**Insurance Claims Fraud Detection using Machine Learning & Business Intelligence**

**Project Overview**

Objective:

This project aims to weave the fine threads of machine learning (ML) and business intelligence (BI) into a powerful fabric that detects fraudulent vehicle insurance claims. It’s designed to shield insurers from losses, accelerate payouts for honest customers, and arm investigators with clarity and conviction.

Problem Statement:

Fraud is not an intermittent leak—it’s a widening tide. In India, experts estimate that 15% of insurance claims are fraudulent, translating to nearly ₹900 crore lost every year . Globally, insurance fraud costs surpass $80 billion annually, with motor insurance fraud rising by 19% in 2023 alone .

From my experience in vehicle claims investigations, I’ve seen how simple claims can be twisted—fake documentation, staged bumps, repeated dents.

Meanwhile, insurers such as Bajaj Allianz and Kotak General Insurance are modernizing settlement with video surveys, AI-based damage assessments, and cashless garage networks. Yet these innovations are double-edged—quick claims, instant pay, but also fertile ground for fraud, especially when repair bills are padded or images are doctored.

International insurers—Progressive (US), Aviva (UK)—deploy telematics, social data analysis, and predictive fraud scoring. This project mirrors those global elite practices while drawing deeply from my domain knowledge to craft something both robust and real.

Business Impact:

Protect the Bottom Line: Stop fraudulent payouts before they eat into profits.

Empower Investigators: Let human teams focus on high-risk cases, not busywork.

Boost Customer Trust: Speed genuine settlements, and let the process feel fair.

Illuminate Fraud Trends: BI dashboards that map fraud hotspots, garage red flags, and claim velocity.

Scope of ML & BI Solution:

ML Component: A binary classifier (fraud vs. genuine) trained on real-world motor claim features—vehicle type, claim type, documentation flags, repair cost anomalies, repeat-claim behavior, image evidence, etc.

BI Component: Rich, interactive dashboards (e.g. Power BI) illustrating claim clusters, risk zones, claim turnaround, and investigator actions.

Deliverables:

A clean, production-ready ML fraud detection pipeline (training, validation, monitoring).

A BI dashboard that brings data to life for stakeholders.

A real-time scoring API for new claims, integrated into claims systems.

**Data Description**

The dataset used for this project contains 15,420 insurance claim records, each describing details about the policyholder, vehicle, accident, and claim. It includes both categorical and numerical features. The target variable is whether the claim was fraudulent (FraudFound\_P).

Key Variables:

Claim Context:

Month, WeekOfMonth, DayOfWeek – Time dimensions of the accident/claim.

AccidentArea – Location type (Urban / Rural).

DayOfWeekClaimed, MonthClaimed, WeekOfMonthClaimed – Report vs. claim timing (fraudsters often delay claims).

Year – Year of accident/claim.

Policyholder Information:

Sex, MaritalStatus, Age, AgeOfPolicyHolder.

PolicyType – Coverage type.

RepNumber – Representative handling the case.

Vehicle Information:

Make – Manufacturer of the vehicle.

VehicleCategory – Category (Sedan, SUV, Commercial, etc.).

VehiclePrice – Declared price of the vehicle.

AgeOfVehicle – Age in years.

NumberOfCars – Total cars owned by the policyholder.

Claim & Fraud Indicators:

Fault – Fault assignment (Policyholder / Third Party).

Deductible – Out-of-pocket amount for policyholder.

DriverRating – Policyholder’s driving risk rating.

Days\_Policy\_Accident – Days since last accident.

Days\_Policy\_Claim – Days since last claim.

PastNumberOfClaims – Historical claim count.

NumberOfSuppliments – Additional claims after initial filing.

AddressChange\_Claim – Whether policyholder changed address recently.

PoliceReportFiled, WitnessPresent – Legal/validation indicators.

Agent/Investigation Related:

AgentType – Type of agent (Internal/External).

Target Variable:

FraudFound\_P – Binary (1 = Fraudulent Claim, 0 = Genuine Claim).

Data Characteristics:

Records: 15,420

Features: 32 independent variables (mix of categorical & numerical).

Target: 1 variable (FraudFound\_P).

Fraud Ratio: Highly imbalanced (fraud cases are significantly fewer than genuine cases, typical of real-world insurance data)

**Exploratory Data Analysis (EDA)**

1 Target Variable Distribution

Analyze FraudFound\_P (fraud vs. non-fraud).

Expect imbalance (fraudulent claims << genuine claims).

Visualization: Bar chart showing % of fraud vs. non-fraud.

Example insight: “Only ~6% of claims are fraudulent, confirming the rarity of fraud and the importance of precision-focused models.”

2 Policyholder Demographics

Age of policyholder: Fraud may cluster in younger (<25) or older (>60) segments.

Marital status & sex: Check whether fraud is more common in single vs married claimants.

Visualization: Histogram & grouped bar chart.

3 Vehicle-Related Insights

Age of vehicle: Older vehicles may have higher fraud claims.

Vehicle price/category: Compare luxury vs economy claims.

Visualization: Boxplots (vehicle price vs fraud), histograms (vehicle age).

4 Claim Behavior Patterns

Claim timing:

Month, DayOfWeekClaimed, WeekOfMonthClaimed.

Fraud may spike at month-end or weekends.

Address change before claim: Fraudsters often change addresses before filing suspicious claims (AddressChange\_Claim).

Visualization: Heatmap / count plots.

5 Policy & Claim History

Past number of claims: Repeat claimants may be riskier.

Deductible vs fraud: Higher deductibles might reduce fraud attempts.

Number of supplements: More supplements → higher fraud probability.

6 Investigation Features

Police report filed: Fraudulent claims often lack police reports.

Witness present: Fraud cases may have fewer witnesses.

Visualization: Stacked bar chart (fraud vs non-fraud with/without witness/police report).

7 Correlation Analysis

Numerical features correlation heatmap: (Age, Deductible, PastClaims, DriverRating).

Helps detect hidden patterns (e.g., low driver rating correlated with fraud).

**Data Processing**

The dataset (fraud\_oracle.csv) contained 15,420 records with 33 features.

Basic cleaning was performed:

Replaced invalid values (?) with NaN.

Removed duplicate records.

Confirmed variable types: 9 numerical, 24 categorical.

**Model Implementation**

**Logistic Regression**

After preprocessing, the first model implemented was Logistic Regression, a strong baseline for binary classification tasks like fraud detection.

This approach was chosen due to its simplicity, interpretability, and efficiency in handling large categorical datasets after encoding.

Steps Followed:

1. Separated features and target:

Target variable: FraudFound\_P (1 = Fraudulent, 0 = Genuine).

Features: All other columns.

2. Identified variable types:

Categorical Features: e.g., Month, DayOfWeek, Make, BasePolicy.

Numerical Features: e.g., Age, Year, DriverRating.

3. Created preprocessing pipeline:

Numerical features → Standardized using StandardScaler.

Categorical features → Encoded using OneHotEncoder (with handle\_unknown="ignore" to handle unseen categories).

4. Combined preprocessing & model into a pipeline:

Used ColumnTransformer to apply transformations.

Trained a LogisticRegression model (solver = "liblinear" for small datasets).

CODE:

import numpy as np

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

# Separate target variable

X = df.drop('FraudFound\_P', axis=1)

y = df['FraudFound\_P']

# Identify categorical and numerical features

categorical\_features = X.select\_dtypes(include=['object', 'category']).columns

numerical\_features = X.select\_dtypes(include=np.number).columns

# Create preprocessing pipelines for numerical and categorical features

numerical\_transformer = StandardScaler()

categorical\_transformer = OneHotEncoder(handle\_unknown='ignore') # handle\_unknown='ignore' for unseen categories

# Create a column transformer to apply different transformations to different columns

preprocessor = ColumnTransformer(

    transformers=[

        ('num', numerical\_transformer, numerical\_features),

        ('cat', categorical\_transformer, categorical\_features)])

# Create the logistic regression model pipeline

model = Pipeline(steps=[('preprocessor', preprocessor),

                        ('classifier', LogisticRegression(solver='liblinear'))]) # Using 'liblinear' solver for smaller datasets

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y) # stratify to maintain proportion of target variable

# Train the model

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

# Make predictions on the test set

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

# Evaluate the model

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred))

print("\nConfusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

**Random Forest Classifier**

Used 100 estimators with random\_state=42.

The dataset was one-hot encoded for categorical variables.

The data was split into 80% training and 20% testing sets.

Evaluation metrics included accuracy, precision, recall, F1-score, and confusion matrix.

CODE:

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

import pandas as pd # Ensure pandas is imported if it wasn't in the previous cell

# Load dataset (assuming 'df' is already loaded and basic cleaning is done)

# If 'df' needs to be reloaded or is not available, uncomment and run this block:

# path = kagglehub.dataset\_download("shivamb/vehicle-claim-fraud-detection") # Assuming path is still valid

# df = pd.read\_csv(path + '/fraud\_oracle.csv')

# df.replace("?", pd.NA, inplace=True)

# df.drop\_duplicates(inplace=True)

# Separate features (X) and target (y)

X = df.drop('FraudFound\_P', axis=1)

y = df['FraudFound\_P']

# Identify categorical columns (object type)

categorical\_cols = X.select\_dtypes(include='object').columns

# Apply one-hot encoding to categorical columns

# handle\_unknown='ignore' is useful if your test set might have categories not seen in the training set

X\_encoded = pd.get\_dummies(X, columns=categorical\_cols, drop\_first=True, dummy\_na=False)

# Split the encoded data

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

# Now proceed with model training

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

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

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

print(classification\_report(y\_test, y\_pred))

**K-Nearest Neighbors (KNN)**

Scaled data using StandardScaler to normalize features.

Tried k=5 neighbors.

Similar evaluation approach with accuracy and classification metrics.

CODE:

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn.neighbors import KNeighborsClassifier

import pandas as pd # Ensure pandas is imported

# Features and label

X = df.drop('FraudFound\_P', axis=1)

y = df['FraudFound\_P']

# Identify categorical columns (object type)

categorical\_cols = X.select\_dtypes(include='object').columns

# Apply one-hot encoding to categorical columns

# This converts string columns like 'Month', 'VehicleCategory', etc. into numerical columns

X\_encoded = pd.get\_dummies(X, columns=categorical\_cols, drop\_first=True, dummy\_na=False)

# Scale features using the encoded data

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X\_encoded) # Use X\_encoded instead of X

# Train-test split using the scaled data

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

# Now you can proceed with training your KNN model

# For example:

# knn\_model = KNeighborsClassifier(n\_neighbors=5)

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

# y\_pred = knn\_model.predict(X\_test)

# print(classification\_report(y\_test, y\_pred))

**Advanced Modeling with Imbalance Handling (SMOTE + XGBoost)**

Insurance fraud data is inherently imbalanced, with far fewer fraudulent claims than genuine ones. To address this, I applied SMOTE (Synthetic Minority Over-sampling Technique), which generates synthetic examples of the minority class, ensuring the model doesn’t get biased toward predicting only non-fraud.

After balancing the dataset, I implemented XGBoost, a powerful gradient boosting algorithm known for handling structured/tabular data effectively. To account for class imbalance, I also tuned the scale\_pos\_weight parameter based on the ratio of genuine to fraudulent claims.

Process followed:

1. Applied SMOTE to oversample fraudulent claims in the training data.

2. Performed One-Hot Encoding on categorical variables.

3. Split the data into training and testing sets (stratified to maintain class distribution).

4. Trained XGBoost Classifier using parameters optimized for imbalance (scale\_pos\_weight).

5. Evaluated the model using:

Confusion Matrix → to analyze false positives/false negatives

Classification Report (Precision, Recall, F1-score)

ROC-AUC Score for overall discrimination ability

Outcome:

Improved recall for fraud detection compared to baseline models.

ROC-AUC score significantly higher, indicating better separation between fraudulent and genuine claims.

XGBoost with SMOTE proved to be the most effective model in capturing fraudulent claims without excessively flagging genuine ones.

**Model Comparison & Hyperparameter Tuning**

After training multiple models (Logistic Regression, Random Forest, KNN, and XGBoost), I performed a systematic comparison to identify the best-performing approach for fraud detection.

Approach:

1. Baseline Models: Logistic Regression, Random Forest, KNN

2. Advanced Model: XGBoost with SMOTE

3. Evaluation Metrics:

Precision → To ensure flagged frauds are truly fraudulent.

Recall (Sensitivity) → To minimize missed fraud cases.

F1-Score → Balance between Precision & Recall.

ROC-AUC → Overall discrimination power.

Model Accuracy Precision (Fraud=1) Recall (Fraud=1) F1-Score

Logistic Regression 0.94 0.67 0.01 0.02

Random Forest Classifier 0.94 1.00 0.01 0.01

KNN 0.93 0.25 0.03 0.05

Logistic Regression + SMOTE 0.96 0.62 0.76 0.68

ROC AUC Comparison

Model ROC AUC

Logistic Regression 0.75

Random Forest Classifier 0.78

KNN 0.70

Logistic Regression + SMOTE 0.966

2. Hyperparameter tuning for XGBoost:

from xgboost import XGBClassifier

from sklearn.model\_selection import RandomizedSearchCV

params = {

    'n\_estimators': [100, 200, 500],

    'max\_depth': [3, 5, 7, 10],

    'learning\_rate': [0.01, 0.05, 0.1, 0.2],

    'subsample': [0.7, 0.8, 1.0],

    'colsample\_bytree': [0.7, 0.8, 1.0],

    'scale\_pos\_weight': [1, 3, 5]  # imbalance handling

}

xgb\_model = XGBClassifier(eval\_metric='logloss', random\_state=42)

random\_search = RandomizedSearchCV(xgb\_model, param\_distributions=params,

                                   n\_iter=20, scoring='f1', cv=3, n\_jobs=-1, verbose=2)

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

best\_xgb = random\_search.best\_estimator\_

print("Best Params:", random\_search.best\_params\_)

**Thershold Optimization**

By default, classifiers like Logistic Regression / XGBoost predict probabilities.

Scikit-learn converts these into class labels with a default threshold of 0.5 → if p ≥ 0.5, predict Fraud.

But in fraud detection, this default is not always optimal.

So you tuned the threshold by maximizing F1-score (balance between Precision & Recall).

CODE:

from sklearn.metrics import precision\_recall\_curve

precision, recall, thresholds = precision\_recall\_curve(y\_test, y\_pred\_proba)

# F1 = 2PR / (P+R)

f1\_scores = 2 \* (precision \* recall) / (precision + recall)

best\_idx = np.argmax(f1\_scores)

best\_threshold = thresholds[best\_idx]

This finds the probability cutoff that gives the best F1-score.

Your best threshold turned out to be 0.61 (not the default 0.50).

Why It’s Important?

Accuracy alone is misleading in fraud detection (because fraud cases are rare).

By optimizing the threshold, you ensure the model prioritizes fraud recall without spamming false alarms.

**Business Impact Metrics**

Fraud detection is never just about accuracy — it’s about money saved.

You frame your results in financial terms:

Example Conversion to Cost Savings

Suppose your model catches 80% of fraudulent claims (Recall = 0.80).

Industry data: average fraudulent claim = ₹1.5 lakhs (adjust per region).If insurer processes 10,000 claims/year with ~5% fraud rate → 500 fraud cases.

Without model: assume only 50% caught = 250 frauds slip → ₹3.75 Cr loss.

With your model (80% recall): only 100 frauds slip → ₹1.5 Cr loss.

Business Impact:

This kind of model saves ₹2.25 Cr annually just by improving recall from 50% → 80%.

Confusion Matrix with Costs

False Negative (missed fraud) = ₹1.5 lakhs loss each

False Positive (wrongly flagged claim) = ~₹10k investigation cost

Translate confusion matrix counts → real money impact.