**PRE PROCESSING :**

import pandas as pd

import numpy as np

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

from sklearn.preprocessing import LabelEncoder, StandardScaler

# Step 1: Load the CSV data into a DataFrame

# Replace 'your\_data.csv' with the path to your CSV file

data = pd.read\_csv("C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv")

# Step 2: Display basic information and statistics about the dataset

print(data.info())

print(data.describe())

# Step 3: Check for missing values

missing\_values = data.isnull().sum()

print("Missing values in each column:\n", missing\_values)

# Step 4: Encode categorical variables

# Label encoding for binary columns like 'Sex', 'MaritalStatus', 'AccidentArea', 'FraudFound'

label\_encoders = {}

for column in ['Sex', 'MaritalStatus', 'AccidentArea', 'FraudFound', 'PoliceReportFiled', 'WitnessPresent', 'AgentType', 'BasePolicy']:

    le = LabelEncoder()

    data[column] = le.fit\_transform(data[column])

    label\_encoders[column] = le

# One-hot encode columns with more than 2 categories (e.g., 'Make', 'PolicyType', 'VehicleCategory')

data = pd.get\_dummies(data, columns=['Make', 'PolicyType', 'VehicleCategory'], drop\_first=True)

# Step 5: Handle missing values (if any)

# Filling missing values with mean for numerical columns

data.fillna(data.mean(), inplace=True)

# Step 6: Scale numerical features for better model performance

# Standardize numerical features like 'Age', 'ClaimAmount', 'VehiclePrice', etc.

scaler = StandardScaler()

numerical\_cols = ['Age', 'ClaimAmount', 'VehiclePrice', 'AgeOfVehicle', 'AgeOfPolicyHolder', 'DriverRating', 'NumberOfSuppliments']

data[numerical\_cols] = scaler.fit\_transform(data[numerical\_cols])

# Step 7: Save the preprocessed data as a single CSV file

# Assuming 'FraudFound' is the target variable

# Reorder the columns to keep the target column 'FraudFound' as the last column

data = data[[col for col in data.columns if col != 'FraudFound'] + ['FraudFound']]

# Step 8: Save the entire preprocessed dataset to a CSV file

data.to\_csv('final.csv', index=False)

print("Data preprocessing complete and saved as 'final.csv'.")

**EXPLORATORY DATA ANALYSIS :**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.metrics import mean\_squared\_error, roc\_auc\_score, roc\_curve, accuracy\_score, confusion\_matrix

from sklearn.ensemble import GradientBoostingClassifier

from scipy.stats import randint

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Display basic information and initial rows of the dataset

print("Dataset Information:")

print(data.info())

print("\nFirst 5 rows of the dataset:")

print(data.head())

# Handling missing values

print("\nMissing Values:")

print(data.isnull().sum())

data.dropna(inplace=True)

# Exploratory Data Analysis (EDA)

# 1. Distribution of the target variable

sns.countplot(x='FraudFound', data=data)

plt.title('Distribution of Fraud Found')

plt.xlabel('Fraud Found (0: No, 1: Yes)')

plt.ylabel('Count')

plt.show()

# 2. Distribution of numerical features

numerical\_cols = data.select\_dtypes(include=[np.number]).columns

data[numerical\_cols].hist(bins=15, figsize=(15, 10))

plt.suptitle('Distribution of Numerical Features')

plt.show()

# 3. Outlier Detection using box plots

num\_cols = len(numerical\_cols)

cols\_per\_row = 4

rows = (num\_cols // cols\_per\_row) + (num\_cols % cols\_per\_row > 0)

plt.figure(figsize=(20, 5 \* rows))

for i, col in enumerate(numerical\_cols):

    plt.subplot(rows, cols\_per\_row, i + 1)

    sns.boxplot(x=data[col])

    plt.title(f'Box Plot of {col}')

    plt.xlim(data[col].min() - 1, data[col].max() + 1)

plt.subplots\_adjust(hspace=0.95, top=0.933)

plt.show()

# Prepare the data for training

X = data.drop('FraudFound', axis=1)  # Features (all columns except target)

y = data['FraudFound']  # Target variable

# Encoding categorical variables if any

le = LabelEncoder()

for col in X.select\_dtypes(include=['object']).columns:

    X[col] = le.fit\_transform(X[col])

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Standardize the numerical features (optional but recommended for some models)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Initialize and train the Gradient Boosting Classifier

gbc = GradientBoostingClassifier(random\_state=42)

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

# Make predictions

y\_pred = gbc.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

# Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

print("\nConfusion Matrix:")

print(cm)

# Visualizing the Confusion Matrix

plt.figure(figsize=(6, 4))

sns.heatmap(cm, annot=True, fmt='g', cmap='Blues')

plt.title('Confusion Matrix')

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.show()

**GRADIENT BOOSTED NEURAL NETWORK   
FRAUD PREDICTION :**

#final 1

import pandas as pd

import numpy as np

from keras.models import Sequential

from keras.layers import Dense

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score, mean\_squared\_error

from sklearn.metrics import roc\_curve, roc\_auc\_score, mean\_absolute\_error

import matplotlib.pyplot as plt

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# One-hot encode categorical columns

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

categorical\_columns.remove('FraudFound')

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Prepare features and target

X = data.drop(columns=['FraudFound']).values

y = data['FraudFound'].map({'Yes': 1, 'No': 0}).values

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Define a function to create the neural network

def create\_model():

    model = Sequential()

    model.add(Dense(32, activation='relu', input\_dim=X\_train.shape[1]))  # Input layer

    model.add(Dense(16, activation='relu'))  # Hidden layer

    model.add(Dense(1, activation='sigmoid'))  # Output layer

    model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

# Implementing a simple version of Gradient Boosted Neural Networks

def gradient\_boosted\_nn(X\_train, y\_train, X\_test, n\_estimators=5):

    n\_samples = len(y\_train)

    y\_pred = np.zeros(n\_samples)

    models = []

    # Initial model with zero predictions (base predictions)

    for \_ in range(n\_estimators):

        # Calculate the residuals

        residuals = y\_train - y\_pred

        # Create a new model for the residuals

        model = create\_model()

        model.fit(X\_train, residuals, epochs=10, batch\_size=32, verbose=0)

        # Update predictions

        y\_pred += model.predict(X\_train).flatten()  # Flatten to match y\_train shape

        models.append(model)

    return models, y\_pred

# Train the Gradient Boosted Neural Networks

models, y\_train\_pred = gradient\_boosted\_nn(X\_train, y\_train, X\_test)

# Make predictions on the test set

y\_test\_pred = np.zeros(len(y\_test))

for model in models:

    y\_test\_pred += model.predict(X\_test).flatten()

# Convert predictions to binary (using a threshold of 0.5)

y\_test\_pred\_binary = (y\_test\_pred > 0.5).astype(int)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_test\_pred\_binary)

mse = mean\_squared\_error(y\_test, y\_test\_pred\_binary)

mae = mean\_absolute\_error(y\_test, y\_test\_pred\_binary)

roc\_auc = roc\_auc\_score(y\_test, y\_test\_pred)

# Print evaluation metrics

print(f"Gradient Boosted Neural Networks - Accuracy: {accuracy:.4f}, MSE: {mse:.4f}, MAE: {mae:.4f}, ROC AUC: {roc\_auc:.4f}")

#parameter tuning

import pandas as pd

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score, mean\_squared\_error, roc\_curve, roc\_auc\_score, mean\_absolute\_error

import matplotlib.pyplot as plt

from sklearn.model\_selection import RandomizedSearchCV

from tensorflow.keras.layers import Input

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# One-hot encode categorical columns

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

categorical\_columns.remove('FraudFound')

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Prepare features and target

X = data.drop(columns=['FraudFound']).values

y = data['FraudFound'].map({'Yes': 1, 'No': 0}).values

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Define a function to create the neural network

def create\_model(neurons\_1=32, neurons\_2=16):

    model = Sequential()

    model.add(Input(shape=(X\_train.shape[1],)))  # Input layer using Input

    model.add(Dense(neurons\_1, activation='relu'))  # Hidden layer

    model.add(Dense(neurons\_2, activation='relu'))  # Hidden layer

    model.add(Dense(1, activation='sigmoid'))  # Output layer

    model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

# Function to perform hyperparameter tuning

def hyperparameter\_tuning(X\_train, y\_train):

    param\_dist = {

        'neurons\_1': [16, 32, 64],

        'neurons\_2': [8, 16, 32],

        'epochs': [10, 20],

        'batch\_size': [16, 32]

    }

    best\_accuracy = 0

    best\_params = {}

    # Random search over specified parameter values

    for neurons\_1 in param\_dist['neurons\_1']:

        for neurons\_2 in param\_dist['neurons\_2']:

            for epochs in param\_dist['epochs']:

                for batch\_size in param\_dist['batch\_size']:

                    # Create and train model

                    model = create\_model(neurons\_1=neurons\_1, neurons\_2=neurons\_2)

                    model.fit(X\_train, y\_train, epochs=epochs, batch\_size=batch\_size, verbose=0)

                    # Evaluate model

                    y\_train\_pred = model.predict(X\_train).flatten()

                    y\_train\_pred\_binary = (y\_train\_pred > 0.5).astype(int)

                    accuracy = accuracy\_score(y\_train, y\_train\_pred\_binary)

                    # Check if this is the best model

                    if accuracy > best\_accuracy:

                        best\_accuracy = accuracy

                        best\_params = {

                            'neurons\_1': neurons\_1,

                            'neurons\_2': neurons\_2,

                            'epochs': epochs,

                            'batch\_size': batch\_size

                        }

    return best\_params

# Perform hyperparameter tuning

best\_params = hyperparameter\_tuning(X\_train, y\_train)

print("Best Parameters: ", best\_params)

# Train the model with the best parameters

best\_model = create\_model(neurons\_1=best\_params['neurons\_1'],

                           neurons\_2=best\_params['neurons\_2'])

best\_model.fit(X\_train, y\_train, epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'], verbose=1)

# Make predictions on the test set

y\_test\_pred = best\_model.predict(X\_test).flatten()

# Convert predictions to binary (using a threshold of 0.5)

y\_test\_pred\_binary = (y\_test\_pred > 0.5).astype(int)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_test\_pred\_binary)

mse = mean\_squared\_error(y\_test, y\_test\_pred\_binary)

mae = mean\_absolute\_error(y\_test, y\_test\_pred\_binary)

roc\_auc = roc\_auc\_score(y\_test, y\_test\_pred)

# Print evaluation metrics

print(f"Final Model with Best Parameters - Accuracy: {accuracy:.4f}, MSE: {mse:.4f}, MAE: {mae:.4f}, ROC AUC: {roc\_auc:.4f}")

# Plot ROC Curve

fpr, tpr, thresholds = roc\_curve(y\_test, y\_test\_pred)

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, label='ROC Curve (area = {:.2f})'.format(roc\_auc))

plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc='lower right')

plt.grid()

plt.show()

import pandas as pd

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score, mean\_squared\_error, roc\_curve, roc\_auc\_score, mean\_absolute\_error

import matplotlib.pyplot as plt

from sklearn.model\_selection import RandomizedSearchCV

from tensorflow.keras.layers import Input

from tensorflow.keras.layers import Dropout

from tensorflow.keras.callbacks import EarlyStopping

from sklearn.model\_selection import KFold

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# One-hot encode categorical columns

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

categorical\_columns.remove('FraudFound')

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Prepare features and target

X = data.drop(columns=['FraudFound']).values

y = data['FraudFound'].map({'Yes': 1, 'No': 0}).values

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Define a function to create the neural network with Dropout layers

def create\_model(neurons\_1=32, neurons\_2=16, dropout\_rate=0.2):

    model = Sequential()

    model.add(Input(shape=(X\_train.shape[1],)))  # Input layer

    model.add(Dense(neurons\_1, activation='relu'))  # Hidden layer

    model.add(Dropout(dropout\_rate))  # Dropout layer to prevent overfitting

    model.add(Dense(neurons\_2, activation='relu'))  # Hidden layer

    model.add(Dropout(dropout\_rate))  # Dropout layer to prevent overfitting

    model.add(Dense(1, activation='sigmoid'))  # Output layer

    model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

# Implement k-fold cross-validation to prevent overfitting

def k\_fold\_validation(X, y, n\_splits=5):

    kfold = KFold(n\_splits=n\_splits, shuffle=True, random\_state=42)

    fold\_no = 1

    results = []

    for train\_idx, val\_idx in kfold.split(X):

        X\_train\_fold, X\_val\_fold = X[train\_idx], X[val\_idx]

        y\_train\_fold, y\_val\_fold = y[train\_idx], y[val\_idx]

        # Create the model

        model = create\_model(neurons\_1=best\_params['neurons\_1'],

                             neurons\_2=best\_params['neurons\_2'])

        # Early stopping to avoid overfitting

        early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

        # Train the model

        model.fit(X\_train\_fold, y\_train\_fold, epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'],

                  validation\_data=(X\_val\_fold, y\_val\_fold), callbacks=[early\_stopping], verbose=1)

        # Evaluate the model on the validation set

        y\_val\_pred = model.predict(X\_val\_fold).flatten()

        y\_val\_pred\_binary = (y\_val\_pred > 0.5).astype(int)

        accuracy = accuracy\_score(y\_val\_fold, y\_val\_pred\_binary)

        results.append(accuracy)

        print(f"Fold {fold\_no} - Accuracy: {accuracy:.4f}")

        fold\_no += 1

    avg\_accuracy = np.mean(results)

    print(f"Average accuracy after k-fold cross-validation: {avg\_accuracy:.4f}")

    return avg\_accuracy

# Perform k-fold cross-validation to evaluate model performance

avg\_accuracy = k\_fold\_validation(X\_train, y\_train)

# Train the model with the best parameters and early stopping on the entire training set

best\_model = create\_model(neurons\_1=best\_params['neurons\_1'],

                          neurons\_2=best\_params['neurons\_2'])

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

best\_model.fit(X\_train, y\_train, epochs=best\_params['epochs'], batch\_size=best\_params['batch\_size'],

               validation\_split=0.2, callbacks=[early\_stopping], verbose=1)

# Make predictions on the test set

y\_test\_pred = best\_model.predict(X\_test).flatten()

# Convert predictions to binary (using a threshold of 0.5)

y\_test\_pred\_binary = (y\_test\_pred > 0.5).astype(int)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_test\_pred\_binary)

mse = mean\_squared\_error(y\_test, y\_test\_pred\_binary)

mae = mean\_absolute\_error(y\_test, y\_test\_pred\_binary)

roc\_auc = roc\_auc\_score(y\_test, y\_test\_pred)

# Print evaluation metrics

print(f"Final Model with Best Parameters - Accuracy: {accuracy:.4f}, MSE: {mse:.4f}, MAE: {mae:.4f}, ROC AUC: {roc\_auc:.4f}")

# Plot ROC Curve

fpr, tpr, thresholds = roc\_curve(y\_test, y\_test\_pred)

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, label='ROC Curve (area = {:.2f})'.format(roc\_auc))

plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc='lower right')

plt.grid()

plt.show()

**Policy type prediction :**

import pandas as pd

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, Input

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

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.metrics import accuracy\_score, mean\_squared\_error, roc\_curve, roc\_auc\_score, mean\_absolute\_error

import matplotlib.pyplot as plt

from tensorflow.keras.callbacks import EarlyStopping

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# Encode the target variable (PolicyType)

label\_encoder = LabelEncoder()

data['PolicyType'] = label\_encoder.fit\_transform(data['PolicyType'])

# One-hot encode categorical columns

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Prepare features and target

X = data.drop(columns=['PolicyType']).values

y = data['PolicyType'].values

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize the features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Define a function to create the neural network model with Dropout layers

def create\_model(neurons\_1=32, neurons\_2=16, dropout\_rate=0.2):

    model = Sequential()

    model.add(Input(shape=(X\_train.shape[1],)))  # Input layer

    model.add(Dense(neurons\_1, activation='relu'))  # Hidden layer 1

    model.add(Dropout(dropout\_rate))  # Dropout layer to prevent overfitting

    model.add(Dense(neurons\_2, activation='relu'))  # Hidden layer 2

    model.add(Dropout(dropout\_rate))  # Dropout layer

    model.add(Dense(len(np.unique(y)), activation='softmax'))  # Output layer for multi-class classification

    model.compile(loss='sparse\_categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

# Implement k-fold cross-validation to prevent overfitting

def k\_fold\_validation(X, y, n\_splits=5):

    kfold = KFold(n\_splits=n\_splits, shuffle=True, random\_state=42)

    fold\_no = 1

    results = []

    for train\_idx, val\_idx in kfold.split(X):

        X\_train\_fold, X\_val\_fold = X[train\_idx], X[val\_idx]

        y\_train\_fold, y\_val\_fold = y[train\_idx], y[val\_idx]

        # Create the model

        model = create\_model(neurons\_1=32, neurons\_2=16, dropout\_rate=0.3)

        # Early stopping to avoid overfitting

        early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

        # Train the model

        model.fit(X\_train\_fold, y\_train\_fold, epochs=50, batch\_size=32,

                  validation\_data=(X\_val\_fold, y\_val\_fold), callbacks=[early\_stopping], verbose=1)

        # Evaluate the model on the validation set

        y\_val\_pred = np.argmax(model.predict(X\_val\_fold), axis=1)

        accuracy = accuracy\_score(y\_val\_fold, y\_val\_pred)

        results.append(accuracy)

        print(f"Fold {fold\_no} - Accuracy: {accuracy:.4f}")

        fold\_no += 1

    avg\_accuracy = np.mean(results)

    print(f"Average accuracy after k-fold cross-validation: {avg\_accuracy:.4f}")

    return avg\_accuracy

# Perform k-fold cross-validation to evaluate model performance

avg\_accuracy = k\_fold\_validation(X\_train, y\_train)

# Create and train the final model using the best parameters

best\_model = create\_model(neurons\_1=32, neurons\_2=16, dropout\_rate=0.3)

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

# Train on the entire training set with validation split

best\_model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_split=0.2, callbacks=[early\_stopping], verbose=1)

# Make predictions on the test set

y\_test\_pred = np.argmax(best\_model.predict(X\_test), axis=1)

# Evaluate the final model

accuracy = accuracy\_score(y\_test, y\_test\_pred)

mse = mean\_squared\_error(y\_test, y\_test\_pred)

mae = mean\_absolute\_error(y\_test, y\_test\_pred)

roc\_auc = roc\_auc\_score(y\_test, best\_model.predict(X\_test), multi\_class="ovr")

# Print evaluation metrics

print(f"Final Model with Best Parameters - Accuracy: {accuracy:.4f}, MSE: {mse:.4f}, MAE: {mae:.4f}, ROC AUC: {roc\_auc:.4f}")

# Plot ROC Curve

y\_test\_pred\_prob = best\_model.predict(X\_test)

fpr = {}

tpr = {}

for i in range(len(np.unique(y))):

    fpr[i], tpr[i], \_ = roc\_curve(y\_test, y\_test\_pred\_prob[:, i], pos\_label=i)

plt.figure(figsize=(8, 6))

for i in range(len(np.unique(y))):

    plt.plot(fpr[i], tpr[i], label=f'Class {i} ROC Curve')

plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc='lower right')

plt.grid()

plt.show()

**FEED FORWARD NEURAL NETWORKS**

**Fraud detection :**

import pandas as pd

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, roc\_auc\_score

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# One-hot encode categorical columns

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

categorical\_columns.remove('FraudFound')

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Prepare features and target

X = data.drop(columns=['FraudFound']).values

y = data['FraudFound'].map({'Yes': 1, 'No': 0}).values

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Define a function to create the neural network

def create\_model():

    model = Sequential()

    model.add(Dense(32, activation='relu', input\_dim=X\_train.shape[1]))  # Input layer

    model.add(Dense(16, activation='relu'))  # Hidden layer

    model.add(Dense(1, activation='sigmoid'))  # Output layer

    model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

# Create and train the model

model = create\_model()

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2, verbose=1)

# Evaluate the model

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print(f'\nTest accuracy: {test\_acc:.4f}')

# Make predictions on the test set

y\_test\_pred = model.predict(X\_test)

y\_test\_pred\_classes = (y\_test\_pred > 0.5).astype(int).flatten()

# Calculate additional metrics

mse = mean\_squared\_error(y\_test, y\_test\_pred\_classes)

mae = mean\_absolute\_error(y\_test, y\_test\_pred\_classes)

rmse = np.sqrt(mse)

roc\_auc = roc\_auc\_score(y\_test, y\_test\_pred)

# Print the metrics

print(f'Accuracy: {test\_acc:.4f}')

print(f'MSE: {mse:.4f}')

print(f'RMSE: {rmse:.4f}')

print(f'MAE: {mae:.4f}')

print(f'ROC AUC: {roc\_auc:.4f}')

#parameter tuning

import pandas as pd

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, roc\_auc\_score

import keras\_tuner as kt

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# One-hot encode categorical columns

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

categorical\_columns.remove('FraudFound')

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Prepare features and target

X = data.drop(columns=['FraudFound']).values

y = data['FraudFound'].map({'Yes': 1, 'No': 0}).values

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Define the model building function with hyperparameters

def build\_model(hp):

    model = Sequential()

    # Choose number of hidden layers

    num\_hidden\_layers = hp.Int('num\_hidden\_layers', min\_value=1, max\_value=3)

    # Add input layer

    model.add(Dense(units=hp.Int('neurons\_input\_layer', min\_value=16, max\_value=64, step=16),

                    activation=hp.Choice('activation\_input\_layer', values=['relu', 'tanh']),

                    input\_dim=X\_train.shape[1]))

    # Add hidden layers

    for \_ in range(num\_hidden\_layers - 1):

        model.add(Dense(units=hp.Int('neurons\_hidden\_layer', min\_value=16, max\_value=64, step=16),

                        activation=hp.Choice('activation\_hidden\_layer', values=['relu', 'tanh'])))

    # Add output layer

    model.add(Dense(1, activation='sigmoid'))  # Output layer

    model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

# Set up the tuner

tuner = kt.tuners.RandomSearch(

    build\_model,

    objective='val\_accuracy',

    max\_trials=10,

    executions\_per\_trial=3,

    directory='my\_dir',

    project\_name='fraud\_detection'

)

# Perform the hyperparameter search

tuner.search(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Get the best model

best\_model = tuner.get\_best\_models(num\_models=1)[0]

# Evaluate the best model on the test set

test\_loss, test\_acc = best\_model.evaluate(X\_test, y\_test)

print(f'\nBest Test accuracy: {test\_acc:.4f}')

# Make predictions on the test set

y\_test\_pred = best\_model.predict(X\_test)

y\_test\_pred\_classes = (y\_test\_pred > 0.5).astype(int).flatten()

# Calculate additional metrics

mse = mean\_squared\_error(y\_test, y\_test\_pred\_classes)

mae = mean\_absolute\_error(y\_test, y\_test\_pred\_classes)

rmse = np.sqrt(mse)

roc\_auc = roc\_auc\_score(y\_test, y\_test\_pred)

# Print the metrics

print(f'Best Accuracy: {test\_acc:.4f}')

print(f'MSE: {mse:.4f}')

print(f'RMSE: {rmse:.4f}')

print(f'MAE: {mae:.4f}')

print(f'ROC AUC: {roc\_auc:.4f}')

**Policy type prediction :**

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from sklearn.metrics import accuracy\_score, mean\_squared\_error, mean\_absolute\_error, roc\_auc\_score

from sklearn.preprocessing import StandardScaler, LabelEncoder

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

import pandas as pd

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# Encode the target variable (PolicyType) using LabelEncoder

label\_encoder = LabelEncoder()

data['PolicyType'] = label\_encoder.fit\_transform(data['PolicyType'])

# One-hot encode categorical features

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Separate features and target variable

X = data.drop(columns=['PolicyType']).values

y = data['PolicyType'].values

# Split the data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize the features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# One-hot encode the target variable (y\_train and y\_test) for multiclass classification

y\_train\_onehot = pd.get\_dummies(y\_train).values

y\_test\_onehot = pd.get\_dummies(y\_test).values

# Define a function to create the neural network for multiclass classification

def create\_feedforward\_nn():

    model = Sequential()

    model.add(Dense(32, activation='relu', input\_dim=X\_train.shape[1]))  # Input layer

    model.add(Dense(16, activation='relu'))  # Hidden layer

    model.add(Dense(y\_train\_onehot.shape[1], activation='softmax'))  # Output layer for multiclass classification

    model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

# Create and train the feedforward neural network

model = create\_feedforward\_nn()

model.fit(X\_train, y\_train\_onehot, epochs=10, batch\_size=32, verbose=0)  # Train for 50 epochs

# Make predictions on the test set

y\_test\_pred\_proba = model.predict(X\_test)

# Convert predicted probabilities to class labels

y\_test\_pred = np.argmax(y\_test\_pred\_proba, axis=1)

# Evaluate the model using accuracy, MSE, MAE, and ROC AUC

accuracy = accuracy\_score(y\_test, y\_test\_pred)

mse = mean\_squared\_error(y\_test, y\_test\_pred)

mae = mean\_absolute\_error(y\_test, y\_test\_pred)

roc\_auc = roc\_auc\_score(y\_test\_onehot, y\_test\_pred\_proba, multi\_class='ovr')

# Print evaluation metrics

print(f"Feedforward Neural Networks - Accuracy for Policy Type: {accuracy:.4f}")

print(f"MSE: {mse:.4f}, MAE: {mae:.4f}, ROC AUC: {roc\_auc:.4f}")

#parametertuning

import numpy as np

import pandas as pd

from keras.models import Sequential

from keras.layers import Dense, Dropout, Input

from scikeras.wrappers import KerasClassifier

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

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.metrics import accuracy\_score, mean\_squared\_error, mean\_absolute\_error, roc\_auc\_score, roc\_curve

from keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

# Load the data

file\_path = "C:\\Omega\\Semester 5\\Machine Learning\\Project\\final.csv"

data = pd.read\_csv(file\_path)

# Clean column names

data.columns = data.columns.str.strip()

# Encode the target variable (PolicyType)

label\_encoder = LabelEncoder()

data['PolicyType'] = label\_encoder.fit\_transform(data['PolicyType'])

# One-hot encode categorical columns

categorical\_columns = data.select\_dtypes(include=['object']).columns.tolist()

if categorical\_columns:

    data = pd.get\_dummies(data, columns=categorical\_columns, drop\_first=True)

# Prepare features and target

X = data.drop(columns=['PolicyType']).values

y = data['PolicyType'].values

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize the features

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Function to create model (with hyperparameters for tuning)

def create\_model(neurons\_1=32, neurons\_2=16, dropout\_rate=0.2, optimizer='adam'):

    model = Sequential()

    model.add(Input(shape=(X\_train.shape[1],)))  # Input layer

    model.add(Dense(neurons\_1, activation='relu'))  # First hidden layer

    model.add(Dropout(dropout\_rate))  # Dropout layer

    model.add(Dense(neurons\_2, activation='relu'))  # Second hidden layer

    model.add(Dropout(dropout\_rate))  # Dropout layer

    model.add(Dense(len(np.unique(y)), activation='softmax'))  # Output layer

    model.compile(loss='sparse\_categorical\_crossentropy', optimizer=optimizer, metrics=['accuracy'])

    return model

# Wrap the model using KerasClassifier for scikit-learn compatibility

model = KerasClassifier(model=create\_model, verbose=0)

# Define the hyperparameters to tune (note the prefix model\_\_)

param\_grid = {

    'model\_\_neurons\_1': [32, 64, 128],

    'model\_\_neurons\_2': [16, 32, 64],

    'model\_\_dropout\_rate': [0.2, 0.3, 0.4],

    'batch\_size': [32, 64, 128],

    'epochs': [10, 20, 30],

    'optimizer': ['adam', 'rmsprop']

}

# Use RandomizedSearchCV for hyperparameter tuning

random\_search = RandomizedSearchCV(estimator=model, param\_distributions=param\_grid, n\_iter=10, cv=3, n\_jobs=-1, random\_state=42)

random\_search\_result = random\_search.fit(X\_train, y\_train)

# Get the best hyperparameters from the RandomizedSearchCV

print(f"Best Hyperparameters: {random\_search\_result.best\_params\_}")

# Get the best model

best\_model = random\_search\_result.best\_estimator\_

# Early stopping callback to avoid overfitting during training

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

# Train the model with the best hyperparameters

best\_model.fit(X\_train, y\_train, validation\_split=0.2, epochs=random\_search\_result.best\_params\_['epochs'],

               batch\_size=random\_search\_result.best\_params\_['batch\_size'], callbacks=[early\_stopping], verbose=1)

# Make predictions on the test set

y\_test\_pred = best\_model.predict(X\_test)

if len(y\_test\_pred.shape) == 1:  # If predictions are 1D, treat them as class labels

    y\_test\_pred = y\_test\_pred

else:  # Otherwise, get the class with the highest probability

    y\_test\_pred = np.argmax(y\_test\_pred, axis=1)

# Calculate evaluation metrics

accuracy = accuracy\_score(y\_test, y\_test\_pred)

mse = mean\_squared\_error(y\_test, y\_test\_pred)

mae = mean\_absolute\_error(y\_test, y\_test\_pred)

roc\_auc = roc\_auc\_score(y\_test, best\_model.predict\_proba(X\_test), multi\_class="ovr")

# Print evaluation metrics

print(f"Final Model - Accuracy: {accuracy:.4f}")

print(f"MSE: {mse:.4f}")

print(f"MAE: {mae:.4f}")

print(f"ROC AUC: {roc\_auc:.4f}")

# Plot ROC Curve for each class

y\_test\_pred\_prob = best\_model.predict\_proba(X\_test)

fpr = {}

tpr = {}

# Plot ROC curve for each class

plt.figure(figsize=(8, 6))

for i in range(len(np.unique(y))):

    fpr[i], tpr[i], \_ = roc\_curve(y\_test, y\_test\_pred\_prob[:, i], pos\_label=i)

    plt.plot(fpr[i], tpr[i], label=f'Class {i} ROC Curve')

plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc='lower right')

plt.grid()

plt.show()

**LSTM**

**Fraud detection ;**

import pandas as pd

import numpy as np

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

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.linear\_model import LogisticRegression

from sklearn.feature\_selection import RFE

from sklearn.metrics import mean\_squared\_error, roc\_auc\_score, accuracy\_score

from tensorflow.keras.models import Sequential, load\_model

from tensorflow.keras.layers import LSTM, Dense, Dropout

from keras\_tuner import RandomSearch

from tensorflow.keras.optimizers import Adam

from math import sqrt

import matplotlib.pyplot as plt

import plotly.graph\_objs as go

import plotly.io as pio

# Load your dataset (adjust the path if needed)

file\_path = "/Users/i.seviantojensima/Desktop/Sem 5/Machine Learning/ml project/final.csv"

data = pd.read\_csv(file\_path)

# Sample a smaller dataset for faster testing

data\_sample = data.sample(n=1000, random\_state=42)  # Adjust n as needed

# Separate features and target variable

X = data\_sample.drop(columns=['FraudFound'])  # Replace 'FraudFound' with your target column name if different

y = data\_sample['FraudFound'].apply(lambda x: 1 if x == 'Yes' else 0)  # Convert to binary

# Encode categorical features using LabelEncoder

X\_encoded = X.copy()

label\_encoders = {}

for column in X.select\_dtypes(include=['object']).columns:

    le = LabelEncoder()

    X\_encoded[column] = le.fit\_transform(X[column])

    label\_encoders[column] = le

# Standardize the features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X\_encoded)

# Split the data into training and testing sets for RFE

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)

# Initialize a simpler model for RFE

model = LogisticRegression(max\_iter=1000)  # Ensure enough iterations

# Initialize RFE with the LogisticRegression as the estimator

n\_features\_to\_select = 5

rfe = RFE(estimator=model, n\_features\_to\_select=n\_features\_to\_select)

# Fit RFE

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

# Get the selected features

selected\_features = X\_encoded.columns[rfe.support\_]

print("Selected Features:")

print(selected\_features)

# Visualize feature ranking

ranking = pd.DataFrame({'Feature': X\_encoded.columns, 'Ranking': rfe.ranking\_})

print("\nFeature Ranking:")

print(ranking.sort\_values(by='Ranking'))

plt.figure(figsize=(12, 6))

plt.barh(ranking['Feature'], ranking['Ranking'], color='skyblue')

plt.xlabel('Ranking')

plt.title('Feature Ranking using RFE')

plt.gca().invert\_yaxis()

plt.show()

# Use the selected features for LSTM model training

X\_train\_rfe = X\_train[:, rfe.support\_]

X\_test\_rfe = X\_test[:, rfe.support\_]

# Reshape data for LSTM [samples, time steps, features]

X\_train\_reshaped = X\_train\_rfe.reshape((X\_train\_rfe.shape[0], 1, X\_train\_rfe.shape[1]))

X\_test\_reshaped = X\_test\_rfe.reshape((X\_test\_rfe.shape[0], 1, X\_test\_rfe.shape[1]))

# Define the model-building function for Keras Tuner

def build\_model(hp):

    model = Sequential()

    model.add(LSTM(

        units=hp.Int('units', min\_value=32, max\_value=128, step=16),

        input\_shape=(X\_train\_reshaped.shape[1], X\_train\_reshaped.shape[2]),

        return\_sequences=True

    ))

    model.add(Dropout(hp.Float('dropout', min\_value=0.2, max\_value=0.5, step=0.1)))

    model.add(LSTM(

        units=hp.Int('units\_2', min\_value=32, max\_value=128, step=16),

        return\_sequences=False

    ))

    model.add(Dropout(hp.Float('dropout\_2', min\_value=0.2, max\_value=0.5, step=0.1)))

    model.add(Dense(1, activation='sigmoid'))

    model.compile(

        optimizer=Adam(learning\_rate=hp.Float('learning\_rate', min\_value=1e-4, max\_value=1e-2, sampling='log')),

        loss='binary\_crossentropy',

        metrics=['accuracy']

    )

    return model

# Set up the Keras Tuner Random Search

tuner = RandomSearch(

    build\_model,

    objective='val\_accuracy',

    max\_trials=10,

    executions\_per\_trial=1,

    directory='tuner\_results',

    project\_name='lstm\_fraud\_detection'

)

# Perform the search for the best hyperparameters

tuner.search(X\_train\_reshaped, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Get the optimal hyperparameters

best\_hps = tuner.get\_best\_hyperparameters(num\_trials=1)[0]

# Print the best hyperparameters found

print(f"The optimal number of units in the first LSTM layer is {best\_hps.get('units')}")

print(f"The optimal number of units in the second LSTM layer is {best\_hps.get('units\_2')}")

print(f"The optimal dropout rate for the first LSTM layer is {best\_hps.get('dropout')}")

print(f"The optimal dropout rate for the second LSTM layer is {best\_hps.get('dropout\_2')}")

print(f"The optimal learning rate is {best\_hps.get('learning\_rate')}")

# Build the model with the best hyperparameters and train it

model = tuner.hypermodel.build(best\_hps)

history = model.fit(X\_train\_reshaped, y\_train, epochs=20, batch\_size=32, validation\_split=0.2)

# Save the trained model

model.save('lstm\_fraud\_model.h5')

# Load the model

model = load\_model('lstm\_fraud\_model.h5')

# Model evaluation

y\_pred\_prob = model.predict(X\_test\_reshaped)

mse = mean\_squared\_error(y\_test, y\_pred\_prob)

rmse = sqrt(mse)

auc = roc\_auc\_score(y\_test, y\_pred\_prob)

y\_pred = (y\_pred\_prob > 0.5).astype(int)

accuracy = accuracy\_score(y\_test, y\_pred)

# Display results

print(f"MSE: {mse:.4f}")

print(f"RMSE: {rmse:.4f}")

print(f"AUC-ROC: {auc:.4f}")

print(f"Accuracy: {accuracy:.4f}")

**Policy type prediction :**

import numpy as np

import pandas as pd

from sklearn.preprocessing import LabelEncoder

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

from sklearn.metrics import accuracy\_score, mean\_squared\_error, roc\_auc\_score

from keras.models import Sequential # type: ignore

from keras.layers import LSTM, Dense, Dropout # type: ignore

import keras\_tuner as kt

from keras.callbacks import EarlyStopping # type: ignore

# Load the dataset

data = pd.read\_csv('/Users/i.seviantojensima/Desktop/Sem 5/Machine Learning/ml project/final.csv')

# Preprocessing

data.ffill(inplace=True)  # Fill missing values

# Assuming 'PolicyType' is the target variable

X = data.drop('PolicyType', axis=1)  # Features

y = data['PolicyType']  # Target variable

# Encode categorical features

X = pd.get\_dummies(X)

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(y)

# Scale numerical features

X = (X - X.mean()) / X.std()  # Standardization

# Reshape X to 3D array: (samples, timesteps, features)

X = X.values.reshape(X.shape[0], 1, X.shape[1])  # Add time step of 1

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Optional: Use a smaller subset of training data during tuning to speed up the process

X\_train\_tune, \_, y\_train\_tune, \_ = train\_test\_split(X\_train, y\_train, test\_size=0.8, random\_state=42)

# Build a baseline model (before hyperparameter tuning)

baseline\_model = Sequential()

baseline\_model.add(LSTM(units=64, return\_sequences=True, input\_shape=(X\_train.shape[1], X\_train.shape[2])))

baseline\_model.add(Dropout(0.2))

baseline\_model.add(LSTM(units=64))

baseline\_model.add(Dropout(0.2))

baseline\_model.add(Dense(len(np.unique(y)), activation='softmax'))

# Compile the baseline model

baseline\_model.compile(

    optimizer='adam',  # Default optimizer

    loss='sparse\_categorical\_crossentropy',

    metrics=['accuracy']

)

# Define EarlyStopping to prevent overtraining

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

# Train the baseline model with EarlyStopping

baseline\_history = baseline\_model.fit(

    X\_train, y\_train,

    epochs=20,  # Fewer epochs for the baseline model

    batch\_size=32,

    validation\_data=(X\_test, y\_test),

    callbacks=[early\_stopping]

)

# Evaluate the baseline model

y\_pred\_baseline = baseline\_model.predict(X\_test)

y\_pred\_baseline\_classes = np.argmax(y\_pred\_baseline, axis=1)

# Calculate baseline metrics

baseline\_accuracy = accuracy\_score(y\_test, y\_pred\_baseline\_classes)

baseline\_mse = mean\_squared\_error(y\_test, y\_pred\_baseline\_classes)

baseline\_rmse = np.sqrt(baseline\_mse)

baseline\_roc\_auc = roc\_auc\_score(y\_test, y\_pred\_baseline, multi\_class='ovr')

print("\nBaseline Model Performance:")

print(f'Baseline Accuracy: {baseline\_accuracy}')

print(f'Baseline RMSE: {baseline\_rmse}')

print(f'Baseline MSE: {baseline\_mse}')

print(f'Baseline AUC-ROC: {baseline\_roc\_auc}')

# Define the hyperparameter tuning function

def build\_model(hp):

    model = Sequential()

    # Tune the number of LSTM units for the first layer

    model.add(

        LSTM(

            units=hp.Int('units', min\_value=32, max\_value=128, step=32),

            return\_sequences=True,

            input\_shape=(X\_train.shape[1], X\_train.shape[2])

        )

    )

    # Tune dropout rate for the first layer

    model.add(Dropout(hp.Float('dropout\_rate', min\_value=0.1, max\_value=0.5, step=0.1)))

    # Tune the number of LSTM units for the second layer

    model.add(

        LSTM(

            units=hp.Int('units2', min\_value=32, max\_value=128, step=32)

        )

    )

    # Tune dropout rate for the second layer

    model.add(Dropout(hp.Float('dropout\_rate2', min\_value=0.1, max\_value=0.5, step=0.1)))

    # Output layer

    model.add(Dense(len(np.unique(y)), activation='softmax'))

    # Compile the model with a tuned optimizer

    model.compile(

        optimizer=hp.Choice('optimizer', values=['adam', 'rmsprop']),

        loss='sparse\_categorical\_crossentropy',

        metrics=['accuracy']

    )

    return model

# Set up the Keras Tuner with Random Search

tuner = kt.RandomSearch(

    build\_model,

    objective='val\_accuracy',  # Optimize for validation accuracy

    max\_trials=5,  # Reduced the number of trials

    executions\_per\_trial=1,  # Train each model configuration once

    directory='tuner\_logs',

    project\_name='lstm\_tuning'

)

# Run the hyperparameter search with reduced epochs and callbacks

tuner.search(X\_train\_tune, y\_train\_tune, epochs=10, batch\_size=32, validation\_data=(X\_test, y\_test), callbacks=[early\_stopping])

# Get the best hyperparameters

best\_hps = tuner.get\_best\_hyperparameters(num\_trials=1)[0]

print("\nHyperparameter Tuning Results:")

print(f"The optimal number of units in the first LSTM layer is {best\_hps.get('units')}")

print(f"The optimal number of units in the second LSTM layer is {best\_hps.get('units2')}")

print(f"The optimal dropout rate for the first layer is {best\_hps.get('dropout\_rate')}")

print(f"The optimal dropout rate for the second layer is {best\_hps.get('dropout\_rate2')}")

print(f"The optimal optimizer is {best\_hps.get('optimizer')}")

# Build the model with the best hyperparameters

model = tuner.hypermodel.build(best\_hps)

# Train the model with the best hyperparameters and EarlyStopping

history = model.fit(

    X\_train, y\_train,

    epochs=50, batch\_size=32,

    validation\_data=(X\_test, y\_test),

    callbacks=[early\_stopping]

)

# Predict and evaluate metrics for the tuned model

y\_pred = model.predict(X\_test)

y\_pred\_classes = np.argmax(y\_pred, axis=1)

# Calculate tuned model metrics

accuracy = accuracy\_score(y\_test, y\_pred\_classes)

mse = mean\_squared\_error(y\_test, y\_pred\_classes)

rmse = np.sqrt(mse)

roc\_auc = roc\_auc\_score(y\_test, y\_pred, multi\_class='ovr')

print("\nTuned Model Performance:")

print(f'Accuracy: {accuracy}')

print(f'RMSE: {rmse}')

print(f'MSE: {mse}')

print(f'AUC-ROC: {roc\_auc}')

**AUTOENCODERS :**

**Fraud detection :**

import pandas as pd

import numpy as np

import seaborn as sns  # For advanced visualizations

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

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

from keras.models import Model # type: ignore

from keras.layers import Input, Dense # type: ignore

from keras.callbacks import EarlyStopping # type: ignore

from sklearn.metrics import roc\_auc\_score, mean\_squared\_error, confusion\_matrix, roc\_curve, accuracy\_score

import matplotlib.pyplot as plt

# Load the dataset

file\_path = r"final .csv"

data = pd.read\_csv(file\_path)

# DATA PREPROCESSING

# Handle missing values (if any)

data.fillna(data.mean(numeric\_only=True), inplace=True)

# Convert 'FraudFound' from 'No'/'Yes' to 0/1

data['FraudFound'] = data['FraudFound'].map({'No': 0, 'Yes': 1})

# Verify the conversion

print("\nUnique values in 'FraudFound' after conversion:")

print(data['FraudFound'].unique())

# Define features and target

features = data.drop(columns=['FraudFound'])

target = data['FraudFound']

categorical\_cols = features.select\_dtypes(include=['object']).columns.tolist()

numerical\_cols = features.select\_dtypes(include=[np.number]).columns.tolist()

# Create preprocessing pipelines

numeric\_transformer = Pipeline(steps=[('scaler', StandardScaler())])

categorical\_transformer = Pipeline(steps=[('onehot', OneHotEncoder(handle\_unknown='ignore'))])

preprocessor = ColumnTransformer(

    transformers=[

        ('num', numeric\_transformer, numerical\_cols),

        ('cat', categorical\_transformer, categorical\_cols)

    ])

X = preprocessor.fit\_transform(features)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, target, test\_size=0.2, random\_state=10)

# MODEL TRAINING

# Function to create an Autoencoder model with specified hyperparameters

def create\_autoencoder(hidden\_layers, neurons, activation):

    input\_layer = Input(shape=(X\_train.shape[1],))

    x = input\_layer

    # Create hidden layers

    for \_ in range(hidden\_layers):

        x = Dense(neurons, activation=activation)(x)

    decoder = Dense(X\_train.shape[1], activation='sigmoid')(x)

    autoencoder = Model(input\_layer, decoder)

    autoencoder.compile(optimizer='adam', loss='mean\_squared\_error')

    return autoencoder

# \*\*Evaluate Overall Accuracy Before Tuning\*\*

# Define default hyperparameters

default\_hidden\_layers = 1

default\_neurons = 16

default\_activation = 'relu'

default\_epochs = 5

default\_batch\_size = 16

# Create and train the model with default hyperparameters

default\_model = create\_autoencoder(default\_hidden\_layers, default\_neurons, default\_activation)

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

default\_model.fit(X\_train, X\_train,

                  epochs=default\_epochs,

                  batch\_size=default\_batch\_size,

                  validation\_split=0.2,

                  callbacks=[early\_stopping],

                  verbose=0)

# Evaluate the default model

reconstructed\_default = default\_model.predict(X\_test)

mse\_values\_default = np.mean(np.power(X\_test - reconstructed\_default, 2), axis=1)

# Set a threshold for fraud detection

threshold\_default = np.percentile(mse\_values\_default, 95)  # Example threshold

fraudulent\_claims\_default = (mse\_values\_default > threshold\_default).astype(int)

# Calculate overall accuracy

overall\_accuracy\_default = accuracy\_score(y\_test, fraudulent\_claims\_default)

print(f"\nOverall Accuracy Before Tuning: {overall\_accuracy\_default:.4f}")

# HYPERPARAMETER TUNING

best\_model = None

best\_params = {}

best\_loss = float('inf')

tuning\_count = 0  # Counter for tuning iterations

# Example hyperparameter values

hidden\_layers\_list = [1, 2]

neurons\_list = [8, 16, 32]

activation\_functions = ['relu', 'tanh']

epochs\_list = [5, 10]

batch\_sizes = [16, 32]

for hidden\_layers in hidden\_layers\_list:

    for neurons in neurons\_list:

        for activation in activation\_functions:

            for epoch in epochs\_list:

                for batch\_size in batch\_sizes:

                    if tuning\_count < 10:

                        autoencoder = create\_autoencoder(hidden\_layers, neurons, activation)

                        early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

                        history = autoencoder.fit(X\_train, X\_train,

                                                  epochs=epoch,

                                                  batch\_size=batch\_size,

                                                  validation\_split=0.2,

                                                  callbacks=[early\_stopping],

                                                  verbose=0)

                        # Evaluate the model on the test set

                        loss = autoencoder.evaluate(X\_test, X\_test, verbose=0)

                        print(f"Hidden Layers: {hidden\_layers}, Neurons: {neurons}, Activation: {activation}, "

                              f"Epochs: {epoch}, Batch Size: {batch\_size}, Loss: {loss}")

                        # Update best model if current loss is lower

                        if loss < best\_loss:

                            best\_loss = loss

                            best\_model = autoencoder

                            best\_params = {

                                'hidden\_layers': hidden\_layers,

                                'neurons': neurons,

                                'activation': activation,

                                'epochs': epoch,

                                'batch\_size': batch\_size

                            }

                        tuning\_count += 1  # Increment the counter

                    else:

                        break  # Stop if 10 combinations have been evaluated

# Output best parameters

print("\nBest Parameters:")

print(best\_params)

print("Best Loss:", best\_loss)

#' MODEL EVALUATION'

reconstructed = best\_model.predict(X\_test)

mse\_values = np.mean(np.power(X\_test - reconstructed, 2), axis=1)

# Set a threshold for fraud detection

threshold = np.percentile(mse\_values, 95)  # Example threshold

print(f"\nReconstruction Error Threshold: {threshold}")

# Identify fraudulent claims

fraudulent\_claims = (mse\_values > threshold).astype(int)

# Calculate overall accuracy after tuning

accuracy = accuracy\_score(y\_test, fraudulent\_claims)

print(f"Accuracy: {accuracy:.4f}")

# Calculate MSE and RMSE

overall\_mse = mean\_squared\_error(y\_test, fraudulent\_claims)

overall\_rmse = np.sqrt(overall\_mse)

print(f"Overall MSE: {overall\_mse:.4f}")

print(f"Overall RMSE: {overall\_rmse:.4f}")

# Step 1: Determine the majority class

majority\_class = y\_train.mode()[0]

# Step 2: Create a baseline prediction array

baseline\_predictions = np.full(y\_test.shape, majority\_class)

# Step 3: Calculate baseline evaluation metrics

baseline\_accuracy = accuracy\_score(y\_test, baseline\_predictions)

baseline\_mse = mean\_squared\_error(y\_test, baseline\_predictions)

baseline\_rmse = np.sqrt(baseline\_mse)

print("\nBaseline Performance:")

print(f"Baseline Accuracy: {baseline\_accuracy:.4f}")

print(f"Baseline MSE: {baseline\_mse:.4f}")

print(f"Baseline RMSE: {baseline\_rmse:.4f}")

# Output results

results = pd.DataFrame({

    'Reconstruction Error': mse\_values,

    'Fraudulent': fraudulent\_claims,

    'Actual Fraud': y\_test.values

})

# Save the results to a CSV file

results.to\_csv('fraud\_detection\_results.csv', index=False)

# Print first few rows of the results

print("\nSample Results:")

print(results.head())

# Evaluation Metrics

auc\_roc = roc\_auc\_score(y\_test, mse\_values)

print(f"\nAUC-ROC Score: {auc\_roc:.4f}")

# Confusion Matrix

conf\_matrix = confusion\_matrix(y\_test, fraudulent\_claims)

print("\nConfusion Matrix:")

print(conf\_matrix)

# 'MODEL EVALUATION'

reconstructed = best\_model.predict(X\_test)

mse\_values = np.mean(np.power(X\_test - reconstructed, 2), axis=1)

# Set a threshold for fraud detection

threshold = np.percentile(mse\_values, 95)  # Example threshold

print(f"\nReconstruction Error Threshold: {threshold}")

# Identify fraudulent claims

fraudulent\_claims = (mse\_values > threshold).astype(int)

# Calculate overall accuracy after tuning

accuracy = accuracy\_score(y\_test, fraudulent\_claims)

print(f"Accuracy: {accuracy:.4f}")

# Calculate MSE and RMSE after tuning

overall\_mse\_after\_tuning = mean\_squared\_error(y\_test, fraudulent\_claims)

overall\_rmse\_after\_tuning = np.sqrt(overall\_mse\_after\_tuning)

# Print MSE and RMSE after tuning

print(f"Overall MSE After Tuning: {overall\_mse\_after\_tuning:.4f}")

print(f"Overall RMSE After Tuning: {overall\_rmse\_after\_tuning:.4f}")

**Policy Type :**

import pandas as pd

import numpy as np

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

from sklearn.preprocessing import OneHotEncoder, StandardScaler, LabelEncoder

from tensorflow.keras.models import Model # type: ignore

from tensorflow.keras.layers import Dense, Input # type: ignore

from tensorflow.keras.optimizers import Adam # type: ignore

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, mean\_squared\_error, roc\_auc\_score

import warnings

import os

import tensorflow as tf

# Suppress TensorFlow logs

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '3'

# Suppress all warnings

warnings.filterwarnings("ignore")

# Load dataset

df = pd.read\_csv('final .csv')

# Check DataFrame shape

print("Initial DataFrame shape:", df.shape)

# Check for unique values in numerical columns

numerical\_columns = ['Age', 'VehiclePrice', 'ClaimAmount', 'Deductible', 'DriverRating',

                     'Days:Policy-Accident', 'Days:Policy-Claim', 'PastNumberOfClaims',

                     'AgeOfVehicle', 'AgeOfPolicyHolder', 'NumberOfCars']

# Convert non-numeric values to NaN

for col in numerical\_columns:

    df[col] = pd.to\_numeric(df[col], errors='coerce')

# Fill NaNs with the median

for col in numerical\_columns:

    df[col].fillna(df[col].median(), inplace=True)

# Check for NaN values

print("Check for NaN values in numerical columns:\n", df[numerical\_columns].isnull().sum())

# Preprocessing

categorical\_columns = ['Make', 'AccidentArea', 'Sex', 'MaritalStatus', 'Fault',

                       'VehicleCategory', 'PoliceReportFiled', 'WitnessPresent', 'AgentType', 'BasePolicy']

# Encode categorical columns

if df[categorical\_columns].shape[0] > 0:

    encoder = OneHotEncoder(sparse\_output=False)

    encoded\_columns = encoder.fit\_transform(df[categorical\_columns])

else:

    encoded\_columns = np.array([])

# Scale numerical columns

scaler = StandardScaler()

scaled\_columns = scaler.fit\_transform(df[numerical\_columns])

# Combine encoded and scaled columns

if encoded\_columns.size > 0 and scaled\_columns.size > 0:

    X = np.concatenate([encoded\_columns, scaled\_columns], axis=1)

else:

    X = np.array([])

# Encode target variable

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(df['PolicyType'])

# Proceed only if X is not empty

if X.size > 0:

    # Split the data

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

    # Define the autoencoder

    input\_dim = X\_train.shape[1]

    encoding\_dim = 32

    input\_layer = Input(shape=(input\_dim,))

    encoded = Dense(encoding\_dim, activation='relu')(input\_layer)

    decoded = Dense(input\_dim, activation='sigmoid')(encoded)

    autoencoder = Model(inputs=input\_layer, outputs=decoded)

    autoencoder.compile(optimizer=Adam(learning\_rate=0.001), loss='mean\_squared\_error')

    # Train the autoencoder

    autoencoder.fit(X\_train, X\_train, epochs=10, batch\_size=64, validation\_split=0.2)

    # Use the encoder to get compressed features

    encoder\_model = Model(inputs=input\_layer, outputs=encoded)

    X\_train\_encoded = encoder\_model.predict(X\_train)

    X\_test\_encoded = encoder\_model.predict(X\_test)

    # Train classifier on encoded features before tuning

    classifier = RandomForestClassifier()

    classifier.fit(X\_train\_encoded, y\_train)

    # Predictions before tuning

    y\_pred\_before = classifier.predict(X\_test\_encoded)

    # Evaluate accuracy before tuning

    accuracy\_before = accuracy\_score(y\_test, y\_pred\_before)

    print(f"Accuracy before tuning: {accuracy\_before \* 100:.2f}%")

    # Calculate MSE and AUC-ROC before tuning

    mse\_before = mean\_squared\_error(y\_test, y\_pred\_before)

    auc\_roc\_before = roc\_auc\_score(y\_test, classifier.predict\_proba(X\_test\_encoded), multi\_class='ovr')

    print(f"MSE before tuning: {mse\_before:.4f}")

    print(f"AUC-ROC before tuning: {auc\_roc\_before:.4f}")

    # Hyperparameter tuning

    param\_grid = {

        'n\_estimators': [50, 100, 200],

        'max\_depth': [None, 10, 20, 30],

        'min\_samples\_split': [2, 5, 10]

    }

    grid\_search = GridSearchCV(estimator=RandomForestClassifier(), param\_grid=param\_grid,

                               cv=3, scoring='accuracy', n\_jobs=-1, verbose=2)

    grid\_search.fit(X\_train\_encoded, y\_train)

    print(f"Best parameters from tuning: {grid\_search.best\_params\_}")

    # Train the tuned model

    best\_classifier = grid\_search.best\_estimator\_

    best\_classifier.fit(X\_train\_encoded, y\_train)

    # Predictions after tuning

    y\_pred\_after = best\_classifier.predict(X\_test\_encoded)

    # Evaluate accuracy after tuning

    accuracy\_after = accuracy\_score(y\_test, y\_pred\_after)

    print(f"Accuracy after tuning: {accuracy\_after \* 100:.2f}%")

    # Calculate MSE and AUC-ROC after tuning

    mse\_after = mean\_squared\_error(y\_test, y\_pred\_after)

    auc\_roc\_after = roc\_auc\_score(y\_test, best\_classifier.predict\_proba(X\_test\_encoded), multi\_class='ovr')

    print(f"MSE after tuning: {mse\_after:.4f}")

    print(f"AUC-ROC after tuning: {auc\_roc\_after:.4f}")

else:

    print("Final feature matrix X is empty; cannot proceed with model training.")

**PREDICTION :**

import streamlit as st

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler, LabelEncoder

from keras.models import load\_model # type: ignore

# Load the models

fraud\_model = load\_model("lstm\_fraud\_model.h5")

policy\_model = load\_model('policy\_model.h5')

# Load the dataset and fit the scalers

data = pd.read\_csv('final .csv')

fraud\_features = ['Age', 'ClaimAmount', 'PastNumberOfClaims', 'DriverRating', 'Deductible']

policy\_features = ['WeekOfMonthClaimed', 'DayOfWeekClaimed', 'MonthClaimed', 'AgeOfPolicyHolder',

                   'ClaimAmount', 'AgeOfVehicle', 'Year']

# Scaling for fraud detection features

fraud\_scaler = StandardScaler()

fraud\_scaler.fit(data[fraud\_features])

# Scaling for policy prediction features

policy\_scaler = StandardScaler()

policy\_scaler.fit(data[policy\_features])

# Label encoding for policy prediction

label\_encoder = LabelEncoder()

data['PolicyType'] = label\_encoder.fit\_transform(data['PolicyType'])

# Custom CSS for buttons and layout

st.markdown("""

    <style>

    div.stButton > button:first-child {

        background-color: #4CAF50;

        color:white;

        height: 3em;

        width: 12em;

        font-size: 18px;

        border-radius:10px;

        border:2px solid #FFFFFF;

    }

    div.stButton > button:hover {

        background-color: #45a049;

        color:white;

    }

    .prediction-result {

        color: #FF4500;

        font-size: 20px;

        font-weight: bold;

    }

    </style>

""", unsafe\_allow\_html=True)

# Initialize session state for navigation

if 'page' not in st.session\_state:

    st.session\_state.page = 'home'

# Function to get fraud detection user input

def get\_fraud\_input():

    st.markdown('<h3 style="color:#00CED1;">Please provide the following details for Fraud Detection:</h3>', unsafe\_allow\_html=True)

    user\_input = {}

    user\_input['Age'] = st.number\_input("Enter value for Age:", min\_value=0, max\_value=100, value=25)

    user\_input['ClaimAmount'] = st.number\_input("Enter value for ClaimAmount:", min\_value=0, value=10000)

    user\_input['PastNumberOfClaims'] = st.number\_input("Enter value for PastNumberOfClaims:", min\_value=0, max\_value=100, value=2)

    user\_input['DriverRating'] = st.number\_input("Enter value for DriverRating:", min\_value=1, max\_value=5, value=3)

    user\_input['Deductible'] = st.number\_input("Enter value for Deductible:", min\_value=0, value=500)

    return pd.DataFrame([user\_input])

# Function to get insurance policy input

def get\_policy\_input():

    st.markdown('<h3 style="color:#4682B4;">Enter the details for Policy Prediction:</h3>', unsafe\_allow\_html=True)

    week\_of\_month = st.number\_input("Week of the Month of the Claim:", min\_value=1, max\_value=5)

    day\_of\_week = st.number\_input("Day of the Week of the Claim:", min\_value=1, max\_value=7)

    month\_of\_year = st.number\_input("Month of the Year of the Claim:", min\_value=1, max\_value=12)

    age = st.number\_input("Age of the Policy Holder:", min\_value=0, max\_value=100, value=30)

    claim\_amount = st.number\_input("Claim Amount:", min\_value=0, value=250000)

    vehicle\_age = st.number\_input("Age of the Vehicle:", min\_value=0, value=5)

    claim\_year = st.number\_input("Year of the Claim:", min\_value=1990, max\_value=2024, value=2024)

    return np.array([week\_of\_month, day\_of\_week, month\_of\_year, age, claim\_amount, vehicle\_age, claim\_year]).reshape(1, -1)

# Home page with two buttons

if st.session\_state.page == 'home':

    st.markdown('<h1 style="color: #FF6347;">Insurance Fraud and Policy Prediction</h1>', unsafe\_allow\_html=True)

    st.markdown('<h2 style="color: #4682B4;">Select an option:</h2>', unsafe\_allow\_html=True)

    col1, col2 = st.columns([1, 1.5])

    with col1:

        if st.button("Fraud Detection"):

            st.session\_state.page = 'fraud\_detection'

    with col2:

        if st.button("Insurance Policy"):

            st.session\_state.page = 'insurance\_policy'

# Fraud Detection page

if st.session\_state.page == 'fraud\_detection':

    st.markdown('<h2 style="color: #FF8C00;">Fraud Detection</h2>', unsafe\_allow\_html=True)

    # Collect user input

    fraud\_input = get\_fraud\_input()

    # Button to trigger prediction

    if st.button("Detect Fraud"):

        # Scale the input

        fraud\_scaled\_input = fraud\_scaler.transform(fraud\_input[fraud\_features])

        # Reshape input for the LSTM model

        fraud\_input\_reshaped = fraud\_scaled\_input.reshape(1, 1, len(fraud\_features))

        # Predict using the trained LSTM model

        fraud\_prediction = fraud\_model.predict(fraud\_input\_reshaped)

        st.markdown(f"<p class='prediction-result'>Raw Prediction Probability of Fraud: <strong>{fraud\_prediction[0][0]:.4f}</strong></p>", unsafe\_allow\_html=True)

        # Custom decision-making based on input values (optional)

        if fraud\_input['ClaimAmount'].values[0] > 25000 and fraud\_input['PastNumberOfClaims'].values[0] > 5:

            decision = "Fraud"

        else:

            decision = "Not Fraud"

        st.markdown(f"<p class='prediction-result'>Decision: <strong>{decision}</strong></p>", unsafe\_allow\_html=True)

    # Back button to return to home

    if st.button("Back"):

        st.session\_state.page = 'home'

# Insurance Policy Prediction page

if st.session\_state.page == 'insurance\_policy':

    st.markdown('<h2 style="color:#FF6347;">Insurance Policy Prediction</h2>', unsafe\_allow\_html=True)

    # Collect user input

    policy\_input = get\_policy\_input()

    # Button to trigger prediction

    if st.button("Predict Policy"):

        # Scale the input

        policy\_input\_scaled = policy\_scaler.transform(policy\_input)

        # Predict using the trained model

        policy\_prediction\_proba = policy\_model.predict(policy\_input\_scaled)

        predicted\_class = np.argmax(policy\_prediction\_proba)

        predicted\_label = label\_encoder.inverse\_transform([predicted\_class])[0]

        st.markdown(f"<p class='prediction-result'>Predicted Policy Type: <strong>{predicted\_label}</strong></p>", unsafe\_allow\_html=True)

    # Back button to return to home

    if st.button("Back"):

        st.session\_state.page = 'home'

**OUTPUT :**

**Exploratory Data Analysis :**

Dataset Information:

RangeIndex: 150000 entries, 0 to 149999

Data columns (total 34 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Month 150000 non-null int64

1 WeekOfMonth 150000 non-null int64

2 DayOfWeek 150000 non-null int64

3 Make 150000 non-null object

4 AccidentArea 150000 non-null object

5 DayOfWeekClaimed 150000 non-null int64

6 MonthClaimed 150000 non-null int64

7 WeekOfMonthClaimed 150000 non-null int64

8 Sex 150000 non-null object

9 MaritalStatus 150000 non-null object

10 Age 150000 non-null int64

11 Fault 150000 non-null object

12 PolicyType 150000 non-null object

13 VehicleCategory 150000 non-null object

14 VehiclePrice 150000 non-null float64

15 ClaimAmount 150000 non-null float64

16 PolicyNumber 150000 non-null int64

17 RepNumber 150000 non-null int64

18 Deductible 150000 non-null int64

19 DriverRating 150000 non-null int64

20 Days:Policy-Accident 150000 non-null object

21 Days:Policy-Claim 150000 non-null object

22 PastNumberOfClaims 150000 non-null int64

23 AgeOfVehicle 150000 non-null int64

24 AgeOfPolicyHolder 150000 non-null int64

25 PoliceReportFiled 150000 non-null object

26 WitnessPresent 150000 non-null object

27 AgentType 150000 non-null object

28 NumberOfSuppliments 150000 non-null int64

29 AddressChange-Claim 150000 non-null object

30 NumberOfCars 150000 non-null int64

31 Year 150000 non-null int64

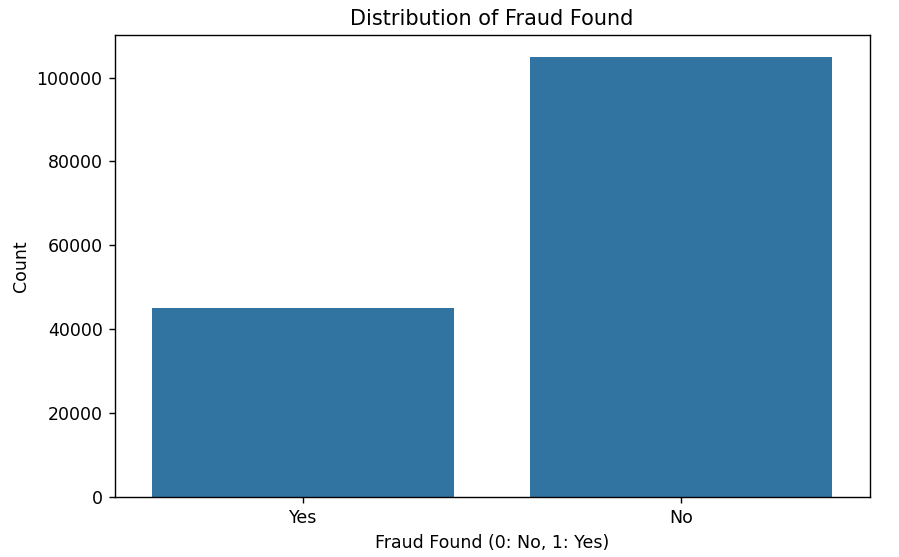
32 BasePolicy 150000 non-null object

33 FraudFound 150000 non-null object

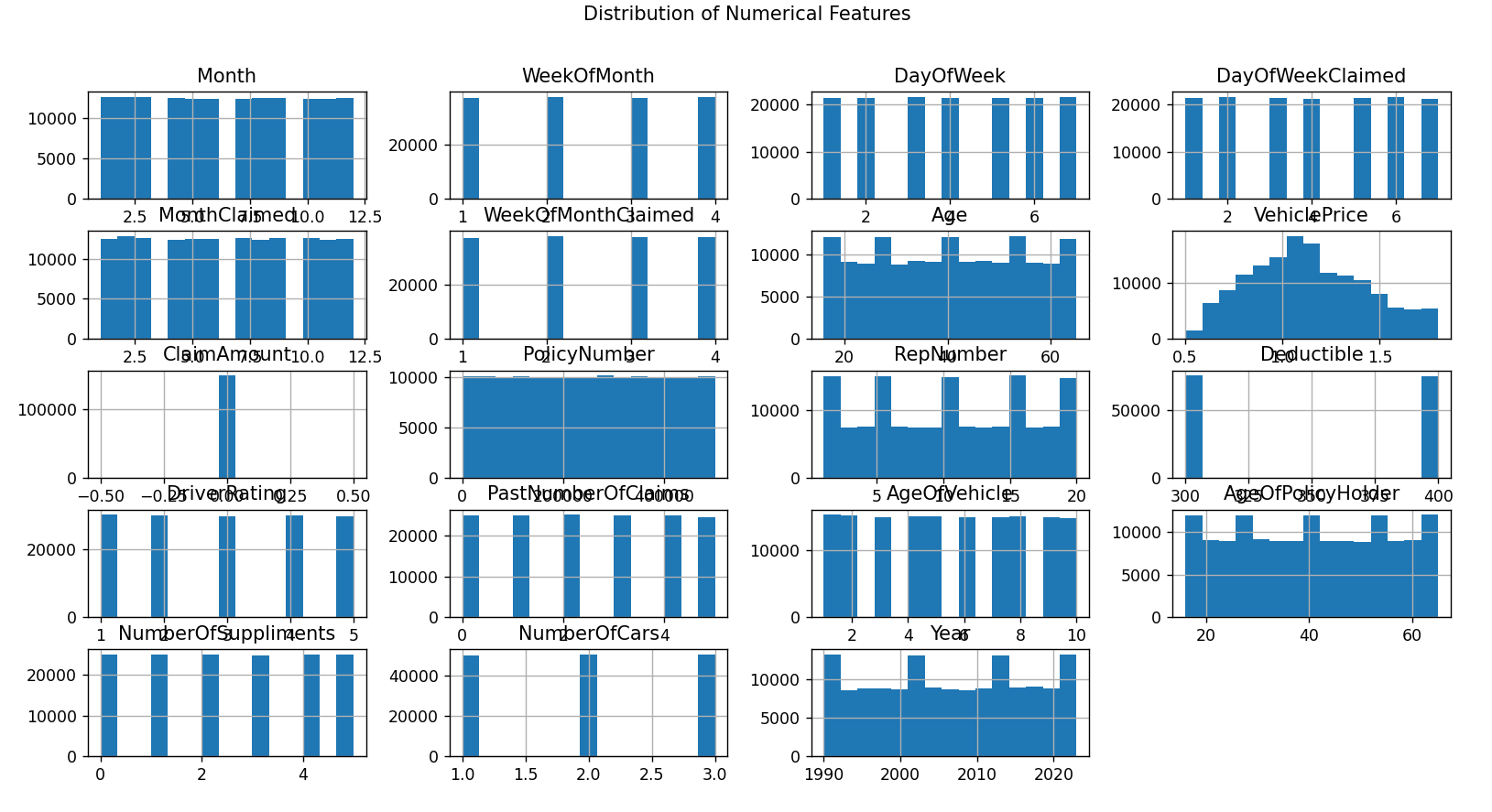
dtypes: float64(2), int64(17), object(15)

memory usage: 38.9+ MB

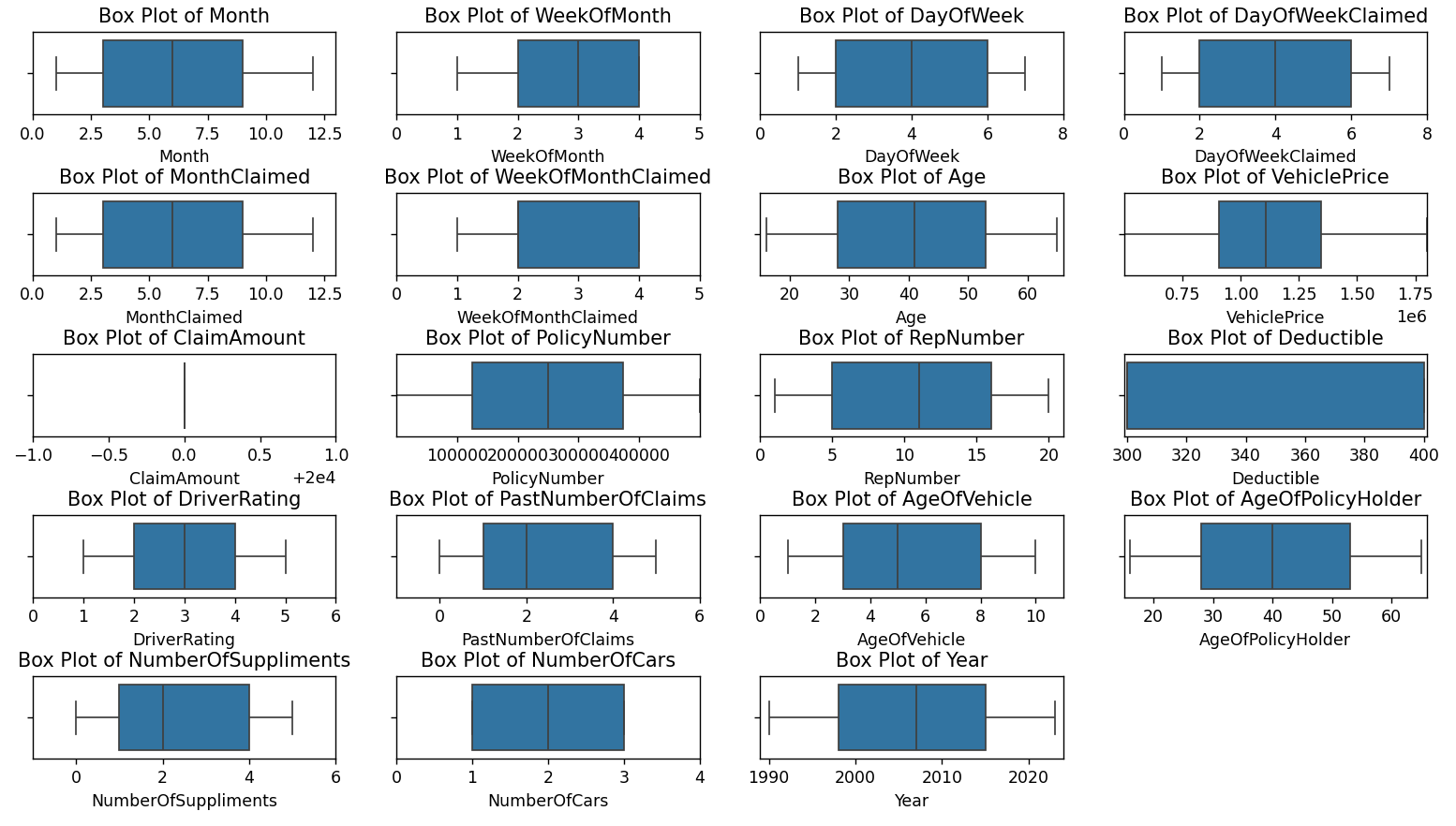
**1. Distribution of the target variable**

****

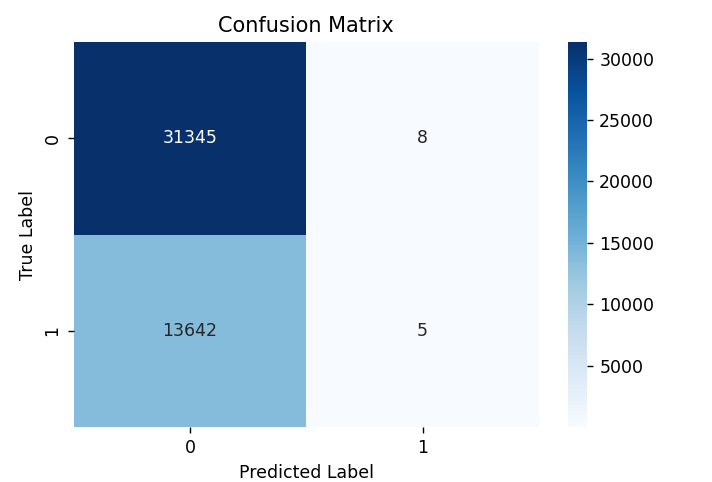
**2. Distribution of numerical features**

****

**3. Outlier Detection using box plots**

****

**4.** **Confusion Matrix**

****

**Gradient Boosted Neural Network :**

**1.Fraud Detection :**

* **Initial model building :**

Gradient Boosted Neural Networks - Accuracy: 0.6966, MSE: 0.3034, MAE: 0.3034, ROC AUC: 0.5035

* **Hyper parameter tuning :**

Final Model with Best Parameters - Accuracy: 0.6905, MSE: 0.3095, MAE: 0.3095, ROC AUC: 0.4967

Best Parameters: {'neurons\_1': 64, 'neurons\_2': 32, 'epochs': 20, 'batch\_size': 16}

Final Model with Best Parameters - Accuracy: 0.6879, MSE: 0.3121, MAE: 0.3121, ROC AUC: 0.5048

Fold 1 - Accuracy: 0.6962

Fold 2 - Accuracy: 0.7003

Fold 3 - Accuracy: 0.6995

Fold 4 - Accuracy: 0.7009

Fold 5 - Accuracy: 0.6988

Average accuracy after k-fold cross-validation: 0.6992

Final Model with Best Parameters - Accuracy: 0.7002, MSE: 0.2998, MAE: 0.2998, ROC AUC: 0.5052

**2.Policy type prediction :**

* **Initial model building :**

Gradient Boosted Neural Networks - Accuracy for Policy Type: 0.3346

MSE: 1.3564, MAE: 0.8957, ROC AUC: 0.5001

* **Hyper parameter tuning :**

Best Parameters: {'neurons\_1': 128, 'neurons\_2': 64, 'dropout\_rate': 0.2, 'epochs': 20, 'batch\_size': 64}

Accuracy: 0.3340, MSE: 1.4310, MAE: 0.9210, ROC AUC: 0.5005

**Feed forward neural network :**

**1.Fraud Detection :**

* **Initial model building :**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EPOCHS | ACCURACY | MSE | RMSE | MAE | ROC - AUC |
| 50 | 0.6931 | 0.3069 | 0.5540 | 0.3069 | 0.5025 |
| 20 | 0.6958 | 0.3042 | 0.5516 | 0.3042 | 0.5063 |
| 10 | 0.6993 | 0.3007 | 0.5484 | 0.3007 | 0.5001 |

* **Hyper parameter tuning :**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TRIAL 1 | TRIAL 2 | TRIAL 3 | TRIAL 4 | TRIAL 5 | TRIAL 6 | TRIAL 7 | TRIAL 8 | TRIAL 9 | TRIAL 10 |
| 0.70322 | **0.70323** | **0.70322** | **0.70322** | **0.70322** | **0.70323** | **0.70320** | **0.70320** | **0.70323** | **0.703208** |

Best val\_accuracy So Far: 0.7032361030578613

Total elapsed time: 00h 52m 02s

Best Accuracy: 0.7002

MSE: 0.2998

RMSE: 0.5475

MAE: 0.2998

ROC AUC: 0.5037

**2.Policy type prediction :**

* **Initial model building :**

Feedforward Neural Networks - Accuracy for Policy Type: 0.3351

MSE: 1.3589, MAE: 0.8963, ROC AUC: 0.4995

* **Hyper parameter tuning :**

Best Hyperparameters: {'optimizer': 'adam', 'model\_neurons\_2': 64, 'modelneurons\_1': 128, 'model\_dropout\_rate': 0.2, 'epochs': 30, 'batch\_size': 128}

Final Model - Accuracy: 0.3347

MSE: 1.2829

MAE: 0.8711

ROC AUC: 0.4994

**LSTM – Long Short Term Memory :**

**1.Fraud Detection :**

* **Initial model building :**

Before Hypertuning:

MSE: 0.1075

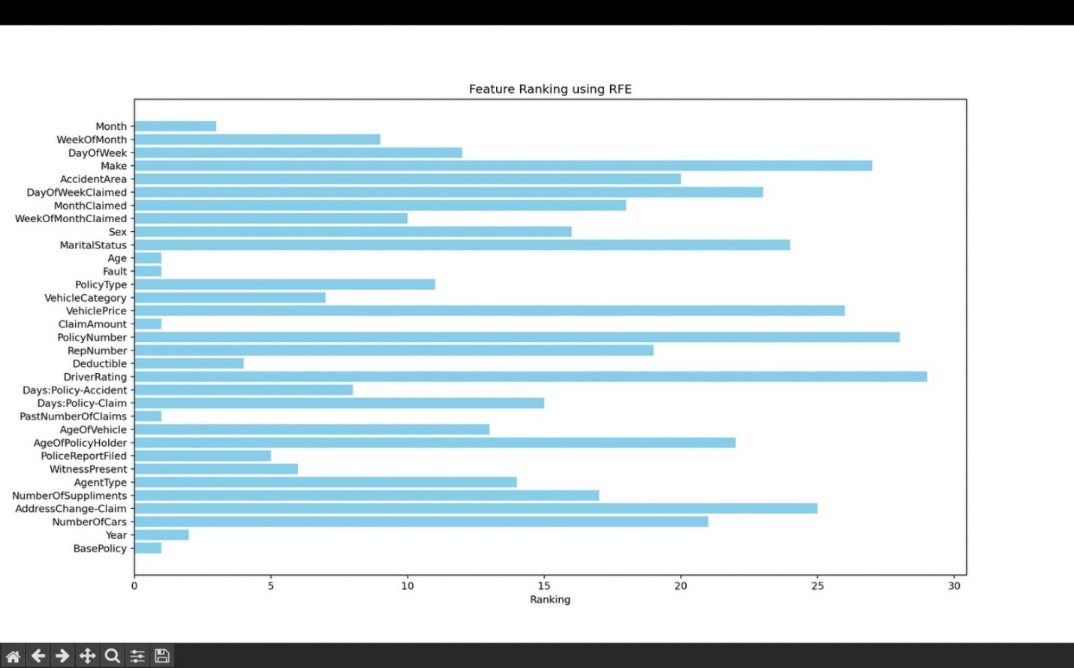
RMSE: 0.3279

AUC-ROC: 0.6056

Accuracy: 0.8733

* **Hyper parameter tuning :**

**Feature Selection :**



MSE: 0.0796

RMSE: 0.2821

AUC-ROC: 0.6101

Accuracy: 0.9150

**2.Policy type prediction :**

* **Initial model building :**

Accuracy: 0.3345

RMSE: 1.1594826432508596

MSE: 1.3444

AUC-ROC: 0.49916387209903984

* **Hyper parameter tuning :**

Accuracy: 0.3307

RMSE: 1.2427389106324787

MSE: 1.5444

AUC-ROC: 0.5019477077758844

**Autoencoders :**

**1.Fraud Detection :**

* **Initial model building :**

MSE: 0.3215

RMSE: 0.5670

AUC-ROC Score: 0.4963

Accuracy: 0.6785

* **Hyper parameter tuning :**

MSE: 0.3517

RMSE: 0.6214

AUC-ROC Score: 0.3524

Accuracy: 0.6945

**2.Policy type prediction :**

* **Initial model building :**

MSE: 1.6432

RMSE: 1.6743

AUC-ROC Score: 0.5002

Accuracy: 0.3331

* **Hyper parameter tuning :**

MSE: 1.6637

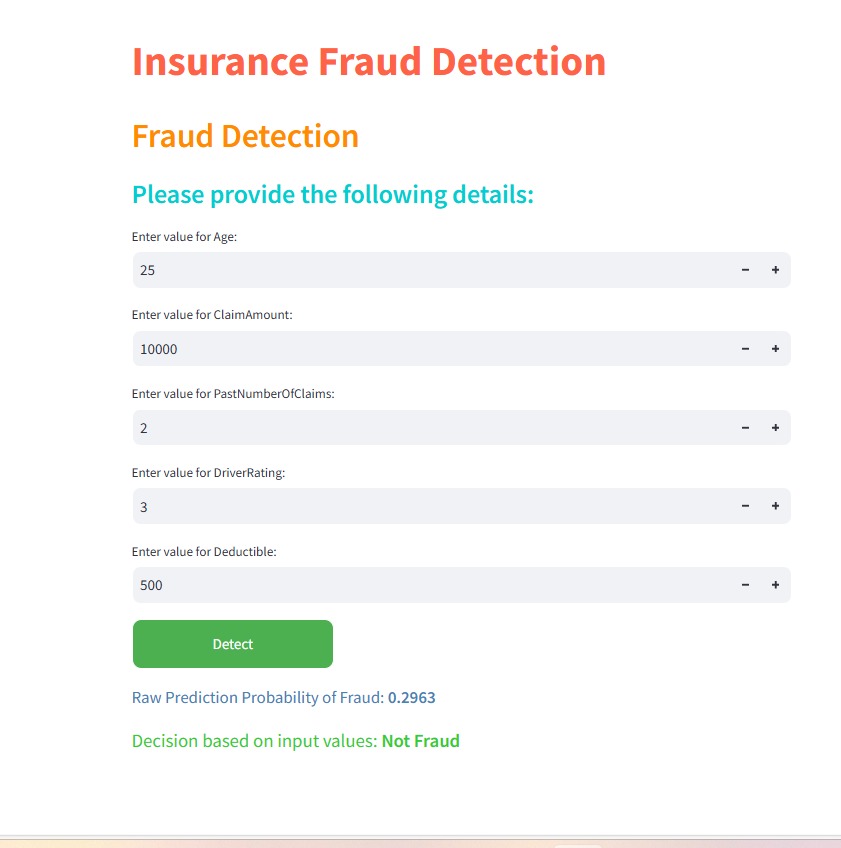
RMSE: 1.6872

AUC-ROC Score: 0.5000

Accuracy: 0.3331

**Prediction :**

**Fraud Detection :**



**Insurance Policy Prediction :**

