_day_night_rear_view_mirror        object
is_ecw                               object
is_speed_alert                       object
ncap_rating                           int64
is_claim                              int64
dtype: object

*****************Duplicate Values Analysis******************
```

0

****



[11]:
```
object_cols = df.select_dtypes(include='O').columns.tolist()
numeric_cols = df.select_dtypes(include=['int','float']).columns.tolist()

print("Object columns are: \n",object_cols,"\n")
print("Numeric columns are: \n",numeric_cols)
```

```
Object columns are:
 ['policy_id', 'area_cluster', 'segment', 'model', 'fuel_type', 'max_torque',
'max_power', 'engine_type', 'is_esc', 'is_adjustable_steering', 'is_tpms',
'is_parking_sensors', 'is_parking_camera', 'rear_brakes_type',
'transmission_type', 'steering_type', 'is_front_fog_lights',
'is_rear_window_wiper', 'is_rear_window_washer', 'is_rear_window_defogger',
'is_brake_assist', 'is_power_door_locks', 'is_central_locking',
'is_power_steering', 'is_driver_seat_height_adjustable',
'is_day_night_rear_view_mirror', 'is_ecw', 'is_speed_alert']

Numeric columns are:
 ['policy_tenure', 'age_of_car', 'age_of_policyholder', 'population_density',
'make', 'airbags', 'displacement', 'cylinder', 'gear_box', 'turning_radius',
'length', 'width', 'height', 'gross_weight', 'ncap_rating', 'is_claim']
```

[13]:
```
boolean_columns = [col for col in df.columns if df[col].isin(['Yes', 'No']).
  ↪all()]
```

```
print("Boolean columns are:", boolean_columns)
```

```
Boolean columns are: ['is_esc', 'is_adjustable_steering', 'is_tpms',
'is_parking_sensors', 'is_parking_camera', 'is_front_fog_lights',
'is_rear_window_wiper', 'is_rear_window_washer', 'is_rear_window_defogger',
'is_brake_assist', 'is_power_door_locks', 'is_central_locking',
'is_power_steering', 'is_driver_seat_height_adjustable',
'is_day_night_rear_view_mirror', 'is_ecw', 'is_speed_alert']
```

[15]:
```
df['max_torque'] = df['max_torque'].str.extract('(\d+\.?\d*)').astype(float)
df['max_power'] = df['max_power'].str.extract('(\d+\.?\d*)').astype(float)
```

[17]:
```
def classify_features(df):
    categorical_features = list()
    non_categorical_features = list()
    discrete_features = list()
    continuous_features = list()
    boolean_features = list()

    for i in df.columns:
        if df[i].dtype == 'object':
            if df[i].nunique() < 10:
                categorical_features.append(i)
            else:
                non_categorical_features.append(i)
        elif df[i].dtype in ['int64','float64']:
            if df[i].nunique() < 10:
                discrete_features.append(i)
            else:
                continuous_features.append(i)
    return categorical_features, non_categorical_features, discrete_features,␣
 ↪continuous_features
```

[19]:
```
categorical, non_categorical, discrete, continuous = classify_features(df)

print("Categorical features are :", categorical,"\n")
print("Non-categorical features are :", non_categorical,"\n")
print("Discrete features are :", discrete,"\n")
print("Continuous features are :", continuous)
```

```
Categorical features are : ['segment', 'fuel_type', 'is_esc',
'is_adjustable_steering', 'is_tpms', 'is_parking_sensors', 'is_parking_camera',
'rear_brakes_type', 'transmission_type', 'steering_type', 'is_front_fog_lights',
'is_rear_window_wiper', 'is_rear_window_washer', 'is_rear_window_defogger',
'is_brake_assist', 'is_power_door_locks', 'is_central_locking',
'is_power_steering', 'is_driver_seat_height_adjustable',
'is_day_night_rear_view_mirror', 'is_ecw', 'is_speed_alert']
```

```
Non-categorical features are : ['policy_id', 'area_cluster', 'model',
'engine_type']

Discrete features are : ['make', 'max_torque', 'max_power', 'airbags',
'displacement', 'cylinder', 'gear_box', 'turning_radius', 'length',
'ncap_rating', 'is_claim']

Continuous features are : ['policy_tenure', 'age_of_car', 'age_of_policyholder',
'population_density', 'width', 'height', 'gross_weight']
```

[23]:
```
num_cols = df[df.select_dtypes([int,float]).columns]
num_cols.describe().T
```

[23]:

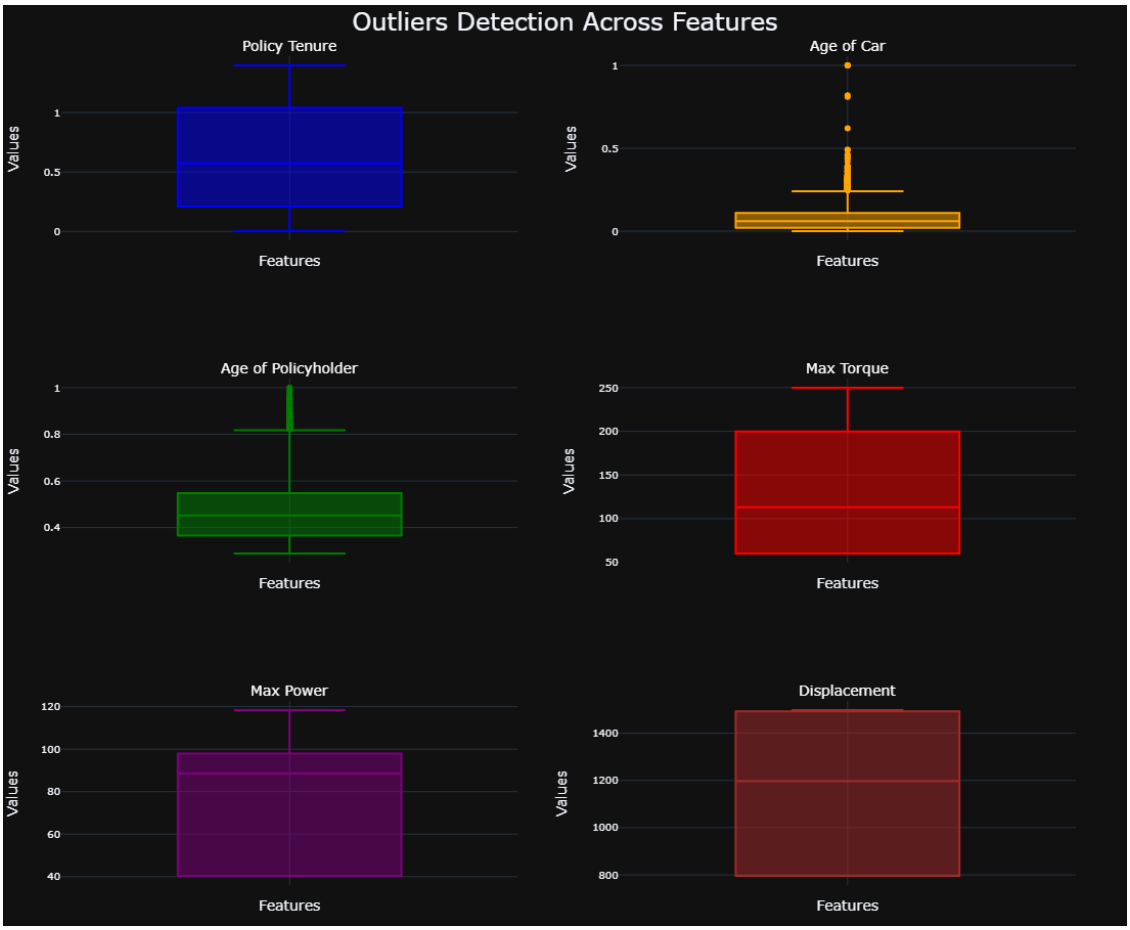
	count	mean	std	min	\
policy_tenure	58592.0	0.611246	0.414156	0.002735	
age_of_car	58592.0	0.069424	0.056721	0.000000	
age_of_policyholder	58592.0	0.469420	0.122886	0.288462	
population_density	58592.0	18826.858667	17660.174792	290.000000	
make	58592.0	1.763722	1.136988	1.000000	
max_torque	58592.0	134.450937	73.146794	60.000000	
max_power	58592.0	78.976765	27.699259	40.360000	
airbags	58592.0	3.137066	1.832641	1.000000	
displacement	58592.0	1162.355851	266.304786	796.000000	
cylinder	58592.0	3.626963	0.483616	3.000000	
gear_box	58592.0	5.245443	0.430353	5.000000	
turning_radius	58592.0	4.852893	0.228061	4.500000	
length	58592.0	3850.476891	311.457119	3445.000000	
width	58592.0	1672.233667	112.089135	1475.000000	
height	58592.0	1553.335370	79.622270	1475.000000	
gross_weight	58592.0	1385.276813	212.423085	1051.000000	
ncap_rating	58592.0	1.759950	1.389576	0.000000	
is_claim	58592.0	0.063968	0.244698	0.000000	

	25%	50%	75%	max
policy_tenure	0.210250	0.573792	1.039104	1.396641
age_of_car	0.020000	0.060000	0.110000	1.000000
age_of_policyholder	0.365385	0.451923	0.548077	1.000000
population_density	6112.000000	8794.000000	27003.000000	73430.000000
make	1.000000	1.000000	3.000000	5.000000
max_torque	60.000000	113.000000	200.000000	250.000000
max_power	40.360000	88.500000	97.890000	118.360000
airbags	2.000000	2.000000	6.000000	6.000000
displacement	796.000000	1197.000000	1493.000000	1498.000000
cylinder	3.000000	4.000000	4.000000	4.000000
gear_box	5.000000	5.000000	5.000000	6.000000
turning_radius	4.600000	4.800000	5.000000	5.200000

```
length              3445.000000  3845.000000   3995.000000   4300.000000
width               1515.000000  1735.000000   1755.000000   1811.000000
height              1475.000000  1530.000000   1635.000000   1825.000000
gross_weight        1185.000000  1335.000000   1510.000000   1720.000000
ncap_rating            0.000000     2.000000      3.000000      5.000000
is_claim               0.000000     0.000000      0.000000      1.000000
```

[25]:
```
fig = make_subplots(
    rows=3,
    cols=2,
    subplot_titles=[
        'Policy Tenure',
        'Age of Car',
        'Age of Policyholder',
        'Max Torque',
        'Max Power',
        'Displacement'
    ]
)

fig.add_trace(go.Box(y= df['policy_tenure'], name='', marker_color='blue'),␣
 ↪row=1, col=1)
fig.add_trace(go.Box(y= df['age_of_car'], name='', marker_color='orange'),␣
 ↪row=1, col=2)
fig.add_trace(go.Box(y= df['age_of_policyholder'], name='',␣
 ↪marker_color='green'), row=2, col=1)
fig.add_trace(go.Box(y= df['max_torque'], name='', marker_color='red'), row=2,␣
 ↪col=2)
fig.add_trace(go.Box(y= df['max_power'], name='', marker_color='purple'),␣
 ↪row=3, col=1)
fig.add_trace(go.Box(y= df['displacement'], name='', marker_color='brown'),␣
 ↪row=3, col=2)

fig.update_layout(
    height=900,
    width=900,
    title_text="Outliers Detection Across Features",
    title_x = 0.5,
    title_font_size=24,
    template='plotly_dark',
    showlegend=False,
    margin=dict(t=50, b=40, l=50, r=50),
)


fig.update_xaxes(showgrid=True, zeroline=False, title_text='Features',␣
 ↪tickfont=dict(size=10))
```

```
fig.update_yaxes(showgrid=True, zeroline=False, title_text='Values',␣
 ↪tickfont=dict(size=10))

fig.update_annotations(
    font=dict(size=14, color='white'),
    align='center'
)

fig.show()
```

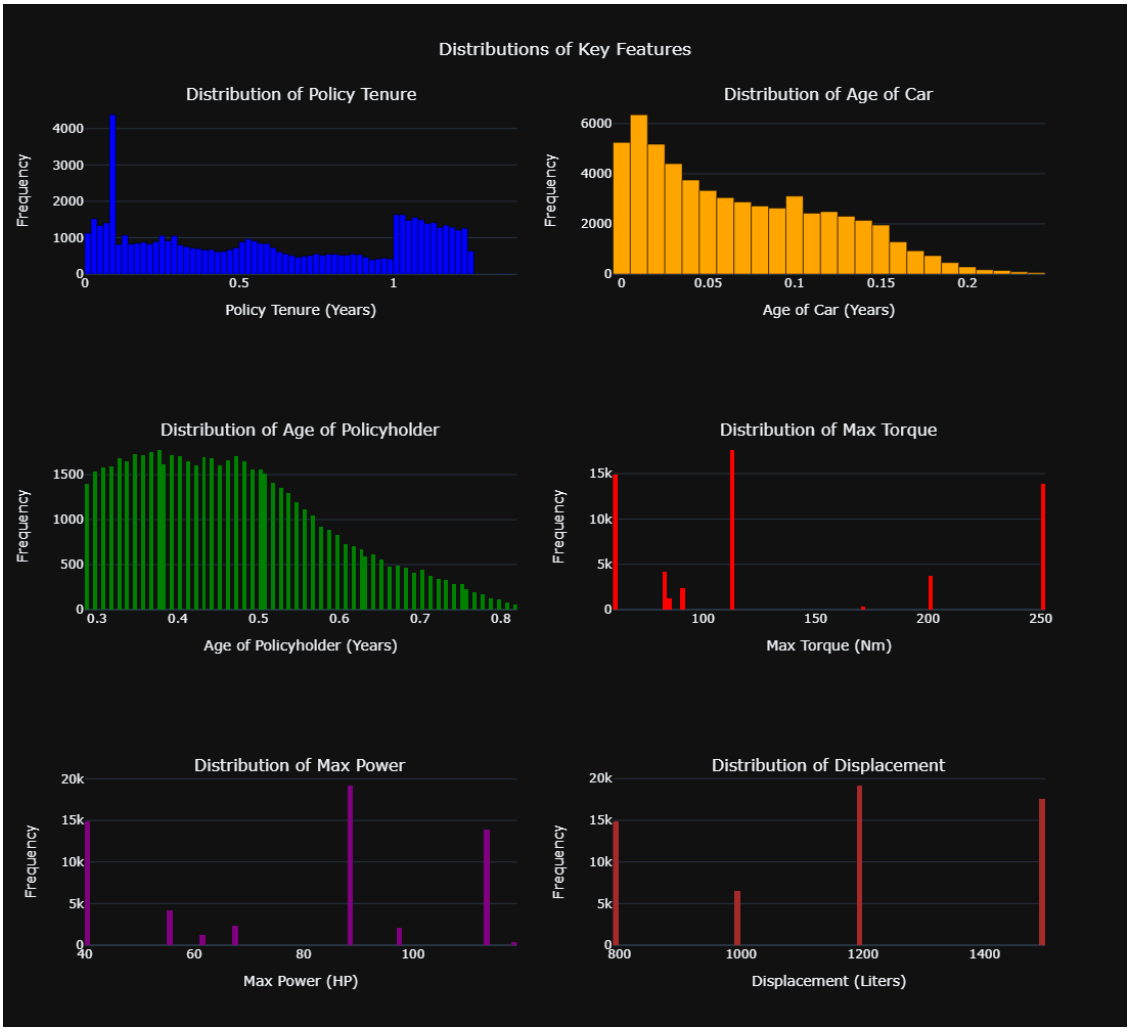


[21]:
```
# Outlier Treatment

def remove_outliers(df,col):
    Q1=df[col].quantile(0.25)
    Q3=df[col].quantile(0.75)
    IQR=Q3-Q1
    UL=Q3+1.5*(IQR)
    LL=Q1-1.5*(IQR)
```

```
    filtered_df = df[(df[col]>=LL) & (df[col]<=UL)]
    return filtered_df

imp_features = ['policy_tenure',
                'age_of_car',
                'age_of_policyholder',
                'max_torque',
                'max_power',
                'displacement']

for cols in imp_features:
    df = remove_outliers(df,cols)
```

[29]:
```
fig = make_subplots(
    rows=3, cols=2,
    subplot_titles=[
        "Distribution of Policy Tenure",
        "Distribution of Age of Car",
        "Distribution of Age of Policyholder",
        "Distribution of Max Torque",
        "Distribution of Max Power",
        "Distribution of Displacement"
    ],
    vertical_spacing=0.2
)

fig.add_trace(go.Histogram(x=df['policy_tenure'], name="Policy Tenure",␣
 ↪marker_color='blue'), row=1, col=1)
fig.update_traces(marker_line_color='black',marker_line_width=0.5,row=1, col=1 )
fig.add_trace(go.Histogram(x=df['age_of_car'], name="Age of Car",␣
 ↪marker_color='orange'), row=1, col=2)
fig.update_traces(marker_line_color='black',marker_line_width=0.5,row=1, col=2 )
fig.add_trace(go.Histogram(x=df['age_of_policyholder'], name="Age of␣
 ↪Policyholder", marker_color='green'), row=2, col=1)
fig.add_trace(go.Histogram(x=df['max_torque'], name="Max Torque",␣
 ↪marker_color='red'), row=2, col=2)
fig.add_trace(go.Histogram(x=df['max_power'], name="Max Power",␣
 ↪marker_color='purple'), row=3, col=1)
fig.add_trace(go.Histogram(x=df['displacement'], name="Displacement",␣
 ↪marker_color='brown'), row=3, col=2)


fig.update_layout(
    title="Distributions of Key Features",
    title_x=0.5,
    height=1000, width=1000,
    showlegend=False,
```

```
    template="plotly_dark"
)

fig.update_xaxes(title_text="Policy Tenure (Years)", row=1, col=1)
fig.update_xaxes(title_text="Age of Car (Years)", row=1, col=2)
fig.update_xaxes(title_text="Age of Policyholder (Years)", row=2, col=1)
fig.update_xaxes(title_text="Max Torque (Nm)", row=2, col=2)
fig.update_xaxes(title_text="Max Power (HP)", row=3, col=1)
fig.update_xaxes(title_text="Displacement (Liters)", row=3, col=2)

fig.update_yaxes(title_text="Frequency")

fig.show()
```



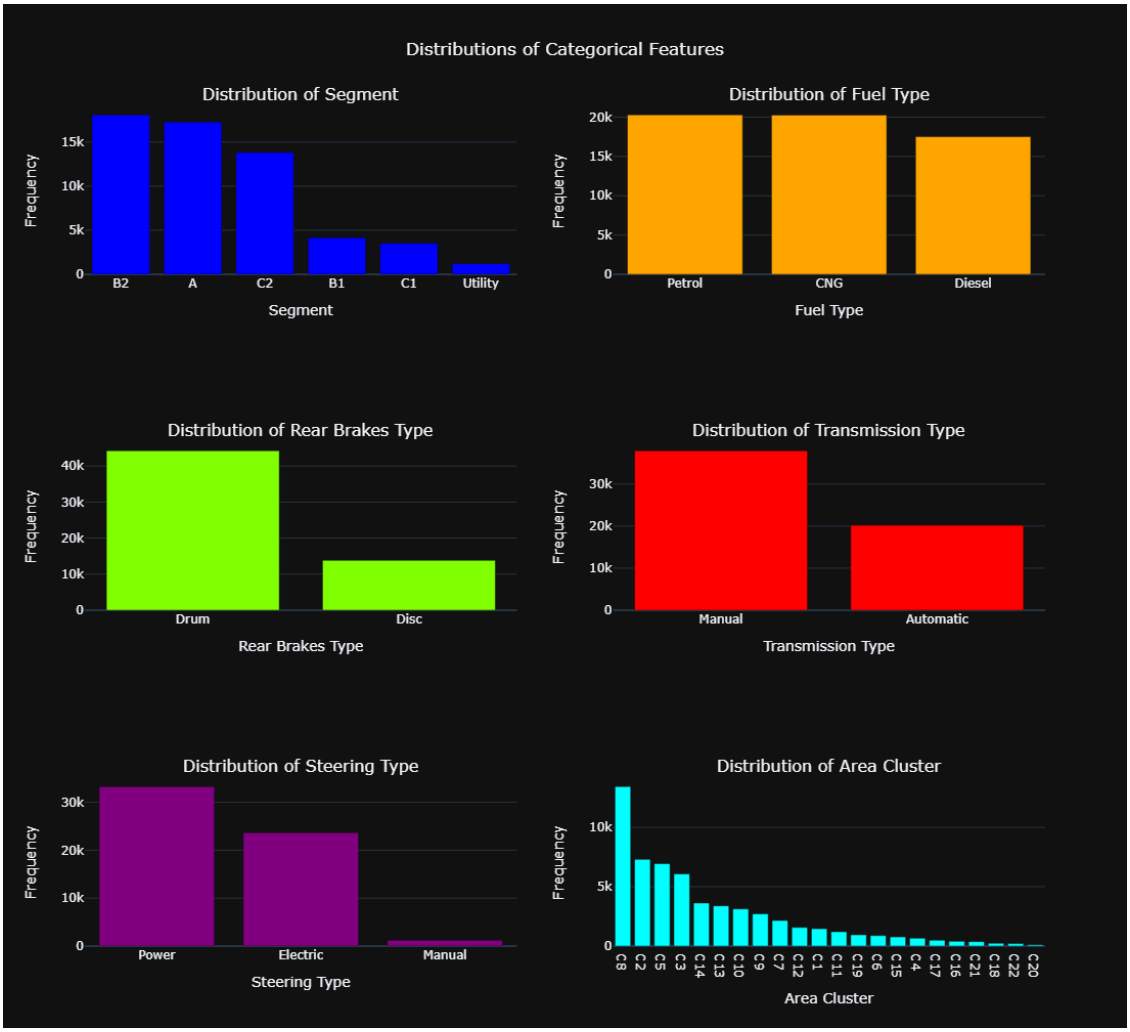
[23]:
```
categorical
```

[23]:
```
['segment',
 'fuel_type',
 'is_esc',
 'is_adjustable_steering',
 'is_tpms',
 'is_parking_sensors',
 'is_parking_camera',
 'rear_brakes_type',
 'transmission_type',
 'steering_type',
 'is_front_fog_lights',
 'is_rear_window_wiper',
 'is_rear_window_washer',
 'is_rear_window_defogger',
 'is_brake_assist',
 'is_power_door_locks',
 'is_central_locking',
 'is_power_steering',
 'is_driver_seat_height_adjustable',
 'is_day_night_rear_view_mirror',
 'is_ecw',
 'is_speed_alert']
```

[33]:
```
fig = make_subplots(
    rows=3, cols=2,
    subplot_titles=[
        "Distribution of Segment",
        "Distribution of Fuel Type",
        "Distribution of Rear Brakes Type",
        "Distribution of Transmission Type",
        "Distribution of Steering Type",
        "Distribution of Area Cluster"
    ],
    vertical_spacing=0.2
)

fig.add_trace(
    go.Bar(
        x=df['segment'].value_counts().index,
        y=df['segment'].value_counts().values,
        name="Segment",
        marker_color='blue'
    ),
    row=1, col=1
)

fig.add_trace(
```

```
    go.Bar(
        x=df['fuel_type'].value_counts().index,
        y=df['fuel_type'].value_counts().values,
        name="Fuel Type",
        marker_color='orange'
    ),
    row=1, col=2
)

fig.add_trace(
    go.Bar(
        x=df['rear_brakes_type'].value_counts().index,
        y=df['rear_brakes_type'].value_counts().values,
        name="Rear Brakes Type",
        marker_color='chartreuse'
    ),
    row=2, col=1
)

fig.add_trace(
    go.Bar(
        x=df['transmission_type'].value_counts().index,
        y=df['transmission_type'].value_counts().values,
        name="Transmission Type",
        marker_color='red'
    ),
    row=2, col=2
)

fig.add_trace(
    go.Bar(
        x=df['steering_type'].value_counts().index,
        y=df['steering_type'].value_counts().values,
        name="Steering Type",
        marker_color='purple'
    ),
    row=3, col=1
)

fig.add_trace(
    go.Bar(
        x=df['area_cluster'].value_counts().index,
        y=df['area_cluster'].value_counts().values,
        name="Area Cluster",
        marker_color='cyan'
    ),
    row=3, col=2
```

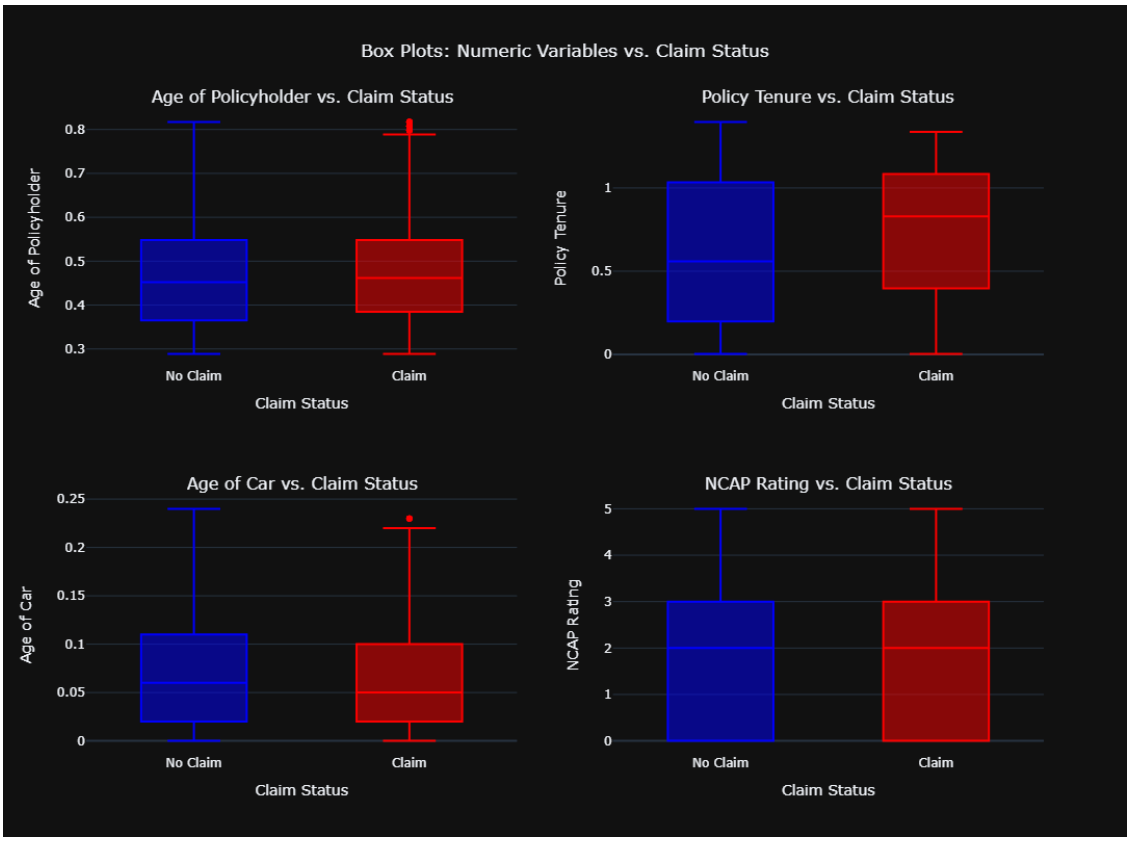
```
)

fig.update_layout(
    title="Distributions of Categorical Features",
    title_x=0.5,
    height=1000, width=1000,
    showlegend=False,
    template="plotly_dark"
)

fig.update_xaxes(title_text="Segment", row=1, col=1)
fig.update_xaxes(title_text="Fuel Type", row=1, col=2)
fig.update_xaxes(title_text="Rear Brakes Type", row=2, col=1)
fig.update_xaxes(title_text="Transmission Type", row=2, col=2)
fig.update_xaxes(title_text="Steering Type", row=3, col=1)
fig.update_xaxes(title_text="Area Cluster", row=3, col=2)

fig.update_yaxes(title_text="Frequency")
fig.show()
```



[35]:
```
fig = make_subplots(
    rows=2, cols=2,
    subplot_titles=[
        'Age of Policyholder vs. Claim Status',
        'Policy Tenure vs. Claim Status',
        'Age of Car vs. Claim Status',
        'NCAP Rating vs. Claim Status'
    ],
    vertical_spacing=0.2
)

fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 0, 'age_of_policyholder'],
        name='No Claim',
```

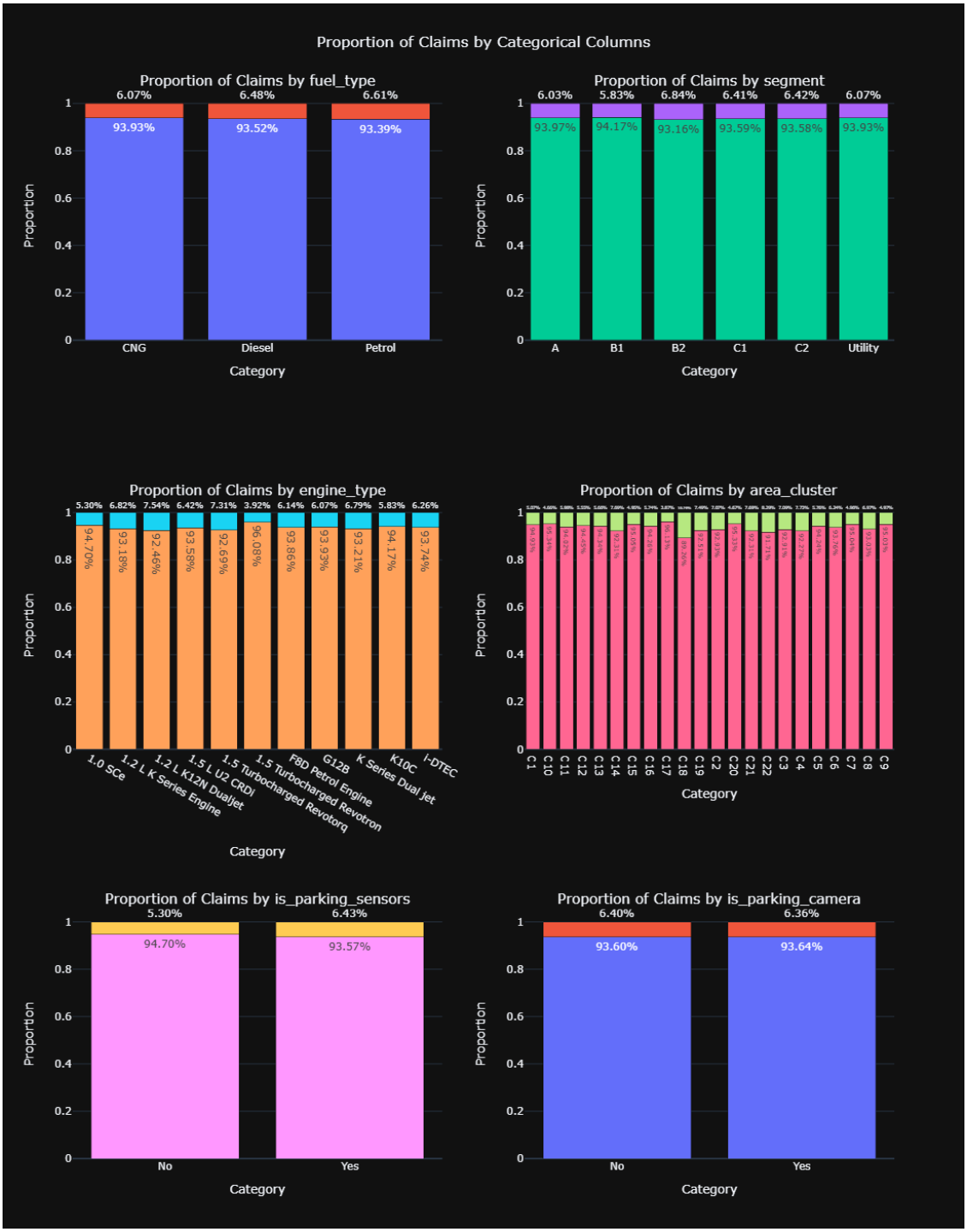
```
        marker_color='blue'
    ),
    row=1, col=1
)
fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 1, 'age_of_policyholder'],
        name='Claim',
        marker_color='red'
    ),
    row=1, col=1
)

fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 0, 'policy_tenure'],
        name='No Claim',
        marker_color='blue'
    ),
    row=1, col=2
)
fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 1, 'policy_tenure'],
        name='Claim',
        marker_color='red'
    ),
    row=1, col=2
)

fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 0, 'age_of_car'],
        name='No Claim',
        marker_color='blue'
    ),
    row=2, col=1
)
fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 1, 'age_of_car'],
        name='Claim',
        marker_color='red'
    ),
    row=2, col=1
)
```

```
fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 0, 'ncap_rating'],
        name='No Claim',
        marker_color='blue'
    ),
    row=2, col=2
)
fig.add_trace(
    go.Box(
        y=df.loc[df['is_claim'] == 1, 'ncap_rating'],
        name='Claim',
        marker_color='red'
    ),
    row=2, col=2
)

# Update layout
fig.update_layout(
    title_text="Box Plots: Numeric Variables vs. Claim Status",
    title_x=0.5,
    height=800,
    width=1000,
    showlegend=False,
    template="plotly_dark"
)

# Update axis labels
fig.update_xaxes(title_text="Claim Status", row=1, col=1)
fig.update_yaxes(title_text="Age of Policyholder", row=1, col=1)

fig.update_xaxes(title_text="Claim Status", row=1, col=2)
fig.update_yaxes(title_text="Policy Tenure", row=1, col=2)

fig.update_xaxes(title_text="Claim Status", row=2, col=1)
fig.update_yaxes(title_text="Age of Car", row=2, col=1)

fig.update_xaxes(title_text="Claim Status", row=2, col=2)
fig.update_yaxes(title_text="NCAP Rating", row=2, col=2)

# Show the figure
fig.show()
```



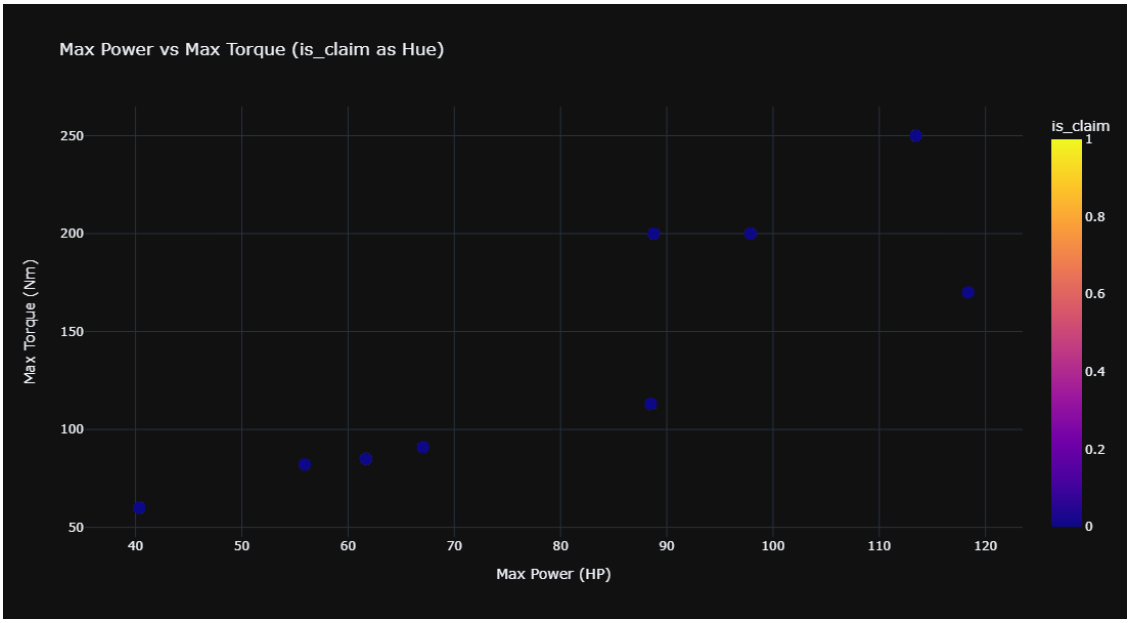
[37]:
```
categorical_columns = ['fuel_type', 'segment', 'engine_type', 'area_cluster',␣
 ↪'is_parking_sensors', 'is_parking_camera']
fig = make_subplots(
    rows=3, cols=2,
    subplot_titles=[f'Proportion of Claims by {col}' for col in␣
 ↪categorical_columns],
    vertical_spacing=0.15
)

row, col = 1, 1
for col_name in categorical_columns:
    category_data = (
        df.groupby([col_name, 'is_claim'])
        .size()
        .reset_index(name='count')
    )
    category_data['proportion'] = category_data['count'] / category_data.
 ↪groupby(col_name)['count'].transform('sum')

    for claim_status in [0, 1]:
```

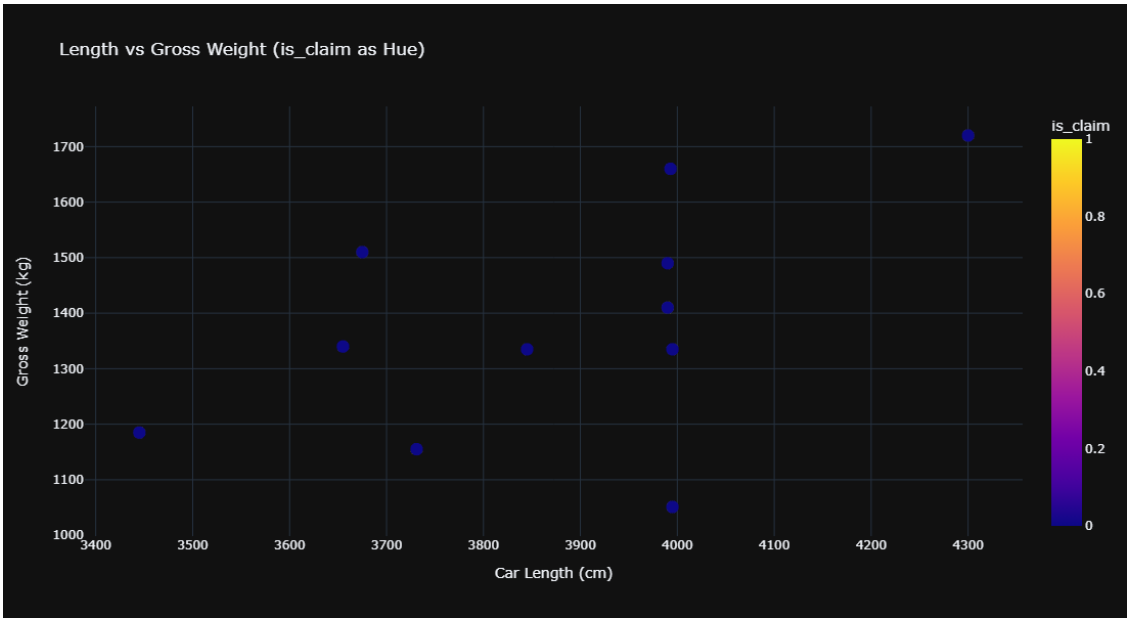
```
        filtered_data = category_data[category_data['is_claim'] == claim_status]
        fig.add_trace(
            go.Bar(
                x=filtered_data[col_name],
                y=filtered_data['proportion'],
                name=f'Claim Status {claim_status}',
                text=filtered_data['proportion'],
                texttemplate='%{text:.2%}',
                textposition='outside'
            ),
            row=row, col=col
        )

    if col == 2:
        row += 1
        col = 1
    else:
        col += 1

fig.update_layout(
    title="Proportion of Claims by Categorical Columns",
    title_x=0.5,
    height=1400,
    width=1000,
    showlegend = False,
    barmode="stack",
    template="plotly_dark"
)

fig.update_xaxes(title_text="Category")
fig.update_yaxes(title_text="Proportion")

fig.show()
```

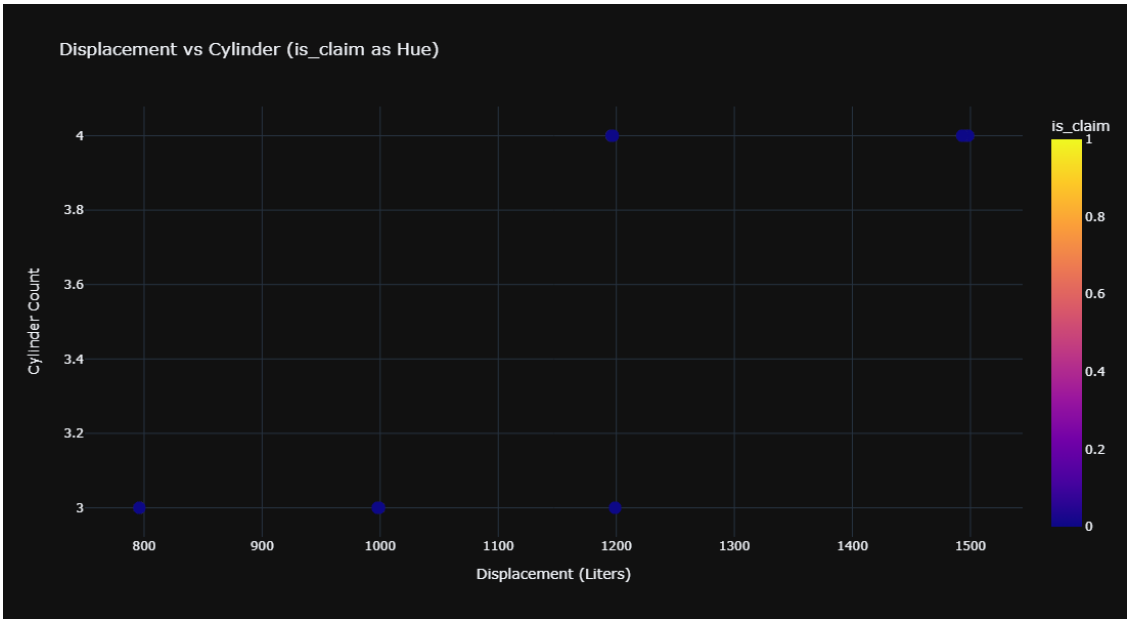


[39]:
```
fig1 = px.scatter(
    df,
    x='max_power',
    y='max_torque',
```

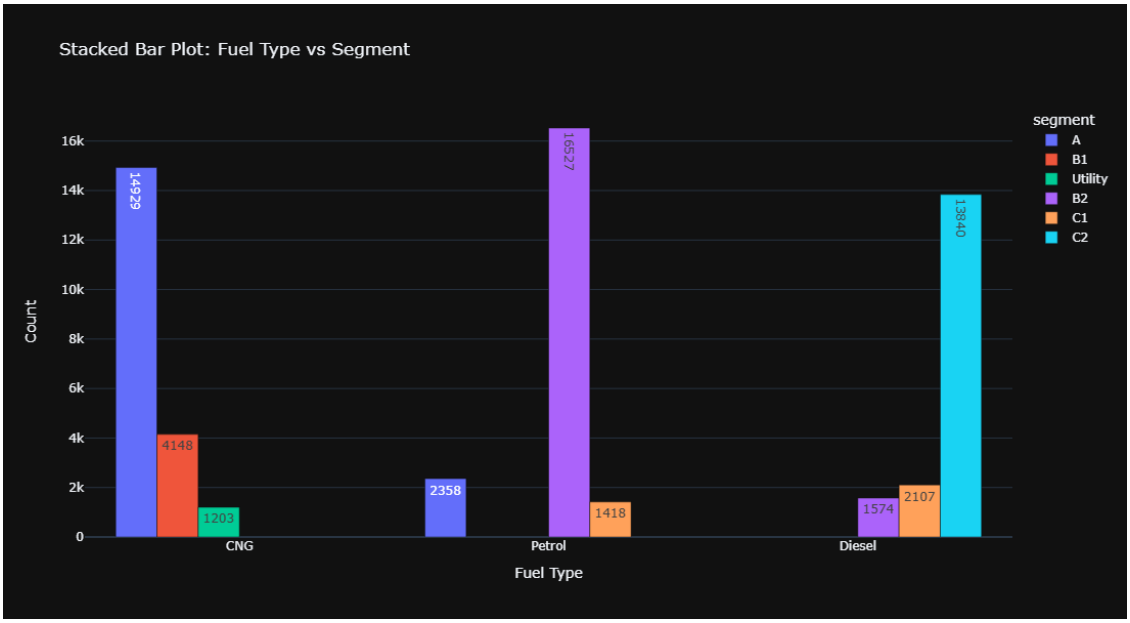
```
    color= 'is_claim',
    title='Max Power vs Max Torque (is_claim as Hue)',
    labels={'max_power': 'Max Power (HP)', 'max_torque': 'Max Torque (Nm)'}
)
fig1.update_traces(marker=dict(size=12))
fig1.update_layout(template='plotly_dark', height=600, width=800)
fig2 = px.scatter(
    df,
    x='length',
    y='gross_weight',
    color='is_claim',
    title='Length vs Gross Weight (is_claim as Hue)',
    labels={'length': 'Car Length (cm)', 'gross_weight': 'Gross Weight (kg)'}
)
fig2.update_traces(marker=dict(size=12))
fig2.update_layout(template='plotly_dark', height=600, width=800)
fig3 = px.scatter(
    df,
    x='displacement',
    y='cylinder',
    color='is_claim',
    title='Displacement vs Cylinder (is_claim as Hue)',
    labels={'displacement': 'Displacement (Liters)', 'cylinder': 'Cylinder␣
 ↪Count'}
)
fig3.update_traces(marker=dict(size=12))
fig3.update_layout(template='plotly_dark', height=600, width=800)

fig1.show()
fig2.show()
fig3.show()
```


Max Power vs Max Torque (is_claim as Hue)
Max Torque (Nm)
250
200
150
100
50
40
50
60
70
80
90
100
110
120
Max Power (HP)
is_claim
1
0.8
0.6
0.4
0.2
0

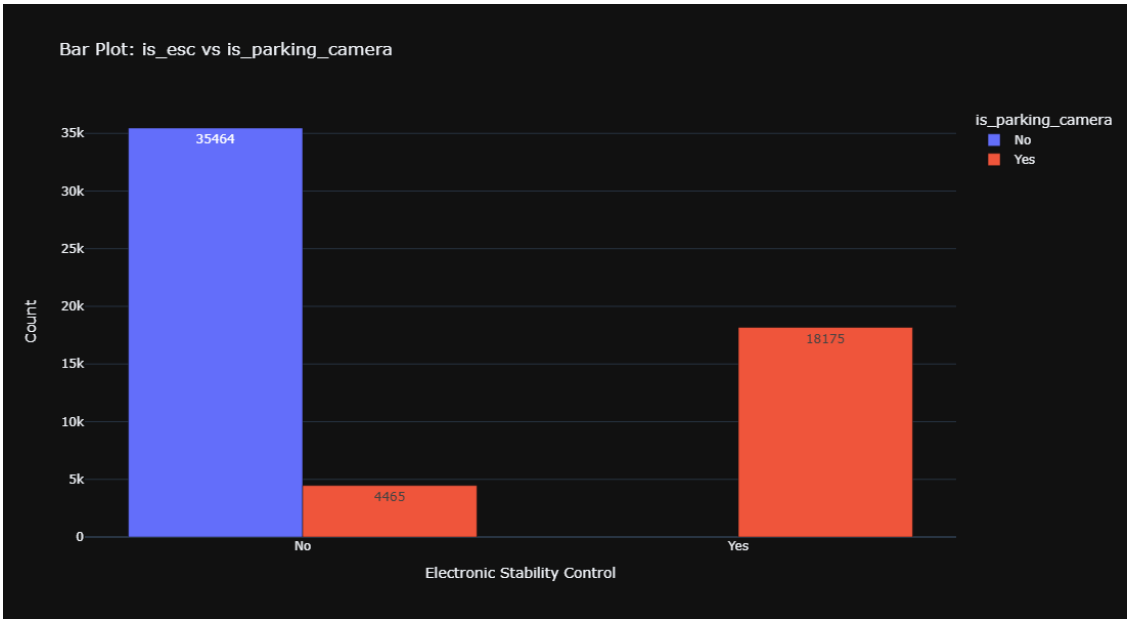

Length vs Gross Weight (is_claim as Hue)
Gross Weight (kg)
1700
1600
1500
1400
1300
1200
1100
1000
3400
3500
3600
3700
3800
3900
4000
4100
4200
4300
Car Length (cm)
is_claim
1
0.8
0.6
0.4
0.2
0



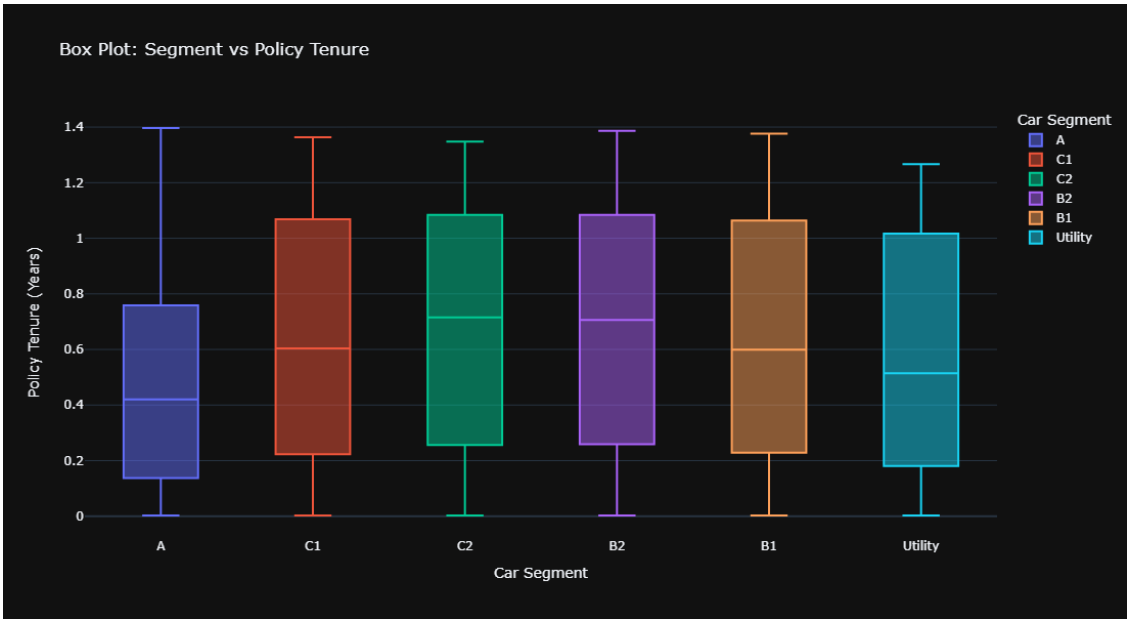
[40]:
```
grouped_data = df.groupby(['fuel_type', 'segment']).size().
  ↪reset_index(name='count')

fig = px.bar(
    grouped_data,
    x='fuel_type',
    y='count',
    color='segment',
    barmode='group',
    title="Stacked Bar Plot: Fuel Type vs Segment",
    text='count',
    height= 600,
    width= 800
)

fig.update_layout(xaxis_title="Fuel Type",␣
  ↪yaxis_title="Count",template='plotly_dark')
fig.show()
```

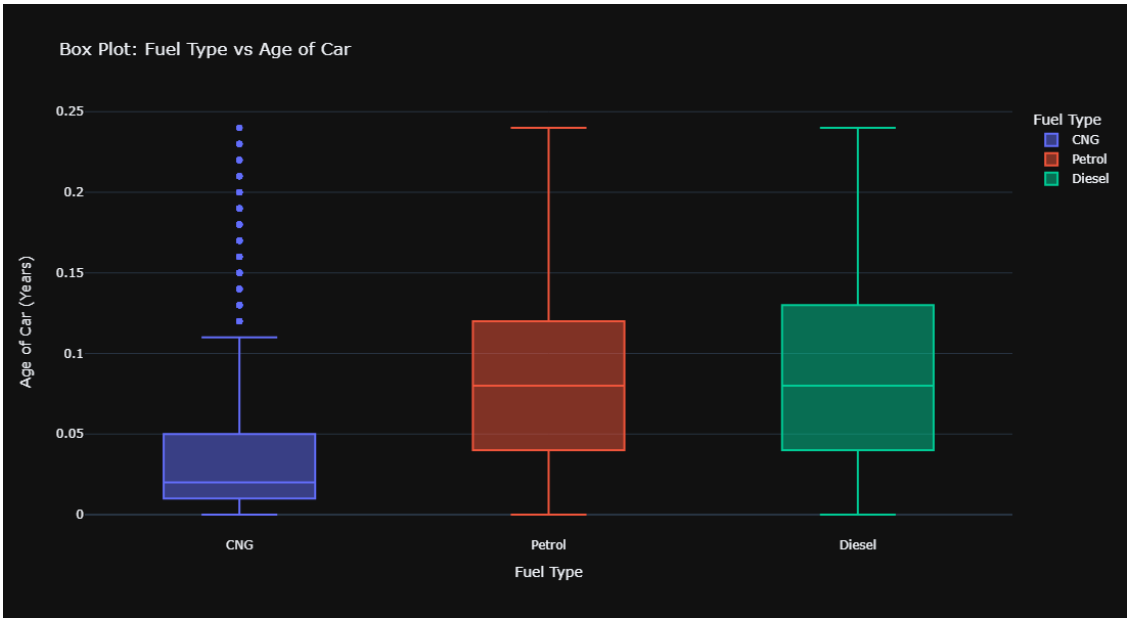


[43]:
```
grouped_data_ = df.groupby(['is_esc', 'is_parking_camera']).size().
 ↪reset_index(name='count')

fig = px.bar(
    grouped_data_,
    x='is_esc',
    y='count',
    color='is_parking_camera',
    barmode='group',
    title="Bar Plot: is_esc vs is_parking_camera",
    text='count',
    height= 600,
    width= 800
)

fig.update_layout(xaxis_title="Electronic Stability Control ",␣
 ↪yaxis_title="Count",template='plotly_dark')
fig.show()
```



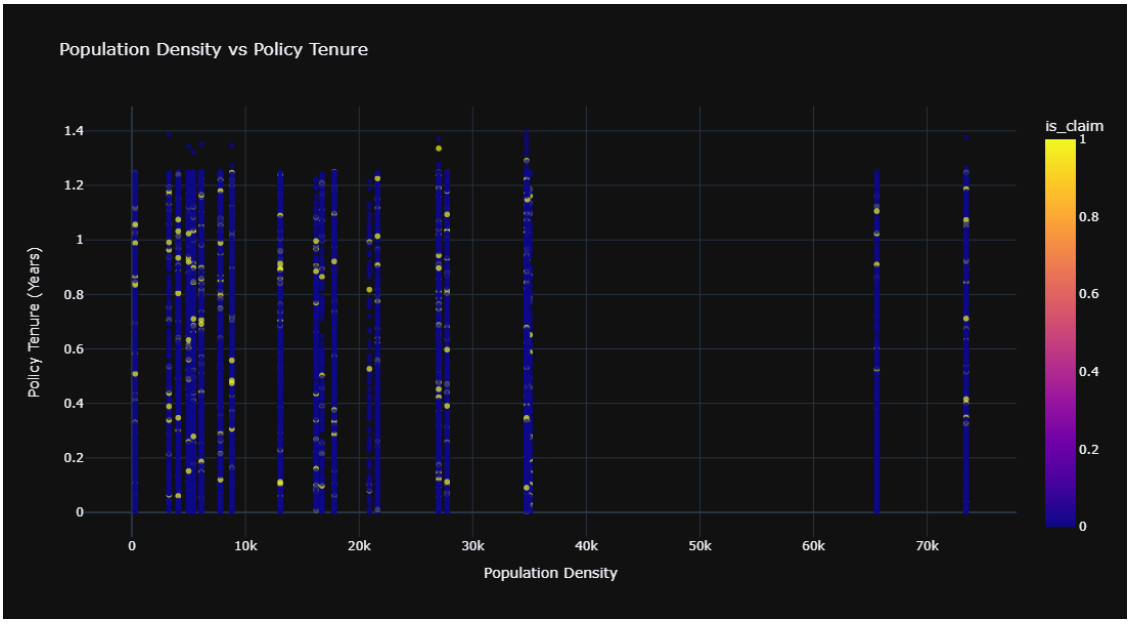
[45]:
```
# Box Plot: Segment vs. Policy Tenure
fig1 = px.box(
    df,
    x='segment',
    y='policy_tenure',
    color='segment',
    title="Box Plot: Segment vs Policy Tenure",
    labels={'policy_tenure': 'Policy Tenure (Years)', 'segment': 'Car Segment'},
    height= 600,
    width=800,
    template= "plotly_dark"
)
fig1.show()

# Box Plot: Fuel Type vs. Age of Car
fig2 = px.box(
    df,
    x='fuel_type',
    y='age_of_car',
    color='fuel_type',
    title="Box Plot: Fuel Type vs Age of Car",
    labels={'age_of_car': 'Age of Car (Years)', 'fuel_type': 'Fuel Type'},
    height= 600,
    width=800,
    template= "plotly_dark"
)
fig2.show()
```

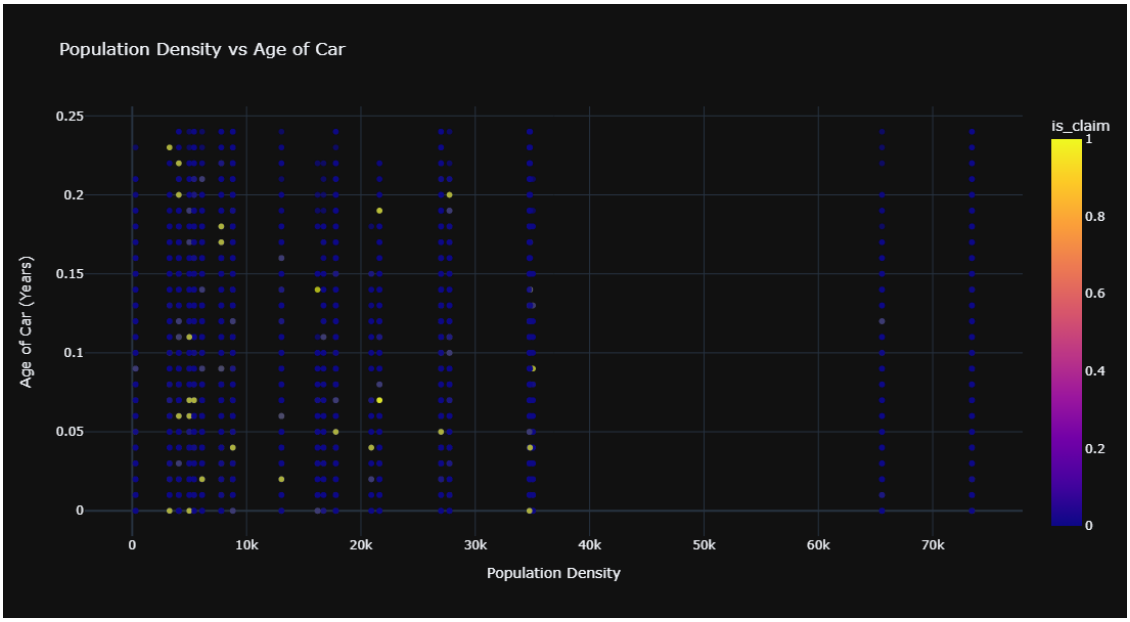


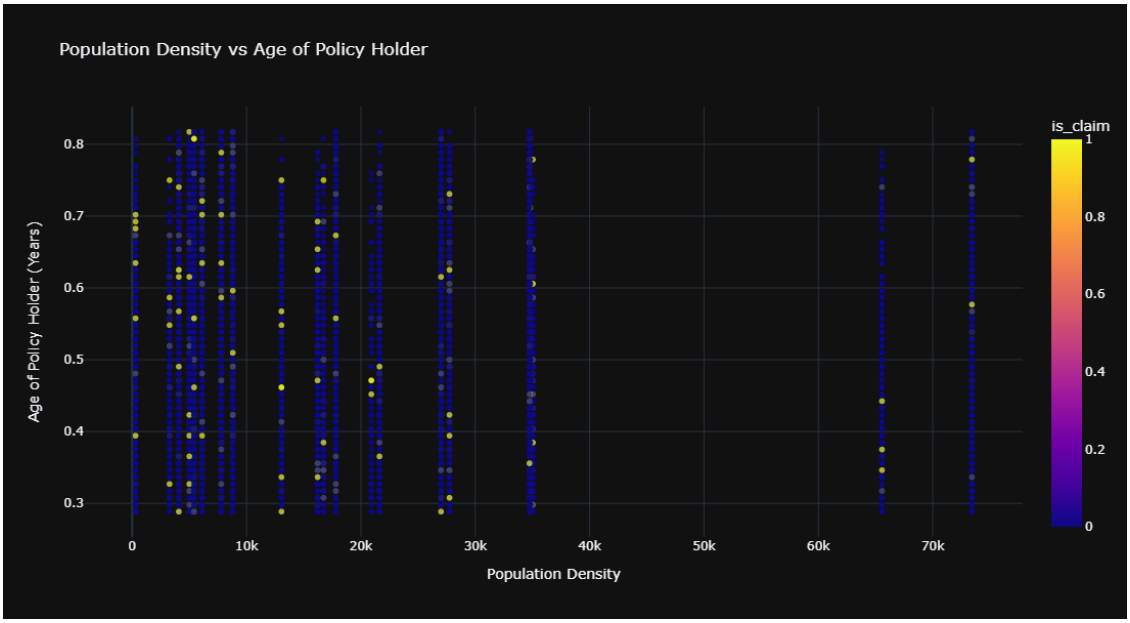


[47]:
```
fig1 = px.scatter(
    df,
    x='population_density',
    y='policy_tenure',
    color='is_claim',
    title="Population Density vs Policy Tenure",
    template= "plotly_dark",
```

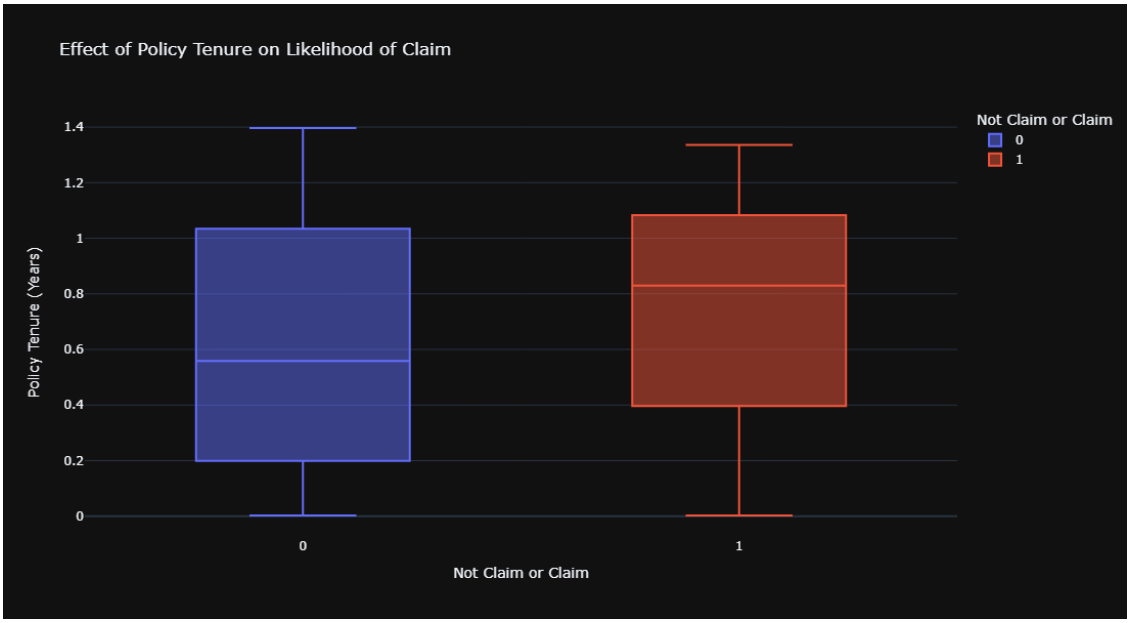
```
    labels={'population_density': 'Population Density', 'policy_tenure':␣
 ↪'Policy Tenure (Years)'},
    height= 600,
    width= 800
)
fig1.update_traces(marker=dict(opacity= 0.7))
fig1.show()

fig2 = px.scatter(
    df,
    x='population_density',
    y='age_of_car',
    color='is_claim',
    title="Population Density vs Age of Car",
    template= "plotly_dark",
    labels={'population_density': 'Population Density', 'age_of_car': 'Age of␣
 ↪Car (Years)'},
    height = 600,
    width = 800
)
fig2.update_traces(marker=dict(opacity=0.7))
fig2.show()

fig3 = px.scatter(
    df,
    x='population_density',
    y='age_of_policyholder',
    color='is_claim',
    title="Population Density vs Age of Policy Holder",
    template= "plotly_dark",
    labels={'population_density': 'Population Density', 'age_of_policyholder':␣
 ↪'Age of Policy Holder (Years)'},
    height = 600,
    width = 800
)
fig3.update_traces(marker=dict(opacity=0.7))
fig3.show()
```


Population Density vs Policy Tenure
Policy Tenure (Years)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
10k
20k
30k
40k
50k
60k
70k
Population Density
is_claim
1
0.8
0.6
0.4
0.2
0

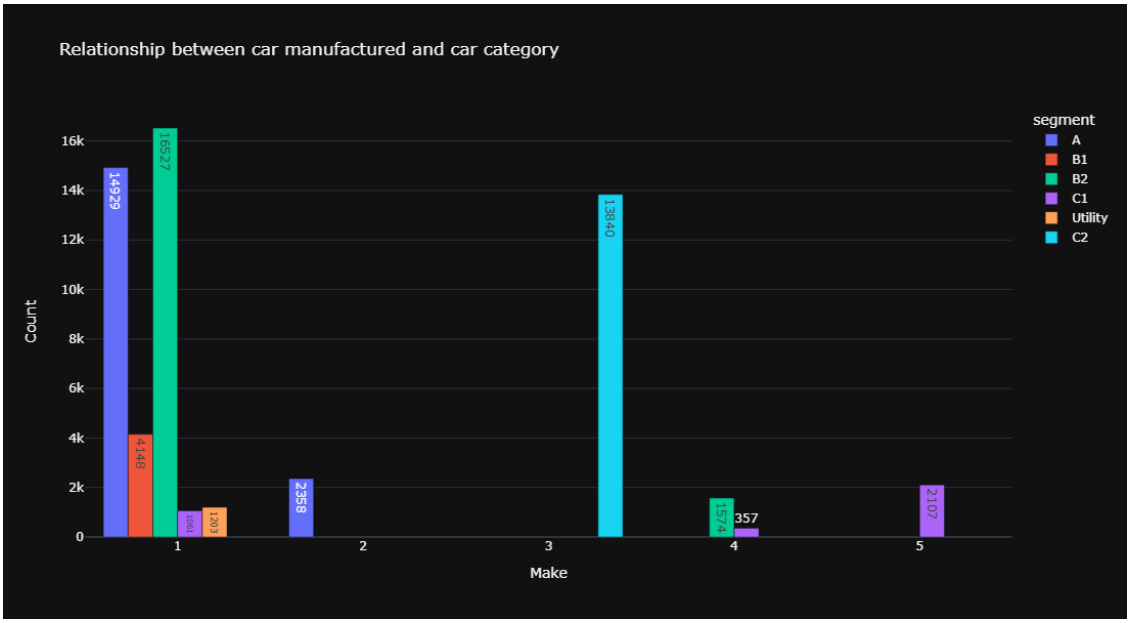

Population Density vs Age of Car
Age of Car (Years)
0.25
0.2
0.15
0.1
0.05
0
0
10k
20k
30k
40k
50k
60k
70k
Population Density
is_claim
1
0.8
0.6
0.4
0.2
0



[48]:
```
fig = px.box(
    df,
    x='is_claim',
    y='policy_tenure',
    color='is_claim',
    title="Effect of Policy Tenure on Likelihood of Claim",
    labels={'policy_tenure': 'Policy Tenure (Years)', 'is_claim': 'Not Claim or␣
 ↪Claim'},
    height= 600,
    width=800,
    template= "plotly_dark"
)
fig.show()
```

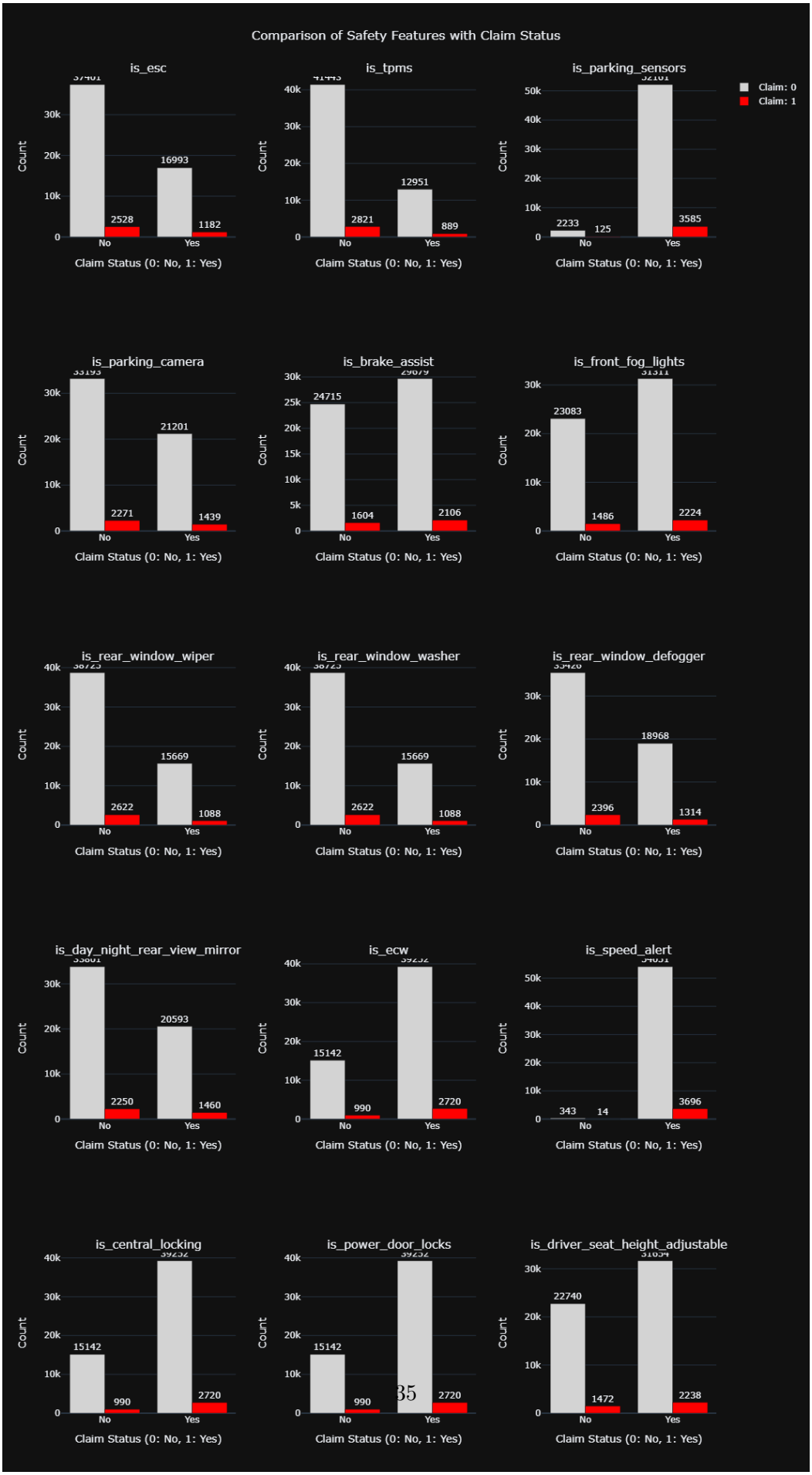


[51]:
```
#Relationship between car manufactured and car category(segment)

grouped_data = df.groupby(['make', 'segment']).size().reset_index(name='count')

fig = px.bar(
    grouped_data,
    x='make',
    y='count',
    color='segment',
    barmode='group',
    title="Relationship between car manufactured and car category",
    text='count',
    height= 600,
    width= 800
)

fig.update_layout(xaxis_title="Make",␣
 ↪yaxis_title="Count",template='plotly_dark')
fig.show()
```

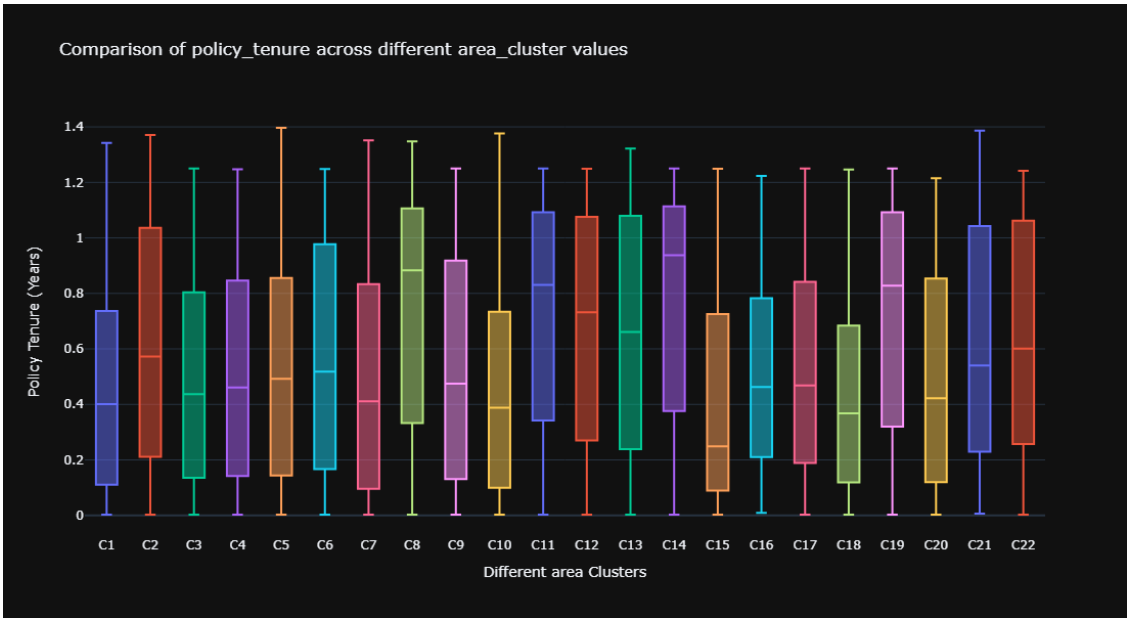


[29]:
```
# Comparison of Safety Features with Claim Status
safety_features = [
    'is_esc', 'is_tpms', 'is_parking_sensors', 'is_parking_camera',
    'is_brake_assist', 'is_front_fog_lights', 'is_rear_window_wiper',
    'is_rear_window_washer', 'is_rear_window_defogger',
    'is_day_night_rear_view_mirror', 'is_ecw', 'is_speed_alert',
    'is_central_locking', 'is_power_door_locks',␣
 ↪'is_driver_seat_height_adjustable'
]

rows = (len(safety_features) + 2) // 3
fig = make_subplots(
    rows=rows, cols=3,
    subplot_titles=safety_features,
    vertical_spacing=0.1,
    horizontal_spacing=0.1
)

row, col = 1, 1
for feature in safety_features:
    grouped = df.groupby([feature, 'is_claim']).size().reset_index(name='count')
    for claim_status in [0, 1]:
        subset = grouped[grouped['is_claim'] == claim_status]
        fig.add_trace(
            go.Bar(
                x=subset[feature],
                y=subset['count'],
```

```
                name=f"Claim: {claim_status}",
                text=subset['count'],
                textposition="outside",
                marker_color='red' if claim_status == 1 else 'lightgray',
                showlegend=(row == 1 and col == 1)  # Show legend only once
            ),
            row=row, col=col
        )


    col += 1
    if col > 3:
        col = 1
        row += 1


fig.update_layout(
    height=400 * rows,
    width=800,
    title_text="Comparison of Safety Features with Claim Status",
    title_x=0.5,
    barmode="group",
    template="plotly_dark",
    showlegend=True
)


fig.update_xaxes(title_text="Claim Status (0: No, 1: Yes)")
fig.update_yaxes(title_text="Count")

fig.show()
```

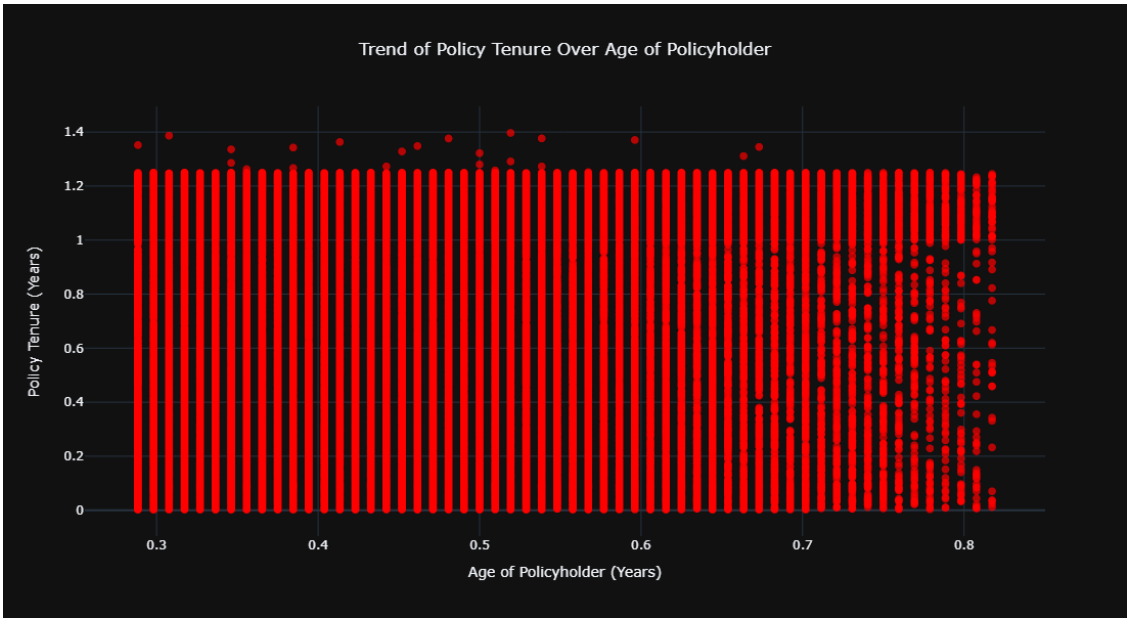

Comparison of Safety Features with Claim Status
Claim: 0
Claim: 1
is_esc
2528
16993
1182
is_tpms
2821
12951
889
is_parking_sensors
2233
125
3585
is_parking_camera
2271
21201
1439
is_brake_assist
24715
1604
2106
is_front_fog_lights
23083
1486
2224
is_rear_window_wiper
2622
15669
1088
is_rear_window_washer
2622
15669
1088
is_rear_window_defogger
2396
18968
1314
is_day_night_rear_view_mirror
2250
20593
1460
is_ecw
15142
990
2720
is_speed_alert
343
14
3696
is_central_locking
15142
990
2720
is_power_door_locks
15142
990
2720
is_driver_seat_height_adjustable
22740
1472
2238
Count
Claim Status (0: No, 1: Yes)
No
Yes

[55]:
```
# Comparison of policy_tenure across different area_cluster values
fig = px.box(
    df,
    x='area_cluster',
    y='policy_tenure',
    color='area_cluster',
    title="Comparison of policy_tenure across different area_cluster values",
    labels={'policy_tenure': 'Policy Tenure (Years)', 'area_cluster':␣
 ↪'Different area Clusters'},
    height= 600,
    width=1000,
    template= "plotly_dark"
)
fig.update_layout(showlegend = False)
fig.show()
```



[57]:
```
#Trend of Policy Tenure Over the Years

fig = px.scatter(
    df,
    x='age_of_policyholder',
    y='policy_tenure',
    title="Trend of Policy Tenure Over Age of Policyholder",
```

```
    labels={'age_of_policyholder': 'Age of Policyholder (Years)',␣
 ↪'policy_tenure': 'Policy Tenure (Years)'},
    template="plotly_dark",
    color_discrete_sequence=["red"],
    height=600,
    width=800
)

fig.update_traces(
    mode='markers',
    marker=dict(size=8, opacity=0.7)
)
fig.update_layout(
    title_x=0.5,
    xaxis_title="Age of Policyholder (Years)",
    yaxis_title="Policy Tenure (Years)"
)

fig.show()
```



[31]:
```
fig = px.scatter_3d(df,
                    x='age_of_policyholder',
                    y='age_of_car',
                    z='displacement',
                    color='is_claim',
                    title="3D Scatter Plot of Insurance Data",
                    labels={'age_of_policyholder': 'Age of Policyholder',
```

```
                            'age_of_car': 'Age of Car',
                            'displacement': 'Car Displacement'},
                     template= "plotly_dark",
                     height= 800,
                     width = 1000)

fig.show()
```

[33]:
```
fig = px.scatter_3d(df,
                     x='length',
                     y='width',
                     z='height',
                     color='is_claim',
                     title='Relationship between Length, Width, and Height of␣
 ↪the Car',
                     template='plotly_dark',
                     height= 800,
                     width = 1000)
fig.show()
```

1.8 Feature Engineering

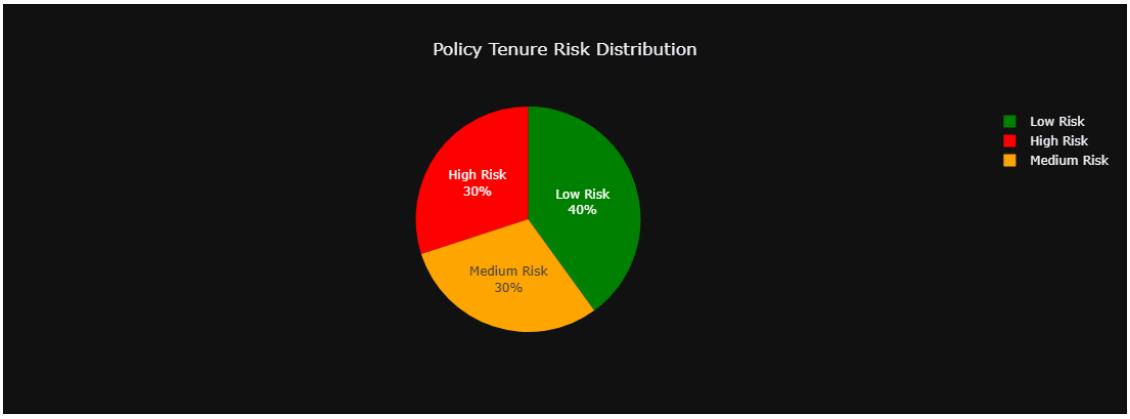
Policy Tenure Risk:

- Longer policy tenure might indicate more experienced or loyal policyholders, potentially lowering risk.
- Feature: Categorize policy_tenure into risk levels.

[35]:
```
df['tenure_risk'] = pd.cut(df['policy_tenure'],
                           bins=[0.0, 0.270744,0.768856, 1.40],
                           labels=['High Risk', 'Medium Risk', 'Low Risk'])
```

[103]:
```
tenure_risk_counts = df['tenure_risk'].value_counts()
risk_data = pd.DataFrame({
    'Tenure Risk': tenure_risk_counts.index,
    'Count': tenure_risk_counts.values
})

fig = px.pie(
    risk_data,
    names='Tenure Risk',
    values='Count',
    title='Policy Tenure Risk Distribution',
    color='Tenure Risk',
    height= 400,
    width= 600,
    color_discrete_map={
        'High Risk': 'red',
```

```
        'Medium Risk': 'orange',
        'Low Risk': 'green'
    }
)
fig.update_traces(textposition='inside', textinfo='percent+label')
fig.update_layout(template='plotly_dark',title_x=0.5)
fig.show()
```



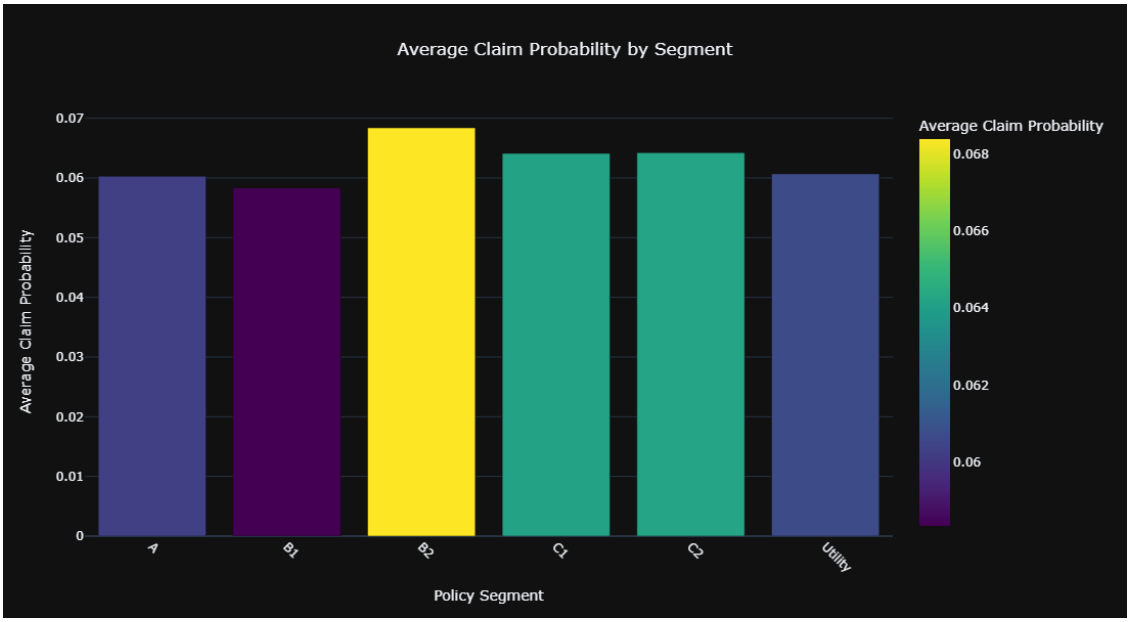
Segment Risk:

- Different car category may have varying claim probabilities based on customer behavior.
- Feature: Calculate segment-wise claim rates.

[39]:
```
segment_risk = df.groupby('segment')['is_claim'].mean()
df['segment_risk'] = df['segment'].map(segment_risk)
```

[111]:
```
segment_risk = df.groupby('segment')['is_claim'].mean()
segment_risk_df = segment_risk.reset_index()
fig = px.bar(
    segment_risk_df,
    x='segment',
    y='is_claim',
    title='Average Claim Probability by Segment',
    labels={'is_claim': 'Average Claim Probability', 'segment': 'Policy␣
 ↪Segment'},
    color='is_claim',
    color_continuous_scale='Viridis',
    height = 600,
    width = 800
)
fig.update_layout(
    template='plotly_dark',
    title_x = 0.5,
```

```
    xaxis_title='Policy Segment',
    yaxis_title='Average Claim Probability',
    xaxis_tickangle=45
)
fig.show()
```



Age of Car:

- Older cars might have a higher likelihood of breakdowns or accidents.

[47]:
```
df['car_age_risk'] = pd.cut(df['age_of_car'],
                            bins=[-1, 0.03, 0.08, 1.01],
                            labels=['Low Risk', 'Medium Risk', 'High Risk'])
```

Safety Features Score:

- Combine binary indicators of safety features (e.g., is_esc, is_tpms) into a composite score.
- Feature: Sum of safety features.

[49]:
```
boolean_columns = [col for col in df.columns if df[col].isin(['Yes', 'No']).
↪all()]
for col in boolean_columns:
    df[col] = df[col].map({'Yes':1,'No':0})
```

[51]:
```
safety_features
```

[51]:
```
['is_esc',
 'is_tpms',
 'is_parking_sensors',
 'is_parking_camera',
 'is_brake_assist',
 'is_front_fog_lights',
 'is_rear_window_wiper',
 'is_rear_window_washer',
 'is_rear_window_defogger',
 'is_day_night_rear_view_mirror',
 'is_ecw',
 'is_speed_alert',
 'is_central_locking',
 'is_power_door_locks',
 'is_driver_seat_height_adjustable']
```

[53]:
```
df['safety_score'] = df[safety_features].sum(axis=1)
```

Engine Characteristics:

- Cars with higher power (max_power, max_torque) or displacement may pose higher risks due to aggressive driving.
- Feature: Normalize engine-related metrics

[56]:
```
df['engine_risk'] = (df['max_power'].astype(float) + df['max_torque'].
↪astype(float)) / df['displacement']
```

Vehicle Size Risk:

- Larger vehicles (e.g., SUVs) might cause higher damage but may have better safety for drivers.
- Feature: Use dimensions like length, width, and height.

[59]:
```
df['size_risk'] = df['length'] * df['width'] * df['height']
```

[61]:
```
fig = make_subplots(
    rows=2, cols=2,
    subplot_titles=(
        "Car Age Risk Distribution",
        "Safety Score Distribution",
        "Engine Risk vs Car Age Risk",
        "Size Risk Distribution"
    )
)

car_age_risk_counts = df['car_age_risk'].value_counts()
fig.add_trace(
    go.Bar(
        x=car_age_risk_counts.index,
```

```
        y=car_age_risk_counts.values,
        marker_color=['red', 'green', 'orange'],
        name="Car Age Risk"
    ),
    row=1, col=1
)


fig.add_trace(
    go.Histogram(x=df['safety_score'], nbinsx=10, marker_color='blue',␣
 ↪name="Safety Score"),
    row=1, col=2
)
fig.update_traces(marker_line_color='black',
                  marker_line_width=1.5)

fig.add_trace(
    go.Box(
        x=df['car_age_risk'],
        y=df['engine_risk'],
        boxmean='sd',
        marker_color='purple',
        name="Engine Risk"
    ),
    row=2, col=1
)

fig.add_trace(
    go.Histogram(x=df['size_risk'], nbinsx=20, marker_color='brown', name="Size␣
 ↪Risk"),
    row=2, col=2
)
fig.update_traces(marker_line_color='black',
                  marker_line_width=1.5)
fig.update_layout(
    height=800, width=900,
    title_text="Vehicle Characteristics",
    title_x = 0.5,
    template="plotly_dark",
    showlegend=False
)
fig.show()
```

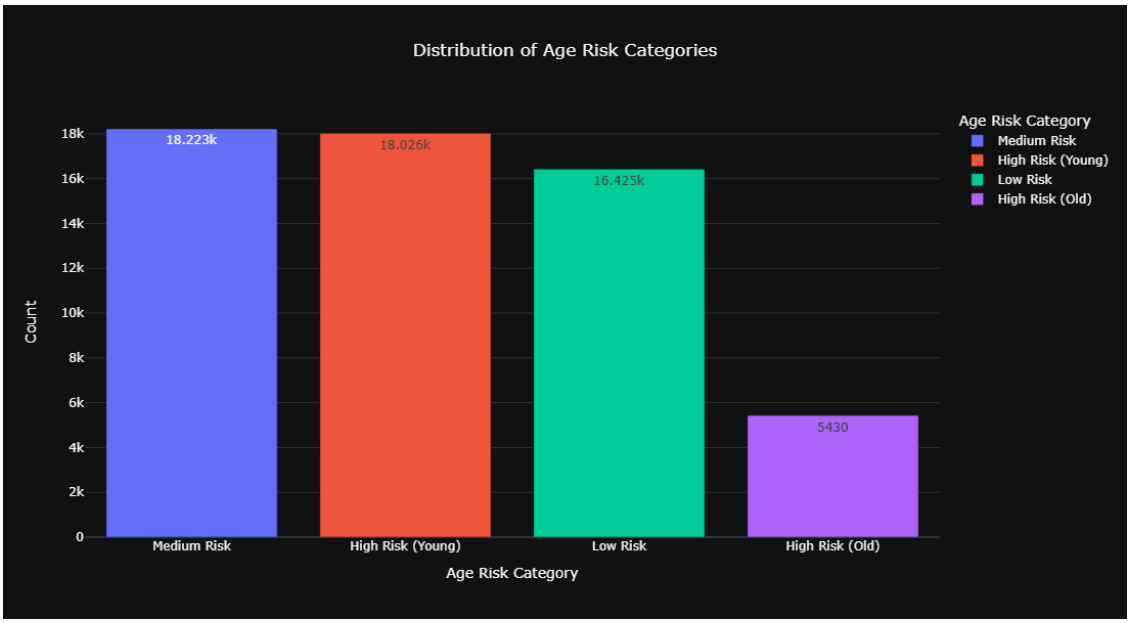
Age of Policyholder:

- Younger drivers (e.g., <25) or older drivers (e.g., >60) may have higher accident probabilities.
- Feature: Categorize age groups.

[65]:
```
bins = [
    df['age_of_policyholder'].quantile(0.0),  # Minimum
    df['age_of_policyholder'].quantile(0.3),  # Up to 30th percentile
    df['age_of_policyholder'].quantile(0.6),  # Up to 60th percentile
    df['age_of_policyholder'].quantile(0.9),  # Up to 90th percentile
    df['age_of_policyholder'].quantile(1.0)   # Maximum
]
labels = ['High Risk (Young)', 'Medium Risk', 'Low Risk', 'High Risk (Old)']
df['age_risk'] = pd.cut(df['age_of_policyholder'], bins=bins, labels=labels,␣
 ↪include_lowest=True)
```

[119]:
```
age_risk_counts = df['age_risk'].value_counts().reset_index()
fig = px.bar(
    age_risk_counts,
    x='age_risk',
    y='count',
    text_auto= True,
```

```
    color='age_risk',
    labels={'age_risk': 'Age Risk Category', 'count': 'Count'},
    title='Distribution of Age Risk Categories',
    height=600,
    width = 800,
    template="plotly_dark"
)
fig.update_layout(title_x=0.5)
fig.show()
```



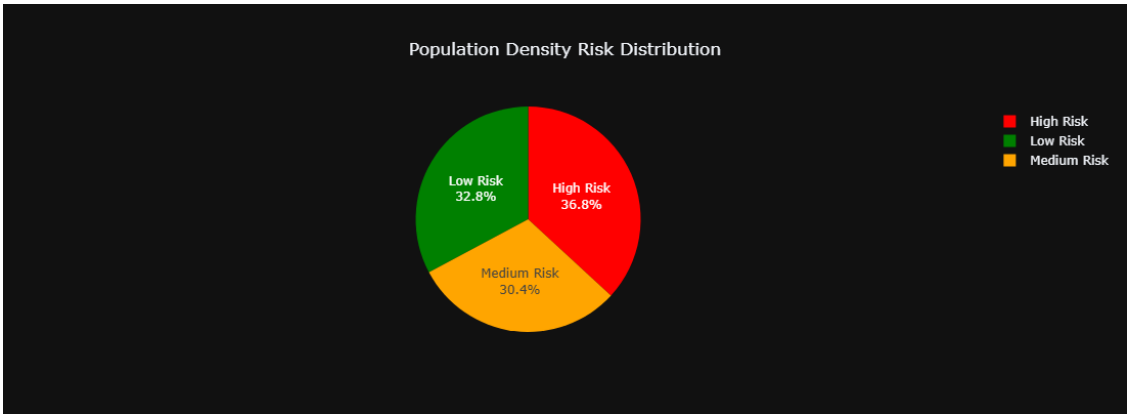
Population Density:

- High-density areas may have a higher likelihood of accidents.
- Feature: Bucket population_density.

[73]:
```
df['density_risk'] = pd.cut(df['population_density'], bins=[289.0, 7788.0,␣
↪17804.0, 73431.0], labels=['Low Risk', 'Medium Risk', 'High Risk'])
```

[109]:
```
density_risk_counts = df['density_risk'].value_counts()
density_risk_data = pd.DataFrame({
    'Density Risk': density_risk_counts.index,
    'Count': density_risk_counts.values
})
fig = px.pie(
    density_risk_data,
    names='Density Risk',
    values='Count',
```

```
    title='Population Density Risk Distribution',
    color='Density Risk',
    height= 400,
    width= 600,
    color_discrete_map={
        'Low Risk': 'green',
        'Medium Risk': 'orange',
        'High Risk': 'red'
    }
)
fig.update_traces(textposition='inside', textinfo='percent+label')
fig.update_layout(template='plotly_dark',title_x=0.5)
fig.show()
```



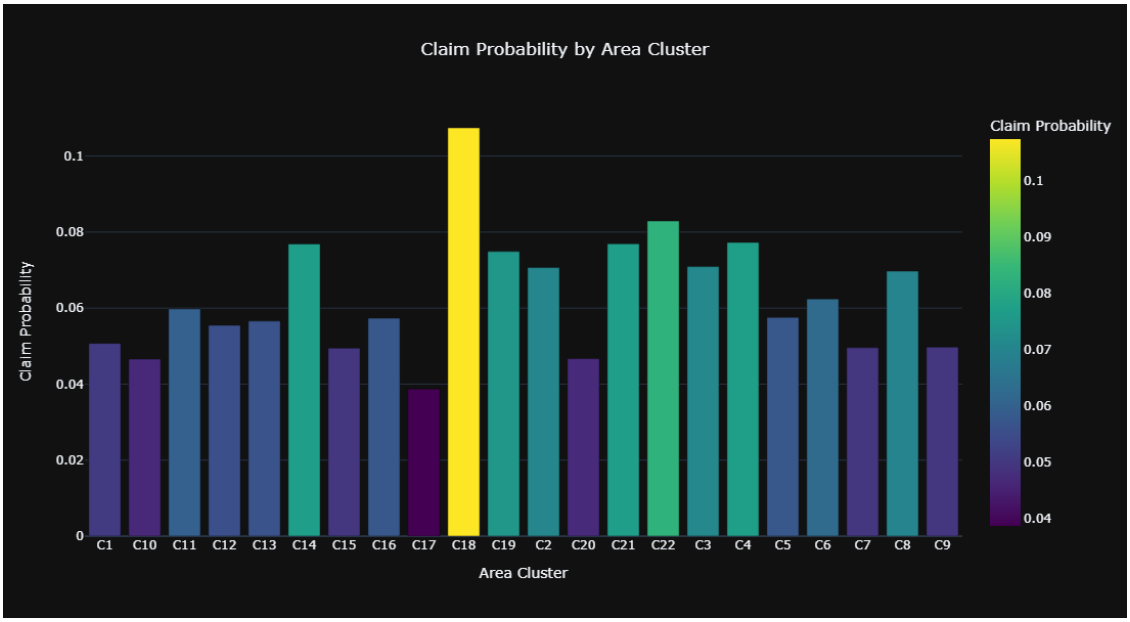
Area Cluster Risk:

- Identify clusters with high claim frequencies.
- Feature: Map average claim rates for area_cluster.

[79]:
```
area_cluster_risk = df.groupby('area_cluster')['is_claim'].mean()
df['area_cluster_risk'] = df['area_cluster'].map(area_cluster_risk)
```

[113]:
```
area_cluster_risk = df.groupby('area_cluster')['is_claim'].mean()
risk_data = pd.DataFrame({
    'Area Cluster': area_cluster_risk.index,
    'Risk (Claim Probability)': area_cluster_risk.values
})
fig = px.bar(
    risk_data,
    x='Area Cluster',
    y='Risk (Claim Probability)',
    title='Claim Probability by Area Cluster',
    labels={'Risk (Claim Probability)': 'Claim Probability'},
```

```
    color='Risk (Claim Probability)',
    height = 600,
    width = 800,
    color_continuous_scale='Viridis'
)

fig.update_layout(
    template='plotly_dark',
    title_x = 0.5,
    xaxis_title='Area Cluster',
    yaxis_title='Claim Probability'
)
fig.show()
```



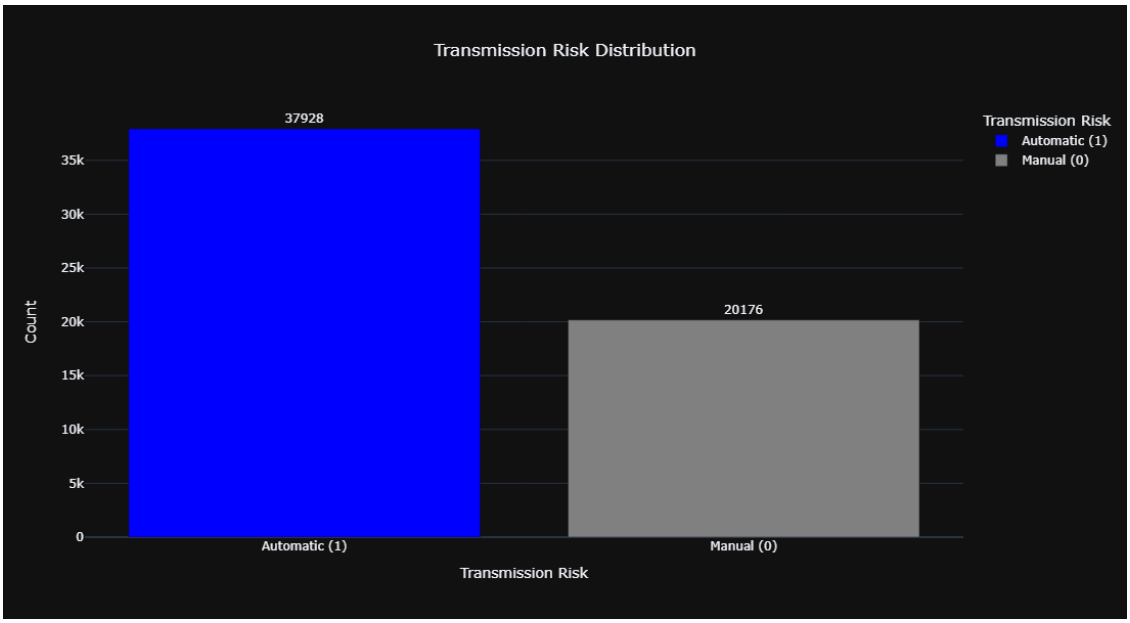
Transmission Type:

- Manual transmission might be associated with higher driver control.
- Feature: Encode transmission_type as binary.

[85]:
```
df['transmission_risk'] = df['transmission_type'].apply(lambda x: 1 if x ==␣
↪'Automatic' else 0)
```

[115]:
```
transmission_risk_counts = df['transmission_risk'].value_counts()
transmission_data = pd.DataFrame({
    'Transmission Risk': ['Automatic (1)', 'Manual (0)'],
    'Count': transmission_risk_counts.values
})
```

```
fig = px.bar(
    transmission_data,
    x='Transmission Risk',
    y='Count',
    title='Transmission Risk Distribution',
    text='Count',
    color='Transmission Risk',
    color_discrete_map={'Automatic (1)': 'blue', 'Manual (0)': 'gray'},
    height= 600,
    width= 800
)
fig.update_traces(textposition='outside')
fig.update_layout(
    xaxis_title='Transmission Risk',
    title_x = 0.5,
    yaxis_title='Count',
    template='plotly_dark'
)
fig.show()
```

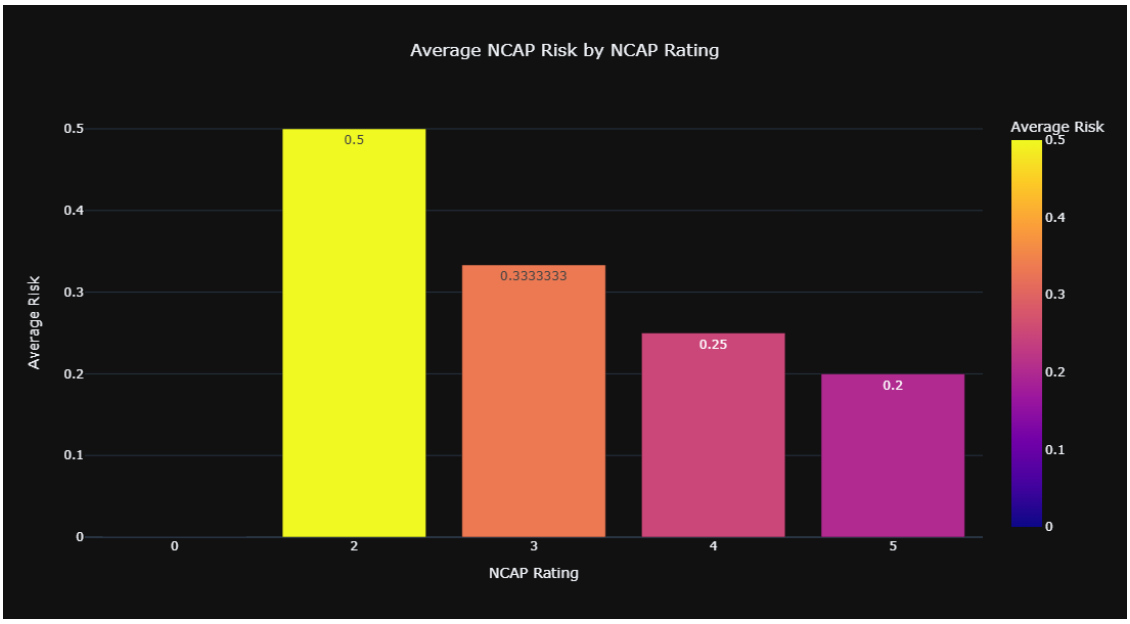


NCAP Rating:

- Higher NCAP ratings imply safer cars.
- Feature: Invert NCAP score for risk assessment.

[89]:
```
df['ncap_risk'] = df['ncap_rating'].apply(lambda x: 1 / x if x > 0 else 0)
```

[117]:
```
ncap_risk_summary = df.groupby('ncap_rating')['ncap_risk'].mean().reset_index()

fig = px.bar(
    ncap_risk_summary,
    x='ncap_rating',
    y='ncap_risk',
    color= 'ncap_risk',
    text_auto= True,
    labels={'ncap_rating': 'NCAP Rating', 'ncap_risk': 'Average Risk'},
    title='Average NCAP Risk by NCAP Rating',
    height = 600,
    width=800,
    template="plotly_dark"
)

fig.update_layout(xaxis=dict(type='category'), title_x=0.5)
fig.show()
```



Policy Risk Score:

- Combine all risk-related features into a single score.
- Feature: Weighted sum of critical features

`safety_score * 0.3`: Safety features often have a significant impact on risk, so they are assigned the highest weight (30%).

`area_cluster_risk * 0.2`: The area where the policyholder lives may moderately influence risk, as some regions may have higher accident or claim rates.

`engine_risk * 0.2`: Engine characteristics impact vehicle performance, which is a medium contributor to risk.

`density_risk.cat.codes * 0.2`: Population density reflects environmental risk factors. The categorical values are encoded numerically to allow mathematical operations.

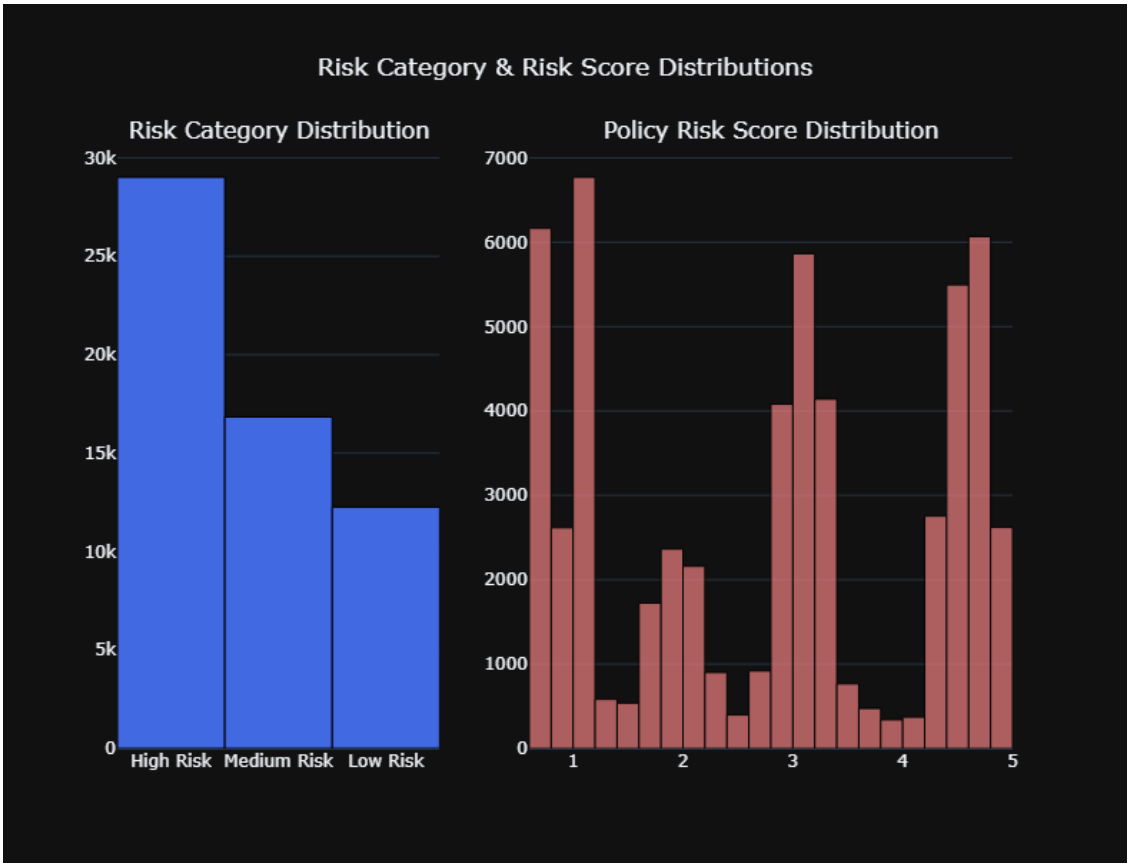
`car_age_risk.cat.codes * 0.1`: Older cars are generally considered riskier, but their contribution to risk is relatively small compared to the other factors, so it has the smallest weight (10%).

[95]:
```
df['policy_risk_score'] = (
    df['safety_score'] * 0.3 +
    df['area_cluster_risk'] * 0.2 +
    df['engine_risk'] * 0.2 +
    df['density_risk'].cat.codes * 0.2 +
    df['car_age_risk'].cat.codes * 0.1
)
```

[97]:
```
df['risk_category'] = pd.cut(df['policy_risk_score'], bins=[0.334424, 1.036729,␣
↪2.949044, 4.870175], labels=['Low Risk', 'Medium Risk', 'High Risk'])
```

[99]:
```
risk_category_counts = df['risk_category'].value_counts()

fig = make_subplots(
    rows=1, cols=2,
    subplot_titles=('Risk Category Distribution', 'Policy Risk Score␣
↪Distribution'),
    column_widths=[0.4, 0.6]
)

fig.add_trace(
    go.Bar(
        x=risk_category_counts.index,
        y=risk_category_counts.values,
        name='Risk Category Distribution',
        marker=dict(color='royalblue')
    ),
    row=1, col=1
)
fig.update_traces(marker_line_color='black',
                  marker_line_width=1)
fig.add_trace(
    go.Histogram(
        x=df['policy_risk_score'],
        name='Policy Risk Score Distribution',
        marker=dict(color='lightcoral', opacity=0.7),
        nbinsx=30
    ),
    row=1, col=2
```

```
)
fig.update_traces(marker_line_color='black',
                  marker_line_width=1)
fig.update_layout(
    title_text='Risk Category & Risk Score Distributions',
    title_x = 0.5,
    showlegend=False,
    height=600,
    width = 900,
    template="plotly_dark"
)

fig.show()
```



[72]:
```
risk_features = ['policy_tenure', 'age_of_car', 'age_of_policyholder',␣
 ↪'population_density',
                 'safety_score', 'engine_risk', 'area_cluster_risk',␣
 ↪'ncap_risk', 'policy_risk_score']

correlation_matrix = df[risk_features + ['is_claim']].corr()
```

```
fig = go.Figure(data=go.Heatmap(
    z=correlation_matrix,
    x=risk_features + ['is_claim'],
    y=risk_features + ['is_claim'],
    colorscale='viridis',
    colorbar=dict(title='Count'),
    xgap = 1,
    ygap = 1,
    text=correlation_matrix,
    texttemplate='%{text:.3f}',
    textfont = dict(size=10)
))

fig.update_layout(
    title='Correlation Matrix',
    xaxis=dict(
        title='Risk Features',
        tickfont = dict(size=10)
    ),
    yaxis=dict(
        title='Risk Features',
        tickfont = dict(size=10)
    ),
    height= 600,
    width= 800,
    template = "plotly_dark"
)

fig.show()
```

[74]:
```
threshold = 0.9
high_corr = np.where((np.abs(correlation_matrix) >= threshold) &␣
 ↪(correlation_matrix != 1))

correlated_pairs = [(risk_features [i], risk_features[j]) for i, j in␣
 ↪zip(*high_corr) if i < j]

print("Highly correlated pairs:")
print(correlated_pairs)

columns_to_drop = set()
for col1, col2 in correlated_pairs:
    columns_to_drop.add(col2)

print("Columns to drop:", columns_to_drop)
```

```
Highly correlated pairs:
[('safety_score', 'policy_risk_score')]
Columns to drop: {'policy_risk_score'}
```

[76]:
```
# df = df.drop(columns=columns_to_drop)
```

[78]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
numerical_pipeline = Pipeline(steps=[
    ('scaler', StandardScaler())
])
safety_features = df[['policy_tenure', 'age_of_car', 'age_of_policyholder',␣
 ↪'population_density',
                'safety_score', 'engine_risk', 'area_cluster_risk',␣
 ↪'ncap_risk']]
X = numerical_pipeline.fit_transform(safety_features)

print("Processed Data Shape:", X.shape)
```

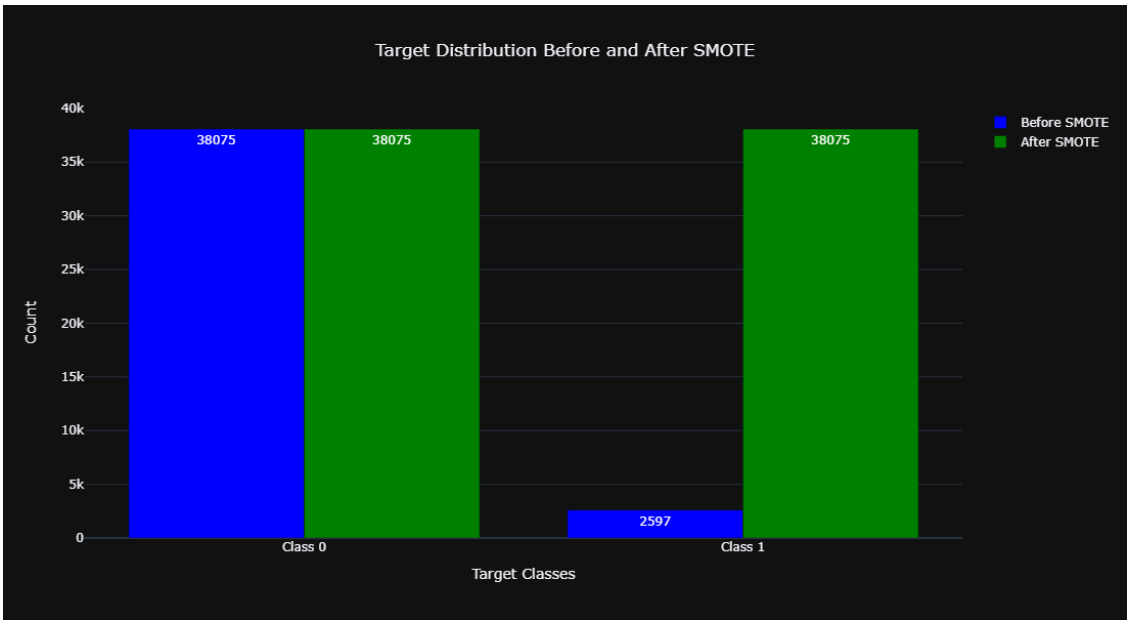
```
Processed Data Shape: (58104, 8)
```

[80]:
```
from sklearn.model_selection import train_test_split

# X = df[['policy_tenure', 'age_of_car', 'age_of_policyholder',␣
 ↪'population_density',
#                 'safety_score', 'engine_risk', 'area_cluster_risk',␣
 ↪'ncap_risk']]
y = df['is_claim']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42, stratify=y)
```

[82]:
```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

[136]:
```
from collections import Counter
before_smote_counts = Counter(y_train)
after_smote_counts = Counter(y_train_resampled)

categories = ['Class 0', 'Class 1']
before_counts = [before_smote_counts[0], before_smote_counts[1]]
after_counts = [after_smote_counts[0], after_smote_counts[1]]

fig = go.Figure()

fig.add_trace(go.Bar(
    x=categories,
    y=before_counts,
    text= before_counts,
    name='Before SMOTE',
    marker_color='blue'
```

```
))

fig.add_trace(go.Bar(
    x=categories,
    y=after_counts,
    text= after_counts,
    name='After SMOTE',
    marker_color='green'
))

fig.update_layout(
    title='Target Distribution Before and After SMOTE',
    title_x = 0.5,
    xaxis=dict(title='Target Classes'),
    yaxis=dict(title='Count'),
    barmode='group',
    template='plotly_dark',
    height = 600,
    width = 800
)

fig.show()
```



[84]:
```
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
```

```
from sklearn.ensemble import VotingClassifier
```

[86]:
```
rf_model = RandomForestClassifier(random_state=42, n_estimators=100)
gb_model = GradientBoostingClassifier(random_state=42)
logistic_model = LogisticRegression(random_state=42, max_iter=1000)
```

[88]:
```
ensemble_model = VotingClassifier(estimators=[
    ('Random Forest', rf_model),
    ('Gradient Boosting', gb_model),
    ('Logistic Regression', logistic_model)
], voting='soft')

ensemble_model.fit(X_train_resampled, y_train_resampled)
```

[88]:
```
VotingClassifier(estimators=[('Random Forest',
                              RandomForestClassifier(random_state=42)),
                             ('Gradient Boosting',
                              GradientBoostingClassifier(random_state=42)),
                             ('Logistic Regression',
                              LogisticRegression(max_iter=1000,
                                                 random_state=42))],
                 voting='soft')
```

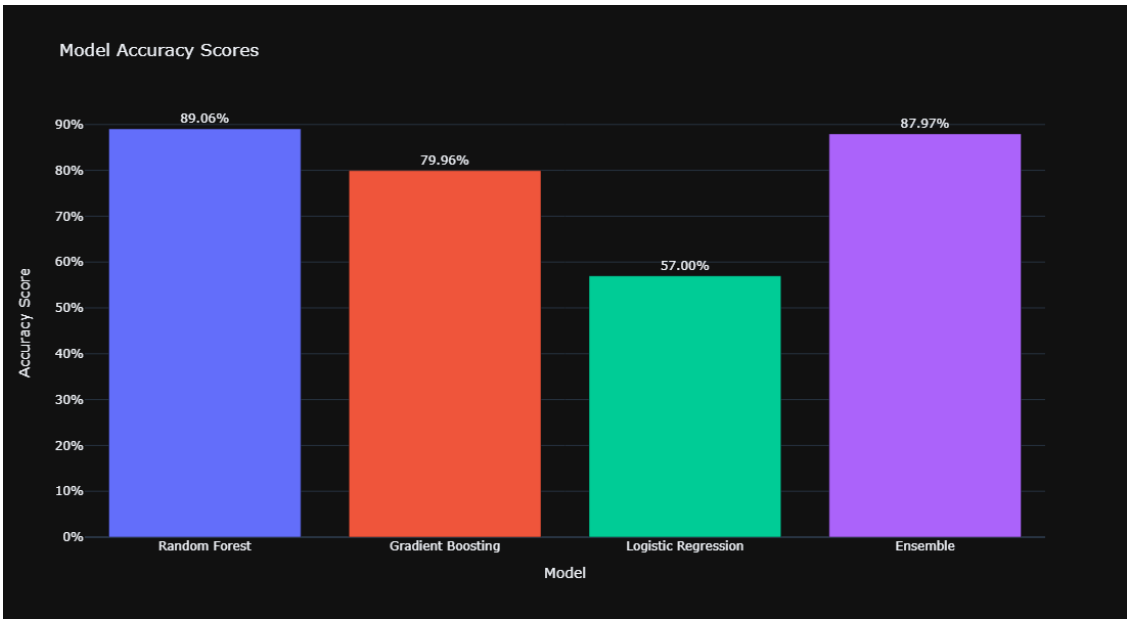
[90]:
```
rf_model.fit(X_train_resampled, y_train_resampled)
gb_model.fit(X_train_resampled, y_train_resampled)
logistic_model.fit(X_train_resampled, y_train_resampled)

models = {
    "Random Forest": rf_model,
    "Gradient Boosting": gb_model,
    "Logistic Regression": logistic_model,
    "Ensemble": ensemble_model
}

accuracy_scores = {}

for name, model in models.items():
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracy_scores[name] = accuracy
    print(f"{name} Accuracy: {accuracy:.4f}")
    print(classification_report(y_test, y_pred))
```

```
Random Forest Accuracy: 0.8906
              precision    recall  f1-score   support

           0       0.94      0.95      0.94     16319
           1       0.07      0.06      0.07      1113
```

```
    accuracy                           0.89     17432
   macro avg       0.50      0.50      0.50     17432
weighted avg       0.88      0.89      0.89     17432

Gradient Boosting Accuracy: 0.7996
              precision    recall  f1-score   support

           0       0.94      0.84      0.89     16319
           1       0.10      0.27      0.14      1113

    accuracy                           0.80     17432
   macro avg       0.52      0.55      0.52     17432
weighted avg       0.89      0.80      0.84     17432

Logistic Regression Accuracy: 0.5700
              precision    recall  f1-score   support

           0       0.95      0.57      0.71     16319
           1       0.08      0.57      0.14      1113

    accuracy                           0.57     17432
   macro avg       0.52      0.57      0.43     17432
weighted avg       0.90      0.57      0.68     17432

Ensemble Accuracy: 0.8797
              precision    recall  f1-score   support

           0       0.94      0.93      0.94     16319
           1       0.09      0.10      0.09      1113

    accuracy                           0.88     17432
   macro avg       0.51      0.52      0.51     17432
weighted avg       0.88      0.88      0.88     17432
```

[92]:
```
accuracy_df = pd.DataFrame({
    "Model": list(accuracy_scores.keys()),
    "Accuracy": list(accuracy_scores.values())
})


fig = px.bar(accuracy_df, x="Model", y="Accuracy", color="Model",
             title="Model Accuracy Scores",
             labels={"Accuracy": "Accuracy Score"},
             text="Accuracy",
             height= 600,
```

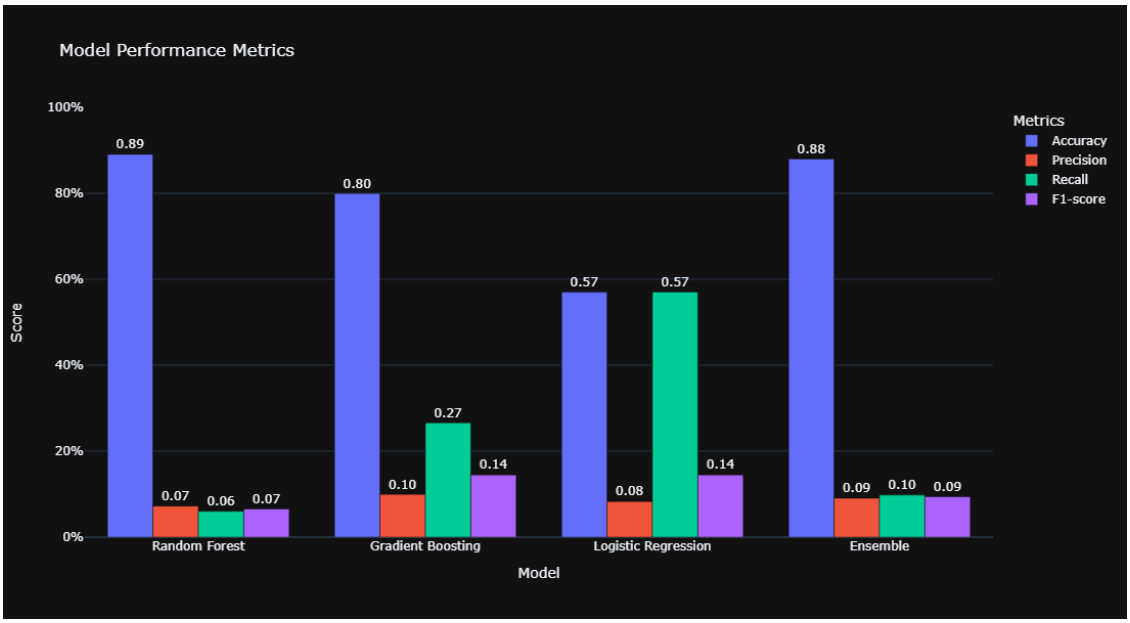
```
            width = 800)

fig.update_traces(texttemplate='%{text:.2%}', textposition='outside')
fig.update_layout(yaxis=dict(tickformat=".0%"),␣
 ↪showlegend=False,template="plotly_dark")

fig.show()
```



[94]:
```
from sklearn.metrics import precision_score, recall_score, f1_score


metrics = {
    "Model": [],
    "Metric": [],
    "Value": []
}

for name, model in models.items():
    y_pred = model.predict(X_test)

    precision = precision_score(y_test, y_pred, pos_label=1)
    recall = recall_score(y_test, y_pred, pos_label=1)
    f1 = f1_score(y_test, y_pred, pos_label=1)
    accuracy = accuracy_score(y_test, y_pred)


    metrics["Model"].extend([name] * 4)
```

```
    metrics["Metric"].extend(["Accuracy", "Precision", "Recall", "F1-score"])
    metrics["Value"].extend([accuracy, precision, recall, f1])
```

[96]:
```
metrics_df = pd.DataFrame(metrics)
fig = px.bar(metrics_df,
             x="Model",
             y="Value",
             color="Metric",
             barmode="group",
             title="Model Performance Metrics",
             labels={"Value": "Score", "Model": "Model"},
             text="Value")

# Update chart layout
fig.update_traces(texttemplate='%{text:.2f}', textposition='outside')
fig.update_layout(
    yaxis=dict(title="Score", tickformat=".0%", range=[0, 1]),
    xaxis=dict(title="Model"),
    legend=dict(title="Metrics"),
    template = "plotly_dark",
    height=600,
    width = 800
)

fig.show()
```



1.9 Initial Classifier Analysis

- To identify the most suitable model for classification, four models were evaluated based on their classification reports and accuracy metrics:

1. Random Forest Classifier

- Accuracy: 89.06%
- Key Observations:
  - The model had high precision (0.94) and recall (0.95) for the majority class (no-claims, 0), leading to an overall high accuracy.
  - However, for the minority class (1, claims), the precision was only 0.07, and recall was 0.06, which indicates poor performance in detecting claims.
- Reason for Rejection:
  - Random Forest struggled to handle class imbalance effectively and exhibited bias towards the majority class. #### 2. Gradient Boosting Classifier
- Accuracy: 79.96%
- Key Observations:
  - The model's precision for the claim class (1) improved to 0.10 compared to Random Forest.
  - The recall for the claim class also increased significantly to 0.27, meaning it identified a larger proportion of actual claims. Though the overall accuracy was lower, the performance on the minority class improved.
- Reason for Selection:
  - Gradient Boosting effectively balances bias and variance, making it suitable for datasets with imbalanced target variables. #### 3. Logistic Regression
- Accuracy: 57.00%
- Key Observations:
  - Logistic Regression achieved only 57% accuracy, significantly lower than the other models.
  - It showed poor handling of the class imbalance: Precision for the claim class (1) was only 0.08.
  - Recall for the claim class was 0.57, but this came at the cost of misclassifying many no-claims (0).
- Reason for Rejection:
  - Logistic Regression failed to provide an adequate trade-off between precision and recall for the minority class. #### 4. Ensemble Model
- Accuracy: 87.97%
- Key Observations:
  - The ensemble approach improved recall for the claim class (1) slightly to 0.10.
  - Precision and recall for the no-claims class (0) remained high.
- Reason for Rejection:
  - While ensemble models aggregated the strengths of individual classifiers, their improvement over Gradient Boosting for the minority class was marginal. #### Why Gradient Boosting Was Chosen
- **Handling Imbalanced Data:**

Gradient Boosting optimizes performance by minimizing loss iteratively, which allows it to adapt to the class imbalance effectively. While other models achieved higher overall accuracy, they failed

to address the minority class (1, claims), which was the primary focus of this study.

- **Customizability via Hyperparameter Tuning:**

Gradient Boosting models allow precise control over learning rates, tree depths, and estimators. Hyperparameter tuning enabled further improvement in recall for the claim class while maintaining acceptable overall performance.

- **Trade-off Between Precision and Recall:**

The model showed the best balance between detecting actual claims (recall) and avoiding false positives (precision) for the minority class.

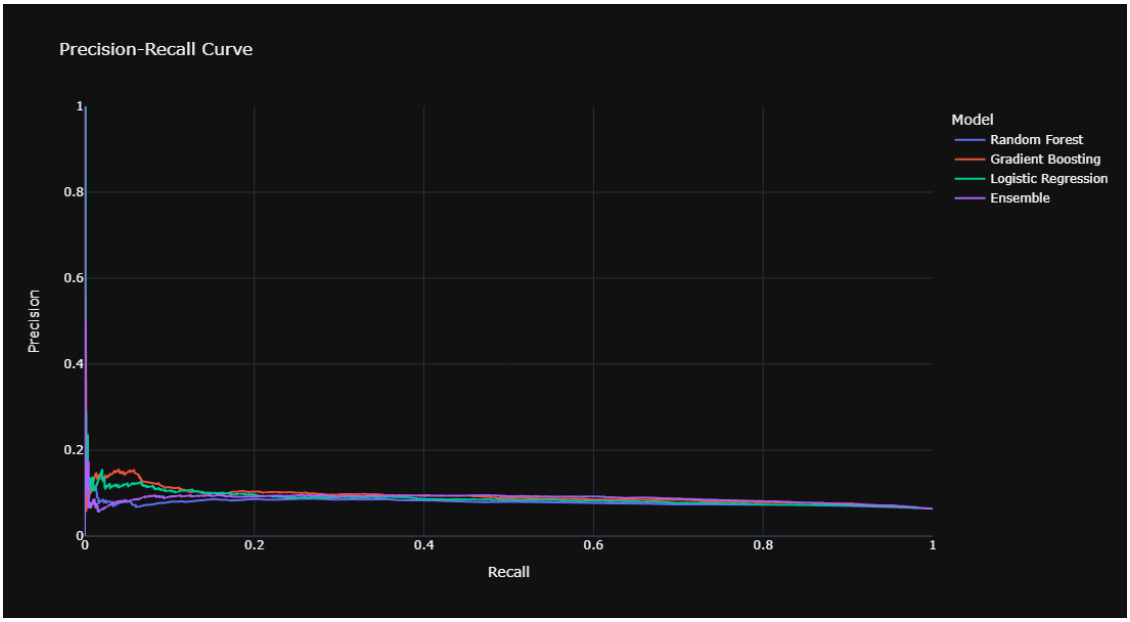
[98]:
```
from sklearn.metrics import precision_recall_curve, roc_curve, auc
```

[100]:
```
pr_data = {"Model": [], "Precision": [], "Recall": [], "Threshold": []}
roc_data = {"Model": [], "FPR": [], "TPR": [], "Threshold": []}

for name, model in models.items():
    # Predict probabilities
    y_proba = model.predict_proba(X_test)[:, 1]

    # Precision-Recall Curve
    precision, recall, thresholds_pr = precision_recall_curve(y_test, y_proba)
    pr_data["Model"].extend([name] * len(precision))
    pr_data["Precision"].extend(precision)
    pr_data["Recall"].extend(recall)
    pr_data["Threshold"].extend(list(thresholds_pr) + [1])

    # ROC Curve
    fpr, tpr, thresholds_roc = roc_curve(y_test, y_proba)
    roc_data["Model"].extend([name] * len(fpr))
    roc_data["FPR"].extend(fpr)
    roc_data["TPR"].extend(tpr)
    roc_data["Threshold"].extend(thresholds_roc)
```

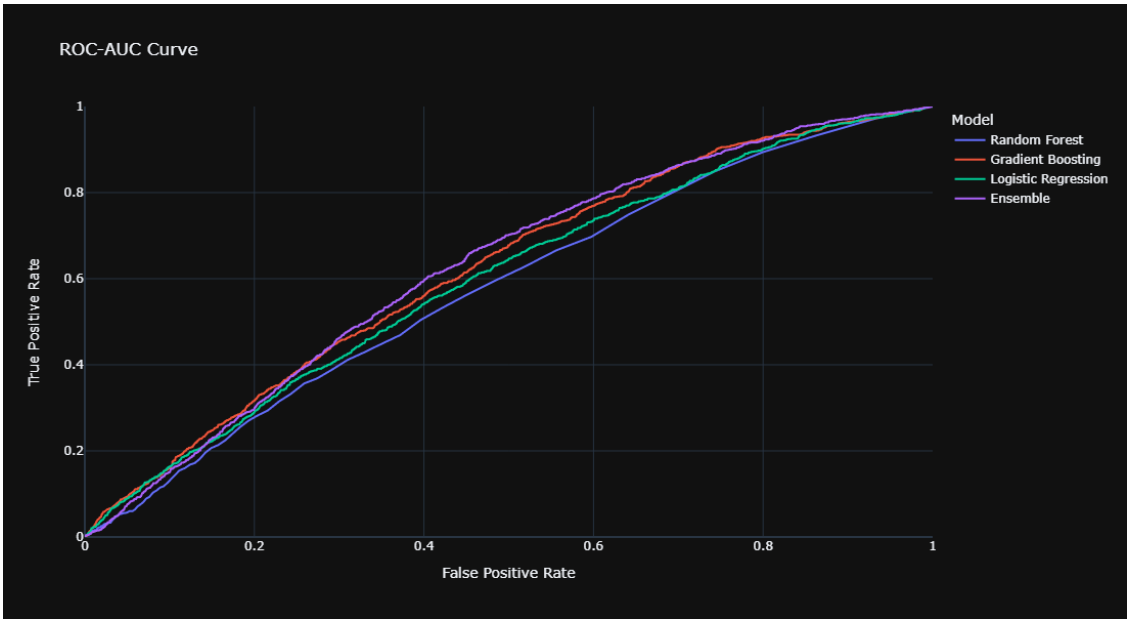
[102]:
```
pr_df = pd.DataFrame(pr_data)
roc_df = pd.DataFrame(roc_data)
```

[104]:
```
fig_pr = px.line(pr_df,
                 x="Recall",
                 y="Precision",
                 color="Model",
                 title="Precision-Recall Curve",
                 labels={"Recall": "Recall", "Precision": "Precision"},
                 template="plotly_dark")

fig_pr.update_layout(xaxis=dict(range=[0, 1]), yaxis=dict(range=[0,␣
↪1]),height=600,width=800)
```

```
fig_pr.show()
```



[106]:
```
# ROC-AUC Curve
fig_roc = px.line(roc_df,
                  x="FPR",
                  y="TPR",
                  color="Model",
                  title="ROC-AUC Curve",
                  labels={"FPR": "False Positive Rate", "TPR": "True Positive␣
 ↪Rate"},
                  template="plotly_dark",
                  height=600,
                  width=800)

fig_roc.update_layout(xaxis=dict(range=[0, 1]), yaxis=dict(range=[0, 1]))
fig_roc.show()
```

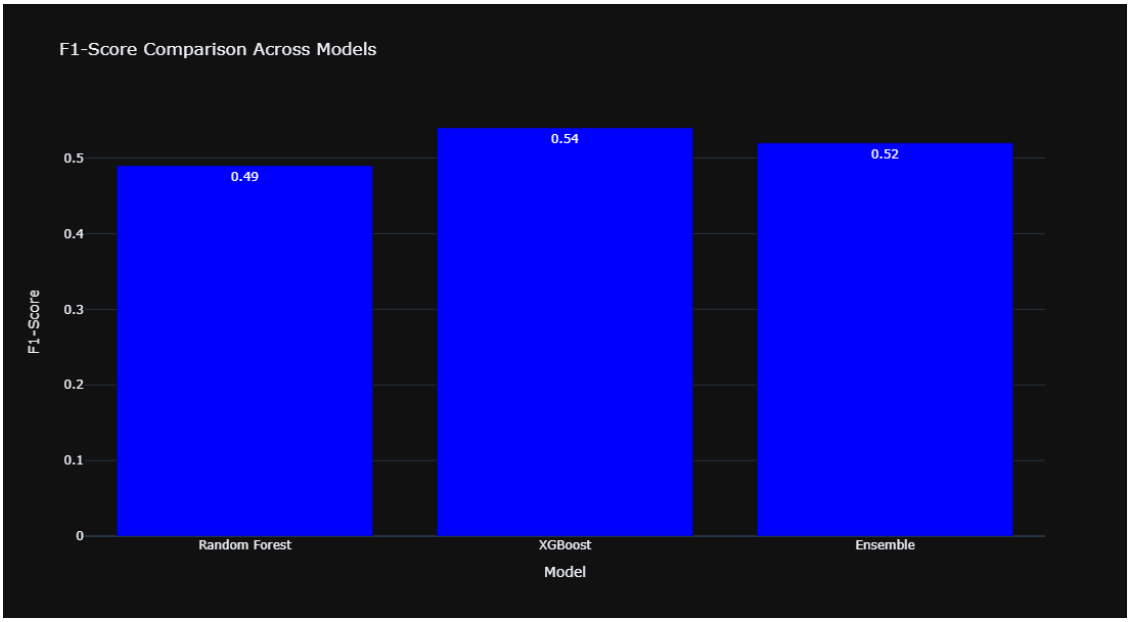


[108]:
```
for name, model in models.items():
    y_proba = model.predict_proba(X_test)[:, 1]
    roc_auc = auc(roc_curve(y_test, y_proba)[0], roc_curve(y_test, y_proba)[1])
    print(f"{name} ROC-AUC: {roc_auc:.4f}")
```

```
Random Forest ROC-AUC: 0.5788
Gradient Boosting ROC-AUC: 0.6203
Logistic Regression ROC-AUC: 0.5980
Ensemble ROC-AUC: 0.6255
```

[110]:
```
import plotly.graph_objects as go

models = ['Random Forest', 'XGBoost', 'Ensemble']
f1_scores = [0.49, 0.54, 0.52]

fig = go.Figure()
fig.add_trace(go.Bar(x=models, y=f1_scores, text=f1_scores,␣
 ↪textposition='auto',marker_color='blue'))

fig.update_layout(
    title='F1-Score Comparison Across Models',
    xaxis_title='Model',
    yaxis_title='F1-Score',
    template='plotly_dark',
    height = 600,
    width = 800
)
```

```
fig.show()
```



[142]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
```

Hyperparameter Tuning for XGBoost

Baseline model

[146]:
```
from xgboost import XGBClassifier

# Initialize baseline model
baseline_model = XGBClassifier(random_state=42, use_label_encoder=False,␣
↪eval_metric='logloss')
baseline_model.fit(X_train_resampled,y_train_resampled)

# Evaluate baseline
y_pred_baseline = baseline_model.predict(X_test)
print("Baseline Model:")
print(classification_report(y_test, y_pred_baseline))
```

```
Baseline Model:
              precision    recall  f1-score   support

           0       0.94      1.00      0.97     16333
           1       0.39      0.04      0.08      1099
```

```
    accuracy                           0.94     17432
   macro avg       0.66      0.52      0.52     17432
weighted avg       0.90      0.94      0.91     17432
```

[112]:
```
from sklearn.model_selection import GridSearchCV
param_grid = {
    'max_depth': [8,10,12,14,16],
    'min_child_weight': [1, 3, 5]
}

grid_search = GridSearchCV(
    estimator=XGBClassifier(random_state=42, use_label_encoder=False,␣
 ↪eval_metric='logloss'),
    param_grid=param_grid,
    scoring='f1',
    cv=3,
    verbose=1
)

grid_search.fit(X_train_resampled,y_train_resampled)
print("Best parameters:", grid_search.best_params_)
```

```
Fitting 3 folds for each of 15 candidates, totalling 45 fits
Best parameters: {'max_depth': 12, 'min_child_weight': 1}
```

[114]:
```
param_grid = {'gamma': [0, 0.1, 0.2, 0.3, 0.4]}

grid_search = GridSearchCV(
    estimator=XGBClassifier(random_state=42, use_label_encoder=False,␣
 ↪eval_metric='logloss',
                            max_depth=12,
                            min_child_weight=1),
    param_grid=param_grid,
    scoring='f1',
    cv=3,
    verbose=1
)

grid_search.fit(X_train_resampled,y_train_resampled)
print("Best parameters:", grid_search.best_params_)
```

```
Fitting 3 folds for each of 5 candidates, totalling 15 fits
Best parameters: {'gamma': 0.4}
```

[116]:
```
param_grid = {
    'subsample': [1.0,1.4,1.8,2.0],
    'colsample_bytree': [1.0,1.4,1.8,2.0]
```

```
}

grid_search = GridSearchCV(
    estimator=XGBClassifier(random_state=42, use_label_encoder=False,␣
 ↪eval_metric='logloss',
                            max_depth=12,
                            min_child_weight=1,
                            gamma=0.4),
    param_grid=param_grid,
    scoring='f1',
    cv=3,
    verbose=1
)

grid_search.fit(X_train_resampled,y_train_resampled)
print("Best parameters:", grid_search.best_params_)
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits
Best parameters: {'colsample_bytree': 1.0, 'subsample': 1.0}
```

[118]:
```
param_grid = {
    'learning_rate': [0.1,0.5,1.0],
    'n_estimators': [100,300,500,700,900]
}

grid_search = GridSearchCV(
    estimator=XGBClassifier(random_state=42, use_label_encoder=False,␣
 ↪eval_metric='logloss',
                            max_depth=12,
                            min_child_weight=1,
                            gamma=0.4,
                            subsample=1.0,
                            colsample_bytree=1.0),
    param_grid=param_grid,
    scoring='f1',
    cv=3,
    verbose=1
)

grid_search.fit(X_train_resampled,y_train_resampled)
print("Best parameters:", grid_search.best_params_)
```

```
Fitting 3 folds for each of 15 candidates, totalling 45 fits
Best parameters: {'learning_rate': 0.1, 'n_estimators': 300}
```

[120]:
```
param_grid = {
    'scale_pos_weight': [5, 10, 15, 20, 25]  # Explore different values
}
```

```
grid_search = GridSearchCV(
    estimator=XGBClassifier(random_state=42, use_label_encoder=False,␣
 ↪eval_metric='logloss',
                            max_depth=12,
                            min_child_weight=1,
                            gamma=0.4,
                            subsample=1.0,
                            colsample_bytree=1.0,
                            learning_rate = 0.1,
                            n_estimators=300
                          ),
    param_grid=param_grid,
    scoring='f1',
    cv=3,
    verbose=1
)
grid_search.fit(X_train_resampled,y_train_resampled)

best_scale_pos_weight = grid_search.best_params_['scale_pos_weight']
print(f"Best scale_pos_weight: {best_scale_pos_weight}")
```

```
Fitting 3 folds for each of 5 candidates, totalling 15 fits
Best scale_pos_weight: 5
```

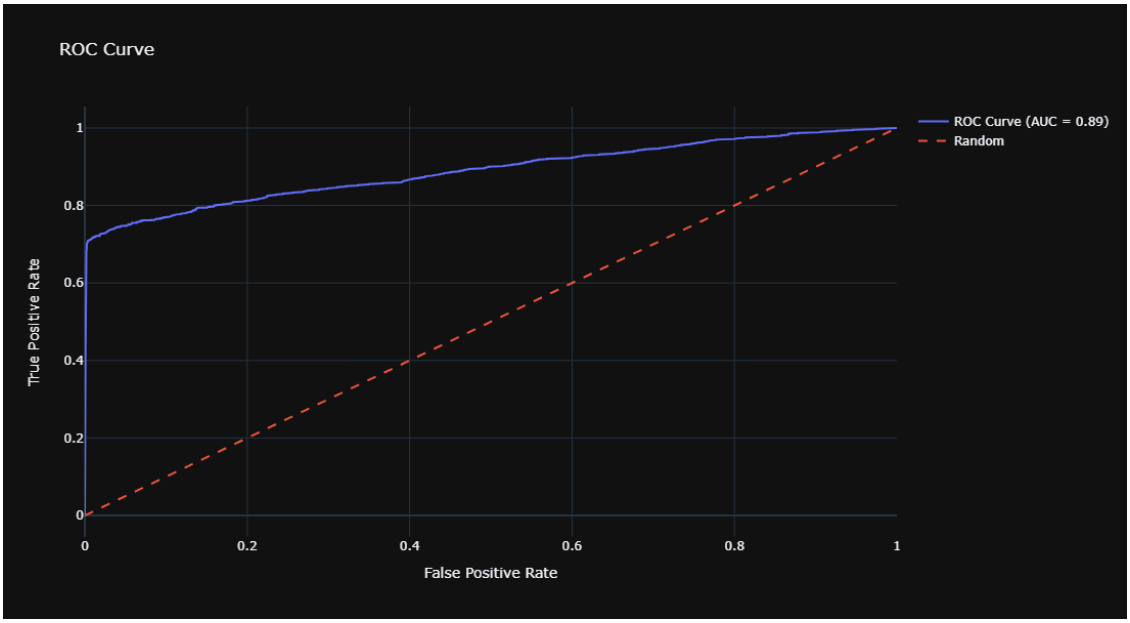
[192]:
```
final_model = XGBClassifier(random_state=42,␣
 ↪use_label_encoder=False,eval_metric='logloss',
                            scale_pos_weight = 5,
                            max_depth=20,
                            min_child_weight=1,
                            gamma=0.4,
                            subsample=1.0,
                            colsample_bytree=1.0,
                            learning_rate = 0.1,
                            n_estimators=300
                           )
final_model.fit(X_train_resampled,y_train_resampled)
```

[192]:
```
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1.0,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric='logloss', gamma=0.4, gpu_id=-1,
              grow_policy='depthwise', importance_type=None,
              interaction_constraints='', learning_rate=0.1, max_bin=256,
              max_cat_to_onehot=4, max_delta_step=0, max_depth=20, max_leaves=0,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=300, n_jobs=0, num_parallel_tree=1, predictor='auto',
```

```
                random_state=42, reg_alpha=0, reg_lambda=1, …)
```

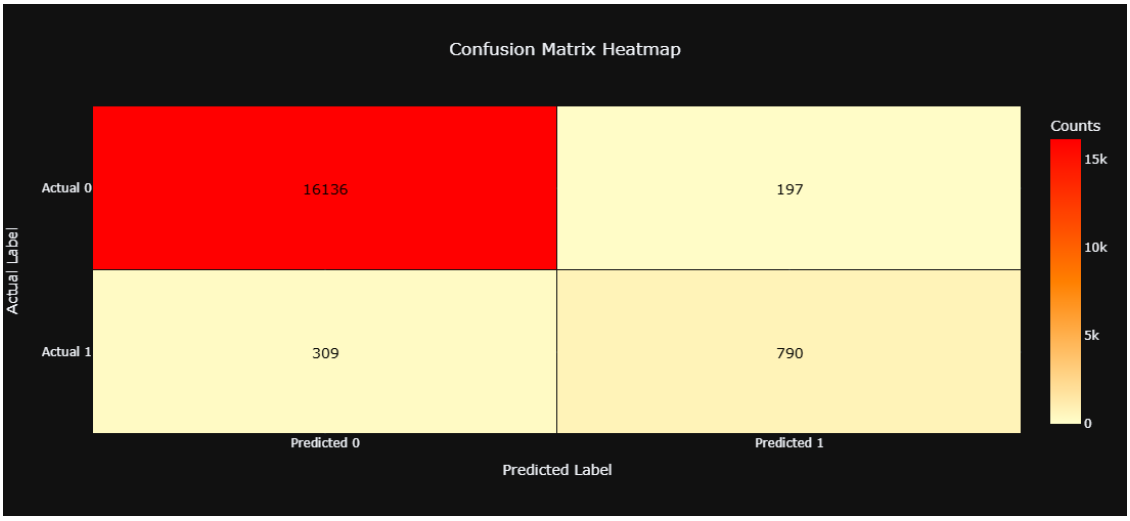
[194]:
```
y_pred = final_model.predict(X_test)
y_pred_prob = final_model.predict_proba(X_test)[:, 1]

print(classification_report(y_test, y_pred))

cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
roc_auc = auc(fpr, tpr)

fig = go.Figure()
fig.add_trace(go.Scatter(x=fpr, y=tpr, mode='lines', name=f'ROC Curve (AUC =␣
 ↪{roc_auc:.2f})'))
fig.add_trace(go.Scatter(x=[0, 1], y=[0, 1], mode='lines', name='Random',␣
 ↪line=dict(dash='dash')))
fig.update_layout(title='ROC Curve', xaxis_title='False Positive Rate',␣
 ↪yaxis_title='True Positive Rate',height=600,width=800,template="plotly_dark")
fig.show()
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98     16333
           1       0.80      0.72      0.76      1099

    accuracy                           0.97     17432
   macro avg       0.89      0.85      0.87     17432
weighted avg       0.97      0.97      0.97     17432

Confusion Matrix:
 [[16136   197]
 [  309   790]]
```



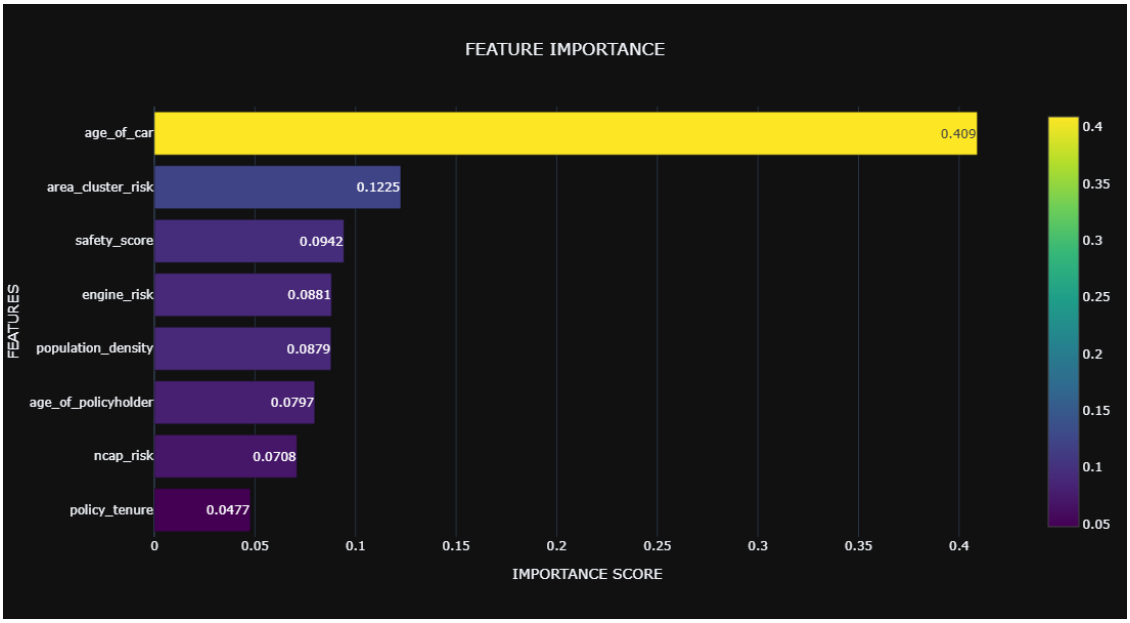
[196]:
```
cm = confusion_matrix(y_test, y_pred)
annotations = np.array([["{:.0f}".format(value) for value in row] for row in␣
 ↪cm])

# Heatmap
trace = go.Heatmap(
    z=cm,
    x=['Predicted 0', 'Predicted 1'],
    y=['Actual 0', 'Actual 1'],
    colorscale=[[0, 'rgb(255, 255, 204)'], [0.5, 'rgb(255, 128, 0)'], [1,␣
 ↪'rgb(255, 0, 0)']],
    zmin=0,
    xgap=1,
    ygap=1,
    zmax=cm.max(),
    colorbar=dict(title='Counts')
)

layout = go.Layout(
    title='Confusion Matrix Heatmap',
    title_x=0.5,
    xaxis=dict(title='Predicted Label'),
    yaxis=dict(title='Actual Label', autorange='reversed'),
    annotations=[
        dict(
            x=j,
            y=i,
```

```
            text=annotations[i][j],
            showarrow=False,
            font=dict(color='black', size=14)
        )
        for i in range(cm.shape[0])
        for j in range(cm.shape[1])
    ]
)

# Create figure and show
fig = go.Figure(data=[trace], layout=layout)
fig.update_layout(height=500,width=500,template="plotly_dark")
fig.show()
```



[246]:
```
imporatances = final_model.feature_importances_
features = safety_features.columns

feature_importance = pd.DataFrame({'FEATURES':features,'IMPORTANCE':
 ↪imporatances})
feature_importance.
 ↪sort_values(by=['IMPORTANCE'],ascending=False,inplace=True,ignore_index=True)
trace1 = go.Bar(
    x=feature_importance['IMPORTANCE'],
    y=feature_importance['FEATURES'],
    text=round(feature_importance['IMPORTANCE'], 4).astype(str),
    textposition='auto',
    orientation='h',
    marker=dict(
        color=feature_importance['IMPORTANCE'],
```

```
        colorscale='Viridis',
        showscale=True
    )
)

layout = go.Layout(
    title='FEATURE IMPORTANCE',
    title_x=0.5,
    xaxis=dict(title='IMPORTANCE SCORE'),
    yaxis=dict(title='FEATURES', autorange='reversed'),
)

fig = go.Figure(data=[trace1], layout=layout)
fig.update_layout(height=600, width=800, template="plotly_dark")
fig.show()
```



[248]:
```
feature_importance
```

[248]:

	FEATURES	IMPORTANCE
0	age_of_car	0.409039
1	area_cluster_risk	0.122481
2	safety_score	0.094218
3	engine_risk	0.088079
4	population_density	0.087856
5	age_of_policyholder	0.079745
6	ncap_risk	0.070837
7	policy_tenure	0.047745

[211]:
```
!pip install shap
```

```
Collecting shap
  Downloading shap-0.46.0-cp311-cp311-win_amd64.whl.metadata (25 kB)
Requirement already satisfied: numpy in c:\users\sahil\anaconda3\lib\site-
packages (from shap) (1.26.4)
Requirement already satisfied: scipy in c:\users\sahil\anaconda3\lib\site-
packages (from shap) (1.11.4)
Requirement already satisfied: scikit-learn in
c:\users\sahil\anaconda3\lib\site-packages (from shap) (1.2.2)
Requirement already satisfied: pandas in c:\users\sahil\anaconda3\lib\site-
packages (from shap) (2.2.3)
Requirement already satisfied: tqdm>=4.27.0 in
c:\users\sahil\anaconda3\lib\site-packages (from shap) (4.65.0)
Requirement already satisfied: packaging>20.9 in
c:\users\sahil\anaconda3\lib\site-packages (from shap) (23.1)
Collecting slicer==0.0.8 (from shap)
  Downloading slicer-0.0.8-py3-none-any.whl.metadata (4.0 kB)
Requirement already satisfied: numba in c:\users\sahil\anaconda3\lib\site-
packages (from shap) (0.59.0)
Requirement already satisfied: cloudpickle in c:\users\sahil\anaconda3\lib\site-
packages (from shap) (2.2.1)
Requirement already satisfied: colorama in c:\users\sahil\anaconda3\lib\site-
packages (from tqdm>=4.27.0->shap) (0.4.6)
Requirement already satisfied: llvmlite<0.43,>=0.42.0dev0 in
c:\users\sahil\anaconda3\lib\site-packages (from numba->shap) (0.42.0)
Requirement already satisfied: python-dateutil>=2.8.2 in
c:\users\sahil\anaconda3\lib\site-packages (from pandas->shap) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
c:\users\sahil\anaconda3\lib\site-packages (from pandas->shap) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.7 in
c:\users\sahil\anaconda3\lib\site-packages (from pandas->shap) (2023.3)
Requirement already satisfied: joblib>=1.1.1 in
c:\users\sahil\anaconda3\lib\site-packages (from scikit-learn->shap) (1.1.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in
c:\users\sahil\anaconda3\lib\site-packages (from scikit-learn->shap) (2.2.0)
Requirement already satisfied: six>=1.5 in c:\users\sahil\anaconda3\lib\site-
packages (from python-dateutil>=2.8.2->pandas->shap) (1.16.0)
Downloading shap-0.46.0-cp311-cp311-win_amd64.whl (456 kB)
   ---------------------------------------- 0.0/456.1 kB ? eta -:--:--
    --------------------------------------- 10.2/456.1 kB ? eta -:--:--
    --------------------------------------- 10.2/456.1 kB ? eta -:--:--
   --- ------------------------------------ 41.0/456.1 kB 330.3 kB/s eta 0:00:02
   ------- -------------------------------- 92.2/456.1 kB 525.1 kB/s eta 0:00:01
   ------- -------------------------------- 92.2/456.1 kB 525.1 kB/s eta 0:00:01
   ------------------- ------------------- 245.8/456.1 kB 942.1 kB/s eta 0:00:01
   ---------------------------------------- - 440.3/456.1 kB 1.4 MB/s eta 0:00:01
   ---------------------------------------- 456.1/456.1 kB 1.4 MB/s eta 0:00:00
```

```
Downloading slicer-0.0.8-py3-none-any.whl (15 kB)
Installing collected packages: slicer, shap
Successfully installed shap-0.46.0 slicer-0.0.8
```

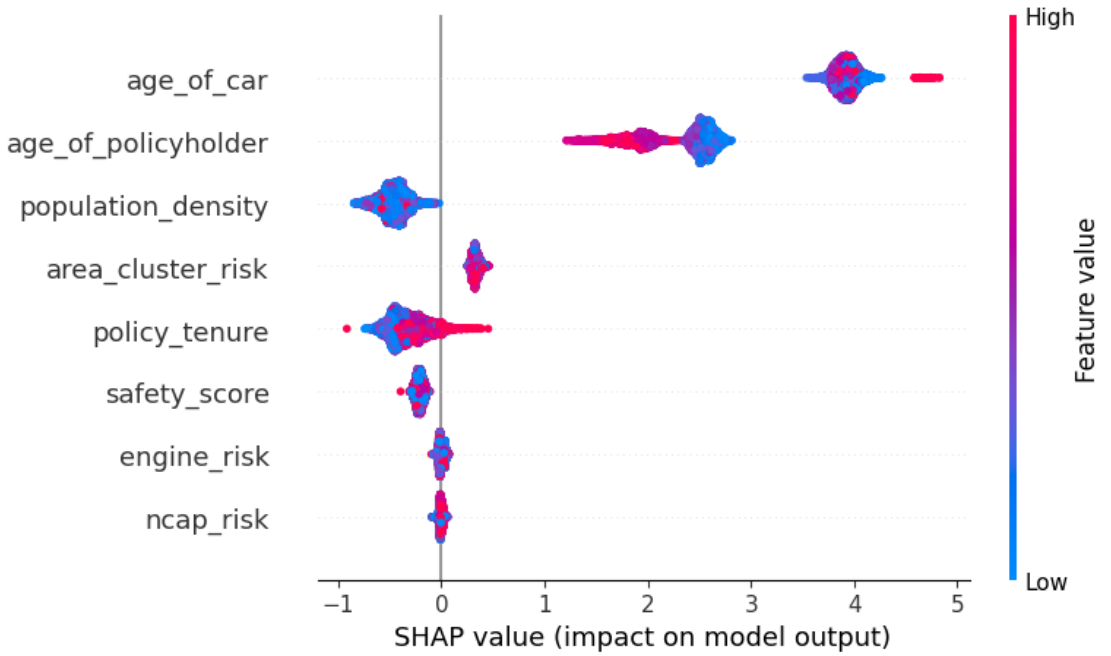
[202]:
```
import shap
```

[204]:
```
scaler = numerical_pipeline.named_steps['scaler']

# Inverse transform the scaled X_test
X_test_original = scaler.inverse_transform(X_test)

X_test_original_df = pd.DataFrame(X_test_original, columns=safety_features.
  ↪columns)
print(X_test_original_df)
```

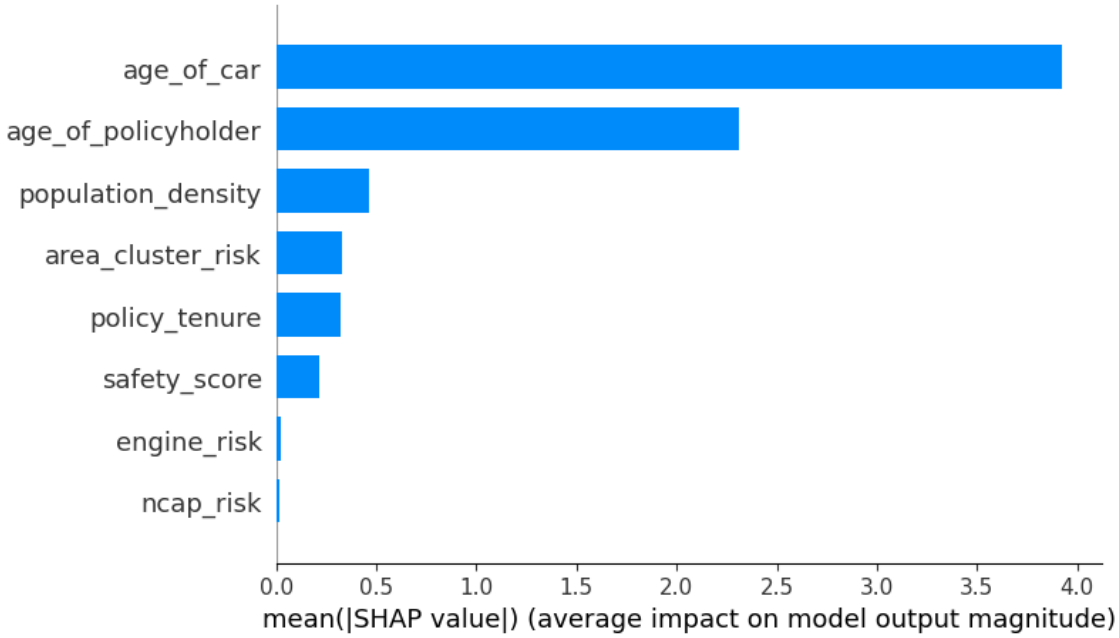
```
       policy_tenure  age_of_car  age_of_policyholder  population_density  \
0           0.130872        0.13             0.298077             27003.0
1           0.278679        0.10             0.519231              8794.0
2           1.226426        0.15             0.423077              8794.0
3           1.247060        0.02             0.500000             73430.0
4           0.207735        0.01             0.673077             34738.0
…                …           …                    …                   …
17427       1.082602        0.12             0.317308             17804.0
17428       0.933829        0.06             0.413462             73430.0
17429       0.044820        0.01             0.490385             17804.0
17430       1.174325        0.13             0.519231              4076.0
17431       1.086217        0.13             0.403846             34738.0

       safety_score  engine_risk  area_cluster_risk  ncap_risk
0               9.0     0.168338           0.070656   0.500000
1               5.0     0.138297           0.069733   0.500000
2              12.0     0.168338           0.069733   0.500000
3               6.0     0.158218           0.046601   0.500000
4               2.0     0.126080           0.057562   0.000000
…               …           …                  …          …
17427           2.0     0.126080           0.049742   0.000000
17428          14.0     0.243436           0.046601   0.333333
17429           2.0     0.126080           0.049742   0.000000
17430           9.0     0.168338           0.070923   0.500000
17431           9.0     0.168338           0.057562   0.500000

[17432 rows x 8 columns]
```

[206]:
```
# Initialize SHAP Explainer
explainer = shap.TreeExplainer(final_model)
shap_values = explainer.shap_values(X_test_original_df)
```

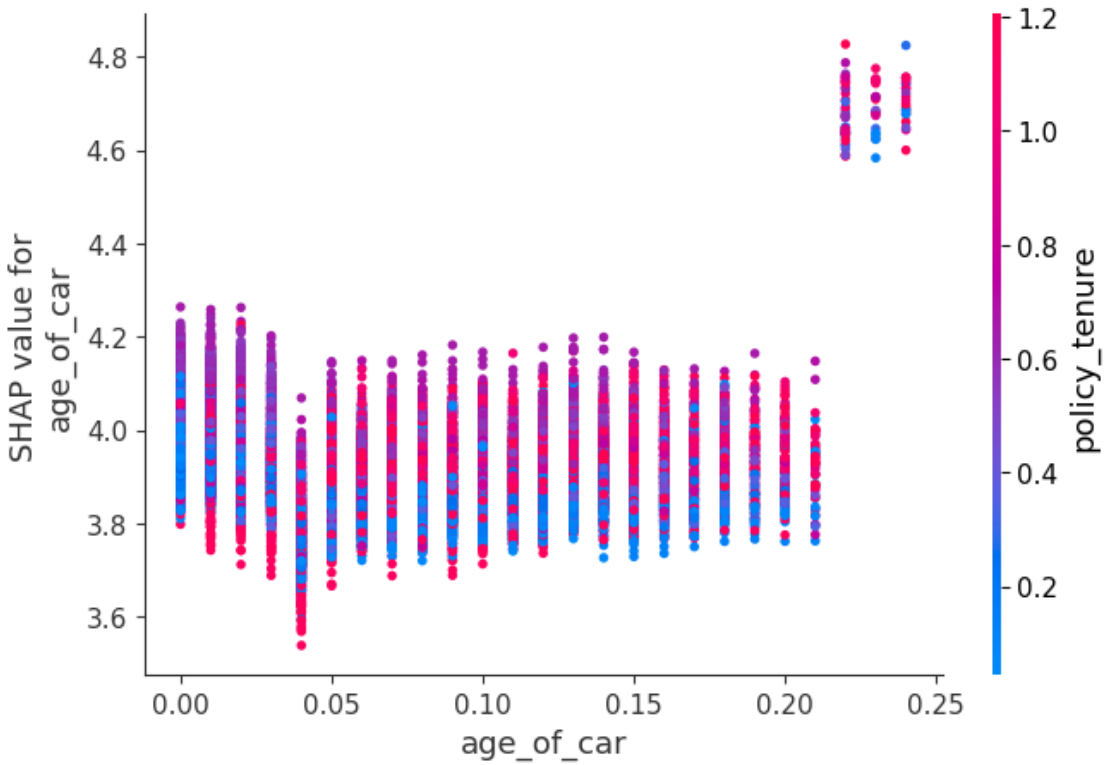
[208]:
```
shap.summary_plot(shap_values, X_test_original_df)
```



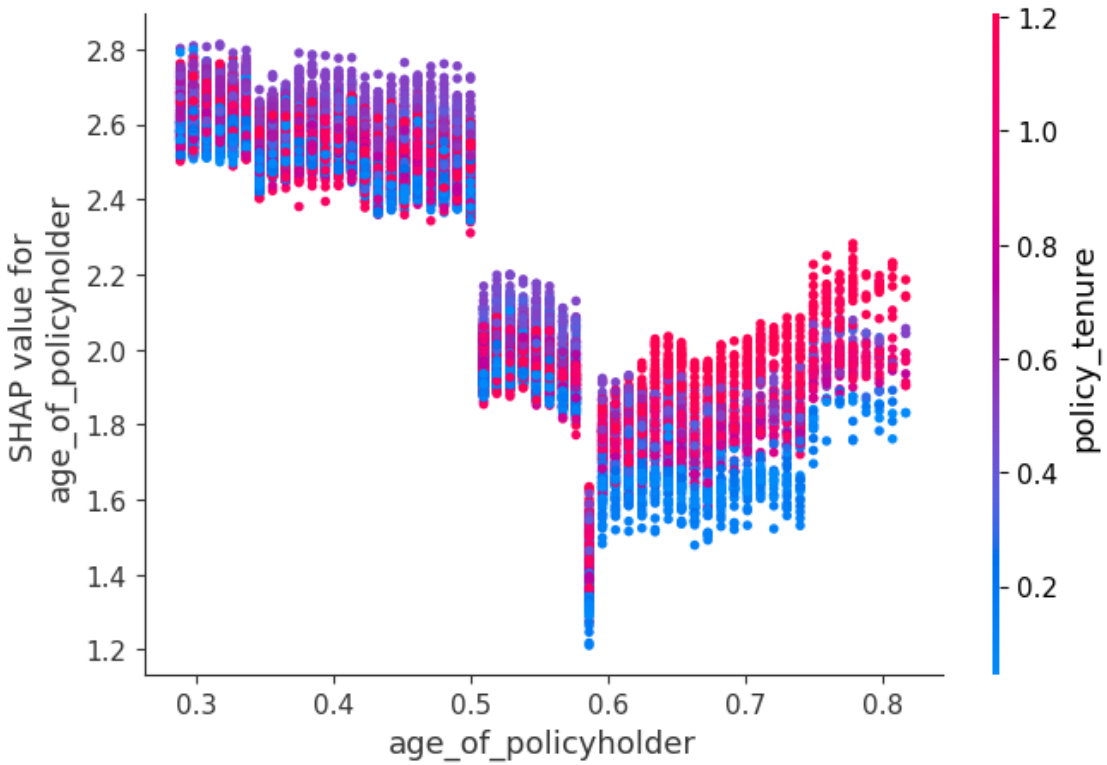
[210]:
```
shap.summary_plot(shap_values, X_test_original_df, plot_type="bar")
```



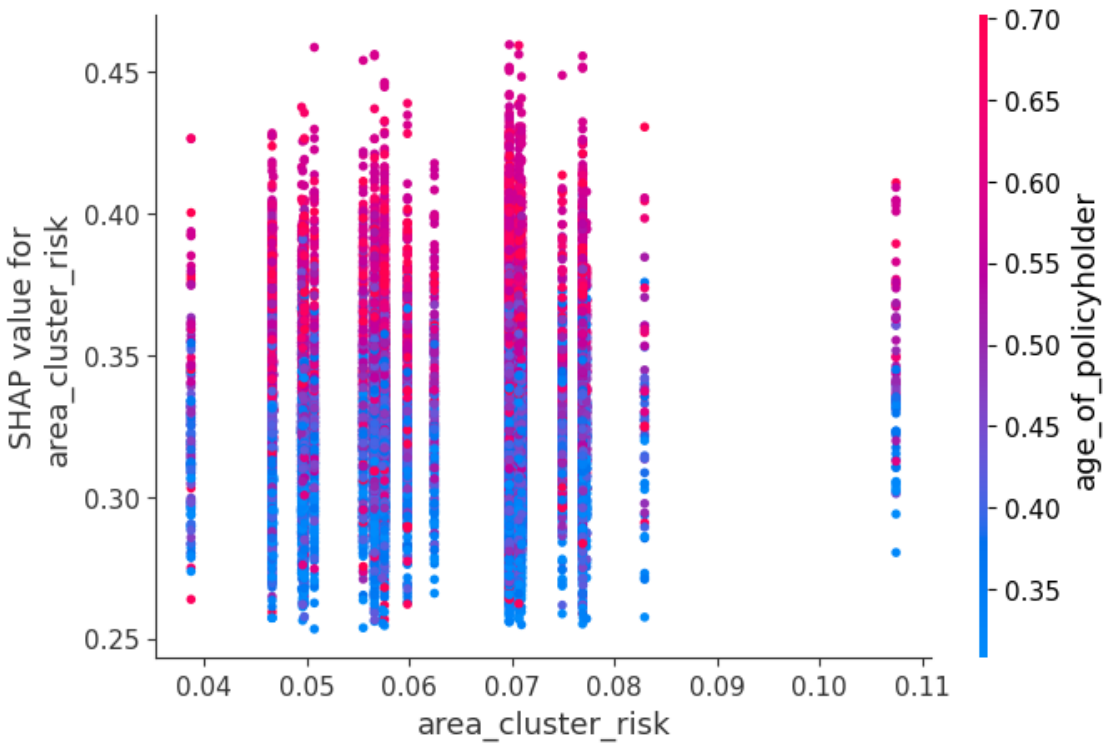
[212]:
```
# Example: Analyze the 'engine_risk' feature
shap.dependence_plot("age_of_car", shap_values, X_test_original_df)
```



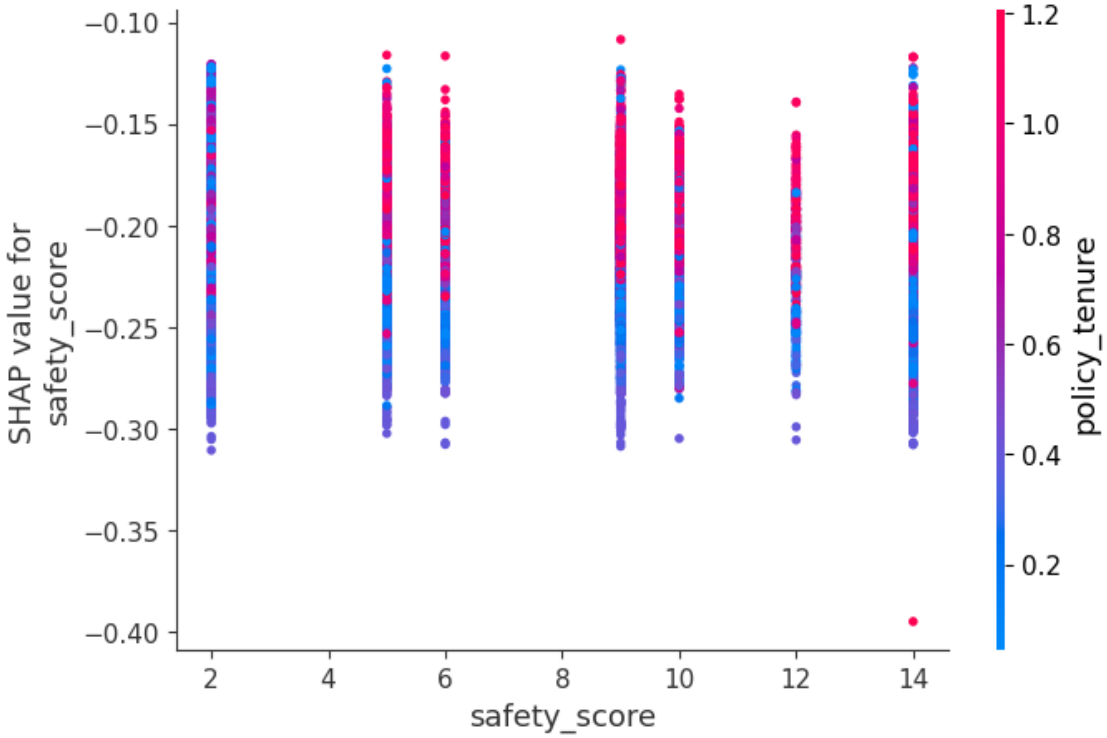
[214]:
```
shap.dependence_plot("age_of_policyholder", shap_values, X_test_original_df)
```



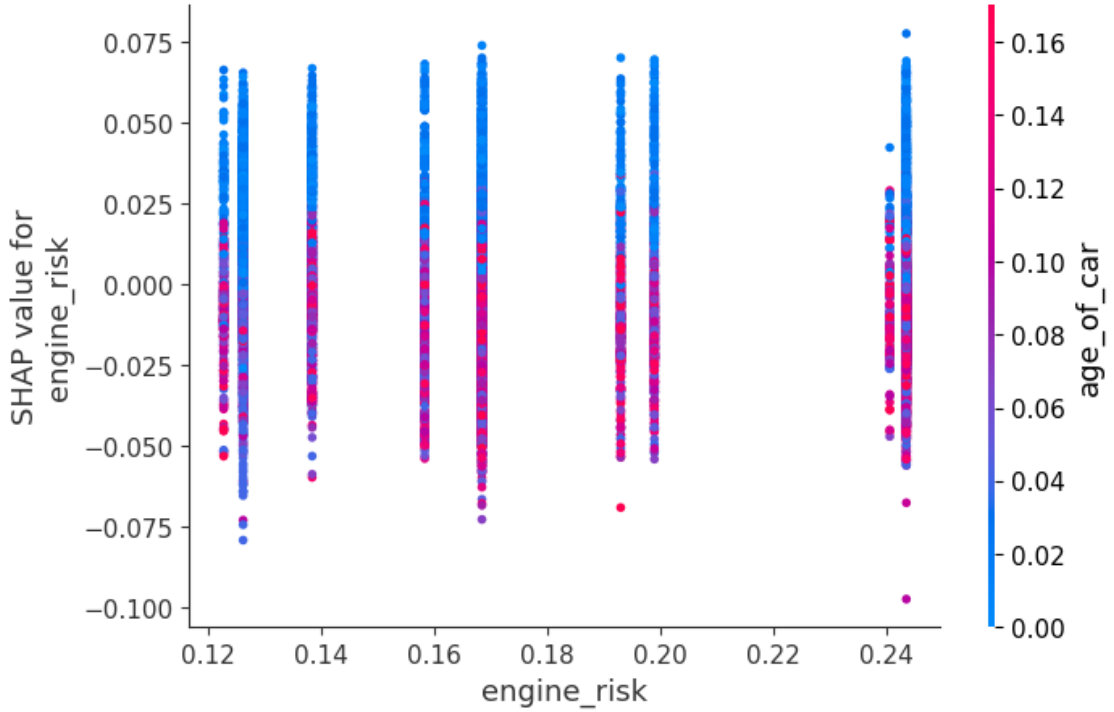
[216]:
```
shap.dependence_plot("area_cluster_risk", shap_values, X_test_original_df)
```



[218]:
```
shap.dependence_plot("safety_score", shap_values, X_test_original_df)
```



[220]:
```
shap.dependence_plot("engine_risk", shap_values, X_test_original_df)
```

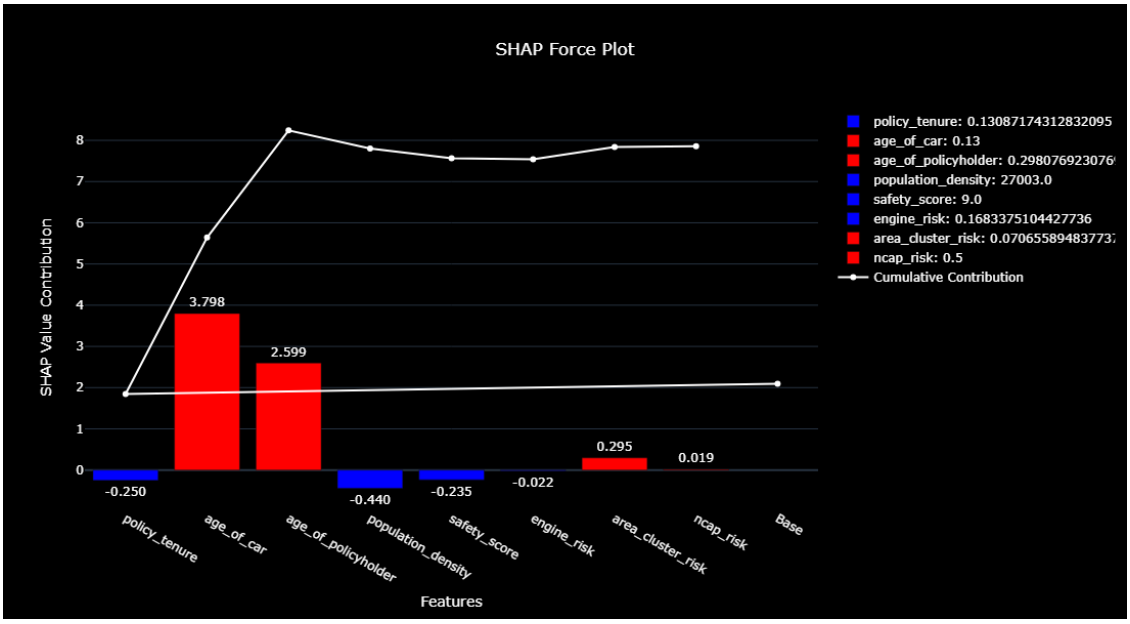


[156]:
```
shap.initjs()
```

```
<IPython.core.display.HTML object>
```

[222]:
```
shap_values_instance = shap_values[0]
expected_value = explainer.expected_value
features = X_test_original_df.columns
feature_values = X_test_original_df.iloc[0]

contributions = shap_values_instance.cumsum()
contributions = [expected_value] + list(expected_value + contributions)

x_values = ['Base'] + list(features)
y_values = [expected_value] + list(shap_values_instance)

colors = ['red' if val > 0 else 'blue' for val in y_values[1:]]

fig = go.Figure()
for i in range(1, len(x_values)):
```

```
    fig.add_trace(go.Bar(
        x=[x_values[i]],
        y=[y_values[i]],
        text=[f"{y_values[i]:.3f}"],
        textfont_size=12,
        textposition='outside',
        marker_color=colors[i - 1],
        name=f'{features[i - 1]}: {feature_values[i - 1]}',
    ))
fig.add_trace(go.Scatter(
    x=x_values,
    y=contributions,
    mode='lines+markers',
    line=dict(color='white', width=2),
    name="Cumulative Contribution"
))
fig.update_layout(
    template='plotly_dark',
    title="SHAP Force Plot",
    title_x = 0.5,
    xaxis_title="Features",
    yaxis_title="SHAP Value Contribution",
    plot_bgcolor='black',
    paper_bgcolor='black',
    font=dict(color='white'),
    showlegend=True,
    height = 600,
    width=800
)
fig.show()
```



1.9.1 Context of SHAP Analysis

The **SHAP (SHapley Additive exPlanations)** analysis aims to interpret the predictions of the insurance claim prediction model. By calculating feature contributions for individual instances and globally, SHAP helps to:

- Understand the influence of features on predictions.
- Identify key drivers of claim probability.
- Assist stakeholders in making data-driven decisions for risk assessment and premium setting.

1.9.2 SHAP Summary Plot

The SHAP summary values represent the average contribution of each feature to the model's predictions across all data points.

SHAP Value Contributions

- `Policy Tenure: -0.250`
  - Small negative contribution to predictions, indicating longer tenure slightly reduces claim likelihood.
- `Age of Car: 3.798`
  - Strong positive contribution; older cars increase the likelihood of a claim significantly.
- `Age of Policyholder: 2.599`
  - Strong positive contribution; older policyholders are more likely to file claims.
- `Population Density: -0.440`
  - Slight negative contribution; areas with higher population density are associated with a reduced claim probability.
- `Safety Score: -0.235`
  - Minimal negative contribution; higher safety scores slightly reduce claim likelihood.

- `Engine Risk: -0.022`
  - Small negative contribution; lower engine risk reduces claim probability slightly.
- `Area Cluster Risk: 0.295`
  - Moderate positive contribution; higher risk in the geographical area increases claim probability.
- `NCAP Risk: 0.019`
  - Moderate positive contribution; higher NCAP risk increases claim probability. #### Insights from the SHAP Summary Plot
- The `Age of Car` and `Age of Policyholder` are the most influential factors, contributing strongly to claim predictions.
- Safety-related features (e.g., `Safety Score`, `Engine Risk`) have minor contributions but align with expectations.
- Environmental factors like `Population Density` and `Area Cluster Risk` demonstrate their influence on claim predictions.

1.9.3 SHAP Force Plot

The `SHAP force plot` provides a detailed view of feature contributions for a specific instance. Each feature either pushes the prediction higher (positive contribution) or lower (negative contribution) relative to the model's baseline prediction.

Force Plot Contributions

- `Policy Tenure: 0.1308`
  - Slight positive push toward increasing claim likelihood.
- `Age of Car: 0.13`
  - Negligible contribution for this instance, despite being influential on average.
- `Age of Policyholder: 0.298`
  - Significant positive push, indicating that this policyholder's age increases the claim likelihood.
- `Population Density: 27003.0`
  - Strong positive contribution, suggesting a high population density is a major risk factor in this specific case.
- `Safety Score: 9.0`
  - Strong positive push, showing a high safety score unusually increases the claim probability in this instance.
- `Engine Risk: 0.1684`
  - Small positive contribution to claim likelihood.
- `Area Cluster Risk: 0.0706`
  - Negligible positive contribution for this instance.
- `NCAP Risk: 0.5`
  - Small positive contribution. #### Insights from the SHAP Force Plot
- The `Population Density` and `Safety Score` exhibit unusually high contributions in this specific instance, deviating from their average trends in the summary plot.
- `Age of Policyholder` consistently demonstrates a strong positive impact, aligning with the overall trend.

1.9.4 Summary of Analysis

- The SHAP summary plot provides a global perspective of feature importance across all data points, highlighting the major and minor contributors to claim predictions.
- The SHAP force plot offers a local perspective, focusing on feature contributions for an individual prediction.
- Features like Age of Car, Age of Policyholder, and Population Density emerge as critical drivers of claim likelihood, indicating areas for further investigation or focus by insurance companies.

[224]:
```
X_ = df[['policy_tenure', 'age_of_car', 'age_of_policyholder',␣
 ↪'population_density',
        'safety_score', 'engine_risk', 'area_cluster_risk', 'ncap_risk']]
df['claim_severity'] = (df['age_of_car'] * 0.1 +
                        df['age_of_policyholder'] * 0.05 +
                        df['population_density'] * 0.2 +
                        df['safety_score'] * -0.2 +
                        df['engine_risk'] * 0.3 +
                        df['area_cluster_risk'] * 0.15 +
                        df['ncap_risk'] * 0.25)
y_ = df['claim_severity']  # Synthetic target variable (claim severity)
```

1.9.5 Why Create a Synthetic Target Variable for Claim Severity?

Absence of a Direct Target Variable:

- In many real-world datasets, the target variable (in this case, claim_severity) might not be directly available or measurable. For example, your dataset might contain the features that influence claim severity (like age_of_car, engine_risk, safety_score), but not the actual severity value (e.g., the cost or damage amount of the claim).
- In such cases, you may simulate a target variable by combining multiple relevant features to create an approximation of claim_severity.

Understanding Feature Relationships:

- By defining a synthetic target (claim_severity) as a weighted combination of multiple features (like age_of_car, engine_risk, and others), I can understand how these features interact and contribute to the severity of claims.
- This allows me to evaluate whether my model can learn these relationships and predict outcomes that are closely aligned with real-world claim severities. #### Data Exploration and Experimentation:
- Creating a synthetic target variable helps me explore and experiment with model building, feature importance, and regression tasks.
- Since the target is artificially generated, the focus is on validating the model's ability to capture relationships between input features and the synthetic target rather than actual claim severity. This serves as a testing ground for my approach before applying it to real-world datasets.

Training and Testing the Model:

- In the absence of real claim severity data, the synthetic target allows me to train machine learning models (e.g., regression models) to learn patterns in the data.
- It enables me to test different algorithms and methods (like GradientBoostingRegressor or XGBoost) to assess their effectiveness at predicting claim severity based on the given features.
-
- The synthetic target variable can help validate the model's generalization capability. By evaluating the model's performance (e.g., R^2 score, mean squared error), you can gauge how well it predicts claim severity for future or unseen data.
- While the synthetic target isn't representative of actual claims, it serves as a proxy for testing model performance.

Refining Feature Importance:

- By examining the model's feature importance (e.g., using XGBoost or Gradient Boosting), I can gain insights into which features are most influential in determining the severity of claims.
- This helps me refine my understanding of which factors drive severity and assists in making data-driven business decisions, such as risk assessment and premium determination.

[226]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split

X_train_, X_test_, y_train_, y_test_ = train_test_split(X_, y_, test_size=0.2,␣
 ↪random_state=42)
gbr_model = GradientBoostingRegressor(random_state=42)
gbr_model.fit(X_train_, y_train_)
```

[226]:
```
GradientBoostingRegressor(random_state=42)
```

[228]:
```
y_pred_ = gbr_model.predict(X_test_)

mse = mean_squared_error(y_test_, y_pred_)
r2 = r2_score(y_test_, y_pred_)

print("Mean Squared Error:", mse)
print("R2 Score:", r2)
```

```
Mean Squared Error: 0.0823440006322034
R2 Score: 0.9999999933702841
```

[236]:
```
from sklearn.model_selection import cross_val_score
```

[ ]:
```
cv_scores = cross_val_score(gbr_model, X_, y_, cv=5,␣
 ↪scoring='neg_mean_squared_error')
mse_scores = -cv_scores

print(f"Cross-Validation MSE Scores: {mse_scores}")
print(f"Mean Cross-Validation MSE: {np.mean(mse_scores)}")
print(f"Standard Deviation of Cross-Validation MSE: {np.std(mse_scores)}")

cv_r2_scores = cross_val_score(gbr_model, X_, y_, cv=5, scoring='r2')
print(f"Cross-Validation R² Scores: {cv_r2_scores}")
print(f"Mean Cross-Validation R²: {np.mean(cv_r2_scores)}")
```

```
Cross-Validation MSE Scores: [0.08949608 0.08603639 0.09494984 0.08356681
0.10409986]
Mean Cross-Validation MSE: 0.09162979705987517
Standard Deviation of Cross-Validation MSE: 0.007312671859384482
Cross-Validation R² Scores: [0.99999999 0.99999999 0.99999999 0.99999999
0.99999999]
Mean Cross-Validation R²: 0.9999999926364838
```

- **Model Stability**: The model is stable, with a low variation in performance across the folds, which suggests that it is generalizing well to unseen data.
- **Good Model Fit**: A low average MSE and consistent results across folds indicate that your model is effectively predicting the `claim_severity` based on the given features.

[293]:
```
X_train_sub, _ , y_train_sub, _ = train_test_split(X_train_, y_train_,␣
 ↪train_size=0.3, random_state=42)
```

[269]:
```
from sklearn.model_selection import GridSearchCV
```

[295]:
```
param_grid_part1 = {
    'n_estimators': [100, 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 5],
    'loss': ['ls', 'huber']
}
```

[297]:
```
grid_search_part1 = GridSearchCV(
    estimator=gbr_model,
    param_grid=param_grid_part1,
    cv=5,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)
grid_search_part1.fit(X_train_sub, y_train_sub)
print("Part 1 Best Parameters:", grid_search_part1.best_params_)
```

```
Part 1 Best Parameters: {'learning_rate': 0.1, 'loss': 'huber', 'max_depth': 5,
'n_estimators': 200}
```

[299]:
```
param_grid_part2 = {
    'n_estimators': [200, 300, 400],
    'learning_rate': [0.1,0.2,0.5],
    'max_depth': [5, 7],
    'subsample': [0.8, 0.9, 1.0],
    'min_samples_split': [2, 5],
    'loss': ['huber']
}

grid_search_part2 = GridSearchCV(
    estimator=gbr_model,
    param_grid=param_grid_part2,
    cv=5,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)
grid_search_part2.fit(X_train_, y_train_)
print("Part 2 Best Parameters:", grid_search_part2.best_params_)
```

```
Part 2 Best Parameters: {'learning_rate': 0.1, 'loss': 'huber', 'max_depth': 5,
'min_samples_split': 5, 'n_estimators': 400, 'subsample': 0.9}
```

[230]:
```
best_gbr = GradientBoostingRegressor(learning_rate= 0.1,
                                     loss= 'huber',
                                     max_depth= 5,
                                     min_samples_split= 5,
                                     n_estimators= 400,
                                     subsample= 0.9,
                                     random_state=42)
best_gbr.fit(X_train_, y_train_)
```

[230]:
```
GradientBoostingRegressor(loss='huber', max_depth=5, min_samples_split=5,
                          n_estimators=400, random_state=42, subsample=0.9)
```

[232]:
```
y_pred_best = best_gbr.predict(X_test_)
mse_best = mean_squared_error(y_test_, y_pred_best)
r2_best = r2_score(y_test_, y_pred_best)

print("Final Model MSE:", mse_best)
print("Final Model R2:", r2_best)
```

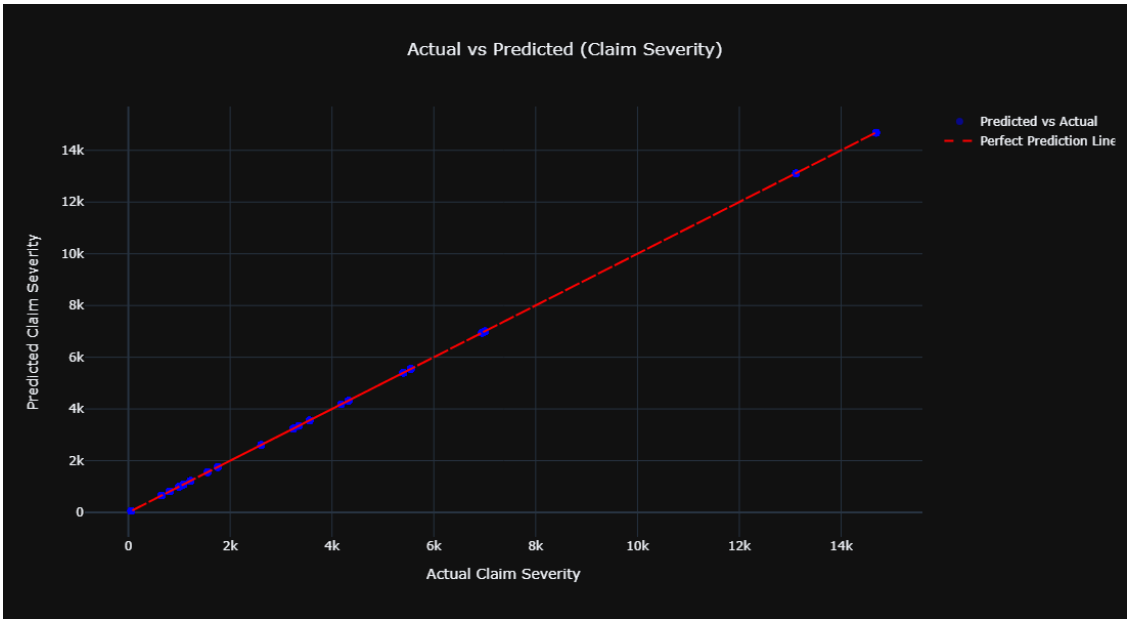
```
Final Model MSE: 0.11053612295234741
Final Model R2: 0.9999999911004678
```

[238]:
```
cv_scores_ = cross_val_score(best_gbr, X_, y_, cv=5,␣
 ↪scoring='neg_mean_squared_error')
mse_scores_ = -cv_scores_
```

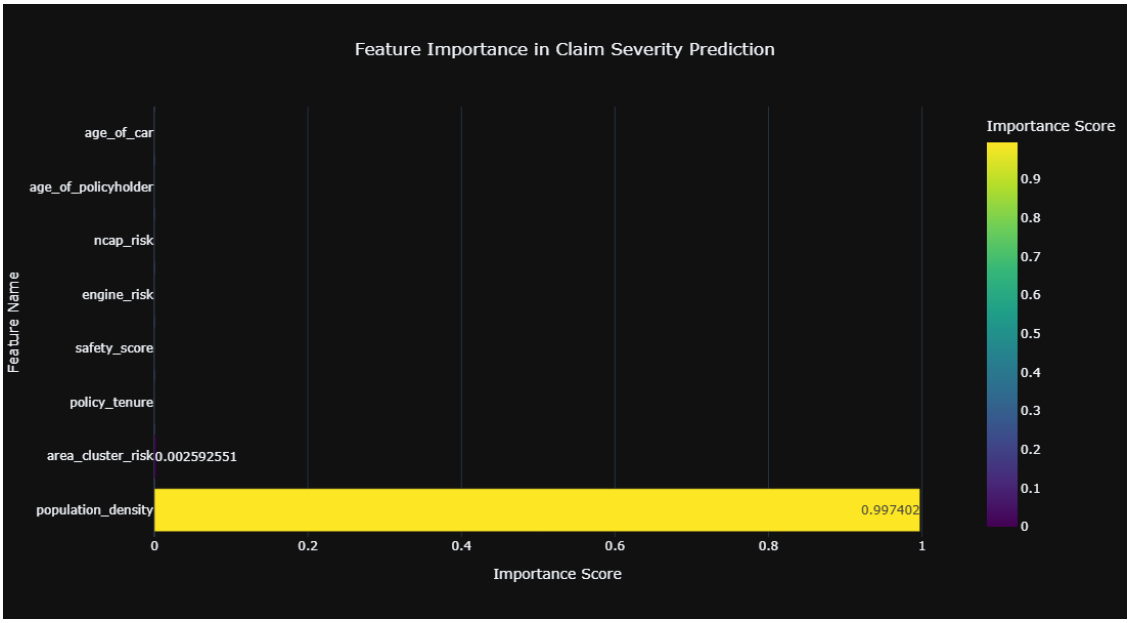
```
print(f"Cross-Validation MSE Scores: {mse_scores_}")
print(f"Mean Cross-Validation MSE: {np.mean(mse_scores_)}")
print(f"Standard Deviation of Cross-Validation MSE: {np.std(mse_scores_)}")

cv_r2_scores_ = cross_val_score(best_gbr, X_, y_, cv=5, scoring='r2')
print(f"Cross-Validation R² Scores: {cv_r2_scores_}")
print(f"Mean Cross-Validation R²: {np.mean(cv_r2_scores_)}")
```

```
Cross-Validation MSE Scores: [2.77458150e+00 6.10421791e-01 9.33210685e-03
1.08664446e+01
 2.19311446e-02]
Mean Cross-Validation MSE: 2.8565422357204864
Standard Deviation of Cross-Validation MSE: 4.131632524385112
Cross-Validation R² Scores: [0.99999979 0.99999995 1.         0.99999915 1.
]
Mean Cross-Validation R²: 0.9999997764941669
```

[240]:
```
fig = go.Figure()

fig.add_trace(go.Scatter(
    x=y_test_,
    y=y_pred_best,
    mode='markers',
    marker=dict(color='blue', opacity=0.5, size=7),
    name="Predicted vs Actual"
))
fig.add_trace(go.Scatter(
    x=y_test_,
    y=y_pred_best,
    mode='lines',
    line=dict(color='red', dash='dash'),
    name="Perfect Prediction Line"
))

fig.update_layout(
    title="Actual vs Predicted (Claim Severity)",
    title_x = 0.5,
    xaxis_title="Actual Claim Severity",
    yaxis_title="Predicted Claim Severity",
    template="plotly_dark",
    height=600,
    width=800
)

fig.show()
```



[242]:
```
feature_importances = best_gbr.feature_importances_

feature_df = pd.DataFrame({
    'Feature': X_.columns,
    'Importance': feature_importances
})

feature_df = feature_df.sort_values(by='Importance', ascending=False)
fig = px.bar(
    feature_df,
    x='Importance',
    y='Feature',
    text_auto= True,
    orientation='h',
    title="Feature Importance in Claim Severity Prediction",
    labels={'Importance': 'Importance Score', 'Feature': 'Feature Name'},
    color='Importance',
    color_continuous_scale='Viridis',
    height=600,
    width =800,
    template="plotly_dark"
)

fig.update_layout(title_x = 0.5)
fig.show()
```



[244]:
```
feature_df
```

[244]:

	Feature	Importance
3	population_density	9.974022e-01
6	area_cluster_risk	2.592551e-03
0	policy_tenure	5.095538e-06
4	safety_score	1.122131e-07
5	engine_risk	7.683842e-09
7	ncap_risk	3.919296e-10
2	age_of_policyholder	1.112184e-11
1	age_of_car	3.016954e-12

1.10 Business Recommendations

1. **Refined Risk Assessment for Policyholders**

- **Action:** Use the predictive model to identify high-risk policyholders based on features like `age_of_car`, `area_cluster_risk`, and `engine_risk`.
- **Impact:** Enable more accurate risk profiling, reducing financial losses from unexpected claims.

2. **Premium Adjustment for Policyholders**

- **Action:** Leverage the claim probability and severity scores to determine fair and competitive premiums.
  - `High-risk policyholders:` Charge higher premiums.
  - `Low-risk policyholders:` Offer discounts or incentives for renewal.
- **Impact:** Improves customer satisfaction by ensuring fairness in pricing while protecting profitability.

3. **Customized Marketing Strategies**

- **Action:** Focus marketing campaigns in areas with high population_density or specific area_cluster regions identified as high-risk.
- **Impact:** Targeted efforts can increase policy sales in underinsured regions while mitigating risk by setting appropriate premiums.

4. **Enhanced Policy Design**

- **Action:** Introduce policy features tailored for high-risk customers,
  - such as: Optional higher deductibles to reduce premium costs.
  - Roadside assistance packages or vehicle monitoring systems to reduce the likelihood of claims.
- **Impact:** Improve customer retention and attract price-sensitive customers.

5. **Safety Improvement Initiatives**

- **Action:** Partner with local authorities or customers to improve vehicle and driver safety, addressing factors like safety_score and engine_risk.
  - Promote road safety campaigns and offer rewards for safe driving.
- **Impact:** Reduce claim frequency, improving profitability and brand image.

6. **Fraud Detection Enhancement**

- **Action:** Use claim probability data to flag potential fraudulent claims based on unusual feature combinations (e.g., very high population_density but low engine_risk).
- **Impact:** Prevent financial losses and strengthen the company's ability to handle fraud efficiently.

7. **Vehicle Feature-Based Policies**

- **Action:** Offer discounts or incentives for vehicles with features that lower claim risks, such as is_esc, is_parking_camera, or is_brake_assist.
- **Impact:** Encourage policyholders to opt for safer cars, reducing overall claims.

8. **Portfolio Optimization**

- **Action:** Analyze claim severity and frequency to identify profitable customer segments.
- **Impact:** Focus on acquiring more customers from these segments while carefully managing exposure to high-risk groups.

9. **Monitoring and Continuous Improvement**

- **Action:** Regularly monitor model predictions and real-world claim outcomes to fine-tune the model and improve accuracy over time.
- **Impact:** Adapt to changing market dynamics and ensure consistent performance.

10. **Long-Term Planning**

- **Action:** Develop strategies to address macroeconomic trends, such as urbanization (increasing population_density) and changing vehicle technologies.
- **Impact:** Stay competitive by preparing for future risks and opportunities.

[ ]: