안건하-전소영-고효진-오예진

Github: <https://github.com/Hyojinko/2021_datascience.git>

#end-to-end process

|  |  |
| --- | --- |
|  | Import warnings  warnings.filterwarnings("ignore") |
|  | import numpy as np |
|  | import pandas as pd |
|  | import matplotlib.pyplot as plt |
|  | import seaborn as sns |
|  | from sklearn import preprocessing |
|  | from sklearn.neighbors import KNeighborsRegressor |
|  | from sklearn.model\_selection import train\_test\_split |
|  | from sklearn.ensemble import BaggingRegressor |
|  | from sklearn.ensemble import RandomForestRegressor |
|  | from sklearn.linear\_model import LinearRegression |
|  | from sklearn.preprocessing import PolynomialFeatures |
|  | from sklearn.cluster import KMeans |
|  | from sklearn.ensemble import BaggingClassifier |
|  | from sklearn.neighbors import KNeighborsClassifier |
|  | from sklearn.ensemble import RandomForestClassifier |
|  | from sklearn.model\_selection import GridSearchCV |
|  | from sklearn.tree import DecisionTreeRegressor |
|  |  |
|  | pd.set\_option('display.max\_row', 100) |
|  | pd.set\_option('display.max\_columns', 100) |
|  |  |
|  | # Read csv |
|  | df = pd.read\_excel('IT\_3.xlsx') |
|  | df.fillna(axis=0, method='ffill', inplace=True) |
|  |  |
|  | df1 = pd.DataFrame(df) |
|  | # # Drop 5 columns |
|  | df1 = df1.drop(['Miles\_driven\_annually', 'Years\_Experience', 'EngineHP', 'credit\_history'], axis=1) |
|  | # Drop rows with Nan |
|  |  |
|  | # Fill missing values |
|  | df1['annual\_claims']=df1['annual\_claims'].astype(np.int64) |
|  | df1['annual\_claims']=df['annual\_claims'].astype('category') |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | # #make target to category |
|  | # targets = df['annual\_claims'].astype(np.int64) |
|  | # targets = targets.astype('category') |
|  |  |
|  |  |
|  | df\_ordinal = df1.copy() |
|  | df\_oneHot = df1.copy() |
|  | df\_label = df1.copy() |
|  |  |
|  | # Convert 'Marital\_Status' feature to numeric values using ordinalEncoder |
|  | ordinalEncoder = preprocessing.OrdinalEncoder() |
|  | X = pd.DataFrame(df1['Marital\_Status']) |
|  | ordinalEncoder.fit(X) |
|  | df\_ordinal['Marital\_Status'] = pd.DataFrame(ordinalEncoder.transform(X)) |
|  |  |
|  | # Convert 'Marital\_Status' feature to numeric values using labelEncoder |
|  | labelEncoder = preprocessing.LabelEncoder() |
|  | labelEncoder.fit(df\_label['Marital\_Status']) |
|  | df\_label['Marital\_Status'] = labelEncoder.transform(df\_label['Marital\_Status']) |
|  |  |
|  |  |
|  | # Convert 'Vehical\_type' feature to numeric values using ordinalEncoder |
|  | ordEnc = preprocessing.OrdinalEncoder() |
|  | X = pd.DataFrame(df1['Vehical\_type']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['Vehical\_type'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'Vehical\_type' feature to numeric values using labelEncoder |
|  | labelEnc = preprocessing.LabelEncoder() |
|  | labelEnc.fit(df1['Vehical\_type']) |
|  | df\_label['Vehical\_type'] = pd.DataFrame(labelEnc.transform(df1['Vehical\_type'])) |
|  |  |
|  |  |
|  | # Convert 'Gender' features to numeric values using ordinalEncoder |
|  | X = pd.DataFrame(df1['Gender']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['Gender'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'Gender' features to numeric values using labelEncoder |
|  | labelEnc.fit(X) |
|  | df\_label['Gender'] = pd.DataFrame(labelEnc.transform(X)) |
|  |  |
|  |  |
|  | # Convert 'State' features to numeric values using ordinalEncoder |
|  | X = pd.DataFrame(df1['State']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['State'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'State' features to numeric values using labelEncoder |
|  | labelEnc.fit(df1['State']) |
|  | df\_label['State'] = pd.DataFrame(labelEnc.transform(df1['State'])) |
|  |  |
|  |  |
|  | # Convert 'Age\_bucket' features to numeric values using ordinalEncoder |
|  | X = pd.DataFrame(df1['Age\_bucket']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['Age\_bucket'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'Age\_bucket' features to numeric values using labelEncoder |
|  | labelEnc.fit(df1['Age\_bucket']) |
|  | df\_label['Age\_bucket'] = pd.DataFrame(labelEnc.transform(df1['Age\_bucket'])) |
|  |  |
|  |  |
|  | # Convert 'EngineHP\_bucket' features to numeric values using ordinalEncoder |
|  | X = pd.DataFrame(df1['EngineHP\_bucket']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['EngineHP\_bucket'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'EngineHP\_bucket' features to numeric values using labelEncoder |
|  | labelEnc.fit(df1['EngineHP\_bucket']) |
|  | df\_label['EngineHP\_bucket'] = pd.DataFrame(labelEnc.transform(df1['EngineHP\_bucket'])) |
|  |  |
|  |  |
|  | # Convert 'Years\_Experience\_bucket' features to numeric values using ordinalEncoder |
|  | X = pd.DataFrame(df1['Years\_Experience\_bucket']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['Years\_Experience\_bucket'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'Years\_Experience\_bucket' features to numeric values using labelEncoder |
|  | labelEnc.fit(df1['Years\_Experience\_bucket']) |
|  | df\_label['Years\_Experience\_bucket'] = pd.DataFrame(labelEnc.transform(df1['Years\_Experience\_bucket'])) |
|  |  |
|  |  |
|  | # Convert 'Miles\_driven\_annually\_bucket' features to numeric values using ordinalEncoder |
|  | X = pd.DataFrame(df1['Miles\_driven\_annually\_bucket']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['Miles\_driven\_annually\_bucket'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'Miles\_driven\_annually\_bucket' features to numeric values using labelEncoder |
|  | labelEnc.fit(df1['Miles\_driven\_annually\_bucket']) |
|  | df\_label['Miles\_driven\_annually\_bucket'] = pd.DataFrame(labelEnc.transform(df1['Miles\_driven\_annually\_bucket'])) |
|  |  |
|  |  |
|  | # Convert 'credit\_history\_bucket' features to numeric values using ordinalEncoder |
|  | X = pd.DataFrame(df1['credit\_history\_bucket']) |
|  | ordEnc.fit(X) |
|  | df\_ordinal['credit\_history\_bucket'] = pd.DataFrame(ordEnc.transform(X)) |
|  |  |
|  | # Convert 'credit\_history\_bucket' features to numeric values using labelEncoder |
|  | labelEnc.fit(df1['credit\_history\_bucket']) |
|  | df\_label['credit\_history\_bucket'] = pd.DataFrame(labelEnc.transform(df1['credit\_history\_bucket'])) |
|  |  |
|  | print("=======isNan") |
|  | print(df\_label.isnull().sum()) |
|  | # 인코딩 거치면 null 값이 생깁니다.. |
|  | df\_label = df\_label.dropna() |
|  |  |
|  | df\_oneHot['annual\_claims']=df\_oneHot['annual\_claims'].astype(np.int64) |
|  |  |
|  | # Getting all the categorical variables in a list |
|  | categoricalColumn = df1.columns[df1.dtypes == np.object].tolist() |
|  | # Convert categorical features to numeric values using oneHotEncoder |
|  | for col in categoricalColumn: |
|  | if(len(df\_oneHot[col].unique()) == 2): |
|  | df\_oneHot[col] = pd.get\_dummies(df\_oneHot[col], drop\_first=True) |
|  |  |
|  | df\_oneHot = pd.get\_dummies(df\_oneHot) |
|  |  |
|  | df1['annual\_claims']=df1['annual\_claims'].astype('category') |
|  | y = df1['annual\_claims'] |
|  | y2 = df1['annual\_claims'] |
|  | # Split the dataset |
|  |  |
|  | X1 = df\_ordinal.drop(['annual\_claims'], 1) |
|  | X2 = df\_label.drop(['annual\_claims'], 1) |
|  | X3 = df\_oneHot.drop(['annual\_claims'], 1) |
|  |  |
|  |  |
|  |  |
|  | X1\_train, X1\_test, y1\_train, y1\_test = train\_test\_split(X1, y, random\_state=0) |
|  | X2\_train, X2\_test, y2\_train, y2\_test = train\_test\_split(X2, y, random\_state=0) |
|  | X3\_train, X3\_test, y3\_train, y3\_test = train\_test\_split(X3, y, random\_state=0) |
|  |  |
|  |  |
|  |  |
|  | # Normalizing the ordinalEncoded dataset using MaxAbsScaler |
|  | scaler = preprocessing.MaxAbsScaler() |
|  | df\_ordinal\_maxAbs\_train = scaler.fit\_transform(X1\_train) |
|  | df\_ordinal\_maxAbs\_train = pd.DataFrame(df\_ordinal\_maxAbs\_train, columns=X1\_train.columns) |
|  |  |
|  | df\_ordinal\_maxAbs\_test = scaler.fit\_transform(X1\_test) |
|  | df\_ordinal\_maxAbs\_test = pd.DataFrame(df\_ordinal\_maxAbs\_test, columns=X1\_test.columns) |
|  | print(df\_ordinal\_maxAbs\_train.head(10)) |
|  |  |
|  |  |
|  |  |
|  | # Normalizing the oneHotEncoded dataset using MaxAbsScaler |
|  | scaler = preprocessing.MaxAbsScaler() |
|  | df\_oneHot\_maxAbs\_train = scaler.fit\_transform(X3\_train) |
|  | df\_oneHot\_maxAbs\_train = pd.DataFrame(df\_oneHot\_maxAbs\_train, columns=X3\_train.columns) |
|  |  |
|  | df\_oneHot\_maxAbs\_test = scaler.fit\_transform(X3\_test) |
|  | df\_oneHot\_maxAbs\_test = pd.DataFrame(df\_oneHot\_maxAbs\_test, columns=X3\_test.columns) |
|  | print(df\_oneHot\_maxAbs\_test.head(10)) |
|  |  |
|  |  |
|  |  |
|  | # Normalizing the labelEncoded dataset using MaxAbsScaler |
|  | scaler = preprocessing.MaxAbsScaler() |
|  | df\_label\_maxAbs\_train = scaler.fit\_transform(X2\_train) |
|  | df\_label\_maxAbs\_train = pd.DataFrame(df\_label\_maxAbs\_train, columns=X2\_train.columns) |
|  |  |
|  | df\_label\_maxAbs\_test = scaler.fit\_transform(X2\_test) |
|  | df\_label\_maxAbs\_test = pd.DataFrame(df\_label\_maxAbs\_test, columns=X2\_test.columns) |
|  | print(df\_label\_maxAbs\_test.head(10)) |
|  |  |
|  |  |
|  | # Normalizing the ordinalEncoded dataset using RobustScaler |
|  | scaler = preprocessing.RobustScaler() |
|  | df\_ordinal\_robust\_train = scaler.fit\_transform(X1\_train) |
|  | df\_ordinal\_robust\_train = pd.DataFrame(df\_ordinal\_robust\_train, columns=X1\_train.columns) |
|  |  |
|  | df\_ordinal\_robust\_test = scaler.fit\_transform(X1\_test) |
|  | df\_ordinal\_robust\_test = pd.DataFrame(df\_ordinal\_robust\_test, columns=X1\_test.columns) |
|  | print(df\_ordinal\_robust\_test.head(10)) |
|  |  |
|  |  |
|  |  |
|  | # Normalizing the oneHotEncoded dataset using RobustScaler |
|  | scaler = preprocessing.RobustScaler() |
|  | df\_oneHot\_robust\_train = scaler.fit\_transform(X3\_train) |
|  | df\_oneHot\_robust\_train = pd.DataFrame(df\_oneHot\_robust\_train, columns=X3\_train.columns) |
|  |  |
|  | df\_oneHot\_robust\_test = scaler.fit\_transform(X3\_test) |
|  | df\_oneHot\_robust\_test = pd.DataFrame(df\_oneHot\_robust\_test, columns=X3\_test.columns) |
|  | print(df\_oneHot\_robust\_test.head(10)) |
|  |  |
|  |  |
|  |  |
|  | # Normalizing the labelEncoded dataset using RobustScaler |
|  | scaler = preprocessing.RobustScaler() |
|  | df\_label\_robust\_train = scaler.fit\_transform(X2\_train) |
|  | df\_label\_robust\_train = pd.DataFrame(df\_label\_robust\_train, columns=X2\_train.columns) |
|  |  |
|  | df\_label\_robust\_test = scaler.fit\_transform(X2\_test) |
|  | df\_label\_robust\_test = pd.DataFrame(df\_label\_robust\_test, columns=X2\_test.columns) |
|  | print(df\_label\_robust\_test.head(10)) |
|  |  |
|  |  |
|  |  |
|  | # Normalizing the ordinalEncoded dataset using MinMaxScaler |
|  | scaler = preprocessing.MinMaxScaler() |
|  | df\_ordinal\_minMax\_train = scaler.fit\_transform(X1\_train) |
|  | df\_ordinal\_minMax\_train = pd.DataFrame(df\_ordinal\_minMax\_train, columns=X1\_train.columns) |
|  |  |
|  |  |
|  | df\_ordinal\_minMax\_test = scaler.fit\_transform(X1\_test) |
|  | df\_ordinal\_minMax\_test = pd.DataFrame(df\_ordinal\_minMax\_test, columns=X1\_test.columns) |
|  | print(df\_ordinal\_minMax\_test.head(10)) |
|  |  |
|  | # Normalizing the oneHotEncoded dataset using MinMaxScaler |
|  | scaler = preprocessing.MinMaxScaler() |
|  | df\_oneHot\_minMax\_train = scaler.fit\_transform(X3\_train) |
|  | df\_oneHot\_minMax\_train = pd.DataFrame(df\_oneHot\_minMax\_train, columns=X3\_train.columns) |
|  |  |
|  | df\_oneHot\_minMax\_test = scaler.fit\_transform(X3\_test) |
|  | df\_oneHot\_minMax\_test = pd.DataFrame(df\_oneHot\_minMax\_test, columns=X3\_test.columns) |
|  | print(df\_oneHot\_minMax\_test.head(10)) |
|  |  |
|  | # Normalizing the labelEncoded dataset using MinMaxScaler |
|  | scaler = preprocessing.MinMaxScaler() |
|  | df\_label\_minMax\_train = scaler.fit\_transform(X2\_train) |
|  | df\_label\_minMax\_train = pd.DataFrame(df\_label\_minMax\_train, columns=X2\_train.columns) |
|  |  |
|  | df\_label\_minMax\_test = scaler.fit\_transform(X2\_test) |
|  | df\_label\_minMax\_test = pd.DataFrame(df\_label\_minMax\_test, columns=X2\_test.columns) |
|  | print(df\_label\_minMax\_test.head(10)) |
|  |  |
|  | # Normalizing the ordinalEncoded dataset using StandardScaler |
|  | scaler = preprocessing.StandardScaler() |
|  | df\_ordinal\_stand\_train = scaler.fit\_transform(X1\_train) |
|  | df\_ordinal\_stand\_train = pd.DataFrame(df\_ordinal\_stand\_train, columns=X1\_train.columns) |
|  |  |
|  | df\_ordinal\_stand\_test = scaler.fit\_transform(X1\_test) |
|  | df\_ordinal\_stand\_test = pd.DataFrame(df\_ordinal\_stand\_test, columns=X1\_test.columns) |
|  | print(df\_ordinal\_stand\_test.head(10)) |
|  |  |
|  | # Normalizing the oneHotEncoded dataset using StandardScaler |
|  | scaler = preprocessing.StandardScaler() |
|  | df\_oneHot\_stand\_train = scaler.fit\_transform(X3\_train) |
|  | df\_oneHot\_stand\_train = pd.DataFrame(df\_oneHot\_stand\_train, columns=X3\_train.columns) |
|  |  |
|  | df\_oneHot\_stand\_test = scaler.fit\_transform(X3\_test) |
|  | df\_oneHot\_stand\_test = pd.DataFrame(df\_oneHot\_stand\_test, columns=X3\_test.columns) |
|  | print(df\_oneHot\_stand\_test.head(10)) |
|  |  |
|  | # Normalizing the labelEncoded dataset using StandardScaler |
|  | scaler = preprocessing.StandardScaler() |
|  | df\_label\_stand\_train = scaler.fit\_transform(X2\_train) |
|  | df\_label\_stand\_train = pd.DataFrame(df\_label\_stand\_train, columns=X2\_train.columns) |
|  |  |
|  | df\_label\_stand\_test = scaler.fit\_transform(X2\_test) |
|  | df\_label\_stand\_test = pd.DataFrame(df\_label\_stand\_test, columns=X2\_test.columns) |
|  | print(df\_label\_stand\_test.head(10)) |
|  |  |
|  |  |
|  |  |
|  |  |
|  | #bar graph of target |
|  | sns.countplot(df['target']) |
|  | plt.show() |
|  |  |
|  | #bar graph of Gender |
|  | sns.countplot(df['Gender']) |
|  | plt.show() |
|  |  |
|  | #bar graph of annual\_claims |
|  | sns.countplot(df['annual\_claims']) |
|  | plt.show() |
|  |  |
|  | #bar graph of Marital\_Status |
|  | sns.countplot(df['Marital\_Status']) |
|  | plt.show() |
|  |  |
|  | #bar graph of Vehical\_type |
|  | sns.countplot(df['Vehical\_type']) |
|  | #plt.show() |
|  |  |
|  |  |
|  | #bar graph of size\_of\_family |
|  | sns.countplot(df['size\_of\_family']) |
|  | #plt.show() |
|  |  |
|  | #bar graph of State |
|  | sns.countplot(df['State']) |
|  | plt.show() |
|  |  |
|  | #Two variable plots |
|  | df\_f = pd.DataFrame(df[df['Gender']=='F']) |
|  | df\_m = pd.DataFrame(df[df['Gender']=='M']) |
|  |  |
|  |  |
|  | #EngineHP histogram by Gender |
|  | plt.hist(df\_f['EngineHP'], bins=10) |
|  | plt.title('Female') |
|  | plt.xlabel('EngineHP') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | plt.hist(df\_m['EngineHP'], bins=10) |
|  | plt.title('Male') |
|  | plt.xlabel('EngineHP') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | #credit history histogram by Gender |
|  | plt.hist(df\_m['credit\_history'], bins=10) |
|  | plt.title('Male') |
|  | plt.xlabel('credit\_history') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | plt.hist(df\_f['credit\_history'], bins=10) |
|  | plt.title('Female') |
|  | plt.xlabel('credit\_history') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  |  |
|  | #years experience histogram by Gender |
|  |  |
|  | plt.hist(df\_f['Years\_Experience'], bins=10) |
|  | plt.title('Female') |
|  | plt.xlabel('Years\_Experience') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | plt.hist(df\_m['Years\_Experience'], bins=10) |
|  | plt.title('Male') |
|  | plt.xlabel('Years\_Experience') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | #annual claims histogram by Gender |
|  |  |
|  |  |
|  | plt.hist(df\_m['annual\_claims'], bins=10) |
|  | plt.title('Male') |
|  | plt.xlabel('annual\_claims') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | plt.hist(df\_f['annual\_claims'], bins=10) |
|  | plt.title('Female') |
|  | plt.xlabel('annual\_claims') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  | #Miles driven annually histogram by Gender |
|  |  |
|  | plt.hist(df\_f['Miles\_driven\_annually'], bins=10) |
|  | plt.title('Female') |
|  | plt.xlabel('Miles\_driven\_annually') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | plt.hist(df\_m['Miles\_driven\_annually'], bins=10) |
|  | plt.title('Male') |
|  | plt.xlabel('Miles\_driven\_annually') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | #Size of family histogram by Gender |
|  |  |
|  | plt.hist(df\_f['size\_of\_family'], bins=10) |
|  | plt.title('Female') |
|  | plt.xlabel('size\_of\_family') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | plt.hist(df\_m['size\_of\_family'], bins=10) |
|  | plt.title('Male') |
|  | plt.xlabel('size\_of\_family') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | #State histogram by Gender |
|  |  |
|  | plt.hist(df\_f['State'], bins=10) |
|  | plt.title('Female') |
|  | plt.xlabel('State') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  | plt.hist(df\_m['State'], bins=10) |
|  | plt.title('Male') |
|  | plt.xlabel('State') |
|  | plt.ylabel('Frequency') |
|  | plt.show() |
|  |  |
|  |  |
|  | #Pie chart of vehicle type |
|  | car = pd.DataFrame(df[df['Vehical\_type']=='Car']) |
|  | van = pd.DataFrame(df[df['Vehical\_type']=='Van']) |
|  | truck = pd.DataFrame(df[df['Vehical\_type']=='Truck']) |
|  | utility = pd.DataFrame(df[df['Vehical\_type']=='Utility']) |
|  |  |
|  | langs = ['Car','Van','Truck','Utility'] |
|  | vehicle\_level = [len(car), len(van), len(truck),len(utility)] |
|  | plt.pie(vehicle\_level,labels=langs,autopct='%1.2f%%') |
|  | plt.show() |
|  |  |
|  | #Pie chart of annual\_claim |
|  | ann\_0 = pd.DataFrame(df[df['annual\_claims']==0]) |
|  | ann\_1 = pd.DataFrame(df[df['annual\_claims']==1]) |
|  | ann\_2 = pd.DataFrame(df[df['annual\_claims']==2]) |
|  | ann\_3 = pd.DataFrame(df[df['annual\_claims']==3]) |
|  | ann\_4 = pd.DataFrame(df[df['annual\_claims']==4]) |
|  |  |
|  | langs = ['0','1','2','3','4'] |
|  | claim\_level = [len(ann\_0), len(ann\_1), len(ann\_2),len(ann\_3),len(ann\_4)] |
|  | plt.pie(claim\_level,labels=langs,autopct='%1.2f%%') |
|  | plt.title('annual claim levels') |
|  | plt.show() |
|  |  |
|  |  |
|  | #Pie chart of size of family |
|  | fam1 = pd.DataFrame(df[df['size\_of\_family']==1]) |
|  | fam2 = pd.DataFrame(df[df['size\_of\_family']==2]) |
|  | fam3 = pd.DataFrame(df[df['size\_of\_family']==3]) |
|  | fam4 = pd.DataFrame(df[df['size\_of\_family']==4]) |
|  | fam5 = pd.DataFrame(df[df['size\_of\_family']==5]) |
|  | fam6 = pd.DataFrame(df[df['size\_of\_family']==6]) |
|  | fam7 = pd.DataFrame(df[df['size\_of\_family']==7]) |
|  | fam8 = pd.DataFrame(df[df['size\_of\_family']==8]) |
|  |  |
|  |  |
|  | langs = ['1','2','3','4','5','6','7','8'] |
|  | family\_level = [len(fam1), len(fam2), len(fam3),len(fam4),len(fam5),len(fam6),len(fam7),len(fam8)] |
|  | plt.pie(family\_level,labels=langs,autopct='%1.2f%%') |
|  | plt.title('famliy size levels') |
|  | plt.show() |
|  |  |
|  | #EngineHP and crdit history scatter plot divided by Gender |
|  |  |
|  | plt.scatter(df\_f['EngineHP'], df\_f['credit\_history'],color='r') |
|  | plt.scatter(df\_m['EngineHP'], df\_m['credit\_history'],color='b') |
|  | plt.xlabel('EngineHP') |
|  | plt.ylabel('Credit history') |
|  | plt.title('Scatter plot divided by Gender') |
|  | plt.show() |
|  |  |
|  | #Years\_Experience and annual claims scatter plot divided by Gender |
|  |  |
|  | plt.scatter(df\_f['Years\_Experience'], df\_f['annual\_claims'],color='r') |
|  | plt.scatter(df\_m['Years\_Experience'], df\_m['annual\_claims'],color='b') |
|  | plt.xlabel('Years\_Experience') |
|  | plt.ylabel('Annual claims ') |
|  | plt.title('Scatter plot divided by Gender') |
|  | plt.show() |
|  |  |
|  | df['annual\_claims']=df1['annual\_claims'].astype(np.int64) |
|  | #Boxplot of annual claims divided by gender |
|  | sns.boxplot(df['Gender'],df['annual\_claims']) |
|  | plt.show() |
|  |  |
|  | #Boxplot of EngineHP divided by gender |
|  | sns.boxplot(df['Gender'],df['EngineHP']) |
|  | plt.show() |
|  |  |
|  |  |
|  | #Boxplot of Years Experience divided by gender |
|  | sns.boxplot(df['Gender'],df['Years\_Experience']) |
|  | plt.show() |
|  |  |
|  | #Boxplot of size of family divided by gender |
|  | sns.boxplot(df['Gender'],df['size\_of\_family']) |
|  | plt.show() |
|  |  |
|  | #heatmap-pearson |
|  | sns.heatmap(df.corr(method='pearson')) |
|  | plt.title("pearson") |
|  | plt.show() |
|  |  |
|  | # Correlation of all features |
|  | plt.figure(figsize=(15,15)) |
|  | sns.heatmap(df.corr(method='pearson'), annot=True, fmt='.2f', linewidths=5, cmap='Blues') |
|  | plt.title("Correlation of all features") |
|  | plt.show() |
|  |  |
|  |  |
|  |  |
|  | # # Rename 'target' and 'annual\_claims' features |
|  | # df2.rename(columns = {'target':'claim\_prediction', 'annual\_claims':'target'}, inplace=True) |
|  |  |
|  | # Drop 'ID' feature |
|  |  |
|  |  |
|  | X\_train = df\_label\_stand\_train |
|  | X\_test = df\_label\_stand\_test |
|  | y\_train = y2\_train |
|  | y\_test = y2\_test |
|  |  |
|  | print(X\_test) |
|  | print(X\_train) |
|  | # |
|  |  |
|  |  |
|  |  |
|  | """ |
|  | model = BaggingClassifier() #grid search 해야함 |
|  | params = {'n\_estimators': [100,125,150], |
|  | 'max\_features': [0.1,0.4, 0.5,1], |
|  | 'max\_samples':[0.1, 0.2, 0.3,0.5,1] |
|  | }; |
|  |  |
|  | print("\n---------- Bagging classifier grid search ----------") |
|  | model\_gscv = GridSearchCV(model,param\_grid = params,cv=5,scoring='accuracy') |
|  | model\_gscv.fit(X\_train,y\_train) |
|  | print("Best param : ",model\_gscv.best\_params\_) |
|  | print("Best score : ",model\_gscv.best\_score\_) |
|  | prediction = model\_gscv.predict(X\_test) |
|  | print(model\_gscv.score(X\_test,y\_test)) |
|  |  |
|  |  |
|  | # Using KNN algorithm |
|  | # Create and train a KNN classifier |
|  | knn = KNeighborsClassifier(n\_neighbors=5) |
|  | knn.fit(X\_train, y\_train) |
|  |  |
|  | y\_prec = knn.predict(X\_test) |
|  | print("\n---------- KNN classifier ----------") |
|  | print(y\_prec[0:100]) |
|  | print("Score: %.2f" % knn.score(X\_test, y\_test)) |
|  |  |
|  |  |
|  | rfModel = RandomForestClassifier() |
|  | params = {'n\_estimators': [100,125,150], |
|  | 'max\_depth': [2,4,6,8], |
|  | 'max\_features': [0.1,0.4, 0.5,1], |
|  | 'max\_samples':[0.1, 0.2, 0.3,0.5,1] |
|  | }; |
|  | print("\n----------Random Forest classifier grid search ----------") |
|  | rfModel\_gscv = GridSearchCV(rfModel,params,scoring = 'r2') |
|  | rfModel\_gscv.fit(X\_train,y\_train) |
|  | y\_predict = rfModel\_gscv.predict(X\_test) |
|  | print("Best param : ",rfModel\_gscv.best\_params\_) |
|  | print("Best score : ",rfModel\_gscv.best\_score\_) |
|  | print(rfModel\_gscv.score(X\_test,y\_test)) |
|  | print("\n\n\n") |
|  |  |
|  | """ |
|  | y\_test = y\_test.astype(np.int64) |
|  | y\_train = y\_train.astype(np.int) |
|  | """ |
|  | models = BaggingRegressor() |
|  | params = {'n\_estimators': [100,125,150], |
|  | 'max\_features': [0.1,0.4, 0.5,1], |
|  | 'max\_samples':[0.1, 0.2, 0.3,0.5,1] |
|  | }; |
|  |  |
|  | print("\n---------- Bagging regressor grid search ----------") |
|  | models\_gscv = GridSearchCV(models,param\_grid = params,cv=5,scoring='r2') |
|  | models\_gscv.fit(X\_train,y\_train) |
|  | print("Best param : ",models\_gscv.best\_params\_) |
|  | print("Best score : ",models\_gscv.best\_score\_) |
|  | print(models\_gscv.score(X\_test,y\_test)) |
|  |  |
|  |  |
|  | # Using KNN algorithm |
|  | # Create and train a KNN classifier |
|  | knnR = KNeighborsRegressor(n\_neighbors=5) |
|  | knnR.fit(X\_train, y\_train) |
|  | print("\n---------- KNN regressor----------") |
|  | print("Score: %.2f" % knnR.score(X\_test, y\_test)) |
|  |  |
|  |  |
|  | rfModelR = RandomForestRegressor() |
|  | params = {'n\_estimators': [100,125,150], |
|  | 'max\_depth': [2,4,6,8], |
|  | 'max\_features': [0.1,0.4, 0.5,1], |
|  | 'max\_samples':[0.1, 0.2, 0.3,0.5,1] |
|  | }; |
|  | print("\n----------Random Forest regressor grid search ----------") |
|  | rfModelR\_gscv = GridSearchCV(rfModelR,params,scoring = 'r2') |
|  | rfModelR\_gscv.fit(X\_train,y\_train) |
|  | print("Best param : ",rfModelR\_gscv.best\_params\_) |
|  | print("Best score : ",rfModelR\_gscv.best\_score\_) |
|  | print(rfModelR\_gscv.score(X\_test,y\_test)) |
|  | print("\n\n\n") |
|  | """ |
|  | # # LinearRegression |
|  | line\_reg = LinearRegression(); |
|  | line\_reg.fit(X\_train, y\_train) |
|  | y\_predict = line\_reg.predict(X\_test) |
|  | print("\n---------- LinearRegression ----------") |
|  | print("y\_predict: \n", y\_predict) |
|  | print("Score: %.2f" % line\_reg.score(X\_test, y\_test)) |
|  | # |
|  | # # Polynomial Regression |
|  | poly\_reg = PolynomialFeatures(degree=2) |
|  | X\_poly\_train = poly\_reg.fit\_transform(X\_train) |
|  | X\_poly\_test = poly\_reg.fit\_transform(X\_test) |
|  | pol\_reg = LinearRegression() |
|  | pol\_reg.fit(X\_poly\_train, y\_train) |
|  | y\_predict = line\_reg.predict(X\_test) |
|  | print("\n---------- Polynomial algorithm ----------") |
|  | print("y\_predict: \n", y\_predict) |
|  | print("Score: %.2f" % pol\_reg.score(X\_poly\_test, y\_test)) |
|  |  |
|  |  |
|  |  |
|  | X = X2 |
|  | y = y2 |
|  |  |
|  | X\_train,X\_test,y\_train,y\_test = train\_test\_split(X, y, random\_state=0) |
|  |  |
|  | X\_train2 = np.array(X\_train) |
|  | X\_test2 = np.array(X\_test) |
|  | y\_test2 = np.array(y\_test) |
|  |  |
|  | #kmeans cluster |
|  | kmeans = KMeans(n\_clusters=2) |
|  | kmeans.fit(X\_train2) |
|  | correct = 0 |
|  | for i in range(len(X\_test2)): |
|  | predict\_me = np.array(X\_test2[i].astype(float)) |
|  | predict\_me = predict\_me.reshape(-1,len(predict\_me)) |
|  | prediction = kmeans.predict(predict\_me) |
|  | if(prediction[0]==y\_test2[i]): |
|  | correct += 1 |
|  | print("Score: %.2f" % (correct/len(X\_test2))) |
|  |  |
|  |  |
|  | #Make own module to predict |
|  | def process\_module(df, targetName): |
|  |  |
|  | # Split the dataset |
|  | y = df[targetName] |
|  | X = df.drop([targetName], 1) |
|  | X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0) |
|  |  |
|  |  |
|  | # Normalization with 4 Scaling methods |
|  | maxAbsScaler = preprocessing.MaxAbsScaler() |
|  | minmaxScaler = preprocessing.MinMaxScaler() |
|  | robustScaler = preprocessing.RobustScaler() |
|  | standardScaler = preprocessing.StandardScaler() |
|  |  |
|  | df\_maxAbs\_scaled\_train = maxAbsScaler.fit\_transform(X\_train) |
|  | df\_maxAbs\_scaled\_train = pd.DataFrame(df\_maxAbs\_scaled\_train, columns=X\_train.columns) |
|  | df\_maxAbs\_scaled\_test = maxAbsScaler.fit\_transform(X\_test) |
|  | df\_maxAbs\_scaled\_test = pd.DataFrame(df\_maxAbs\_scaled\_test, columns=X\_test.columns) |
|  |  |
|  | df\_minMax\_scaled\_train = minmaxScaler.fit\_transform(X\_train) |
|  | df\_minMax\_scaled\_train = pd.DataFrame(df\_minMax\_scaled\_train, columns=X\_train.columns) |
|  | df\_minMax\_scaled\_test = minmaxScaler.fit\_transform(X\_test) |
|  | df\_minMax\_scaled\_test = pd.DataFrame(df\_minMax\_scaled\_test, columns=X\_test.columns) |
|  |  |
|  | df\_robust\_scaled\_train = robustScaler.fit\_transform(X\_train) |
|  | df\_robust\_scaled\_train = pd.DataFrame(df\_robust\_scaled\_train, columns=X\_train.columns) |
|  | df\_robust\_scaled\_test = robustScaler.fit\_transform(X\_test) |
|  | df\_robust\_scaled\_test = pd.DataFrame(df\_robust\_scaled\_test, columns=X\_test.columns) |
|  |  |
|  | df\_standard\_scaled\_train = standardScaler.fit\_transform(X\_train) |
|  | df\_standard\_scaled\_train = pd.DataFrame(df\_standard\_scaled\_train, columns=X\_train.columns) |
|  | df\_standard\_scaled\_test = standardScaler.fit\_transform(X\_test) |
|  | df\_standard\_scaled\_test = pd.DataFrame(df\_standard\_scaled\_test, columns=X\_test.columns) |
|  |  |
|  |  |
|  | # Alogirthm |
|  | print("\n------------------------- Using maxAbs scaled dataset -------------------------") |
|  | max\_score\_maxAbs = algorithm\_module(df\_maxAbs\_scaled\_train, df\_maxAbs\_scaled\_test, y\_train, y\_test) |
|  | print("\n------------------------- Using minMax scaled dataset -------------------------") |
|  | max\_score\_minMax = algorithm\_module(df\_minMax\_scaled\_train, df\_minMax\_scaled\_test, y\_train, y\_test) |
|  | print("\n------------------------- Using robust scaled dataset -------------------------") |
|  | max\_score\_robust = algorithm\_module(df\_robust\_scaled\_train, df\_robust\_scaled\_test, y\_train, y\_test) |
|  | print("\n------------------------- Using standard scaled dataset -------------------------") |
|  | max\_score\_standard = algorithm\_module(df\_standard\_scaled\_train, df\_standard\_scaled\_test, y\_train, y\_test) |
|  |  |
|  |  |
|  | # Result |
|  | max\_score\_result = max(max\_score\_maxAbs, max\_score\_minMax, max\_score\_robust, max\_score\_standard) |
|  | print("\n\n============================== Result ==============================") |
|  | print("Final maximum score: %.6f" % max\_score\_result) |
|  |  |
|  |  |
|  | def algorithm\_module(X\_train, X\_test, y\_train, y\_test): |
|  |  |
|  | # Linear Regression |
|  | line\_reg = LinearRegression() |
|  | line\_reg.fit(X\_train, y\_train) |
|  | y\_prec\_linear = line\_reg.predict(X\_test) |
|  | score\_linear = line\_reg.score(X\_test, y\_test) |
|  | print("\ny\_predict\_linear: \n", y\_prec\_linear[0:50]) |
|  | print("Score: %.6f" % score\_linear) |
|  |  |
|  | # Polynomial Regression |
|  | poly\_reg = PolynomialFeatures(degree=2) |
|  | X\_poly\_train = poly\_reg.fit\_transform(X\_train) |
|  | X\_poly\_test = poly\_reg.fit\_transform(X\_test) |
|  | pol\_reg = LinearRegression() |
|  | pol\_reg.fit(X\_poly\_train, y\_train) |
|  | y\_prec\_poly = line\_reg.predict(X\_test) |
|  | score\_poly = pol\_reg.score(X\_poly\_test, y\_test) |
|  | print("\ny\_predict\_poly: \n", y\_prec\_poly[0:50]) |
|  | print("Score: %.6f" % score\_poly) |
|  |  |
|  | # KNN algorithm |
|  | knn = KNeighborsRegressor(n\_neighbors=5) |
|  | knn.fit(X\_train, y\_train) |
|  | y\_prec\_knn = knn.predict(X\_test) |
|  | score\_knn = knn.score(X\_test, y\_test) |
|  | print("\ny\_predict\_KNN: \n", y\_prec\_knn[0:50]) |
|  | print("Score: %.6f" % score\_knn) |
|  |  |
|  | # Random Forest |
|  | random\_forest = RandomForestRegressor(max\_depth=4, random\_state=0) |
|  | random\_forest.fit(X\_train, y\_train) |
|  | y\_predict\_rf = random\_forest.predict(X\_test) |
|  | score\_rf = random\_forest.score(X\_test, y\_test) |
|  | print("\ny\_predict\_RF: \n", y\_predict\_rf[0:50]) |
|  | print("Score: %.6f" % score\_rf) |
|  |  |
|  | max\_score = max(score\_linear, score\_poly, score\_knn, score\_rf) |
|  | return max\_score |
|  |  |
|  |  |
|  | # Test our model using the ordinal encoded dataset |
|  | df\_test\_model = df\_ordinal.copy() |
|  | print("\n\n============================== Using own module ==============================") |
|  | process\_module(df\_test\_model, 'annual\_claims') |
|  | # process\_module(df\_test\_model, 'target') |

#OpenSource module

import warnings

warnings.filterwarnings("ignore")

import pandas as pd

from sklearn import preprocessing

from sklearn.neighbors import KNeighborsRegressor

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestRegressor

from sklearn.linear\_model import LinearRegression

from sklearn.preprocessing import PolynomialFeatures

def process\_module(df, targetName):

# Split the dataset

y = df[targetName]

X = df.drop([targetName], 1)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

# Normalization with 4 Scaling methods

maxAbsScaler = preprocessing.MaxAbsScaler()

minmaxScaler = preprocessing.MinMaxScaler()

robustScaler = preprocessing.RobustScaler()

standardScaler = preprocessing.StandardScaler()

df\_maxAbs\_scaled\_train = maxAbsScaler.fit\_transform(X\_train)

df\_maxAbs\_scaled\_train = pd.DataFrame(df\_maxAbs\_scaled\_train, columns=X\_train.columns)

df\_maxAbs\_scaled\_test = maxAbsScaler.fit\_transform(X\_test)

df\_maxAbs\_scaled\_test = pd.DataFrame(df\_maxAbs\_scaled\_test, columns=X\_test.columns)

df\_minMax\_scaled\_train = minmaxScaler.fit\_transform(X\_train)

df\_minMax\_scaled\_train = pd.DataFrame(df\_minMax\_scaled\_train, columns=X\_train.columns)

df\_minMax\_scaled\_test = minmaxScaler.fit\_transform(X\_test)

df\_minMax\_scaled\_test = pd.DataFrame(df\_minMax\_scaled\_test, columns=X\_test.columns)

df\_robust\_scaled\_train = robustScaler.fit\_transform(X\_train)

df\_robust\_scaled\_train = pd.DataFrame(df\_robust\_scaled\_train, columns=X\_train.columns)

df\_robust\_scaled\_test = robustScaler.fit\_transform(X\_test)

df\_robust\_scaled\_test = pd.DataFrame(df\_robust\_scaled\_test, columns=X\_test.columns)

df\_standard\_scaled\_train = standardScaler.fit\_transform(X\_train)

df\_standard\_scaled\_train = pd.DataFrame(df\_standard\_scaled\_train, columns=X\_train.columns)

df\_standard\_scaled\_test = standardScaler.fit\_transform(X\_test)

df\_standard\_scaled\_test = pd.DataFrame(df\_standard\_scaled\_test, columns=X\_test.columns)

# Alogrithm

print("\n------------------------- Using maxAbs scaled dataset -------------------------")

max\_score\_maxAbs = algorithm\_module(df\_maxAbs\_scaled\_train, df\_maxAbs\_scaled\_test, y\_train, y\_test)

print("\n------------------------- Using minMax scaled dataset -------------------------")

max\_score\_minMax = algorithm\_module(df\_minMax\_scaled\_train, df\_minMax\_scaled\_test, y\_train, y\_test)

print("\n------------------------- Using robust scaled dataset -------------------------")

max\_score\_robust = algorithm\_module(df\_robust\_scaled\_train, df\_robust\_scaled\_test, y\_train, y\_test)

print("\n------------------------- Using standard scaled dataset -------------------------")

max\_score\_standard = algorithm\_module(df\_standard\_scaled\_train, df\_standard\_scaled\_test, y\_train, y\_test)

# Result

max\_score\_result = max(max\_score\_maxAbs, max\_score\_minMax, max\_score\_robust, max\_score\_standard)

print("\n\n============================== Result ==============================")

print("Final maximum score: %.6f" % max\_score\_result)

def algorithm\_module(X\_train, X\_test, y\_train, y\_test):

# Linear Regression

line\_reg = LinearRegression()

line\_reg.fit(X\_train, y\_train)

y\_prec\_linear = line\_reg.predict(X\_test)

score\_linear = line\_reg.score(X\_test, y\_test)

print("\ny\_predict\_linear: \n", y\_prec\_linear[0:50])

print("Score: %.6f" % score\_linear)

# Polynomial Regression

poly\_reg = PolynomialFeatures(degree=2)

X\_poly\_train = poly\_reg.fit\_transform(X\_train)

X\_poly\_test = poly\_reg.fit\_transform(X\_test)

pol\_reg = LinearRegression()

pol\_reg.fit(X\_poly\_train, y\_train)

y\_prec\_poly = line\_reg.predict(X\_test)

score\_poly = pol\_reg.score(X\_poly\_test, y\_test)

print("\ny\_predict\_poly: \n", y\_prec\_poly[0:50])

print("Score: %.6f" % score\_poly)

# KNN algorithm

knn = KNeighborsRegressor(n\_neighbors=5)

knn.fit(X\_train, y\_train)

y\_prec\_knn = knn.predict(X\_test)

score\_knn = knn.score(X\_test, y\_test)

print("\ny\_predict\_KNN: \n", y\_prec\_knn[0:50])

print("Score: %.6f" % score\_knn)

# Random Forest

random\_forest = RandomForestRegressor(max\_depth=4, random\_state=0)

random\_forest.fit(X\_train, y\_train)

y\_predict\_rf = random\_forest.predict(X\_test)

score\_rf = random\_forest.score(X\_test, y\_test)

print("\ny\_predict\_RF: \n", y\_predict\_rf[0:50])

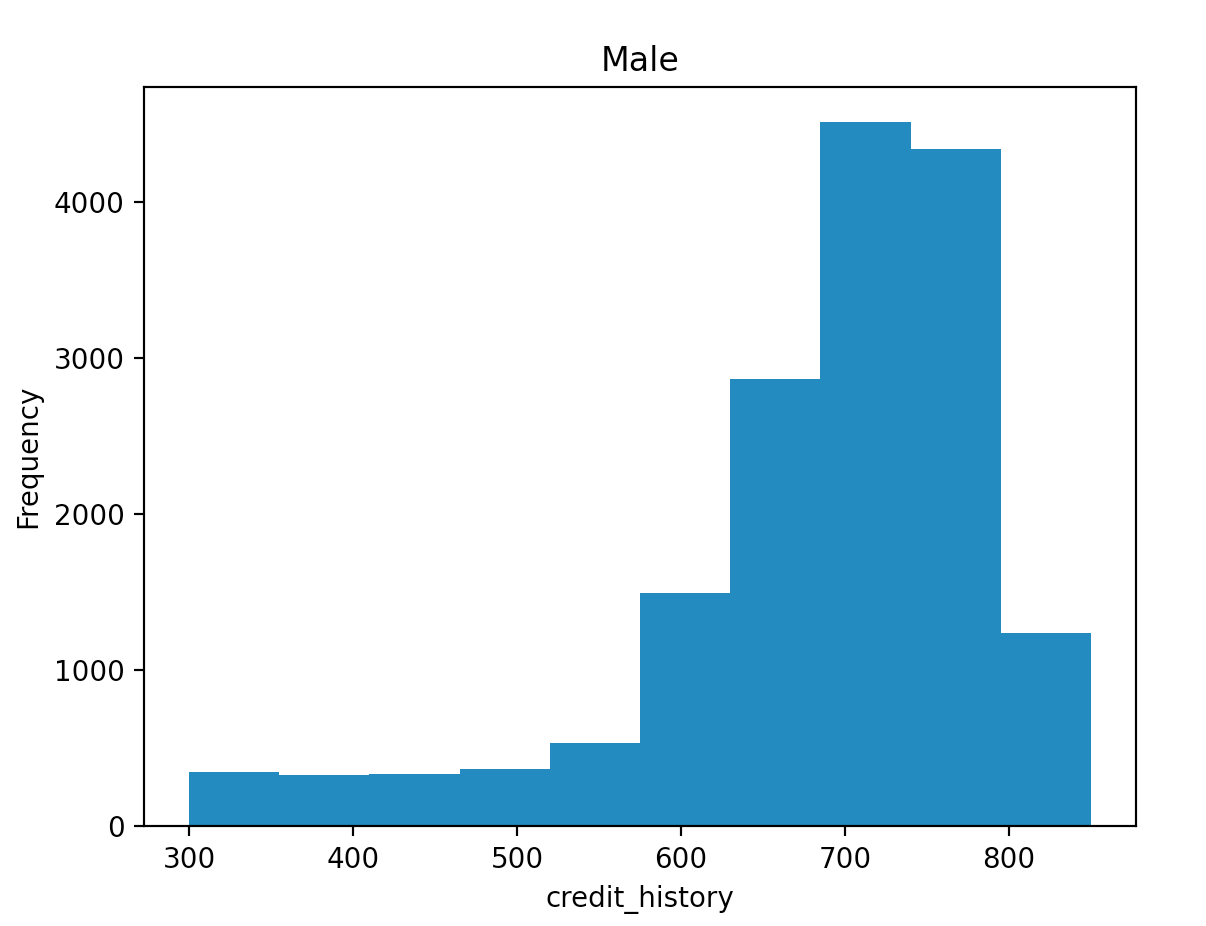
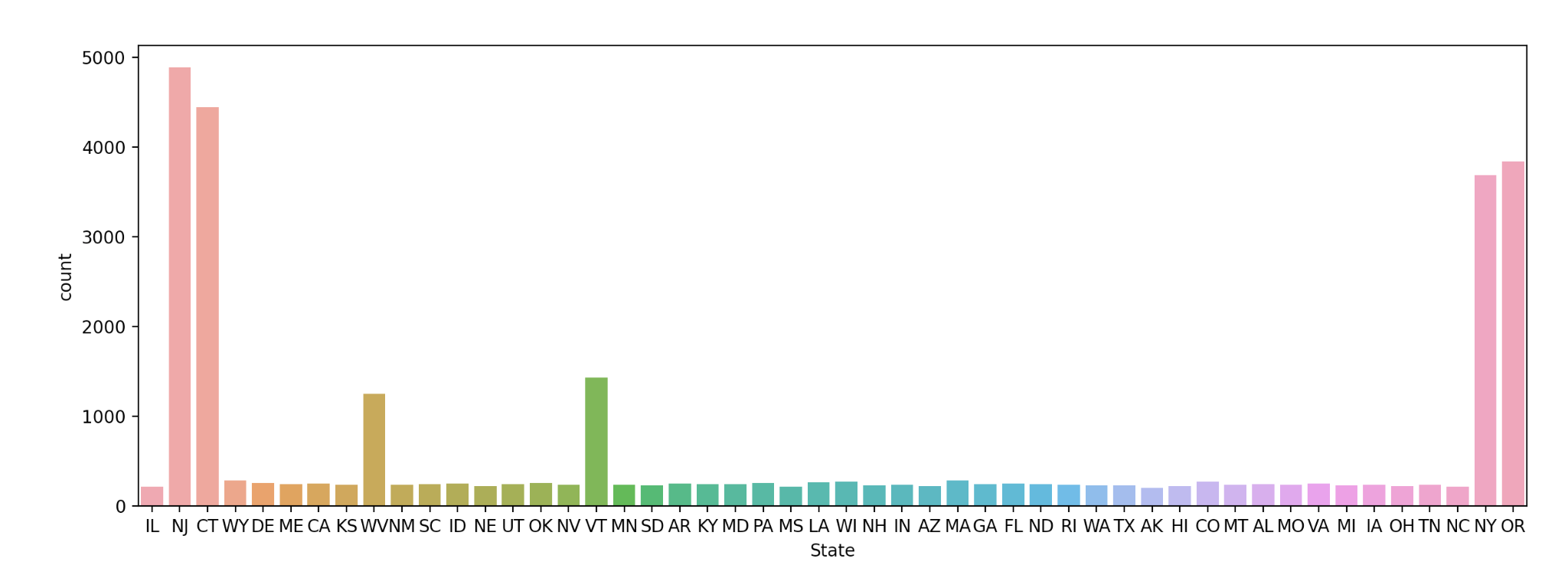
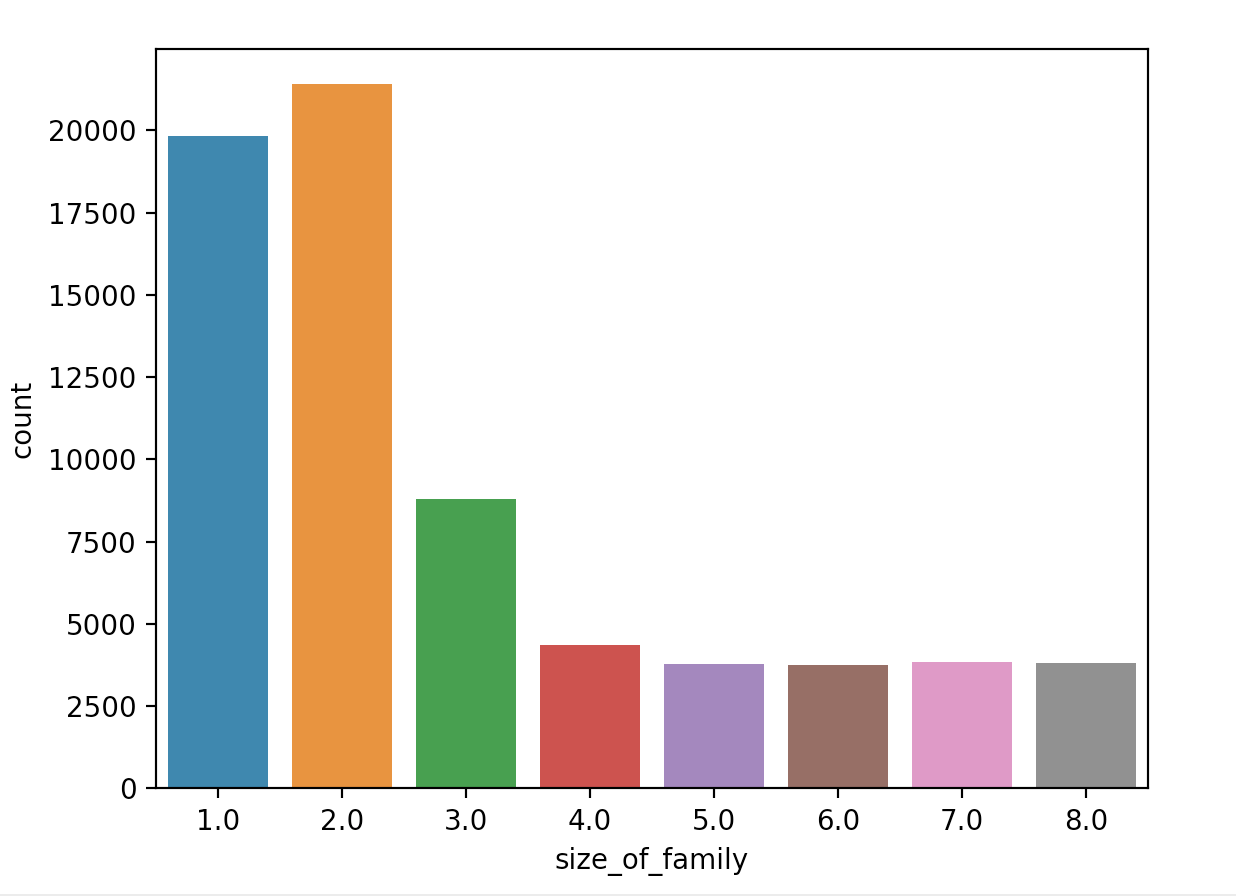
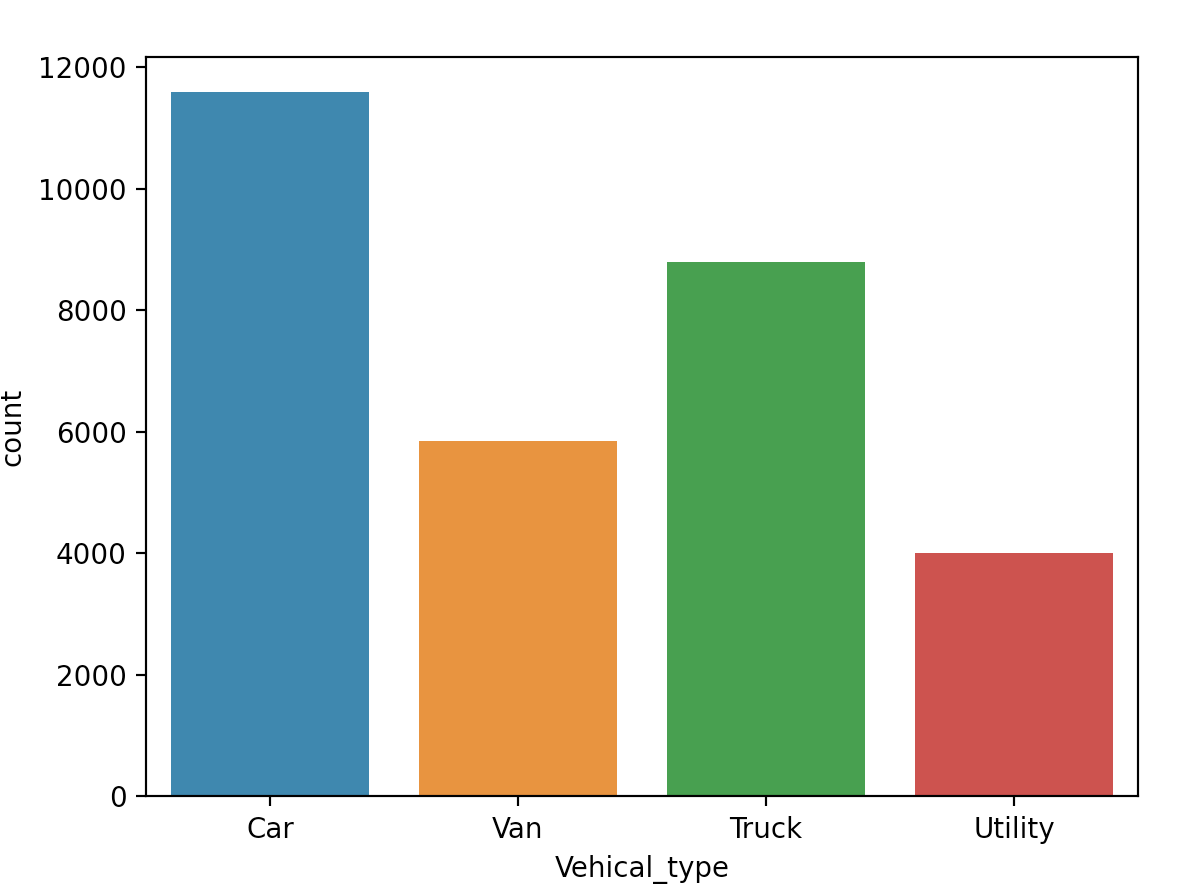
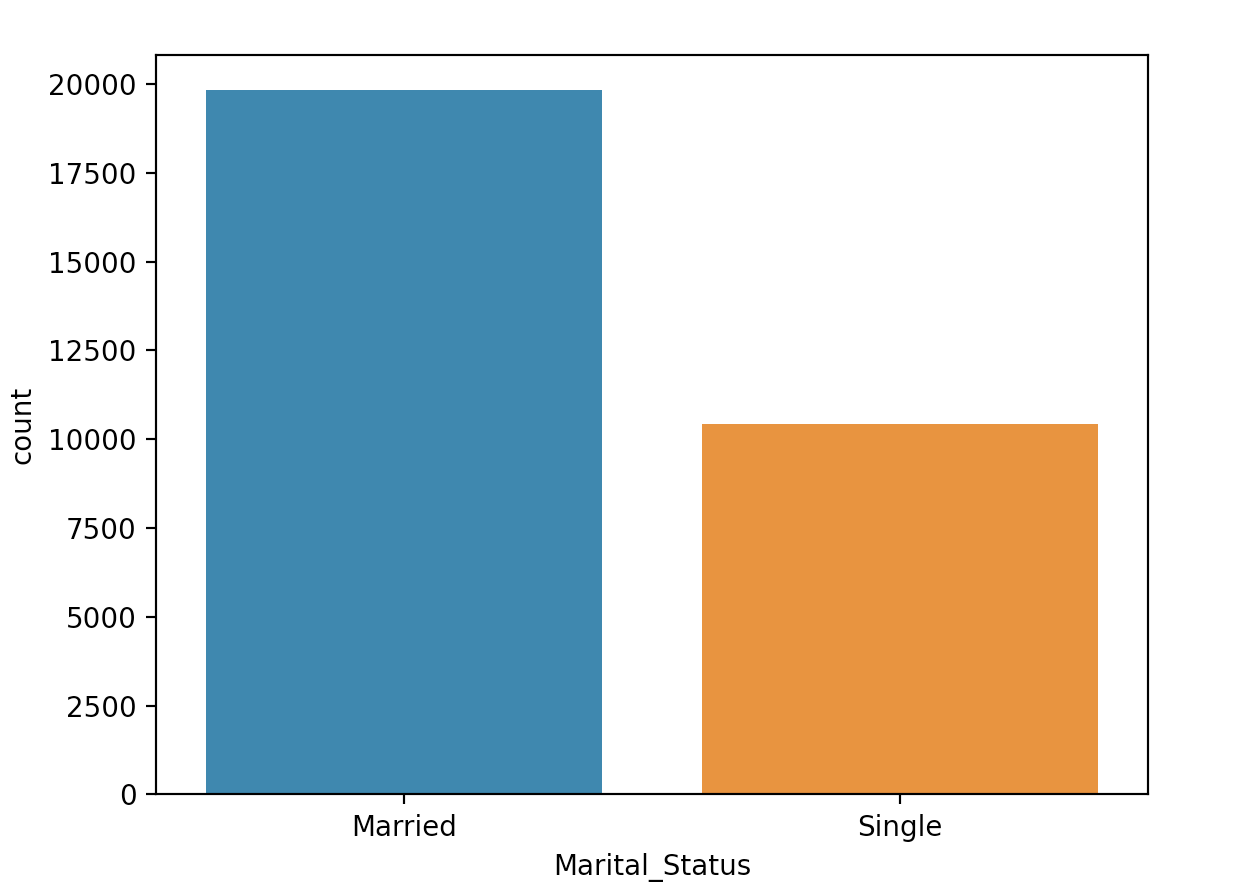
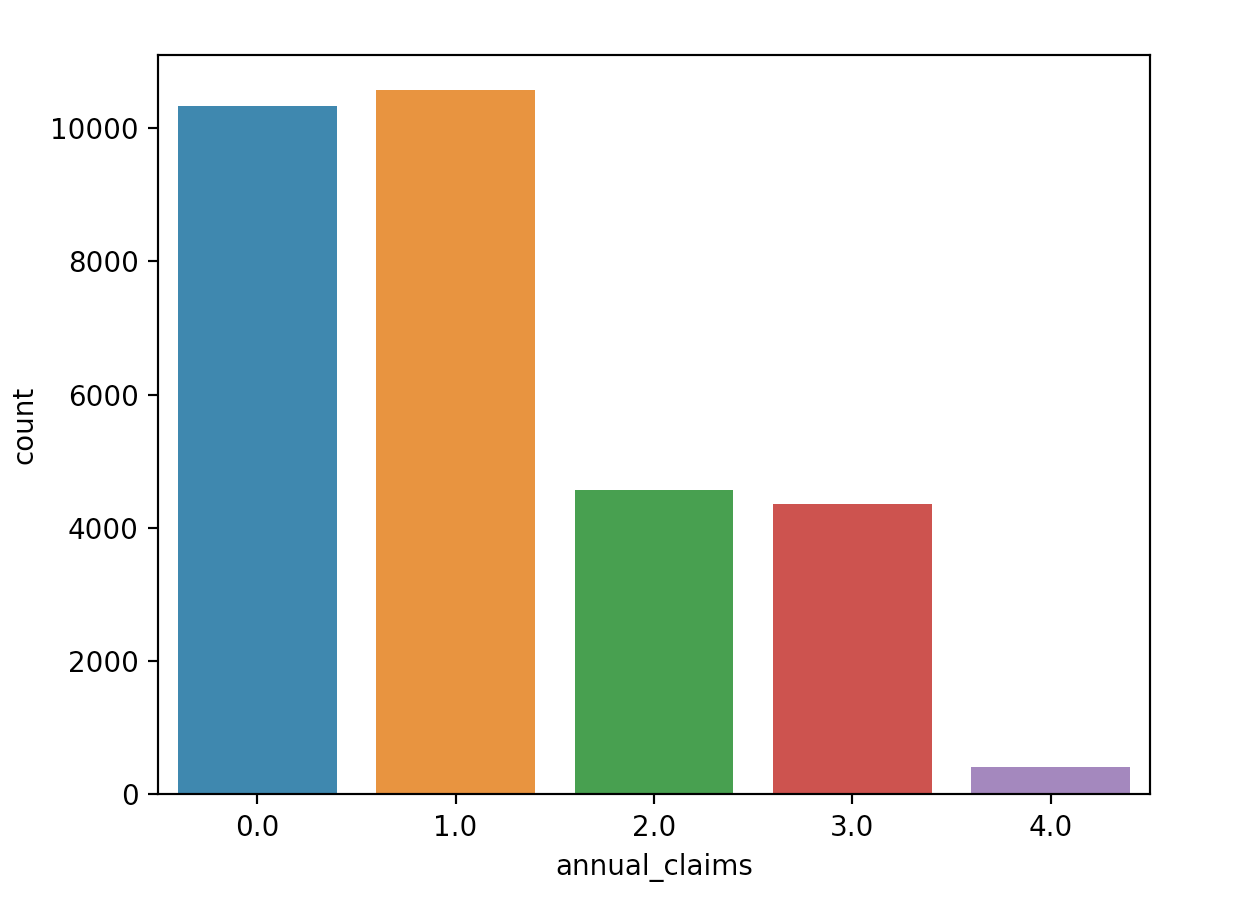
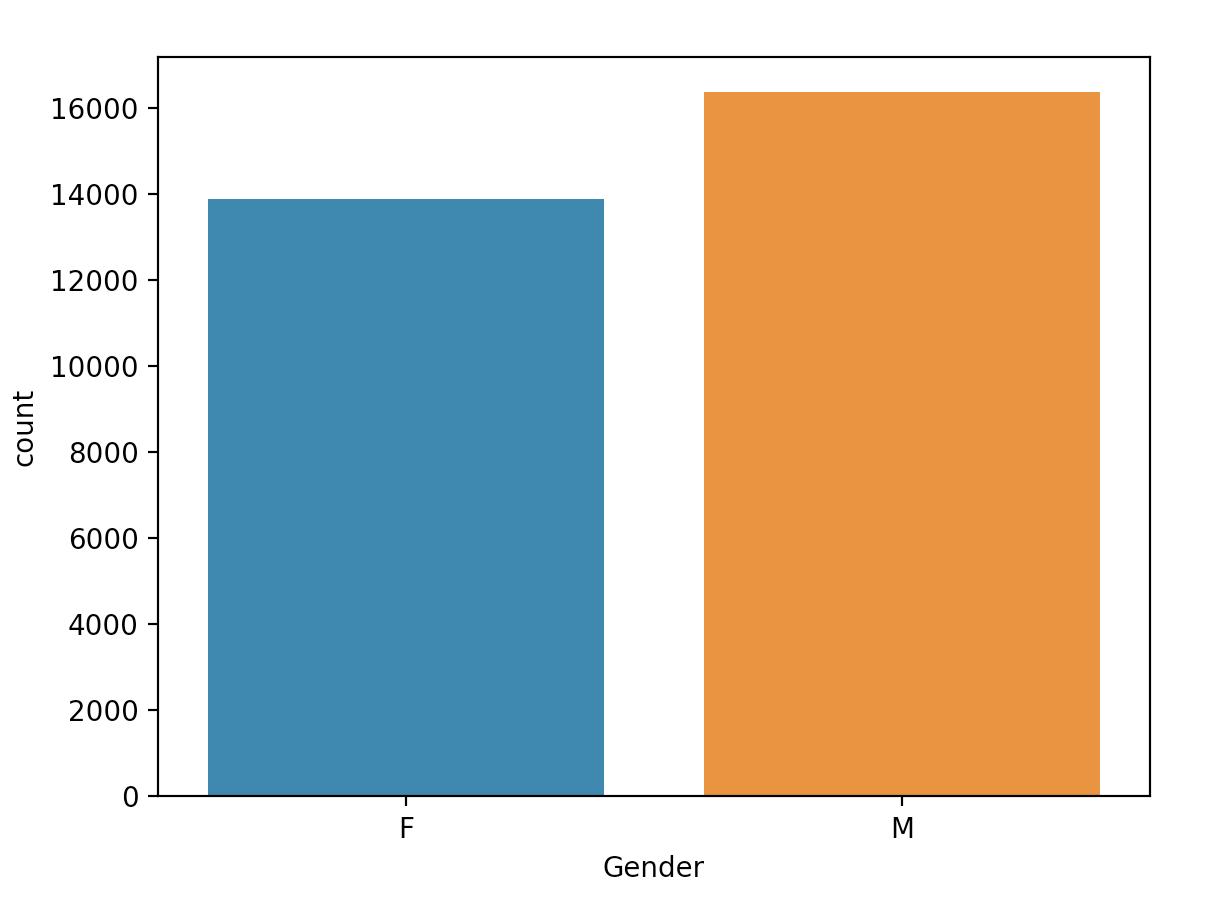
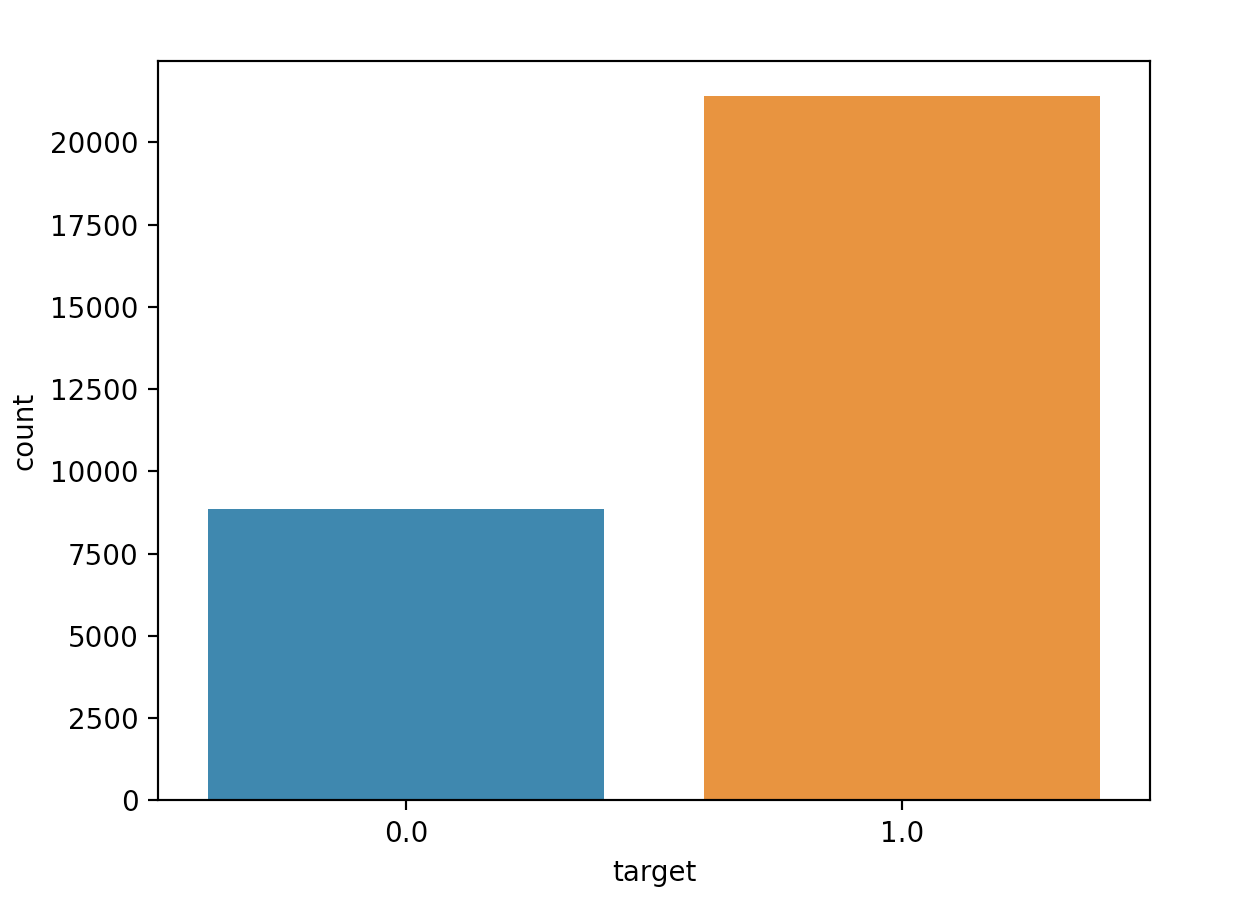
print("Score: %.6f" % score\_rf)

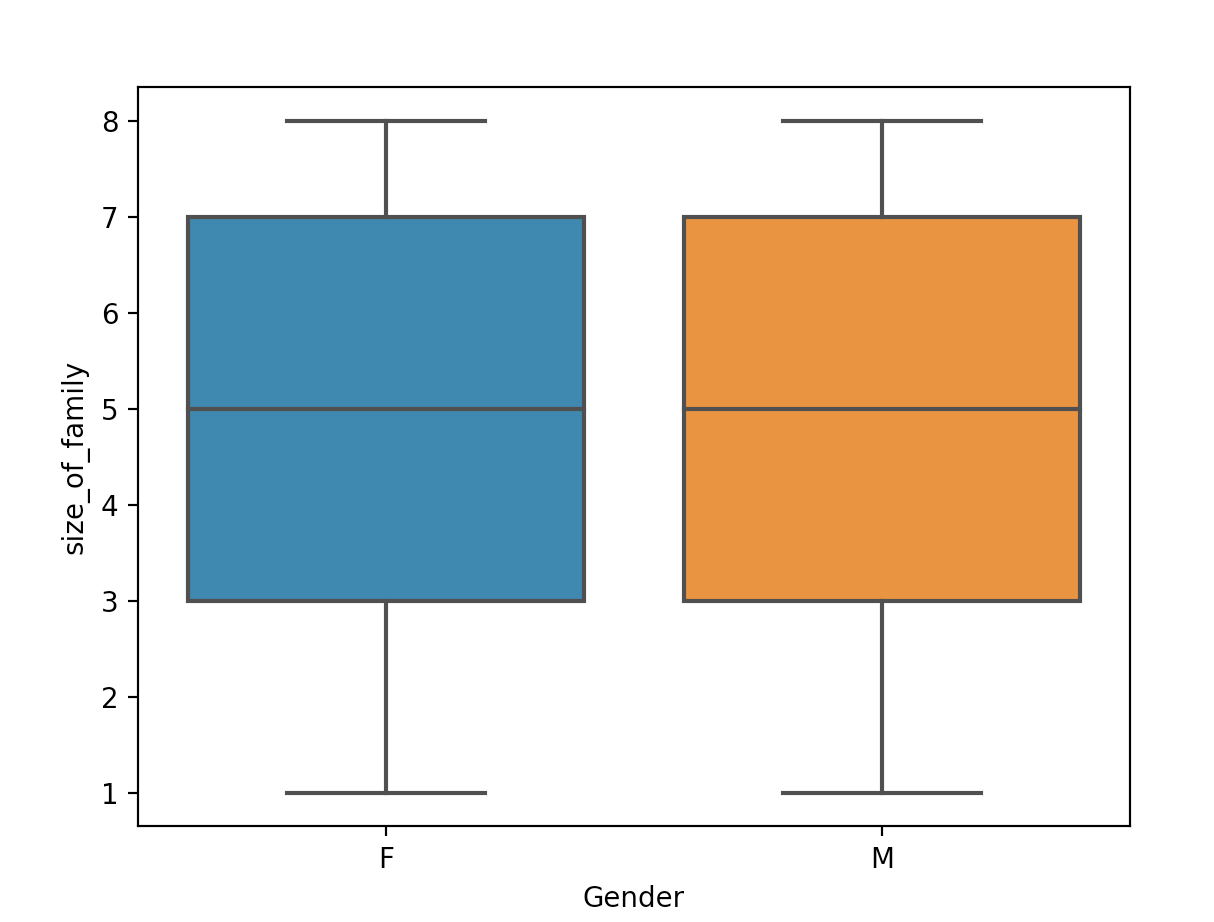
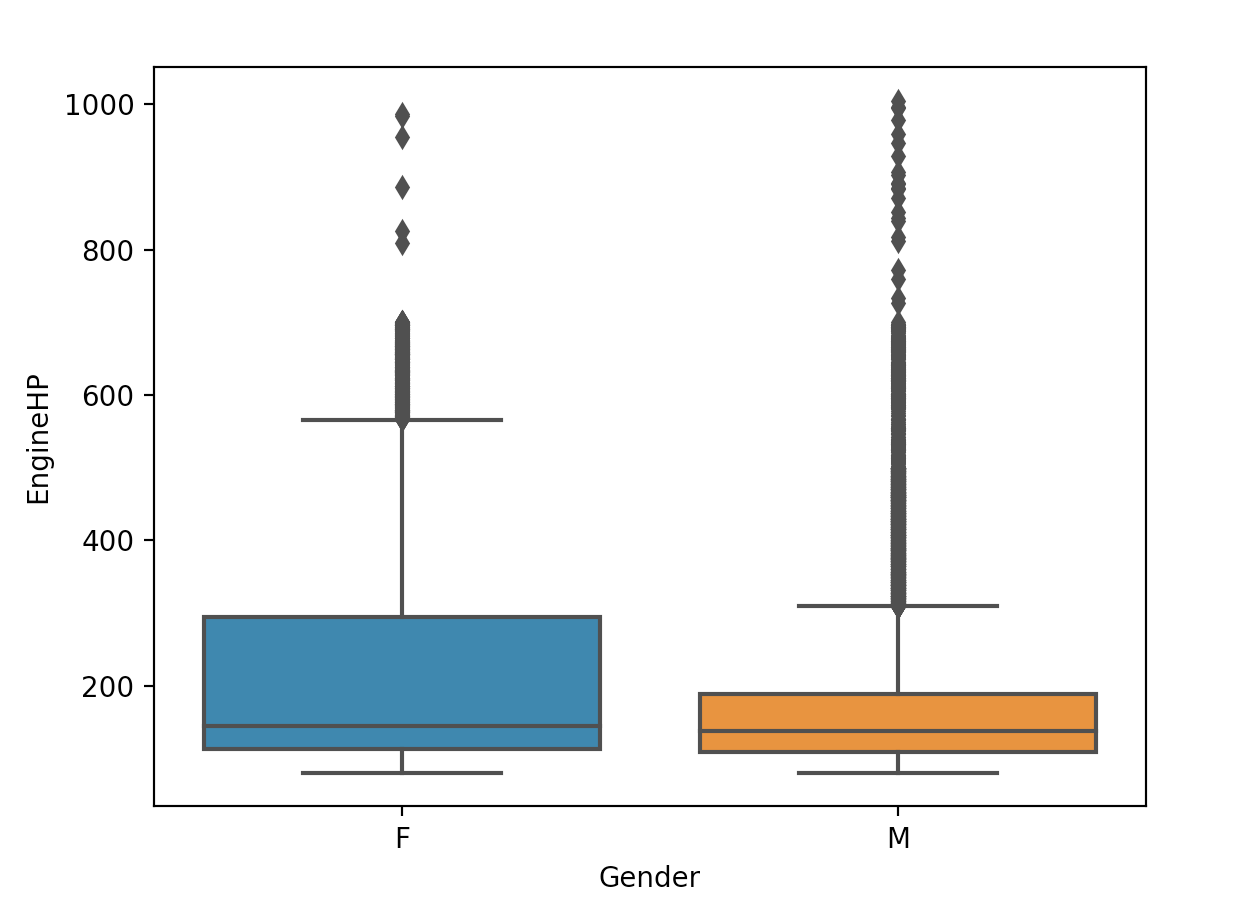
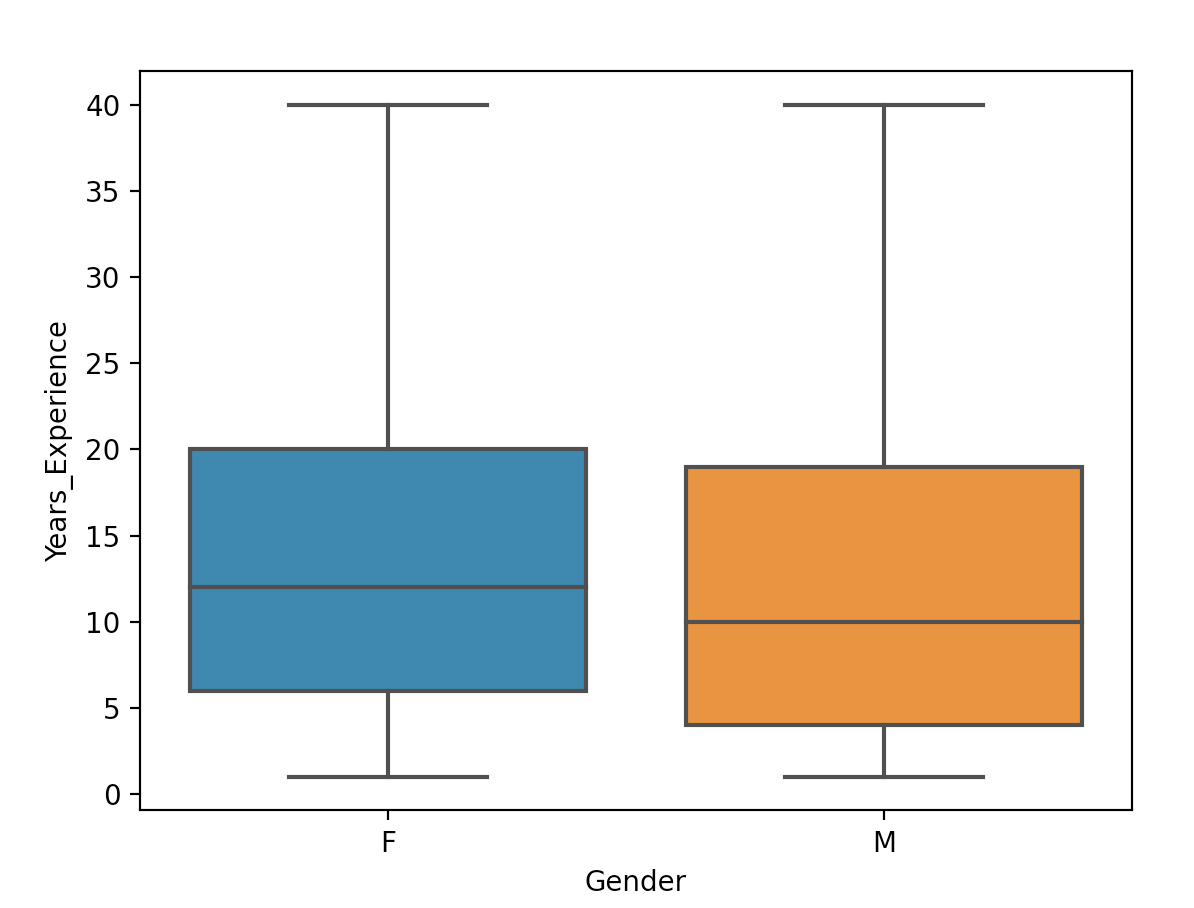
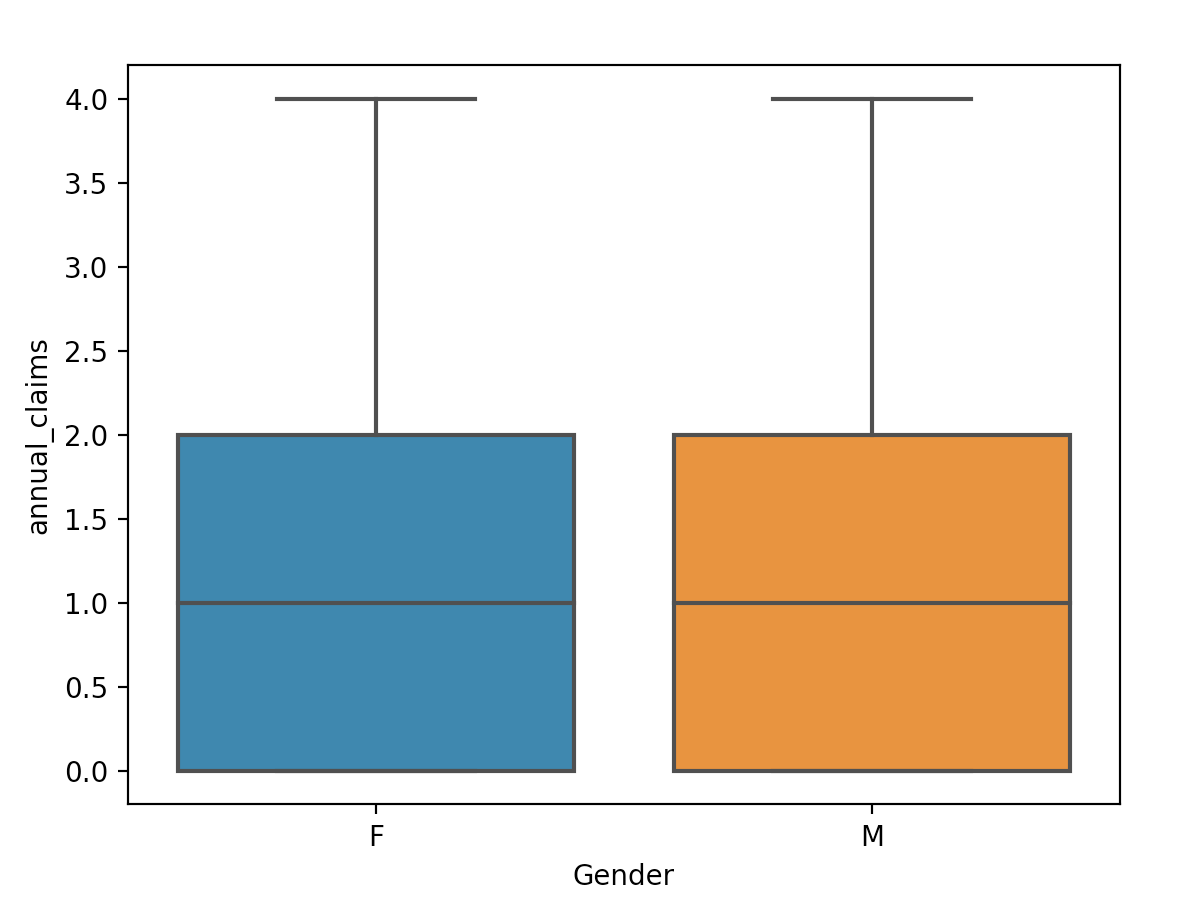
max\_score = max(score\_linear, score\_poly, score\_knn, score\_rf)

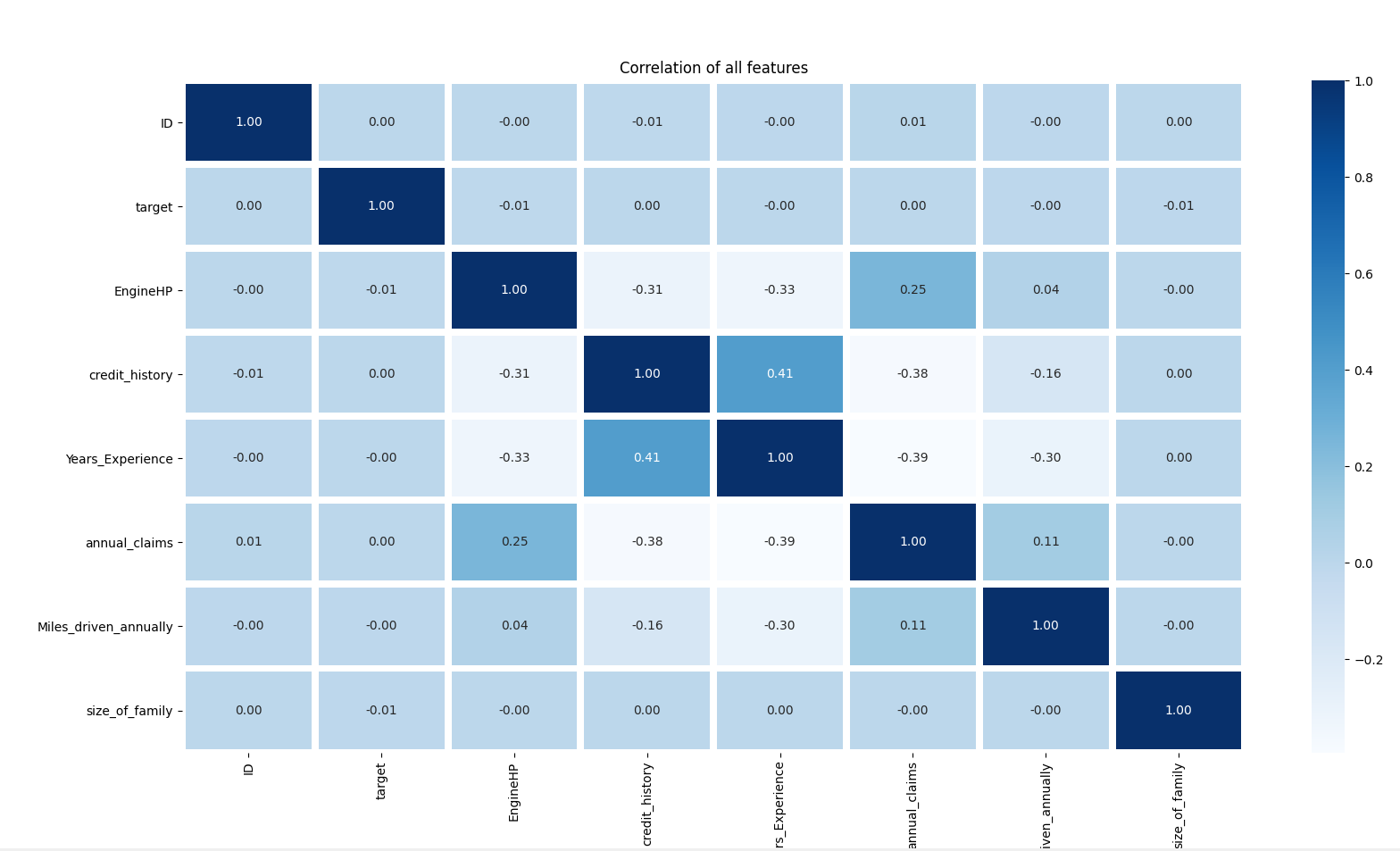
return max\_score

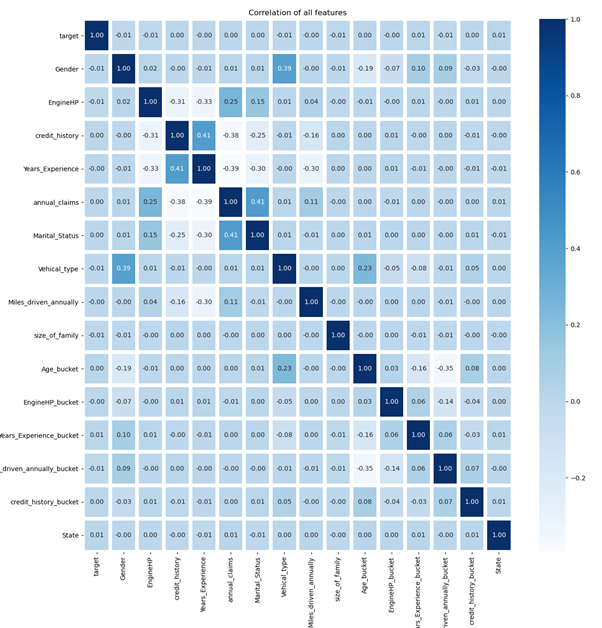
Result

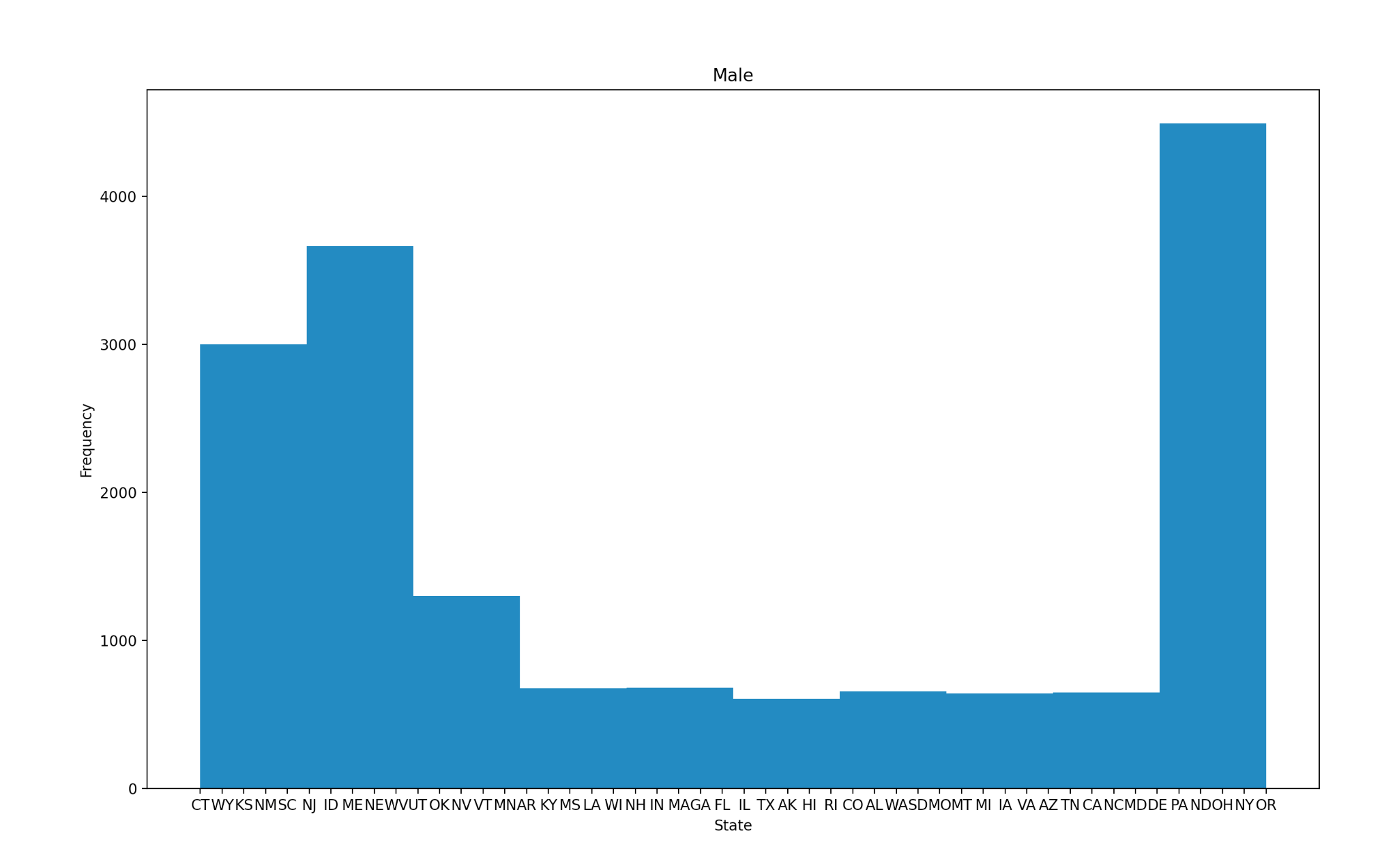
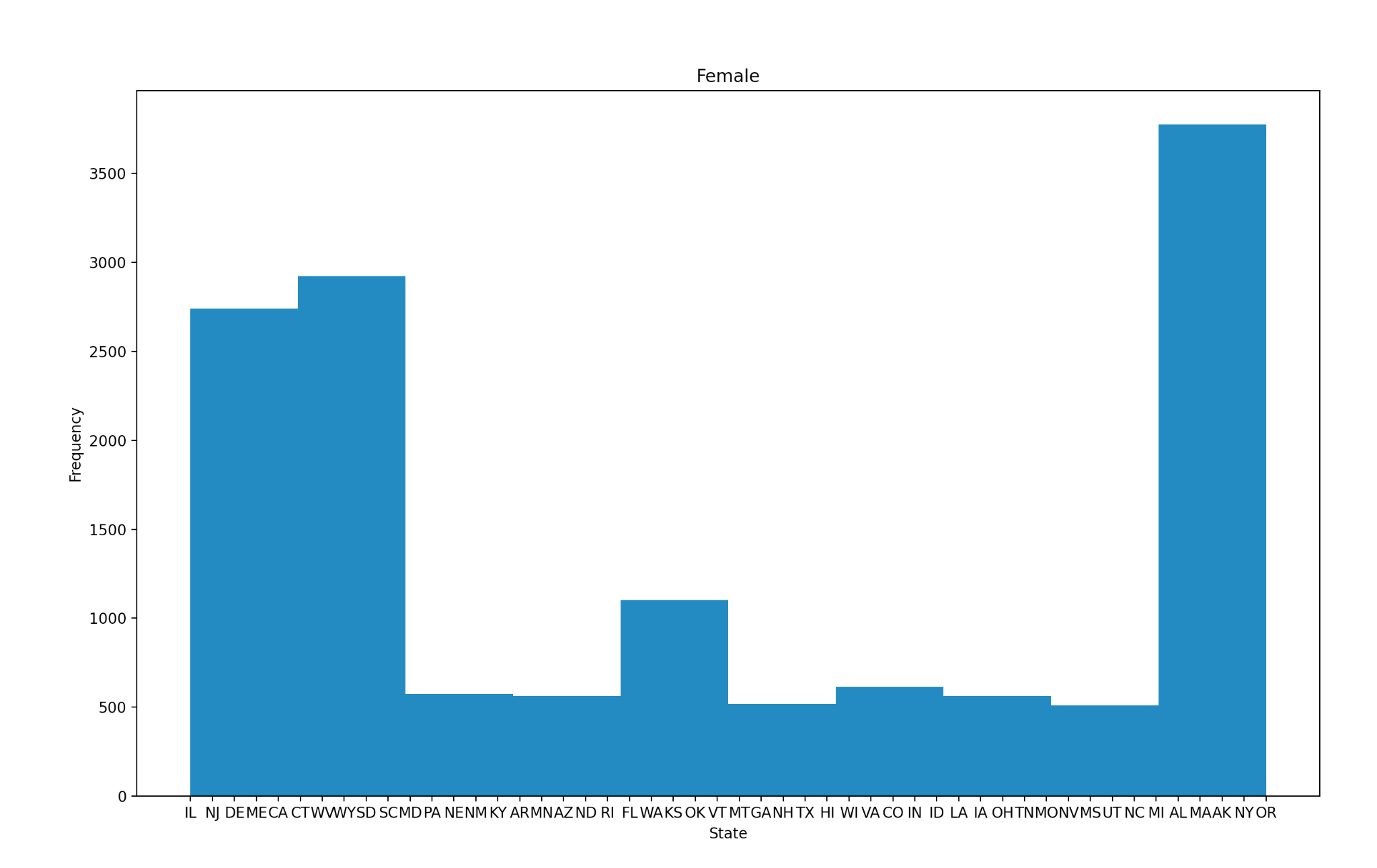
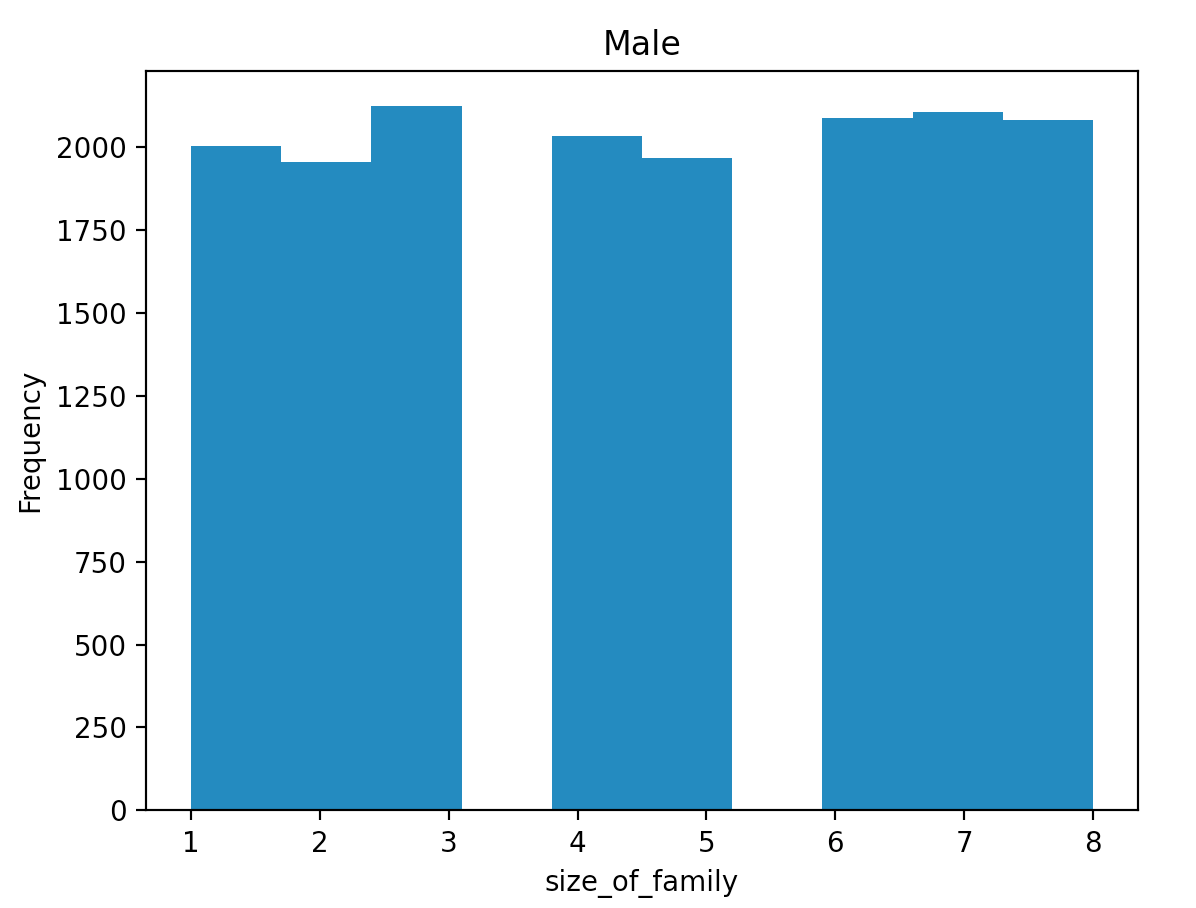
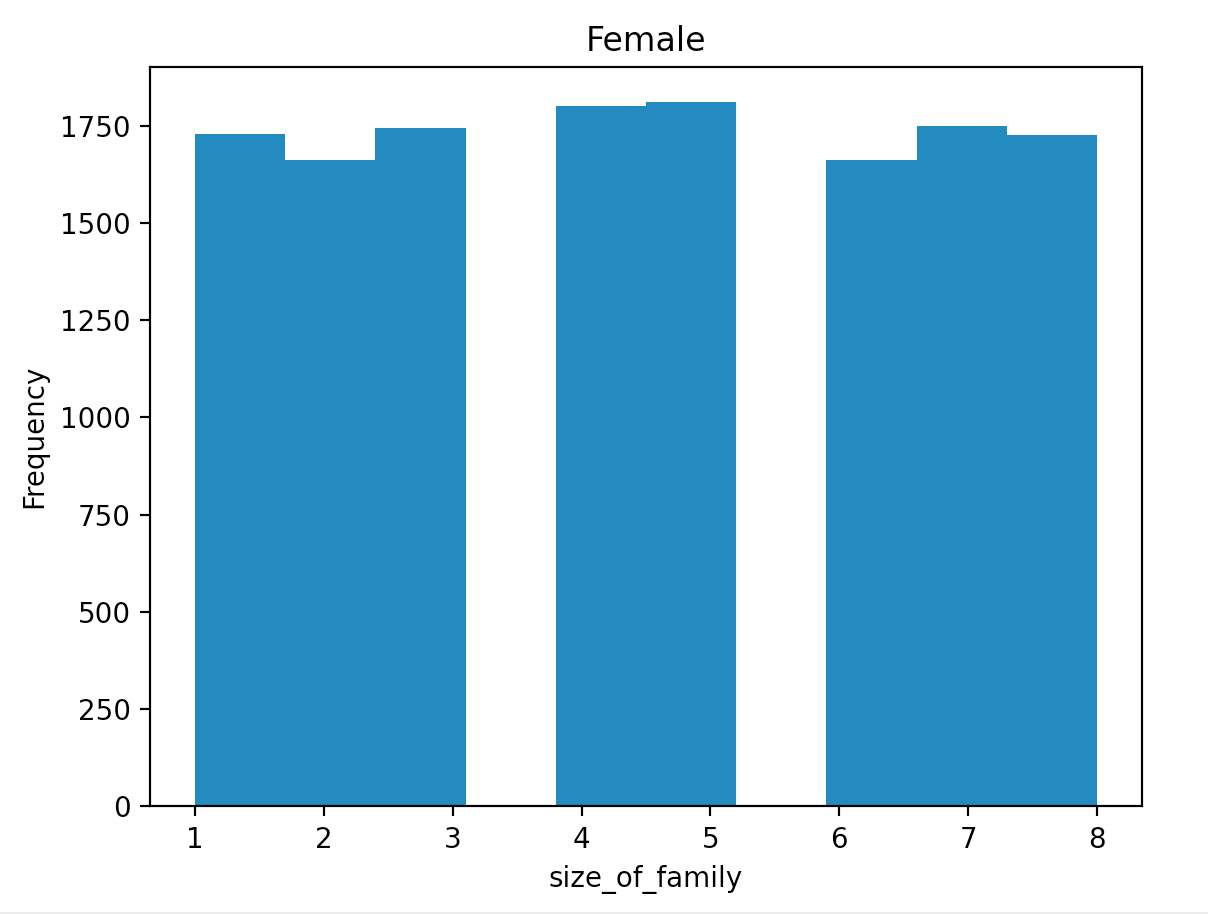
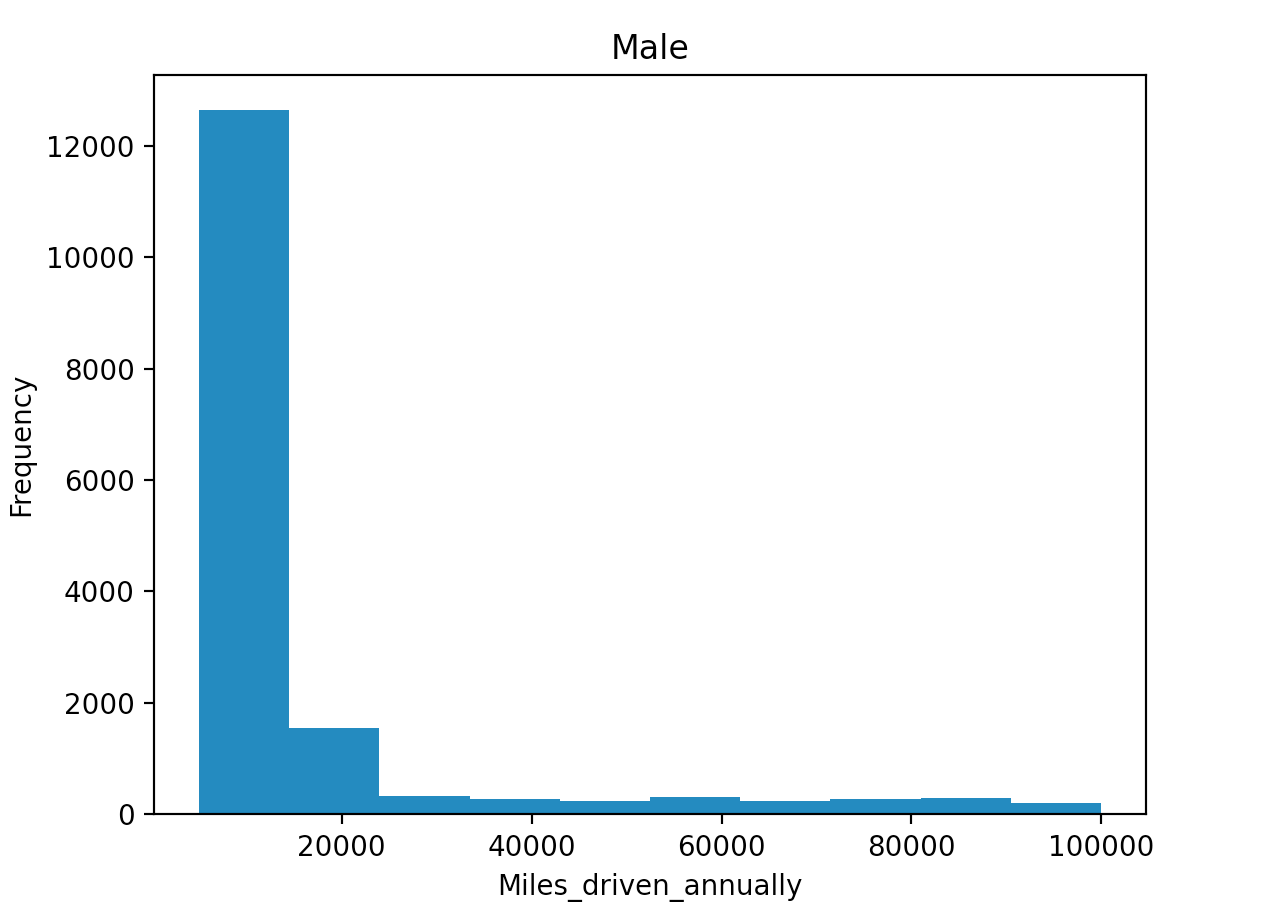
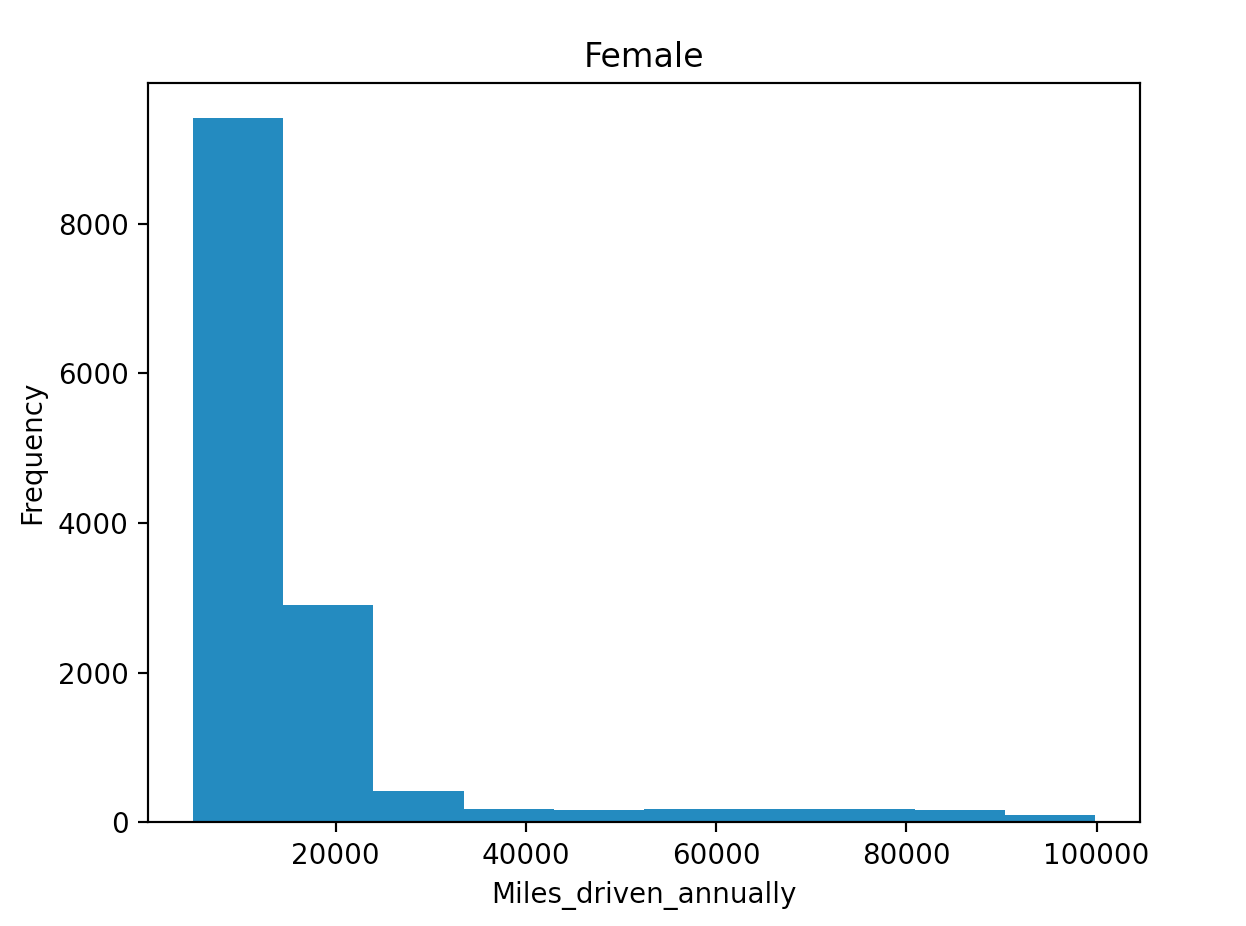
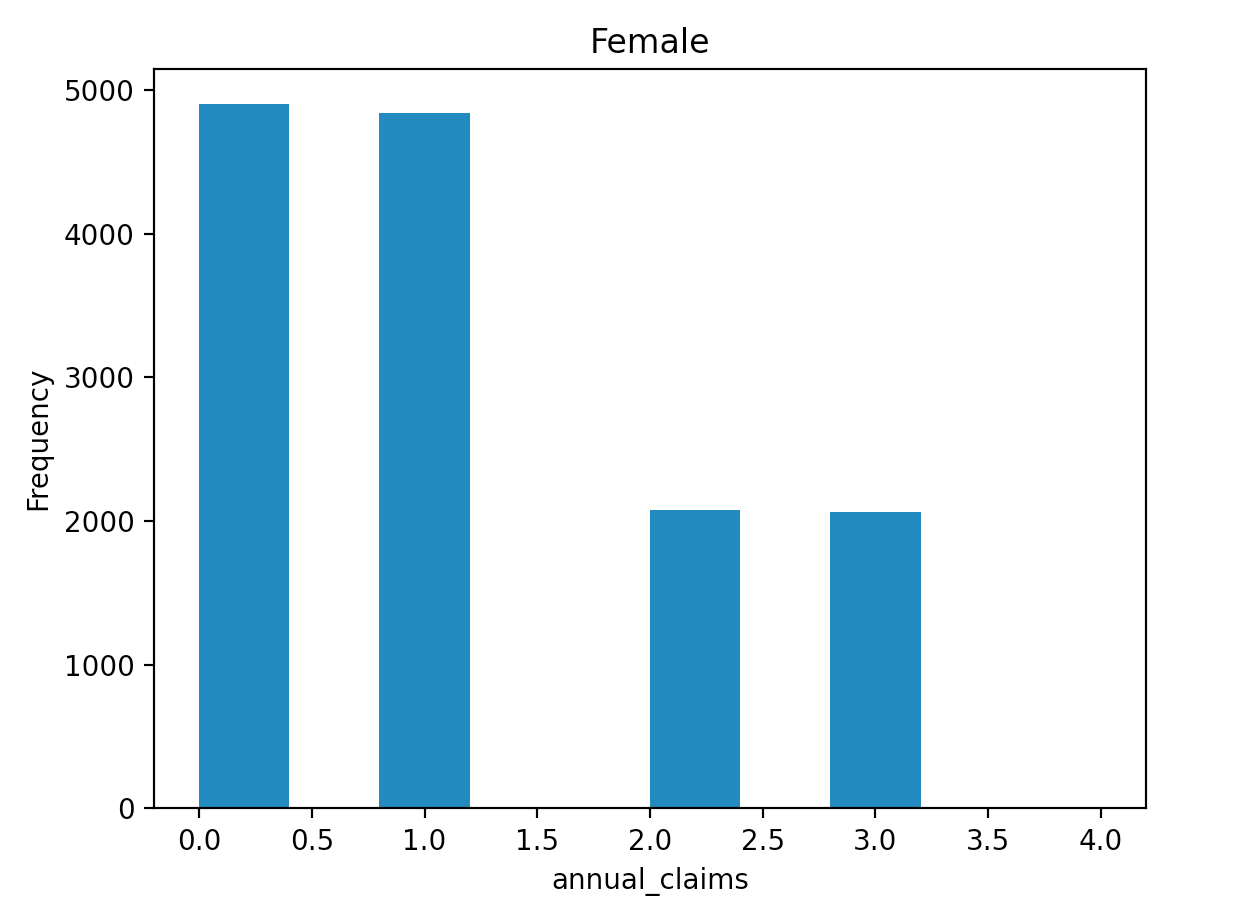
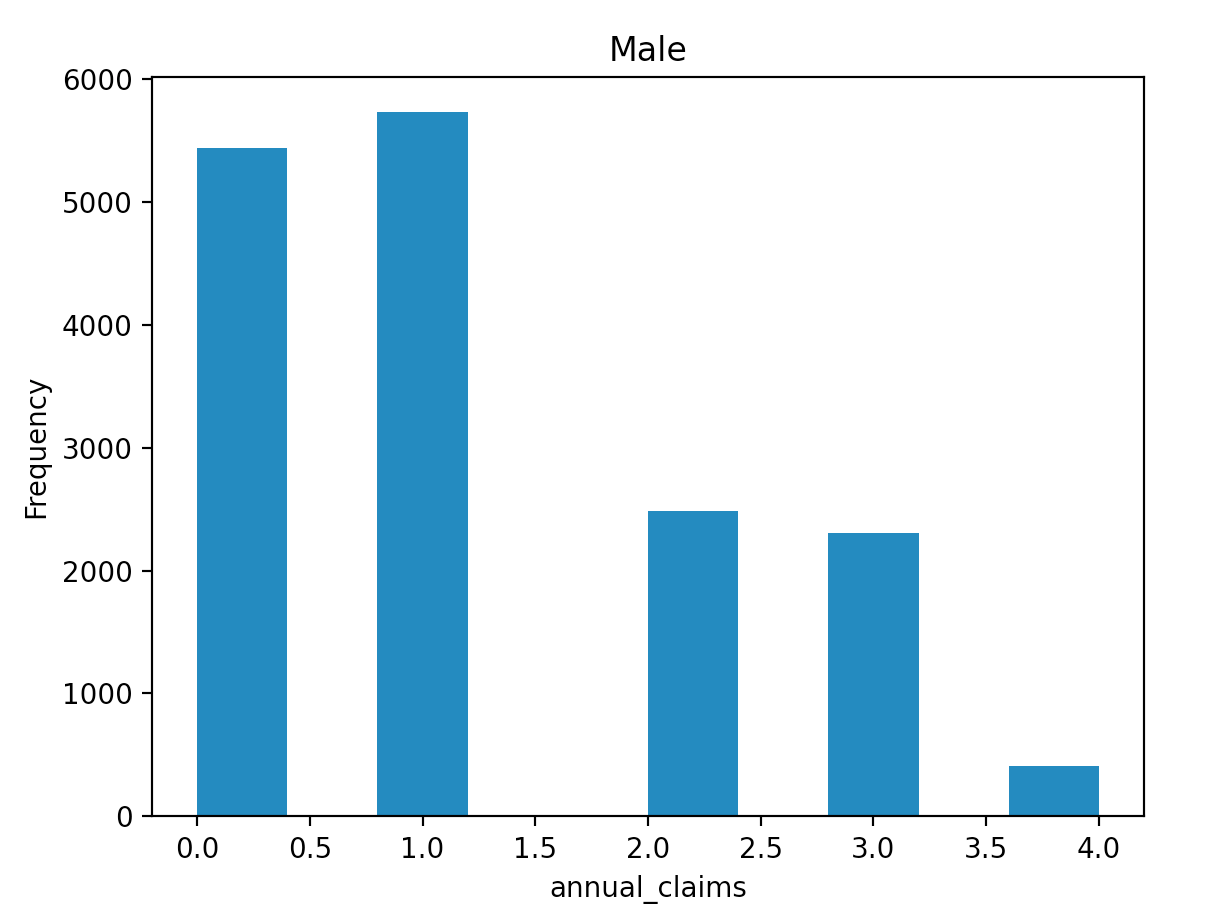
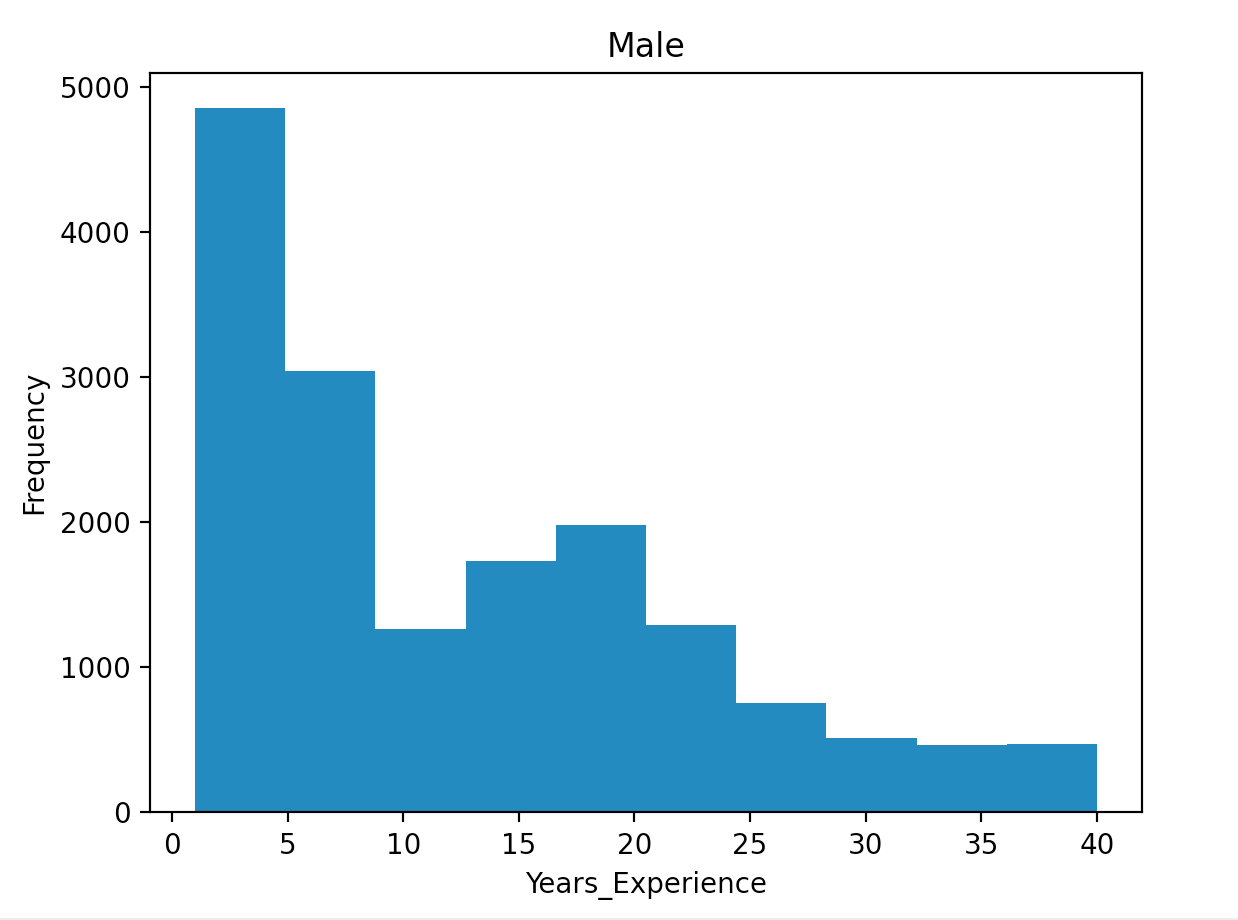
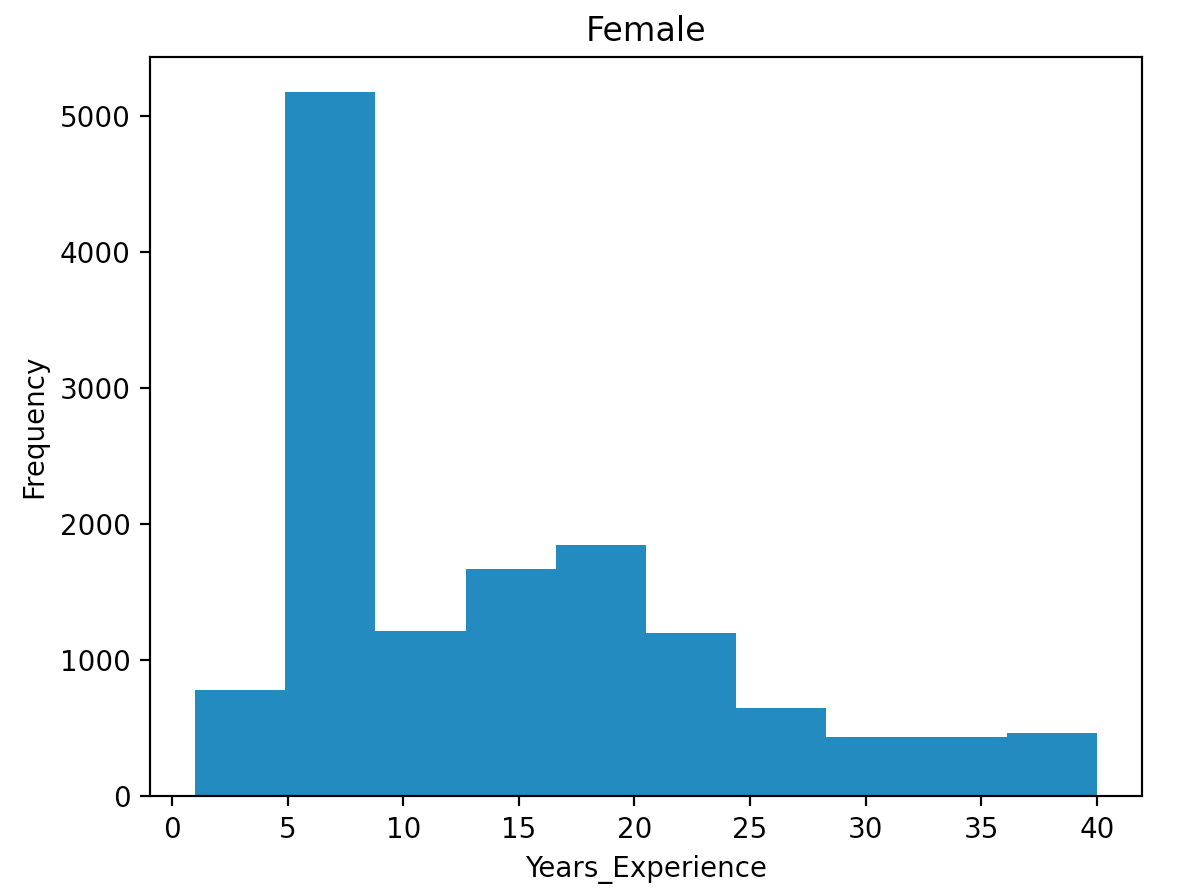
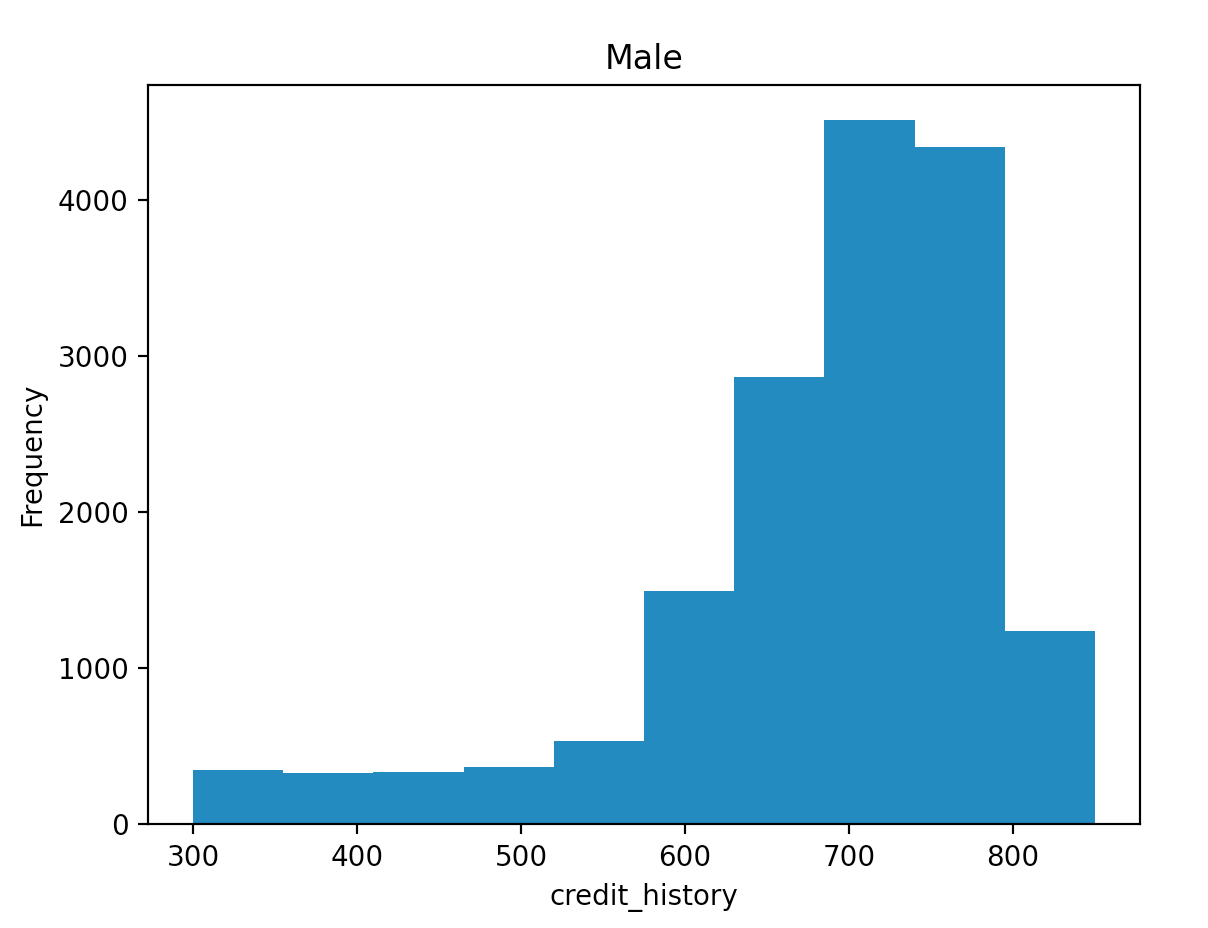
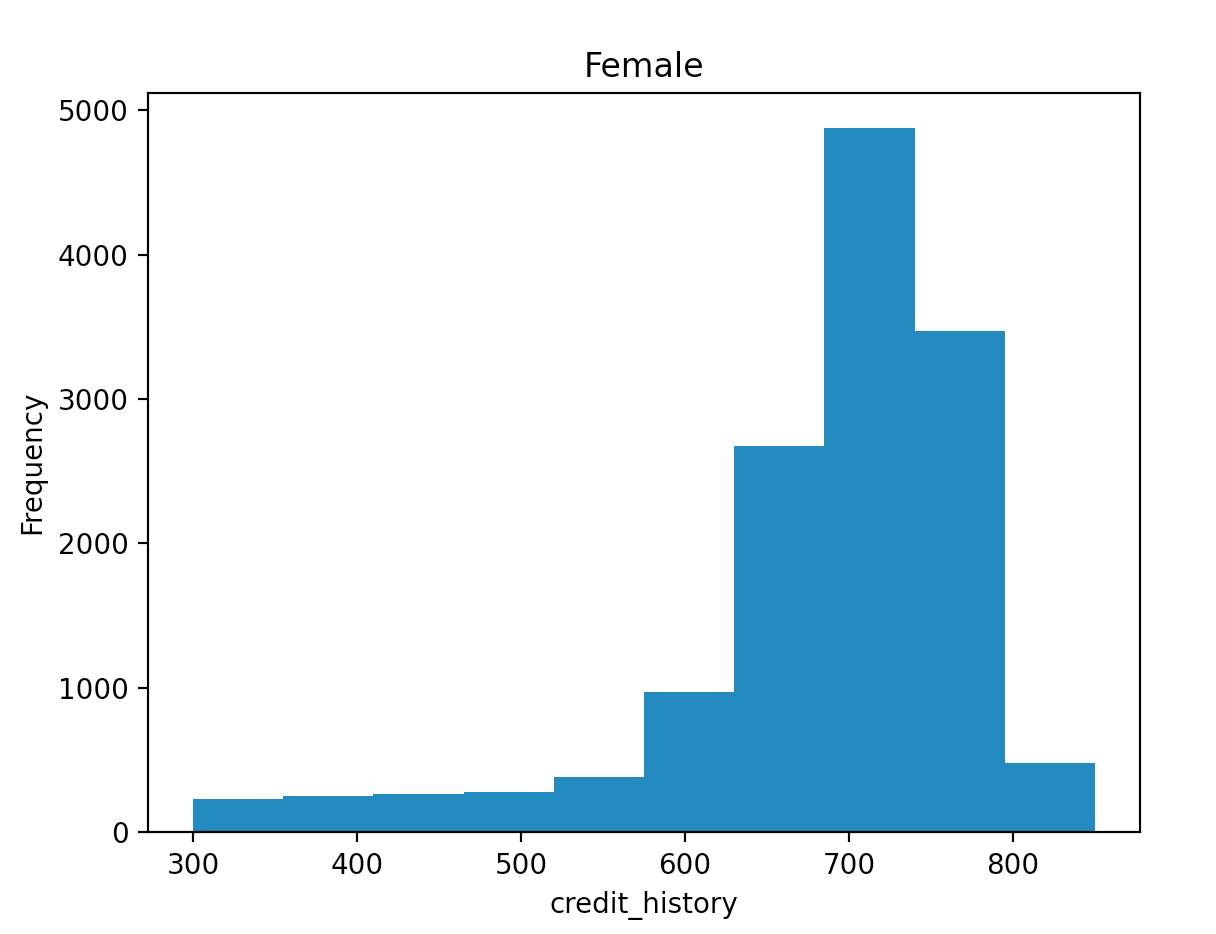
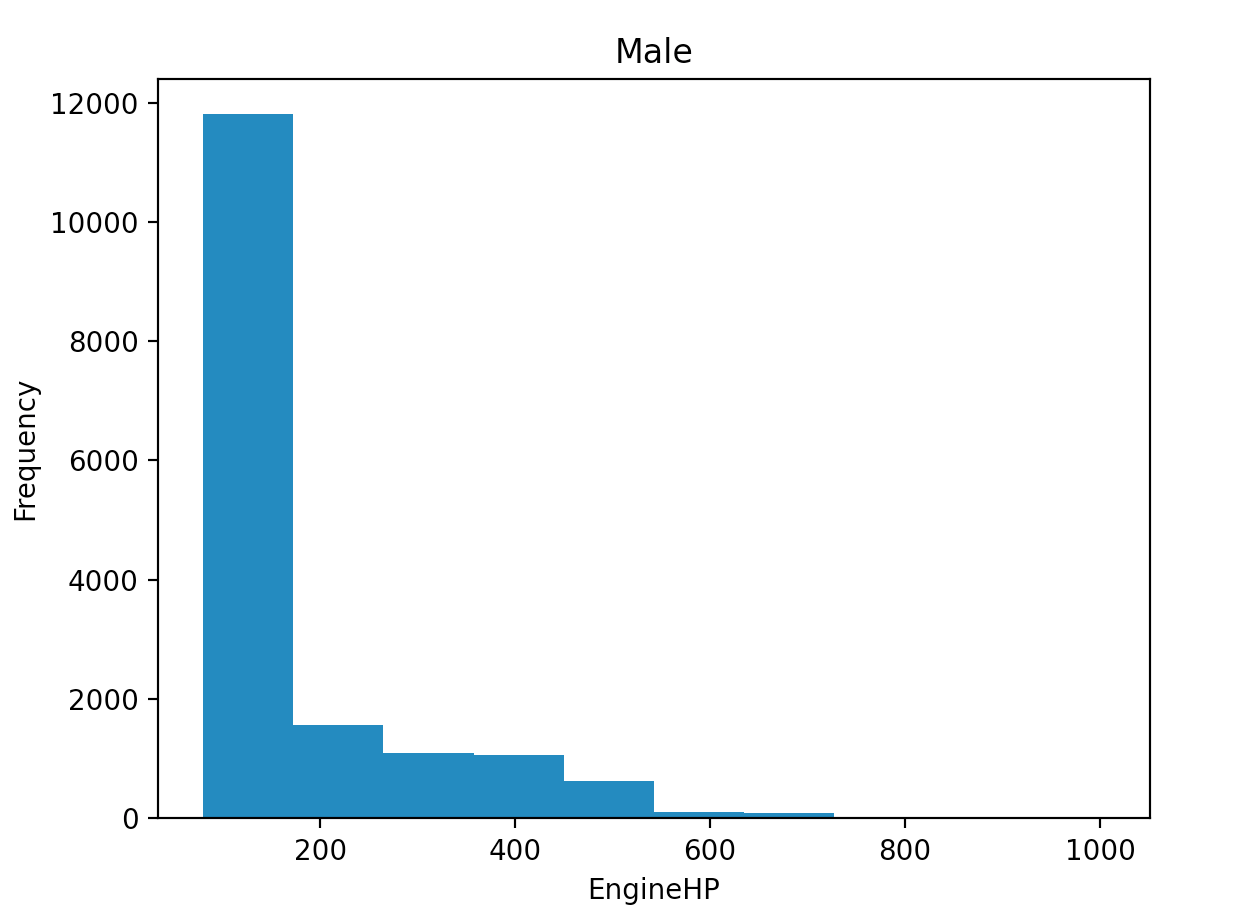
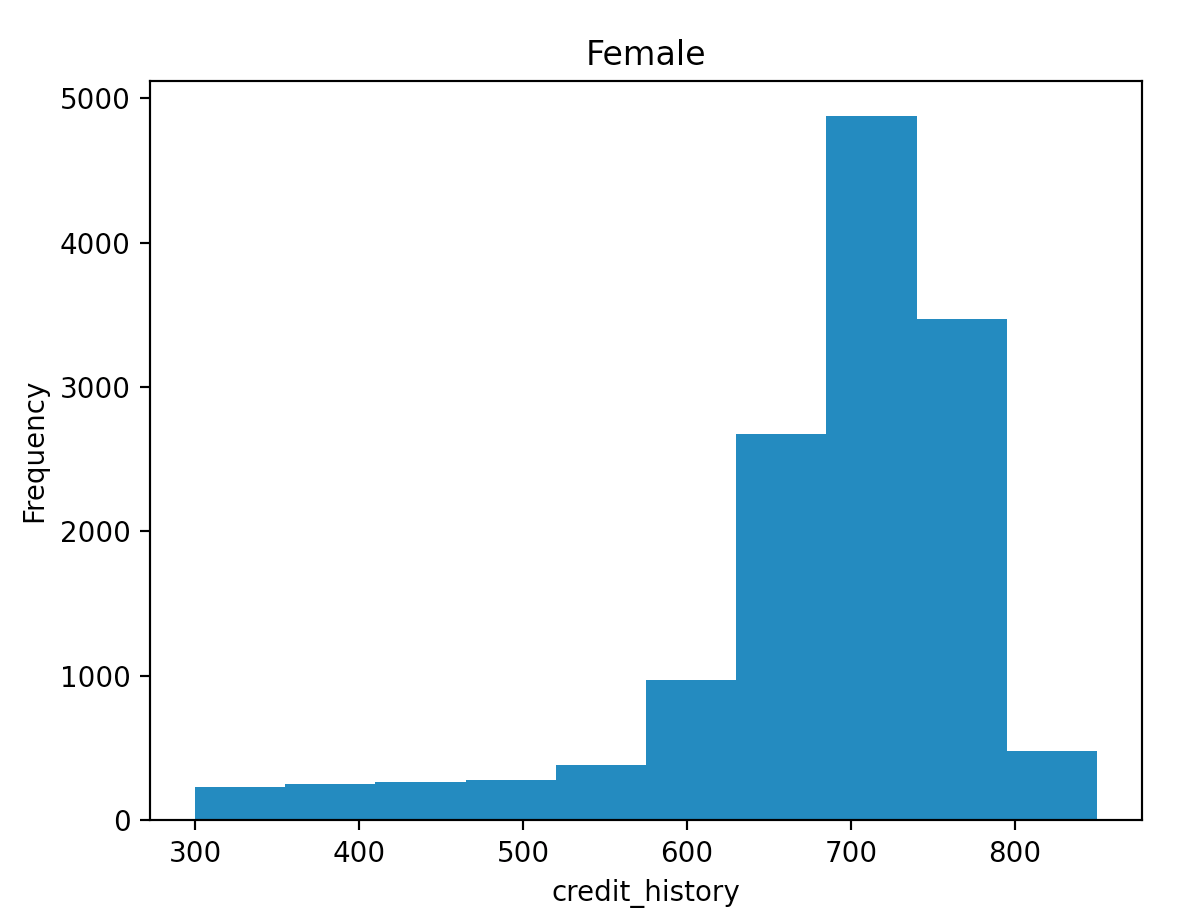
#Graph

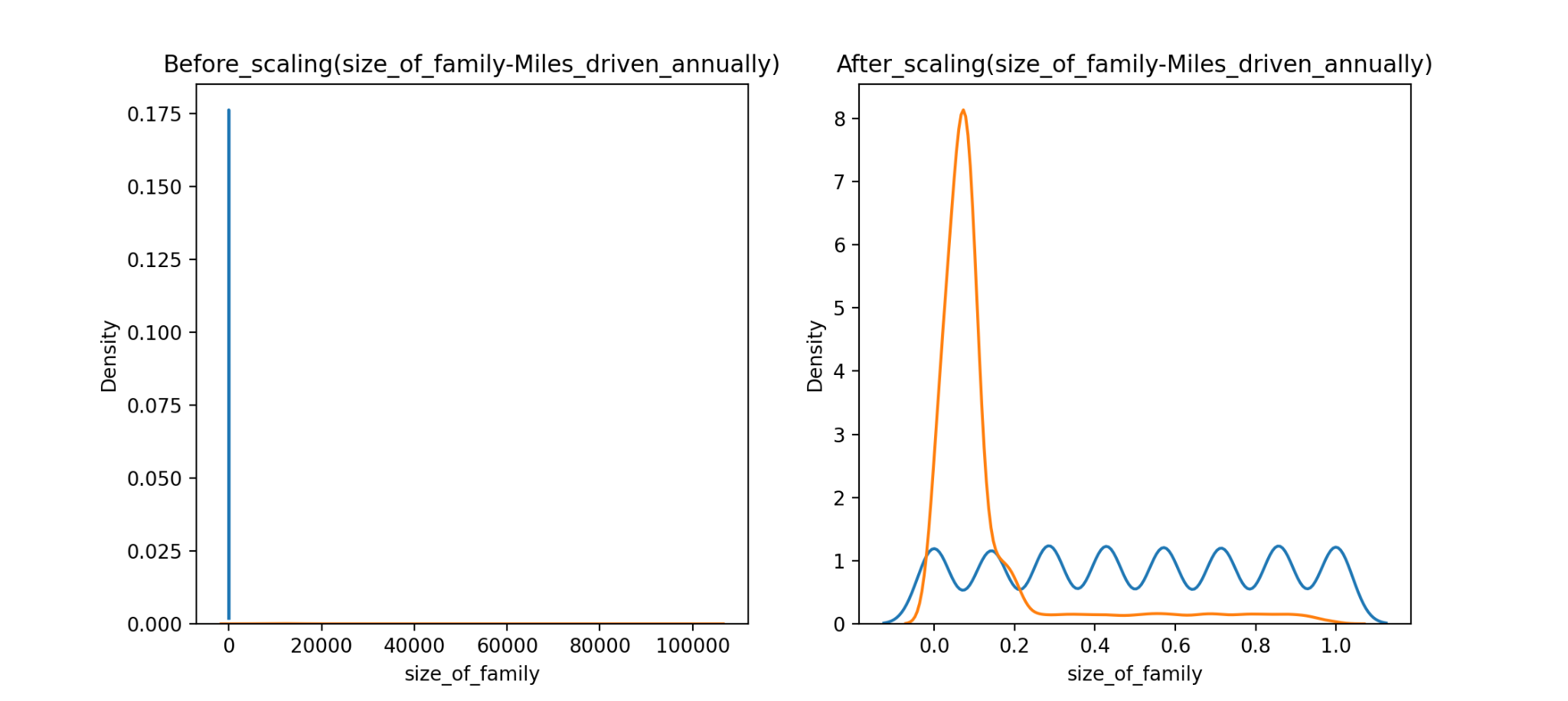
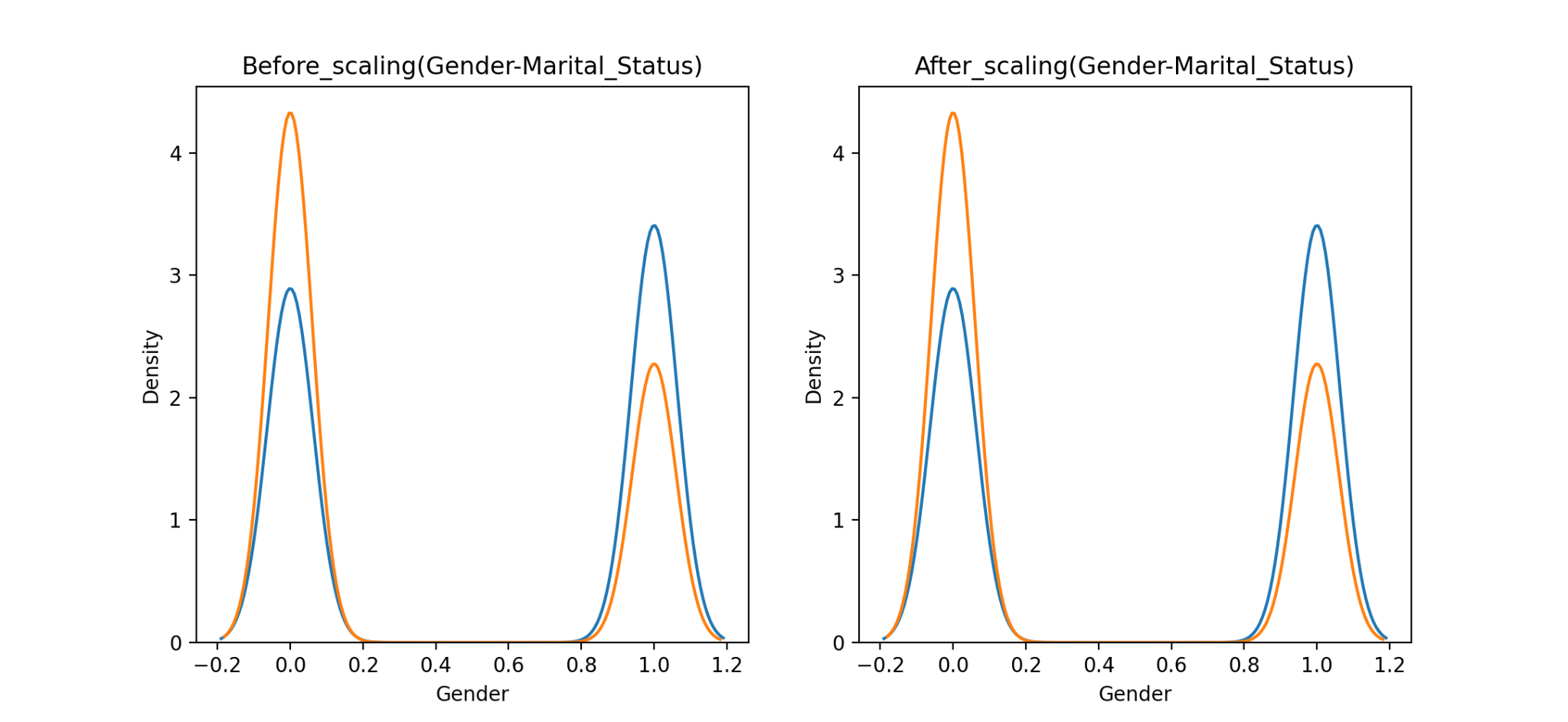
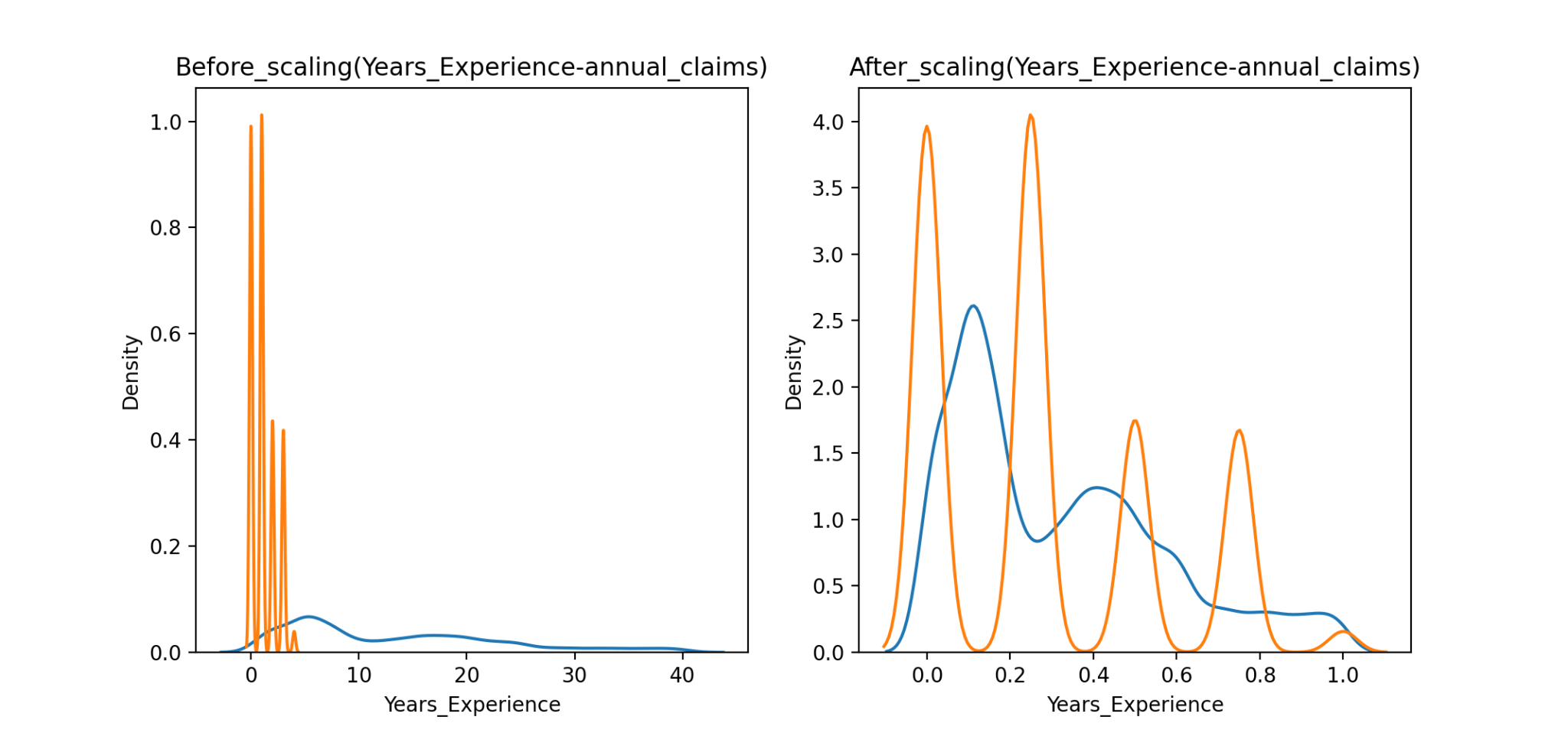
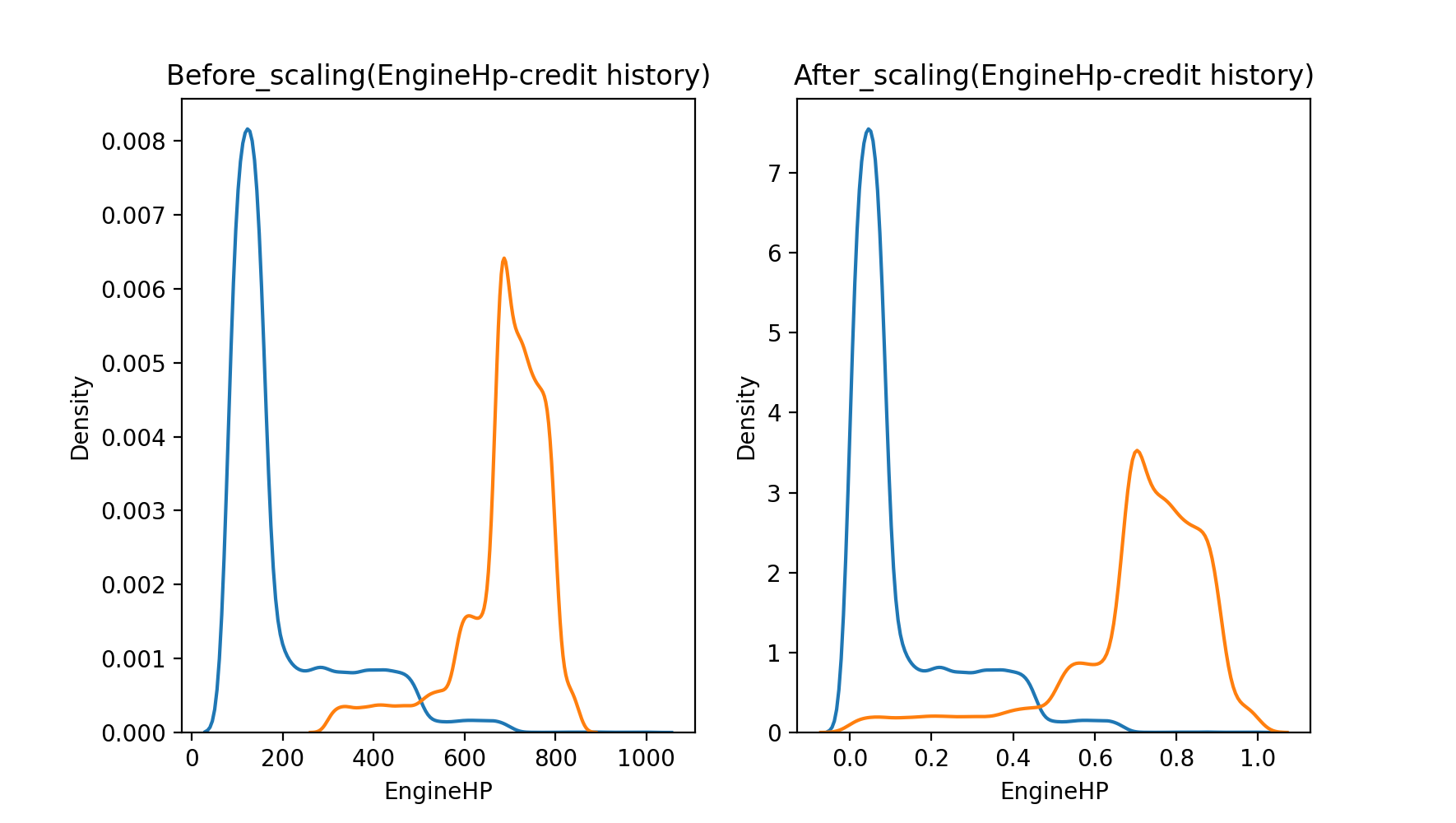
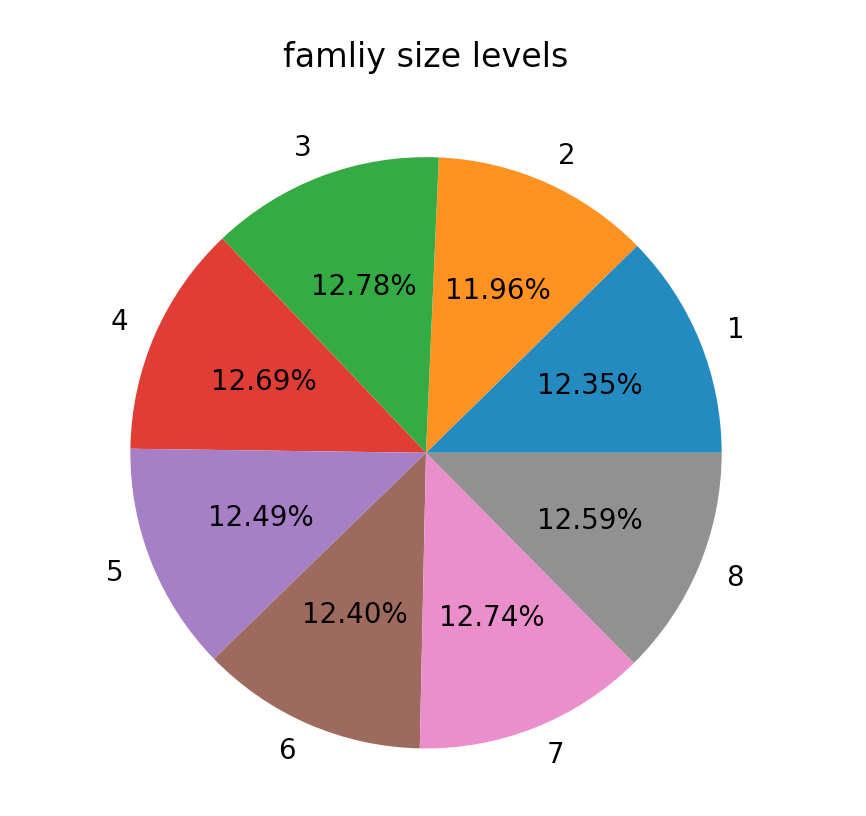
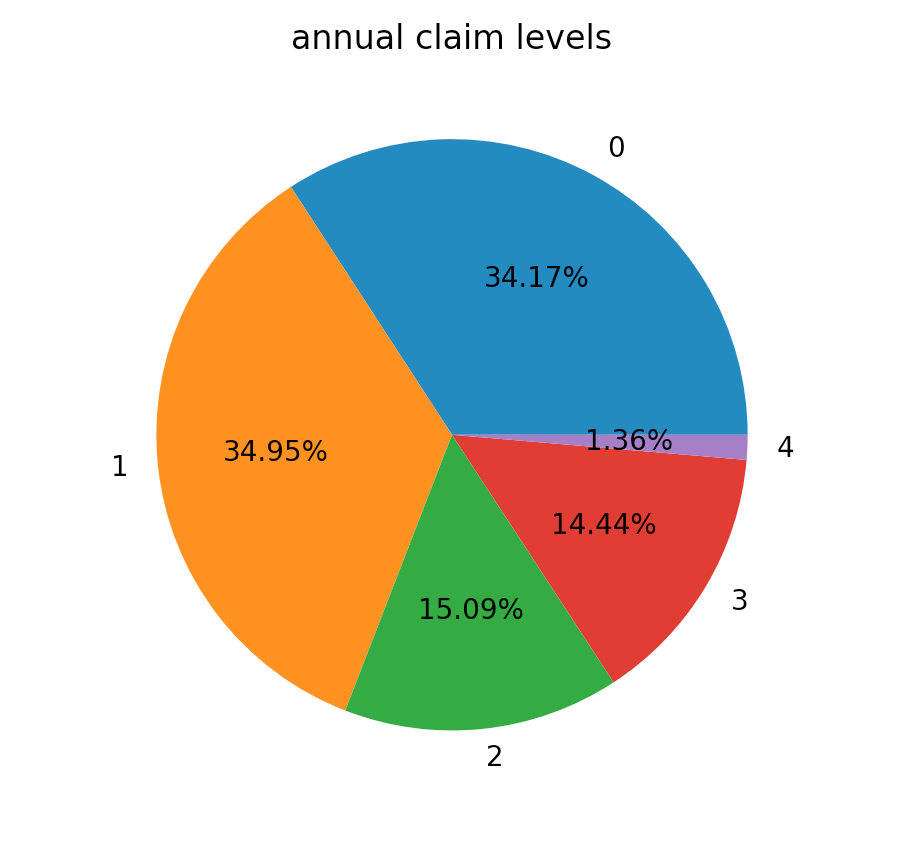
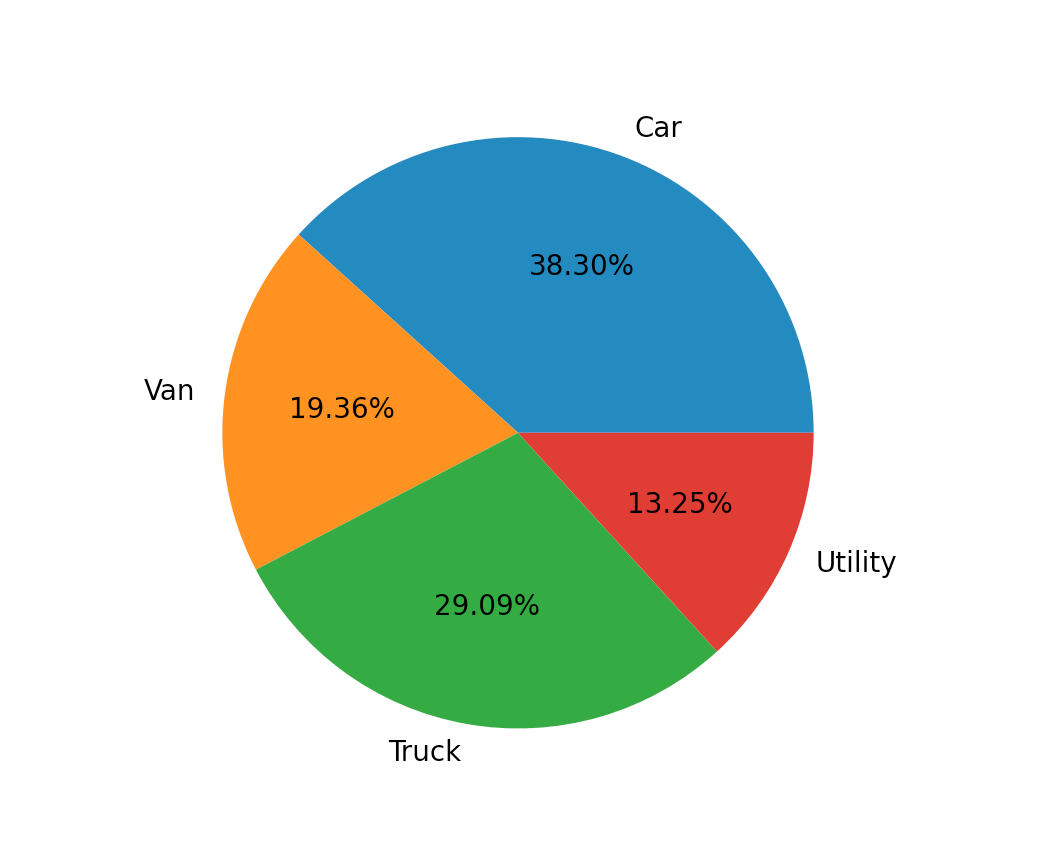




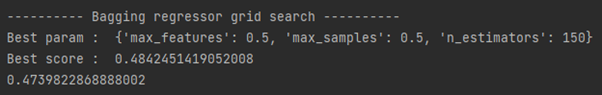




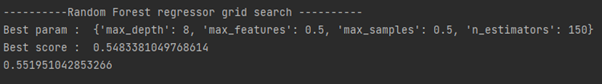




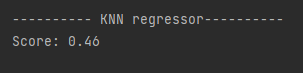
#Bagging regressor (grid search)



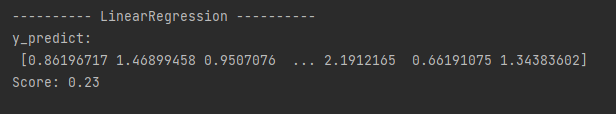
#Random forest regressor (grid search)



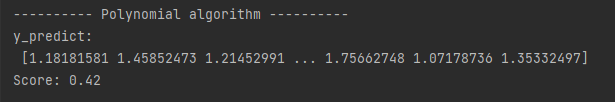
#Knn regressor



#Linear regressor



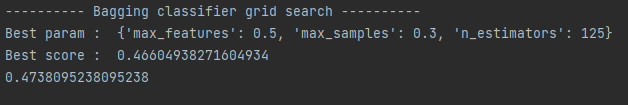
#Polynomial regressor



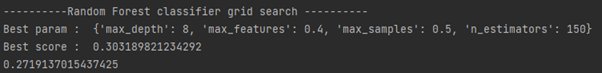
#K-mean clustering



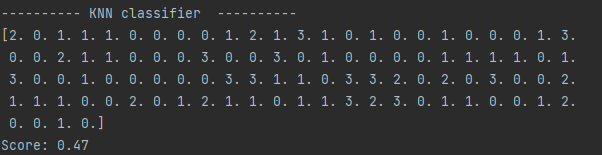
#Bagging classifier (grid search)



#Random forest classifier



#Knn classifier



#Module

