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):
```

```python
    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 :

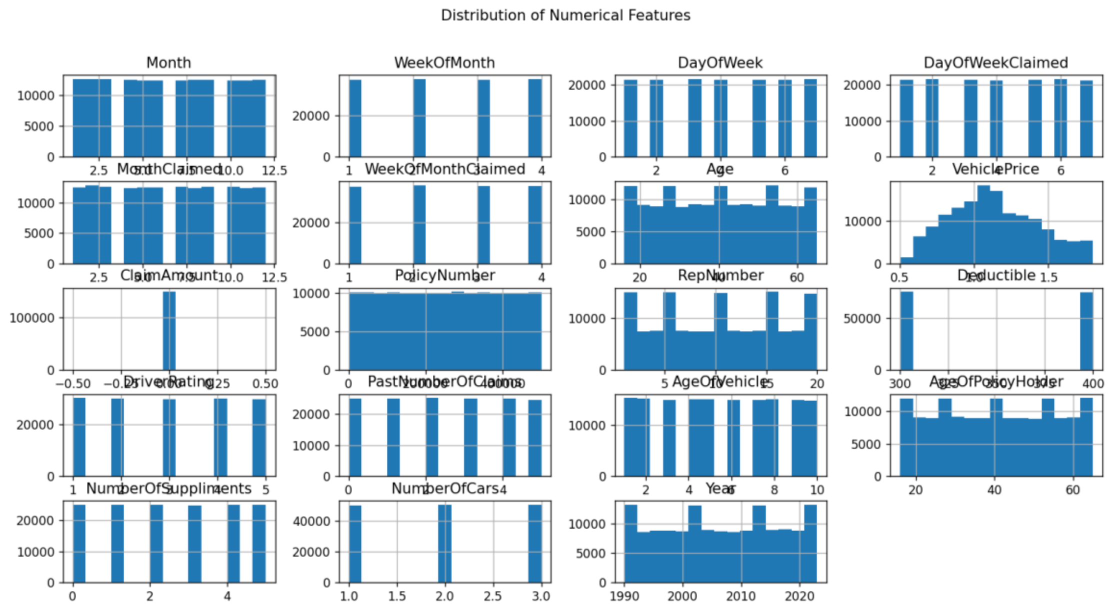
Exploratory Data Analysis :

```
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
```

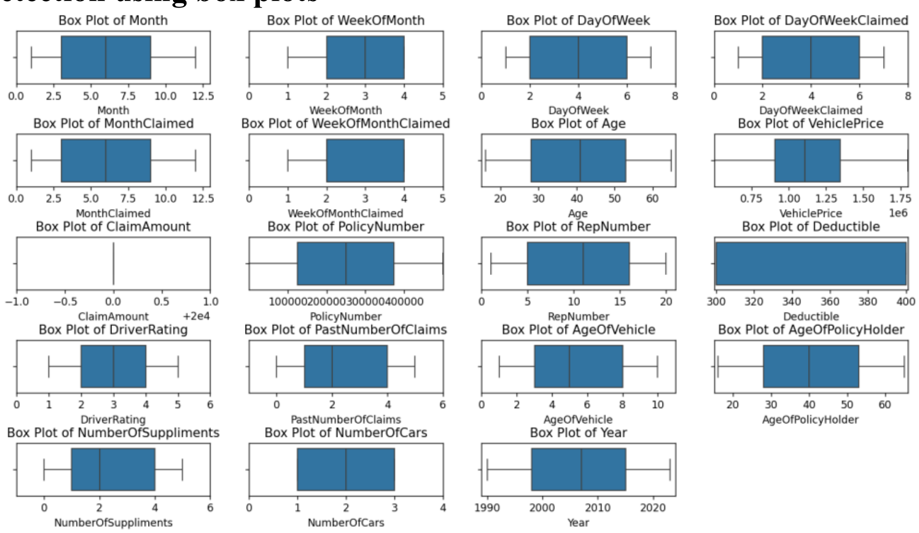
```
 15  ClaimAmount           150000 non-null  float64
 16  PolicyNumber          150000 non-null  int64
 17  RepNumber             150000 non-null  int64
 18  Deductible            150000 non-null  int64
 19  DriverRating          150000 non-null  int64
 20  Days:Policy-Accident  150000 non-null  object
 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

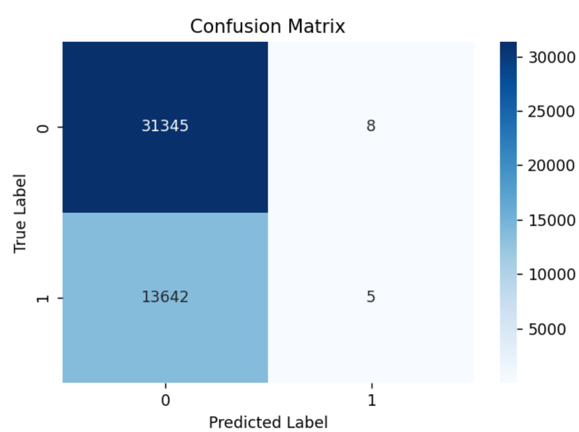
2. Distribution of numerical features



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 :

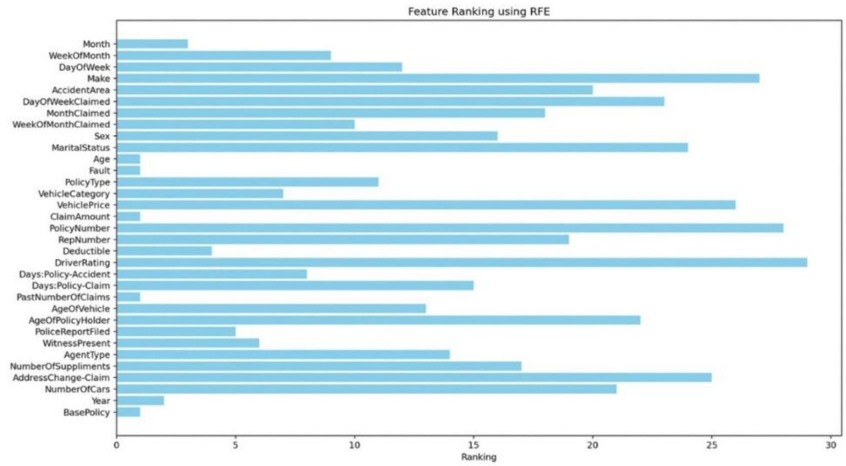
1.Fraud Detection :

- **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 :

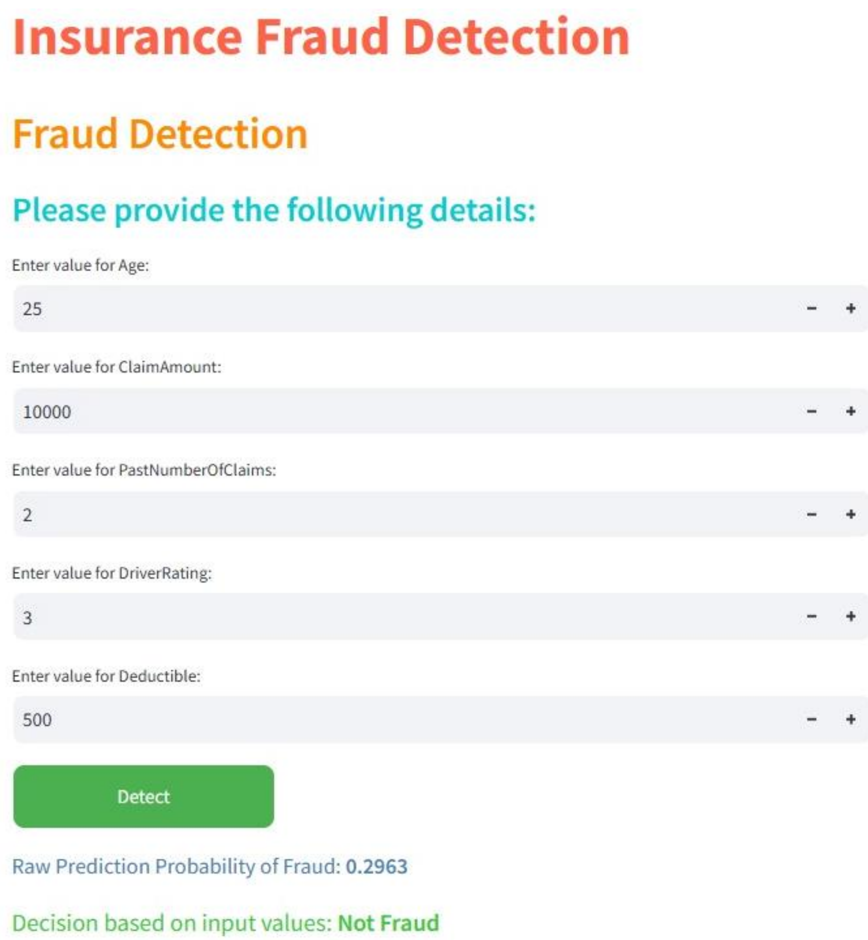
- **Initial model building :**

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 :

Insurance Policy Prediction

Enter the details for Policy Prediction:

Week of the Month of the Claim:

1

Day of the Week of the Claim:

1

Month of the Year of the Claim:

1

Age of the Policy Holder:

30

Claim Amount:

250000

Age of the Vehicle:

5

Year of the Claim:

2024

Predict Policy

Predicted Policy Type: Collision