# Insurance Fraud Detection

## Overview

This project involves analyzing and detecting **fraudulent insurance claims** using a dataset of historical claim information. The dataset used in this project is sourced from (<https://github.com/simranjeet97/Top-Machine-Learning-Algorithms-Python/blob/main/insurance_data.csv>).

Fraud detection is crucial for insurance companies to save costs and maintain credibility. This project utilizes machine learning algorithms to predict whether an insurance claim is fraudulent.

### Objectives:

1. Data Cleaning: Handle missing values and inconsistent data.

2. Data Visualization: Understand trends and patterns in the data.

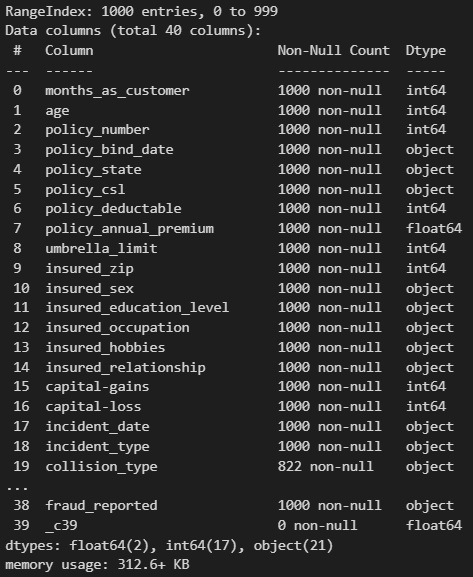
3. Feature Engineering: Prepare the data for machine learning models.

4. Model Training and Evaluation: Build and compare three algorithms—Decision Tree, Random Forest, and Logistic Regression.

5. Model Saving: Save trained models for future use.

### Dataset Information :-

The dataset used for this project contains details about **insurance claims**, including information on the policy, insured individuals, incidents, and the outcome of the claims. This dataset was specifically chosen for its relevance to identifying fraudulent claims. The dataset contains multiple columns representing various attributes:



## Step-by-Step Explanation

### 1. Data Preprocessing

To prepare the dataset for analysis, we focused on cleaning the data and ensuring its quality. This included addressing **missing values** and **removing unnecessary information** to simplify the dataset. Below is an overview of the steps taken:

**1. Importing and Cleaning the Data**

* The dataset was imported, and any missing values represented by **(‘?’)** were replaced with **"NaN"** (a standard way of marking missing data).
* We carefully analyzed the missing data and filled in the gaps using simple methods. For example:
  + Categorical data (like text-based columns) was filled with the most common value, using the statistical method called **mode**.

**2. Handling Missing Values**

When we loaded the dataset, we noticed that some columns contained missing information, which could affect our analysis. Here's how we addressed it:

**Step 1: Identifying Missing Data**

* We created a **bar chart** to visualize the percentage of missing data in each column.
* Some columns, such as **collision\_type**, **property\_damage**, and **police\_report\_available**, had a lot of missing information.

**Step 2: Filling Missing Data**

* For columns like collision\_type, the missing values were replaced with the most **frequent value** in the column.
* For other columns, no changes were made because they were **dropped** in later steps.

**3. Removing Irrelevant Columns**

To simplify the dataset, we **removed several columns** that were either irrelevant or had too much missing information. These include:

* Policy details: policy\_number, policy\_bind\_date, policy\_state, etc.
* Incident location details: incident\_location, incident\_city, etc.
* Vehicle information: auto\_make, auto\_model, auto\_year.
* Miscellaneous: \_c39 (a column with no meaningful data).

By removing these columns, the dataset became cleaner and more focused on the analysis.

**4. Principal Component Analysis (PCA)**

To enhance the dataset for analysis, Principal Component Analysis (PCA) was applied to **reduce** its dimensionality while retaining 95% of the variance. This process simplifies the dataset and helps **improve model performance**.

**Step 1: Splitting the Dataset**

* The data was split into training (80%) and testing (20%) sets

**Step 2: Standardizing the Data**

* To ensure equal contribution of each feature, the data was standardized using **StandardScaler**

**Step 3: Applying PCA**

* PCA was applied to reduce the dimensionality, **retaining 95%** of the variance

**Step 4: Results**

* PCA reduced the dataset's complexity while preserving **important features**, making it easier to manage for further analysis

### 2. Data Visualization:

1. **Analyzed Missing Data**:
   * Created **bar plots** to show the count of missing values for each column, helping to identify which columns required cleaning.
   * Generated a **bar plot** of null value percentages to better understand and justify the preprocessing decisions.
2. **Explored Data Distributions**:

* Visualized the unique value distributions of categorical and numerical columns using **bar plots** to identify potential inconsistencies or trends in the data.

1. **Analyzed Relationships:**

* Created **heatmaps** to examine **correlations** between numerical variables, providing insights into potential relationships that could influence analysis.

1. **Detected Outliers:**

* Used box plots to identify and visualize outliers in numerical columns, ensuring that these anomalies were accounted for during preprocessing.

### 3. Feature Engineering:

### Categorical Data Transformation:

### Applied one-hot encoding to convert categorical variables into numerical form, allowing machine learning models to process them effectively.

### Feature Scaling:

### Scaled numerical features using the StandardScaler to standardize the data. This ensures all numerical features have a mean of 0 and a standard deviation of 1, preventing certain features from dominating others due to differences in scale.

### 4. Machine Learning Models:

* 1. **Decision Tree**:
* Built a basic **Decision Tree** model, which helps make decisions by splitting data into smaller, more manageable groups.
* Improved its performance by adjusting settings (hyperparameters) using **GridSearchCV**, which automatically tests different combinations to find the best one.

* 1. **Random Forest** :
* Used a **Random Forest**, which is like a collection of multiple Decision Trees working together to make better predictions.
* Again, tuned the model's settings using **GridSearchCV** to find the best combination for more accurate results.
  1. **Logistic Regression**:
* Applied a simple **Logistic Regression** model, which is useful for predicting categories (like yes/no).
* After **scaling the data**, this model performed surprisingly well, giving competitive results.

### Model Evaluation:

1. **Evaluated Model Performance**:

* We assessed each model’s performance using key metrics to understand how well they worked:
  + - **Accuracy**: The overall percentage of correct predictions.
    - **Precision**: How many of the predicted positive outcomes were actually positive.
    - **Recall**: How many of the actual positive outcomes were correctly identified.
    - **F1 Score**: A balance between precision and recall, especially useful when there are unequal categories.

1. **Model Comparison**:

* After evaluating the models, we compared them to choose the one with the best performance for deployment.

### 6. Saving the Models

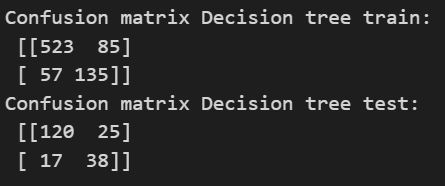
### Saved Trained Models:

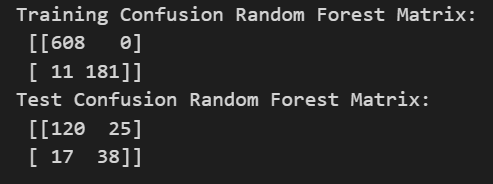
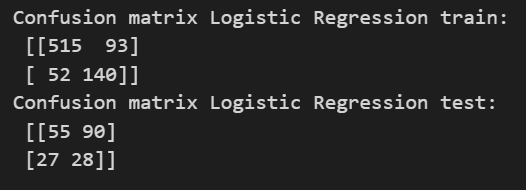
### Stored the trained models (Decision Tree, Random Forest, and Logistic Regression) using the pickle library.

### This allows us to save the models and use them later without retraining, making the process more efficient.

### 7. Confusion Matrix for All Models:

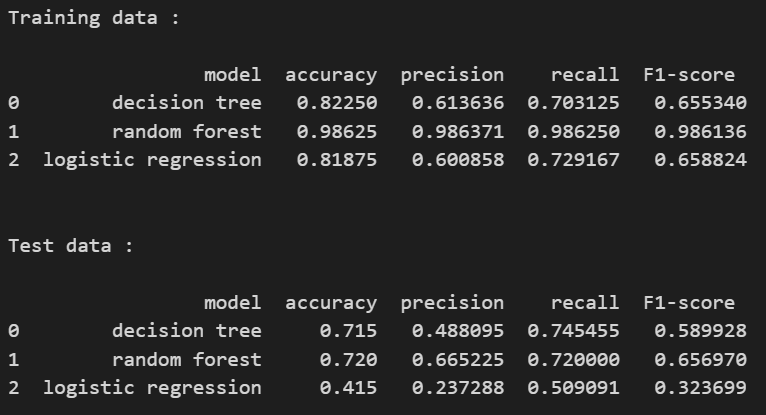
* **Evaluated Model Performance**:
  + Used a **Confusion Matrix** to analyze how well the three models (Random Forest, Decision Tree, and Logistic Regression) performed.
  + The matrix shows:
    - **True Positives (TP)**: Correctly identified fraud cases.
    - **True Negatives (TN)**: Correctly identified non-fraud cases.
    - **False Positives (FP)**: Incorrectly labeled non-fraud cases as fraud.
    - **False Negatives (FN)**: Missed fraud cases.
* **Purpose**:
  + The **confusion matrix** helped assess how accurately each model detected fraudulent insurance claims.





## 

## Results



**Note**: The **Random Forest** model was identified as the most effective based on overall performance.

**How to Run the Project**

1. **Install Prerequisites:**
   * Ensure you have **Python 3** installed.
   * Install the required libraries by running this command in your terminal:



1. **Prepare the Dataset**:
   * Place the dataset file **(‘dataset.csv’)** in the project folder.
   * Ensure the file name matches the one referenced in the code.
2. **Run the Code**:
   * Open the Python script in your preferred IDE (like Jupiternotebook, VS Code) or terminal.
   * Execute the script to start the project.
3. **View Results**:
   * Check the console for accuracy and other performance metrics.
   * **Graphs** and **visualizations** will appear automatically during execution.

### User Interface for Fraud Detection Prediction

The **Streamlit application** provides an easy-to-use interface for predicting fraudulent insurance claims using **pre-trained machine learning models**. The user can input various features of an insurance claim, and based on those inputs, the system will predict whether the claim is fraudulent or not. Here's a breakdown of the code logic and the reasoning behind it:

#### 1. App Initialization

* **Streamlit Setup**: The app is built using **Streamlit**, which provides a simple framework for creating web apps in Python.
* **Model and Scaler Loading**: Upon initialization, the **pre-trained models** (Decision Tree, Random Forest, Logistic Regression) and the scaler are loaded using pickle.load. This allows the app to use the models without retraining them every time the app is run.

#### 2. Defining Features

* **Features Dictionary**: We define a dictionary that lists **all the features** (input fields) expected from the user. Each feature is mapped to its data type (float, int, or bool). This dictionary helps dynamically create the UI elements for user input.
* **Feature Types**: The features cover a wide range of insurance claim details, including policy information, incident severity, and other factors relevant to fraud detection.

#### 3. Input Collection

* **Dynamic UI Elements**: Based on the feature types, Streamlit dynamically generates the input fields:
  + **Boolean Inputs**: For features like "insured\_occupation\_armed-forces", the user can select **"True" or "False"** using a dropdown **(st.selectbox).**
  + **Integer and Float Inputs**: For numerical features like "injury\_claim", "months\_as\_customer", etc., the user enters values using **st.number\_input**, which ensures only valid integers or floats are accepted.
* **Input Collection for All Features**: This loop iterates through the features and creates the appropriate UI component for each one, ensuring the app remains scalable if more features are added in the future.

#### 4. Model Selection

* **Model Dropdown**: The user is presented with a **dropdown (st.selectbox)** to choose between three different models: **Decision Tree, Random Forest, or Logistic Regression.**
* **Model Mapping**: Based on the user’s selection, the app loads the corresponding pre-trained model. This flexibility allows **users to compare** predictions from **different models**.

#### 5. Preprocessing User Input

* **Preprocessing Function**: Once the user inputs their data, the **preprocess\_input** function processes the input to convert the data into a format suitable for the model:
  + Boolean values ("True"/"False") are converted into binary values (1 or 0).
  + Integer and float values are cast appropriately.
  + This ensures that the input data matches the model’s expected format.

#### 6. Prediction and Display

* **Prediction Logic**: When the user clicks the "**Predict**" button, the following steps occur:
  + The user inputs are processed using the preprocess\_input function.
  + The data is then scaled using the preloaded scaler, ensuring that numerical features are scaled consistently with the training data.
  + The selected model (Decision Tree, Random Forest, or Logistic Regression) **predicts** whether the claim is **fraudulent** or **not** using the processed and scaled input data.
* **Prediction Output**: The app then displays the result to the user:
  + If the prediction indicates fraud (1), it will display "**Fraud**".
  + If the prediction indicates no fraud (0), it will display "**Not Fraud**".
  + The prediction result is shown using **st.success** for positive feedback to the user.

#### 7. Conclusion

The app provides an **intuitive**, **interactive** **user interface** for insurance claim **fraud detection**. By selecting features and a model, users can easily input claim data and receive immediate feedback on the likelihood of fraud. The dynamic nature of the app ensures that it is adaptable to various datasets and user inputs.