3.Fraudulent Insurance Claims Dataset

- TC Antony

Project Observation

This project focuses on detecting fraudulent insurance claims using anomaly detection and classification techniques. Key features include claim details, policyholder information, and fraud indicators. Feature engineering introduced a Claim-to-Income ratio and identified claims filed unusually soon after policy issuance. Outlier detection methods like Elliptic Envelope, Isolation Forest, and Local Outlier Factor (LOF) were applied to flag suspicious claims. Finally, a fraud probability score (0-1) was calculated using an ensemble of Random Forest and Neural Networks, providing a robust approach to fraud risk prediction. Initial results indicate strong performance but suggest room for improvement in generalization and overfitting reduction.

6 - 2nd Attempt

i] Feature Selection [184 - 203 Slide]
ii] Model Training [204 - 214 Slide]
iii] Future Prediction [215 - 220 Slide]

7 - Conclusion - [221-222 Slide]

Dataset:

Claim_ID	Policyholder_ID	Claim_Amount	Claim_Type	Suspicious_Flags	Fraud_Label	Year	Month	Day	Month_name
bdd640fb-0667-4ad1-9c80-317fa3b1799d	1a3d1fa7-bc89-40a9-a3b8-c1e9392456de	32151.63	Medical	False	0	2021	7	4	July
8b9d2434-e465-4150-bd9c-66b3ad3c2d6d	17fc695a-07a0-4a6e-8822-e8f36c031199	11548.93	Home Damage	True	0	2020	7	25	July
9a1de644-815e-46d1-bb8f-aa1837f8a88b	b38a088c-a65e-4389-b74d-0fb132e70629	29729.38	Medical	True	0	2020	4	6	April
72ff5d2a-386e-4be0-ab65-a6a48b8148f6	c241330b-01a9-471f-9e8a-774bcf36d58b	11322.58	Home Damage	True	0	2023	1	29	January
6c307511-b2b9-437a-a8df-6ec4ce4a2bbd	c37459ee-f50b-4a63-b71e-cd7b27cd8130	35942.97	Home Damage	False	0	2021	10	31	October

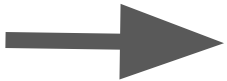
```
[ ]  1 df3.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Unnamed: 0        1000 non-null   int64  
 1   Claim_ID          1000 non-null   object 
 2   Claim_Date        1000 non-null   object 
 3   Policyholder_ID   1000 non-null   object 
 4   Claim_Amount      1000 non-null   float64
 5   Claim_Type        1000 non-null   object 
 6   Suspicious_Flags  1000 non-null   bool   
 7   Fraud_Label       1000 non-null   int64  
dtypes: bool(1), float64(1), int64(2), object(4)
memory usage: 55.8+ KB
```

Extract Day, Month, Year From Date

```
df3['Claim_Date']=pd.to_datetime(df3['Claim_Date'])
df3['Year']=df3['Claim_Date'].dt.year
df3['Month']=df3['Claim_Date'].dt.month
df3['Day']=df3['Claim_Date'].dt.day
df3.head()
```

Claim_Date
2021-07-04
2020-07-25
2020-04-06
2023-01-29
2021-10-31



Year	Month	Day
2021	7	4
2020	7	25
2020	4	6
2023	1	29
2021	10	31

Mapping Month to Month Name:

```
dw_maping={

1:'January',
2:'February',
3:'March',
4:'April',
5:'May',
6:'June',
7:'July',
8:'August',
9:'September',
10:'October',
11:'November',
12:'December'
}
df3['Month_name']=df3['Claim_Date'].dt.month.map(dw_maping)
```

Month
7
7
4
1
10

Month_name
July
July
April
January
October

New Dataset :

	Claim_ID	Policyholder_ID	Claim_Amount	Claim_Type	Suspicious_Flags	Fraud_Label	Year	Month	Day	Month_name
0	bdd640fb-0667-4ad1-9c80-317fa3b1799d	1a3d1fa7-bc89-40a9-a3b8-c1e9392456de	32151.63	Medical	False	0	2021	7	4	July
1	8b9d2434-e465-4150-bd9c-66b3ad3c2d6d	17fc695a-07a0-4a6e-8822-e8f36c031199	11548.93	Home Damage	True	0	2020	7	25	July
2	9a1de644-815e-46d1-bb8f-aa1837f8a88b	b38a088c-a65e-4389-b74d-0fb132e70629	29729.38	Medical	True	0	2020	4	6	April
3	72ff5d2a-386e-4be0-ab65-a6a48b8148f6	c241330b-01a9-471f-9e8a-774bcf36d58b	11322.58	Home Damage	True	0	2023	1	29	January
4	6c307511-b2b9-437a-a8df-6ec4ce4a2bbd	c37459ee-f50b-4a63-b71e-cd7b27cd8130	35942.97	Home Damage	False	0	2021	10	31	October

```
1 df3.nunique()
```

	0
Claim_ID	1000
Policyholder_ID	1000
Claim_Amount	1000
Claim_Type	3
Suspicious_Flags	2
Fraud_Label	2
Year	6
Month	12
Day	31
Month_name	12

Fraud_Label (1 = Fraud, 0 = Genuine)

Primary Analysis (Univariate)

Claim_Amount (Numerical)

Histogram: To check the distribution. Box Plot: To identify outliers.

Claim_Type (Categorical)

Bar Chart: To analyze the frequency of different claim types.

Fraud_Label (Categorical - Target Variable)

Bar Chart: To check fraud vs. non-fraud claims.

Suspicious_Flags (Numerical/Ordinal)

Histogram or Bar Chart: To check distribution of flagged claims.

Date Features (Year, Month, Day)

Line Chart: Trends of claims over time.

Heatmap (Day vs. Month): To find seasonality in claims.

Cross-Analysis (Bivariate & Multivariate)

Claim_Type vs. Fraud_Label

Stacked Bar Chart: To check which claim types are associated with fraud.

Month_name vs. Fraud_Label

Bar Chart: To analyze fraud occurrences per month.

Claim_Amount vs. Fraud_Label

Box Plot: To check if fraudulent claims have different amount distributions.

Claim_Amount vs. Suspicious_Flags

Scatter Plot: To analyze correlation between flagged claims and amount.

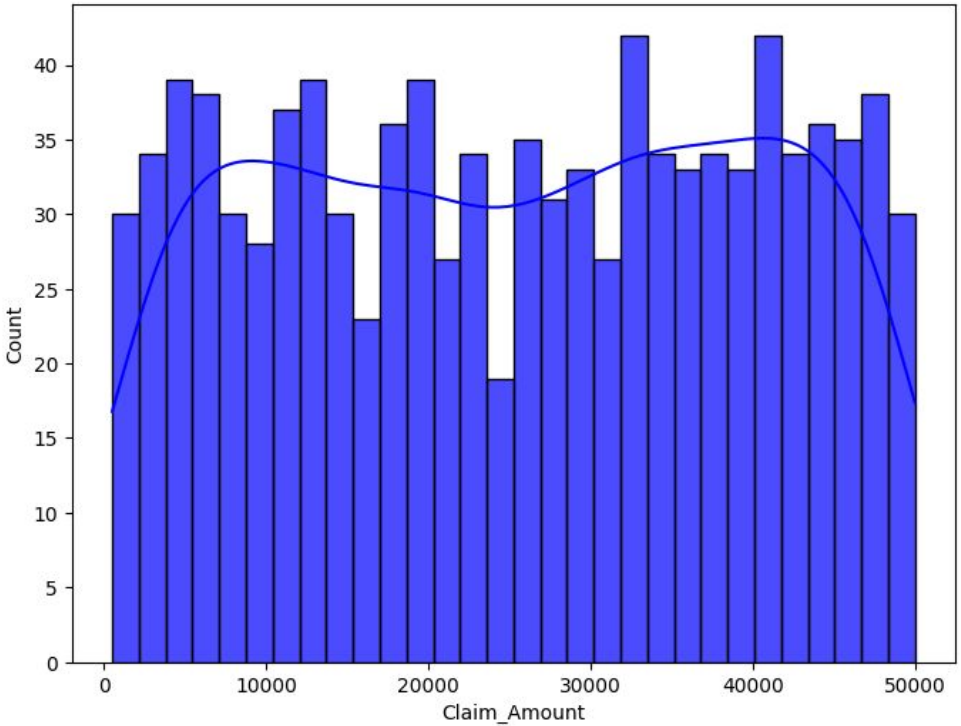
Claim_Amount vs. Claim_Type

Box Plot: To check if claim type influences the amount.

1) Claim_Amount (Numerical)

#Histogram: To check the distribution.

 # Box Plot: To identify outliers.



-0.033663832483501764

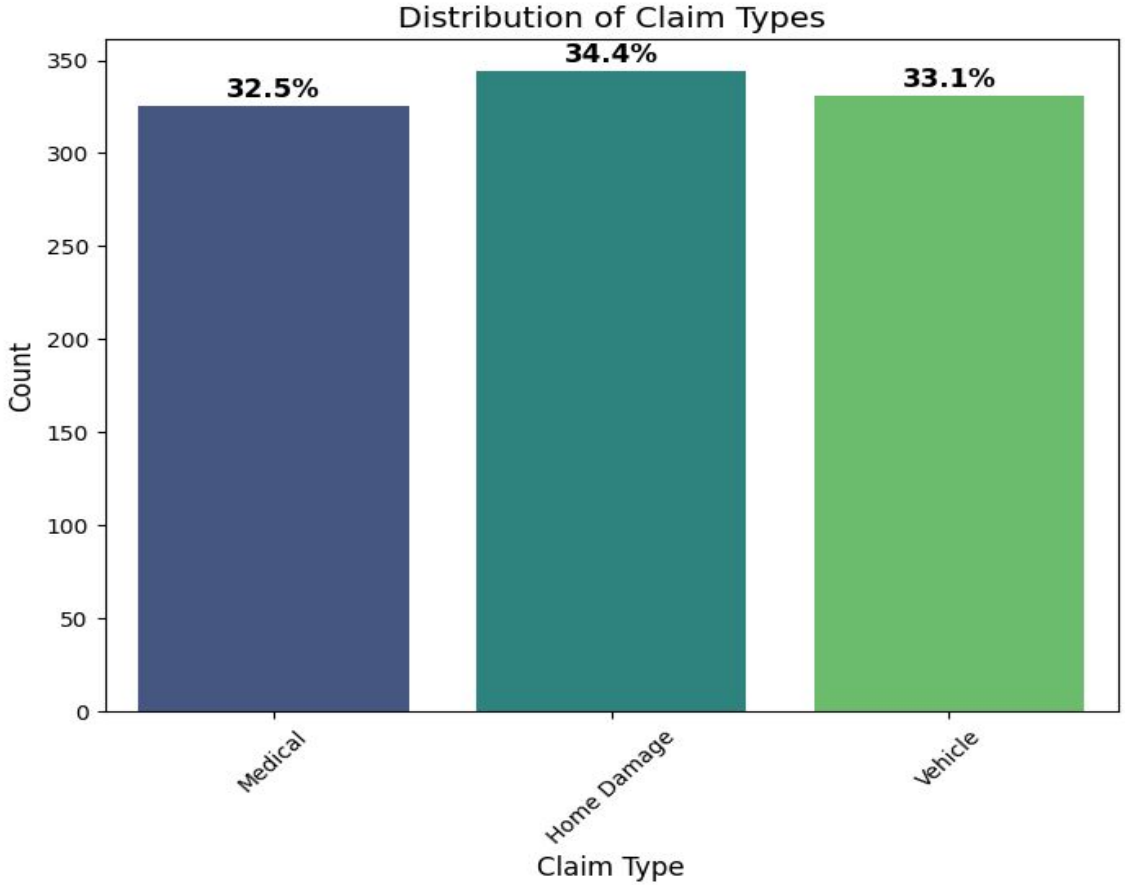
No clear normal distribution:

The data **doesn't resemble a bell curve**, indicating that claim amounts are not normally distributed. **This is common for insurance claims**, as **extreme (high) value claims are less frequent.**

- 0.033663832483501764 is near to 0 -> its slighlty Normal Distribution

2) Claim_Type (Categorical)

Bar Chart: To analyze the frequency of different claim types.



Relatively Even Distribution:

The chart shows a fairly balanced distribution **across the three claim types: Medical, Home Damage, and Vehicle**. None of the categories significantly dominates the others.

Slightly Higher Home Damage Claims:

Visually, the "Home Damage" bar appears to be slightly taller than the other two, suggesting a marginally higher count in this category. However, the difference is not substantial.

Similar Counts for Medical and Vehicle:

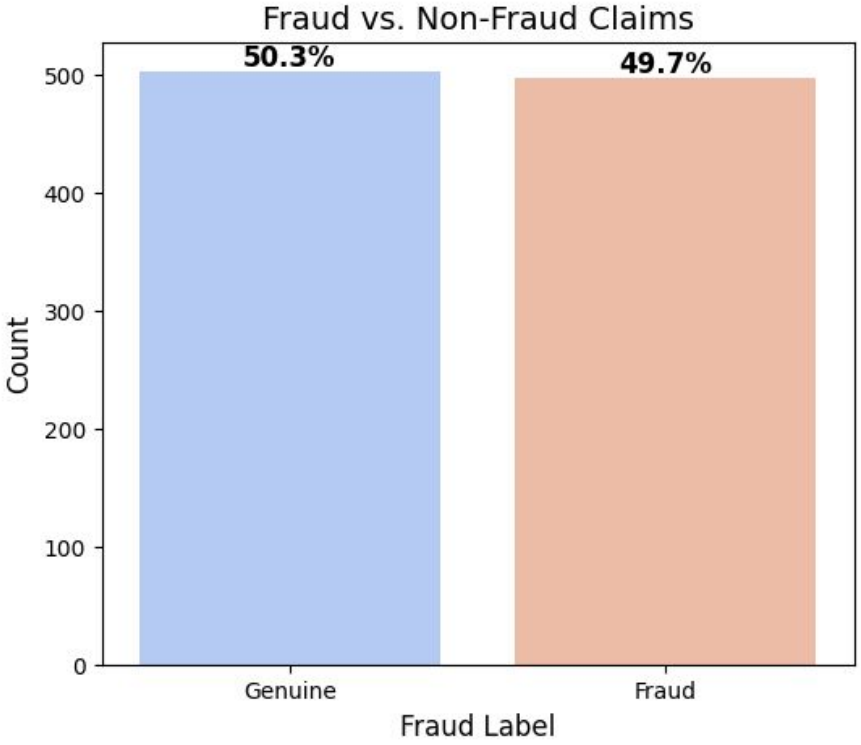
The "Medical" and "Vehicle" claim counts seem to be quite similar, with **"Medical" possibly being slightly lower**.

3) Fraud_Label (Categorical - Target Variable)

Bar Chart: To check fraud vs. non-fraud claims.

Fraud_Label	count
0	503
1	497

dtype: int64



(1 = Fraud, 0 = Genuine)

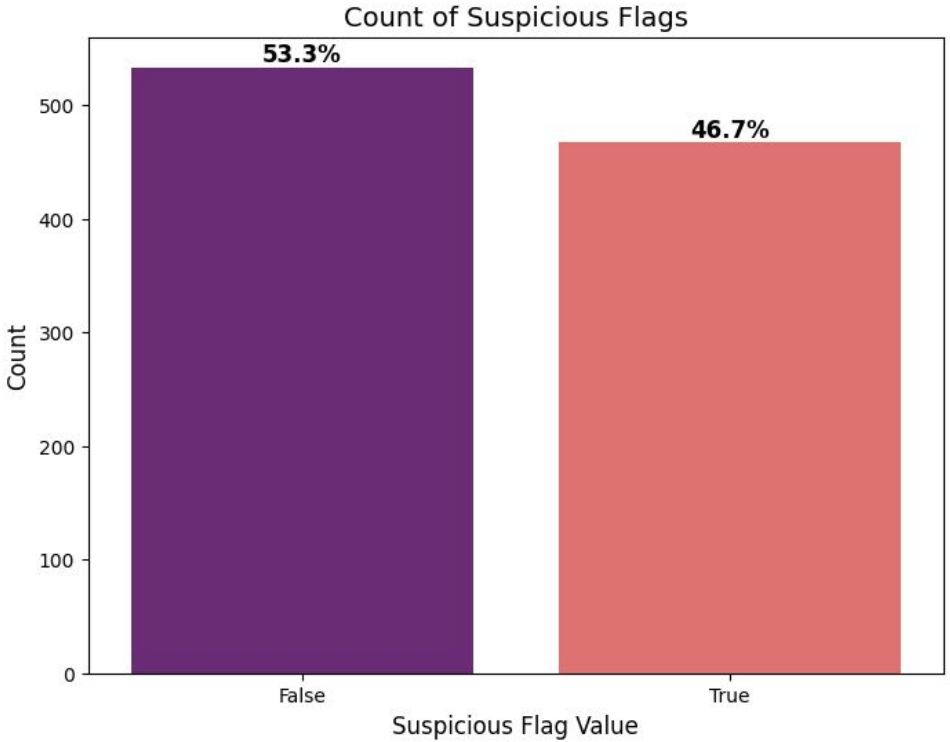
Balanced Classes:

The chart depicts a nearly perfect balance between the number of "Non-Fraud" and "Fraud" claims. The counts for both categories are almost identical.

Geniune has higher than Fraud Slightly

4) Suspicious_Flags (Numerical/Ordinal)

Histogram or Bar Chart: To check distribution of flagged claims.



Insights:

✅ True (1) → The claim is flagged as suspicious (potential fraud or anomaly detected).

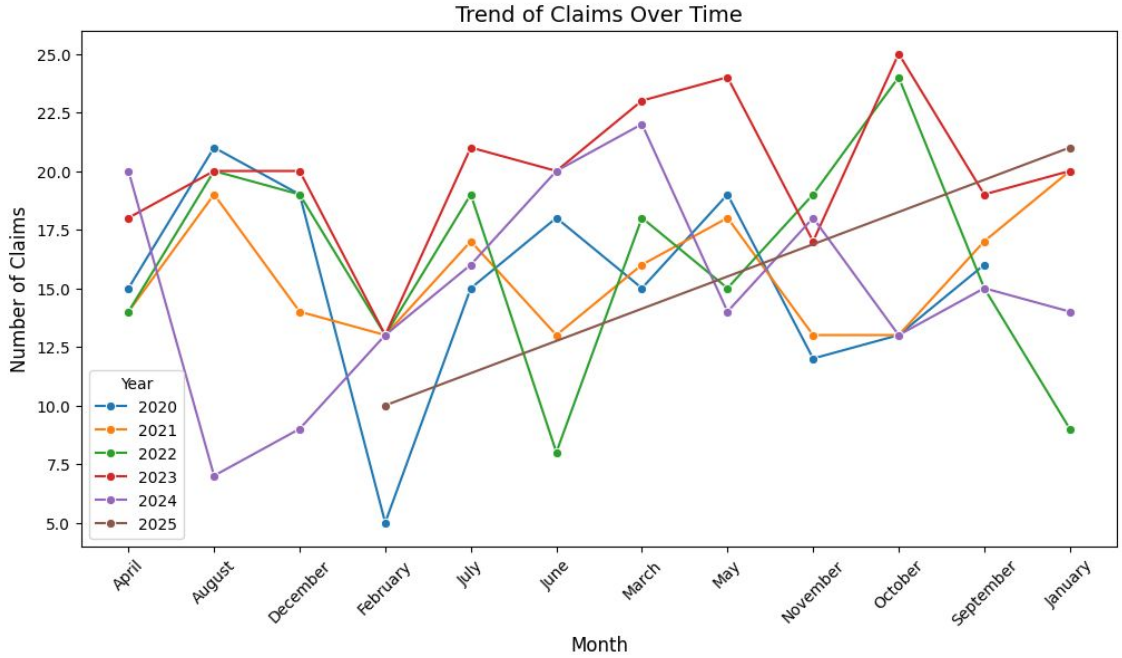
❌ False (0) → The claim is not suspicious (no immediate fraud indicators).

Higher Count of False Flags: The chart clearly shows a higher count of "False" suspicious flags compared to "True" flags.

5) Date Features (Year, Month, Day)

Line Chart: Trends of claims over time.

Heatmap (Day vs. Month): To find seasonality in claims.



Highest Claim Count (Year and Month):

2023 shows the highest claim count overall. Within 2023, **October** appears to have the highest number of claims.

Average Claims:

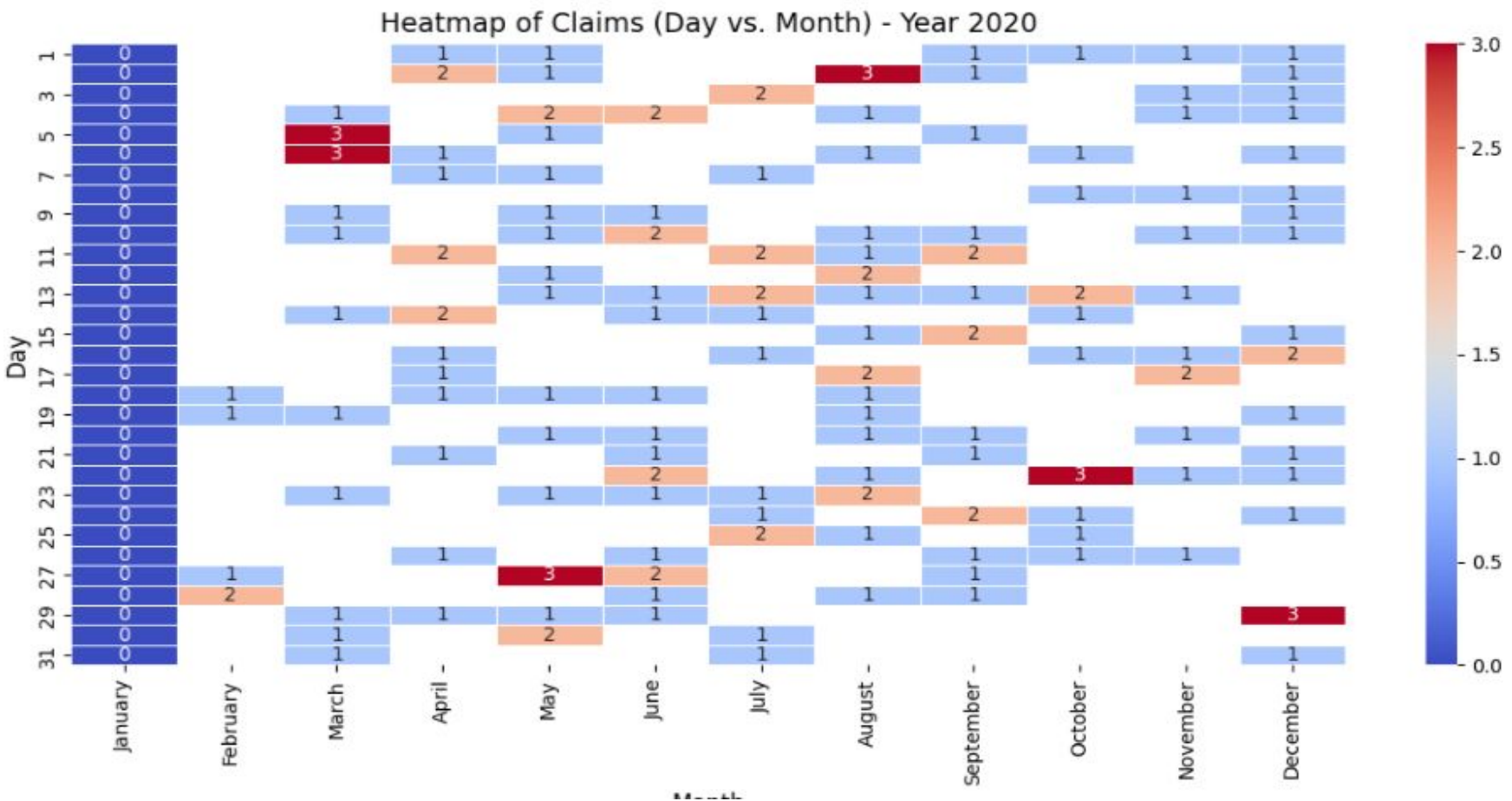
To determine the average claims, we'd ideally need the exact numerical values for each point. However, looking at the chart, we can make an approximation: It seems **the majority of data points fall between 12 and 20 claims**. Eyeballing it, a rough estimate for the overall average might be somewhere in the mid-teens **(around 15-16 claims)**. Keep in mind this is a visual approximation.

Minimum Claims (Year and Month):

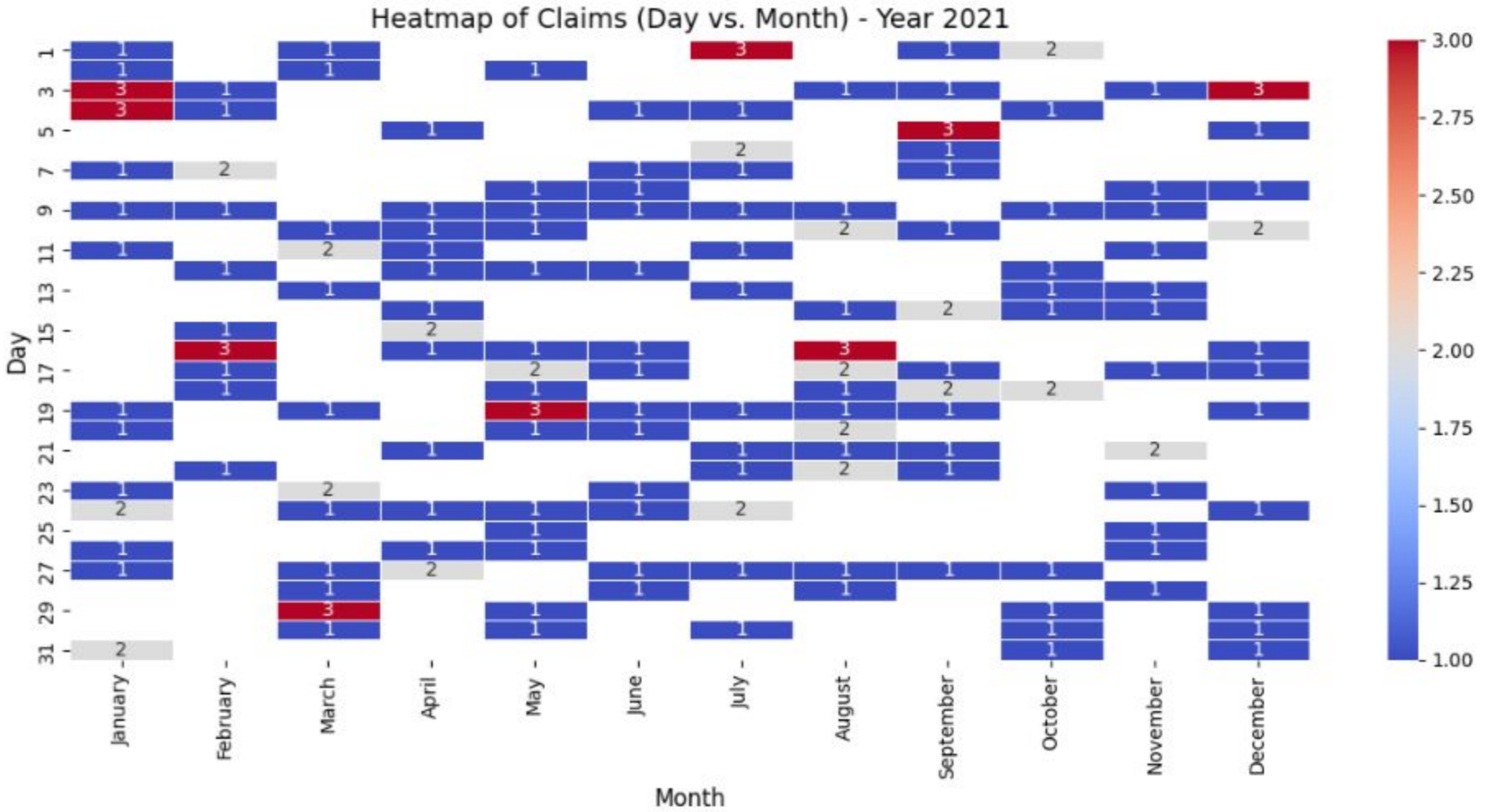
2020 and 2024 both have months with very low claim counts.

**February 2020 appears to have the absolute lowest number of claims.

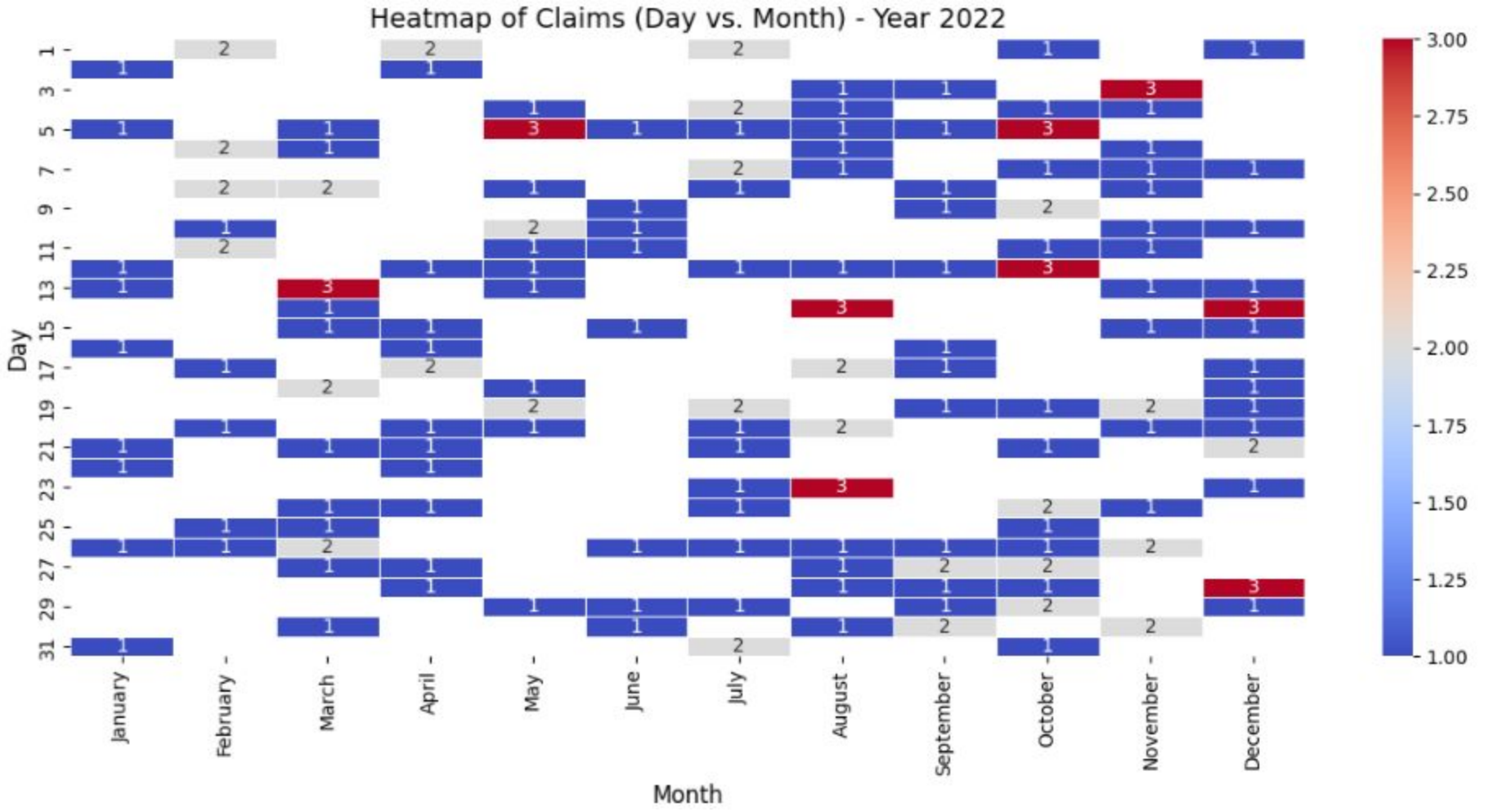
Using Heatmap to Analyse Deeply based on Number of Claims in Month & Year



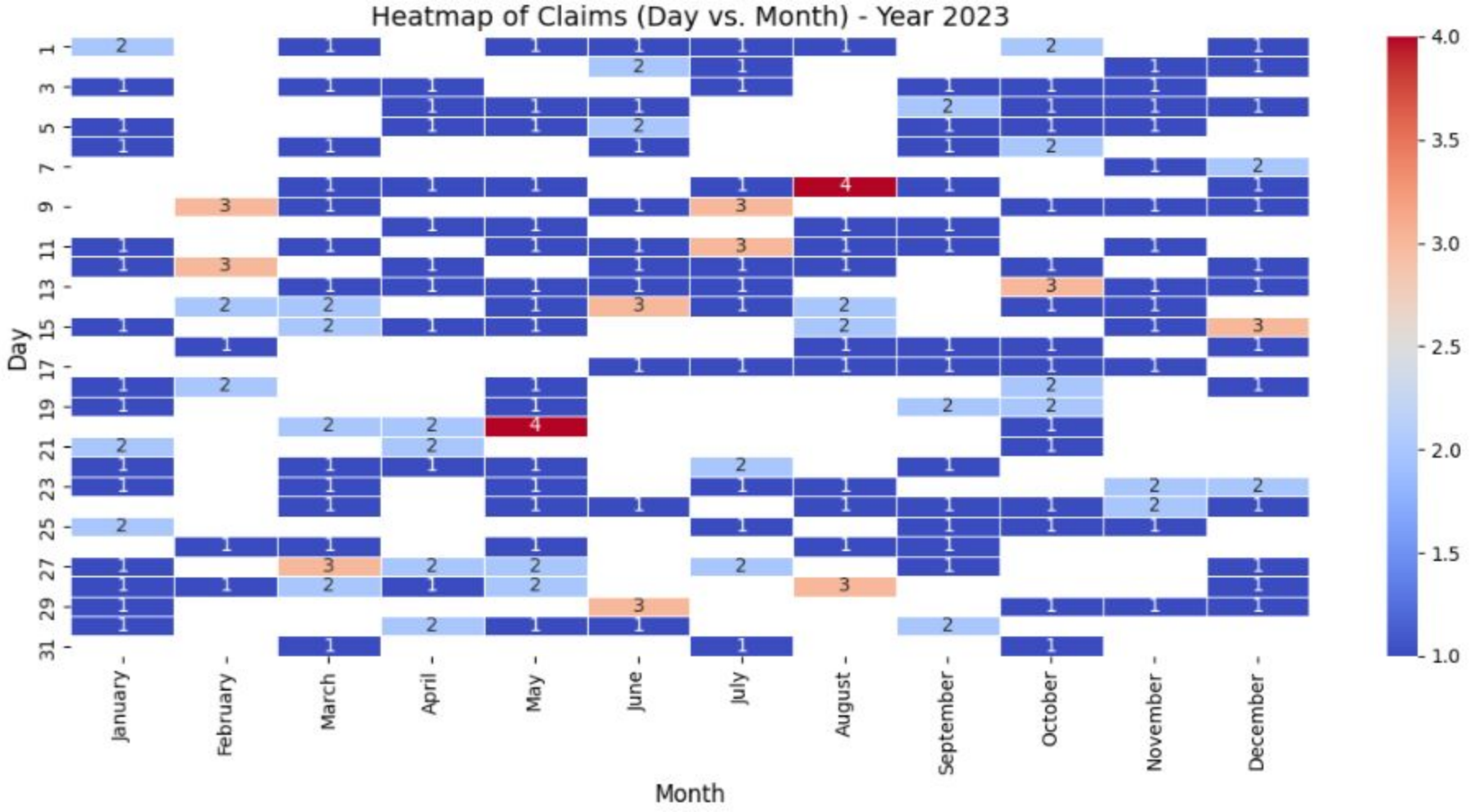
In 2020 highest claim per day is 3 , March, May, August , October and December
Highest number of Claim Month in this Year



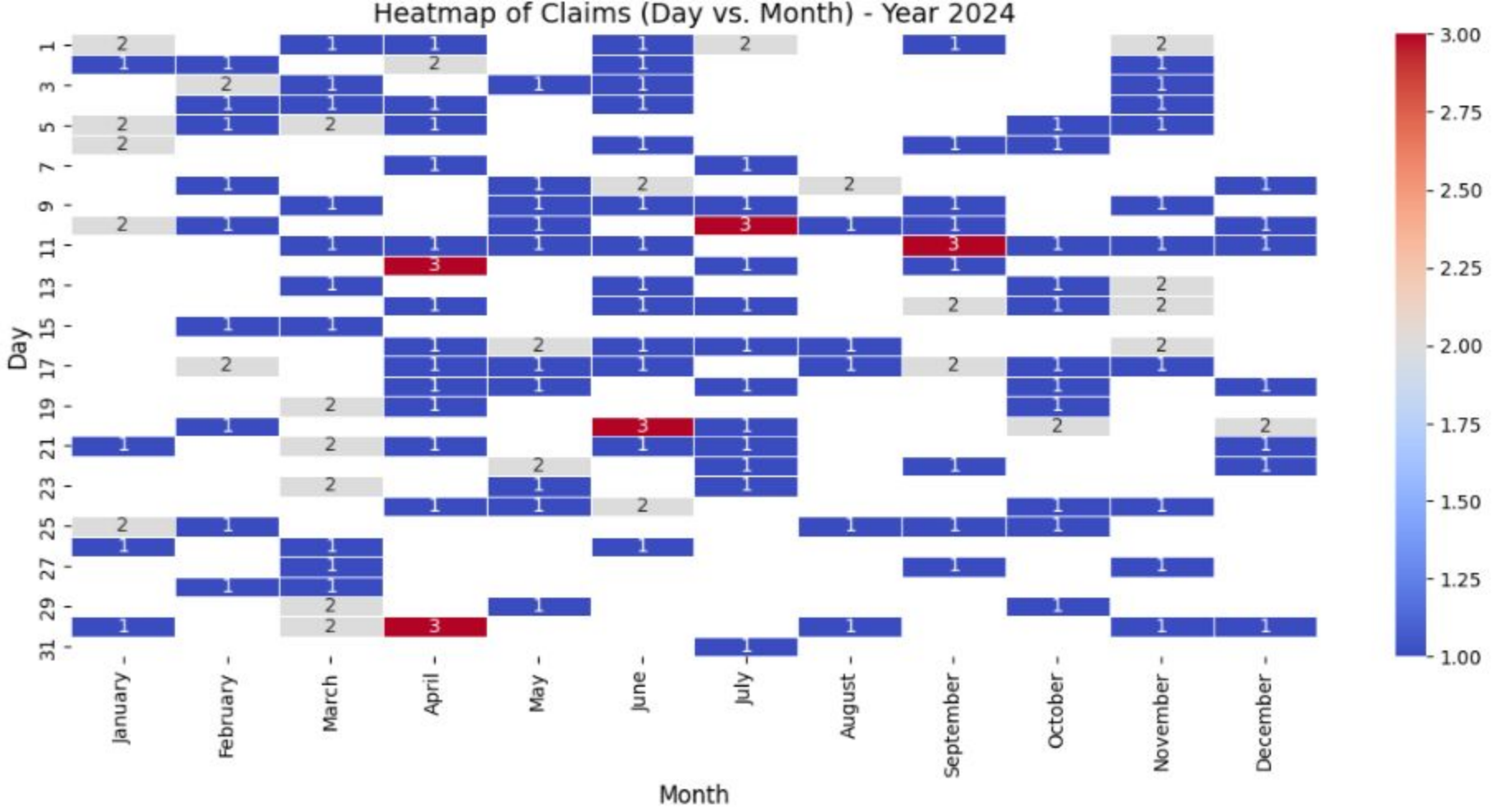
In 2021 - January, August, are Highest Claim



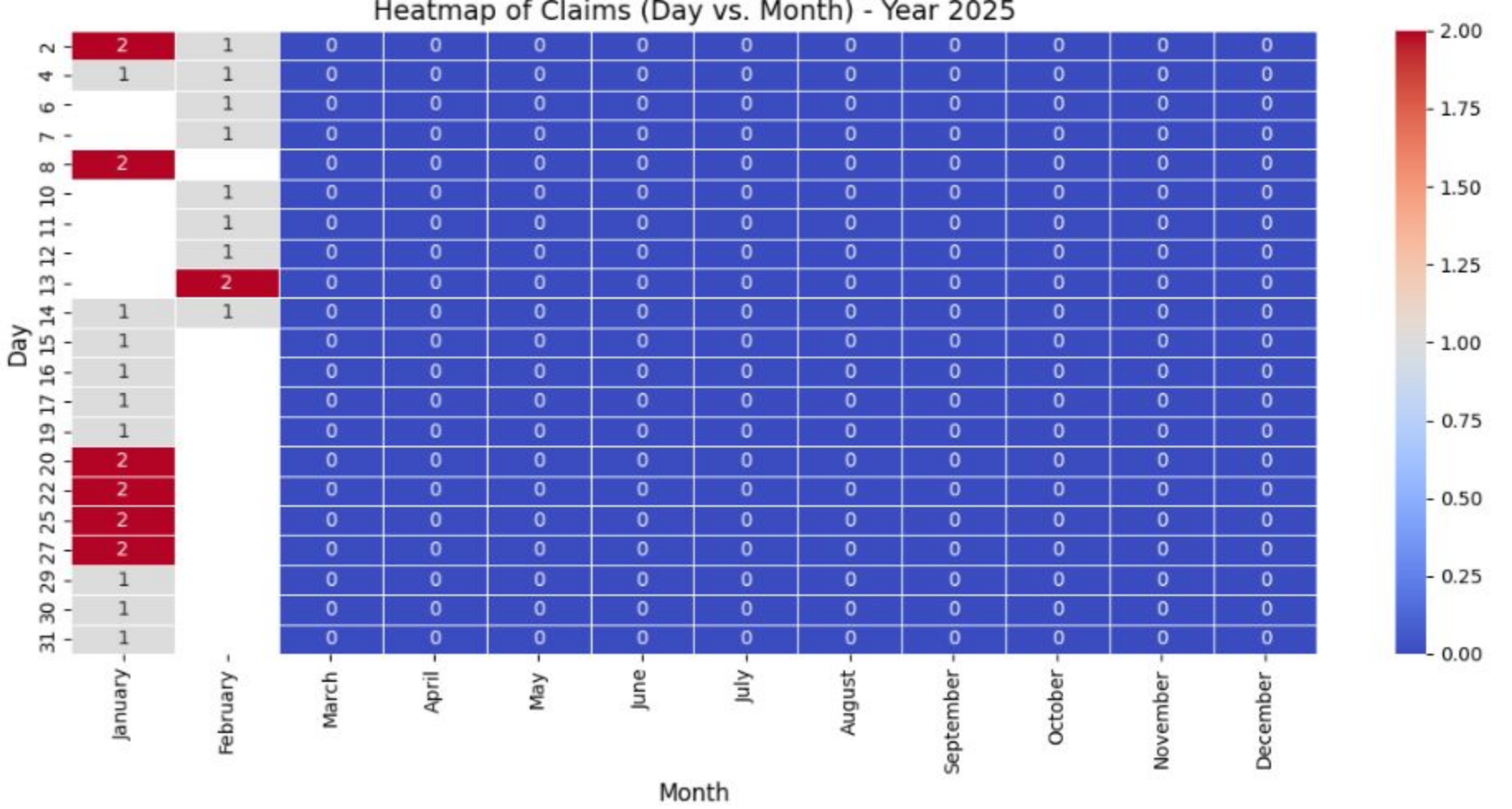
In 2022 August, October, November, December Highest number of Claims



In 2023 Highest number of Claims was Happen - August 9th Day 4 Claims



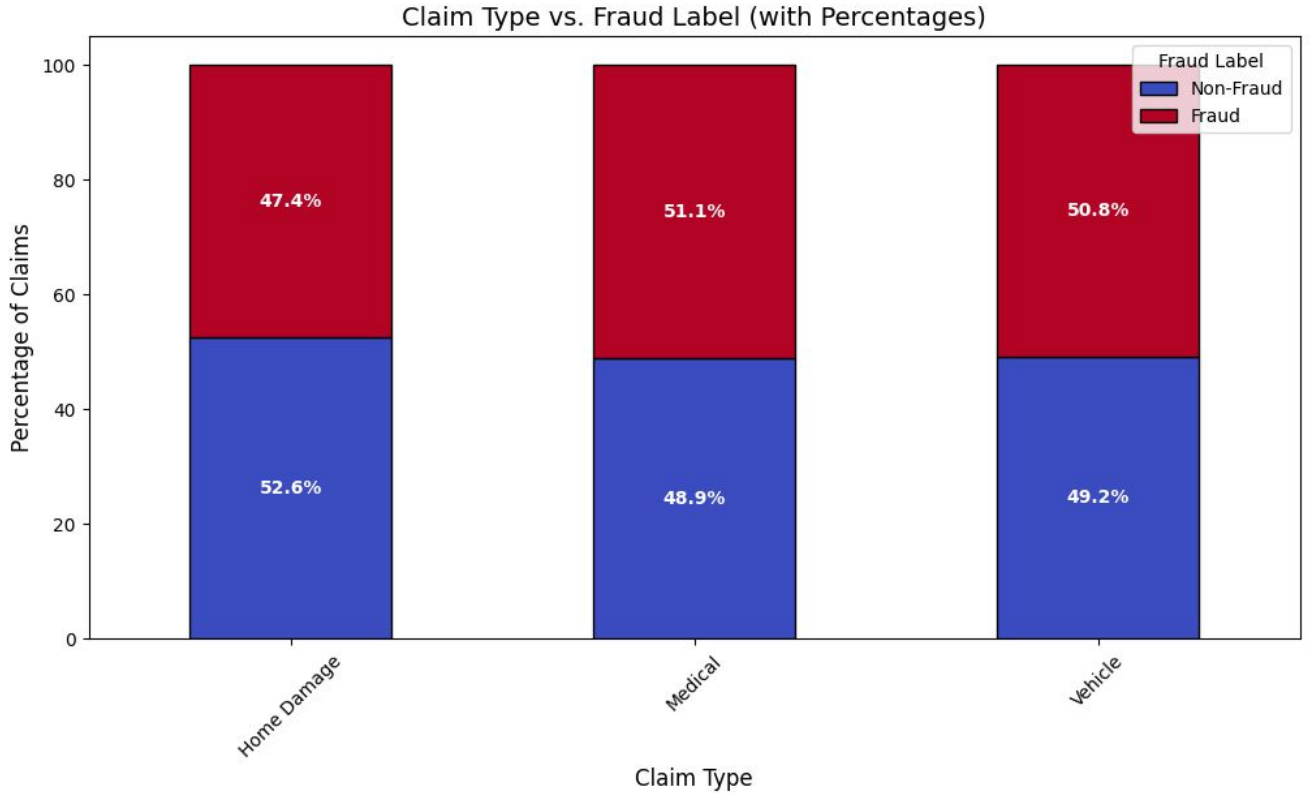
In 2024 has Moderate Claims



Report Taken as Feb-19-2025, January Month Face highest Number of Claims

1) Claim_Type vs. Fraud_Label

Stacked Bar Chart: To check which claim types are associated with fraud.



Insights from the Claim Type vs. Fraud Label Chart:

Dominance of Non-Fraudulent Claims:

For all three claim types **(Home Damage, Medical, and Vehicle)**, **the number of non-fraudulent claims significantly outweighs the number of fraudulent claims**. This is expected in most insurance scenarios.

Relatively Consistent Fraud Proportion:

While the absolute numbers vary, the proportion of fraudulent claims seems relatively consistent across the three claim types. **Medical claims might have a slightly higher proportion of fraud**, but further statistical testing would be needed to confirm this.

Potential for Further Investigation:

The chart highlights areas where further investigation might be warranted. For instance:

Medical Claims:

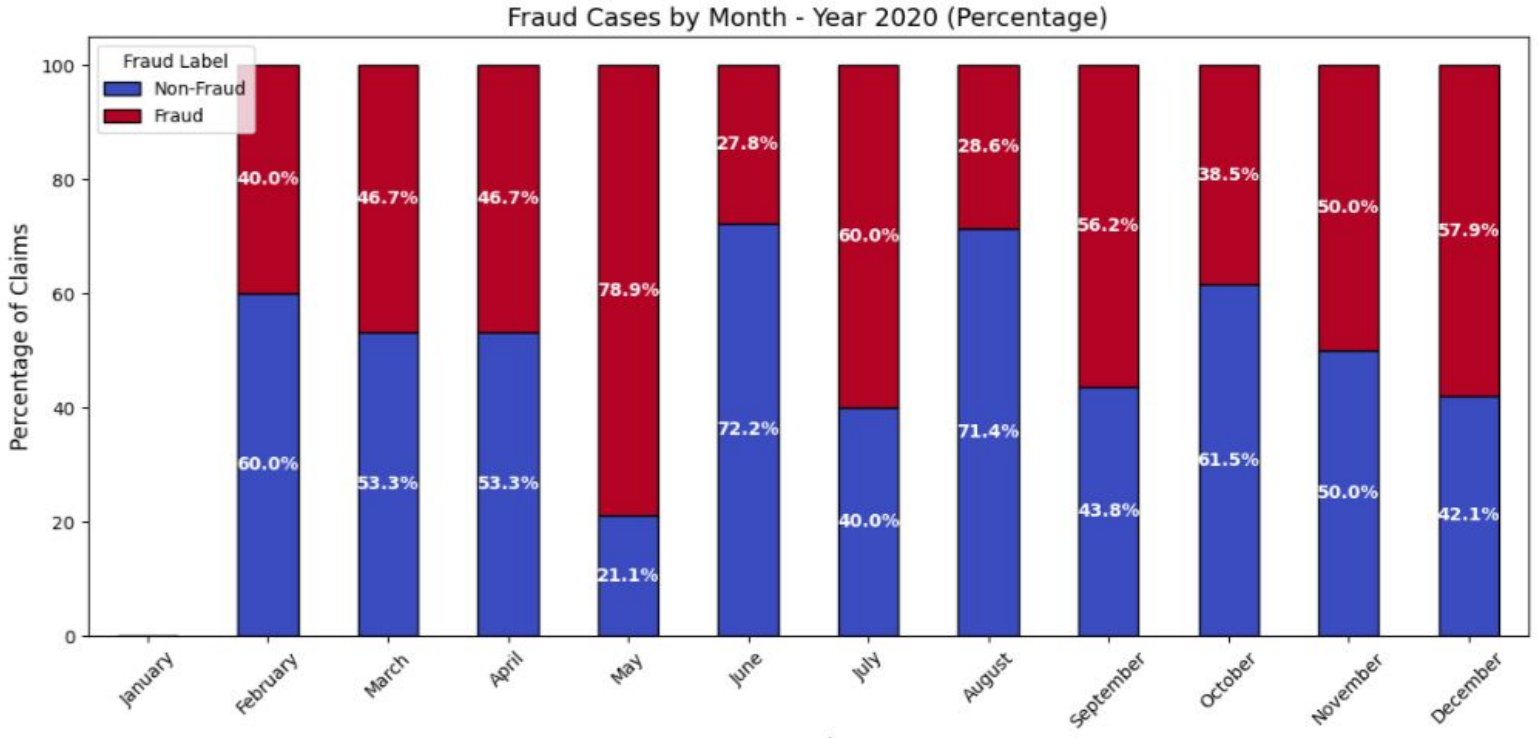
The slightly higher proportion of potential fraud in medical claims could be explored further to identify specific patterns or vulnerabilities.

High-Value Claims:

While not shown in this chart, analyzing fraud specifically within **high-value claims could be insightful**, as these would have a **greater financial impact.**

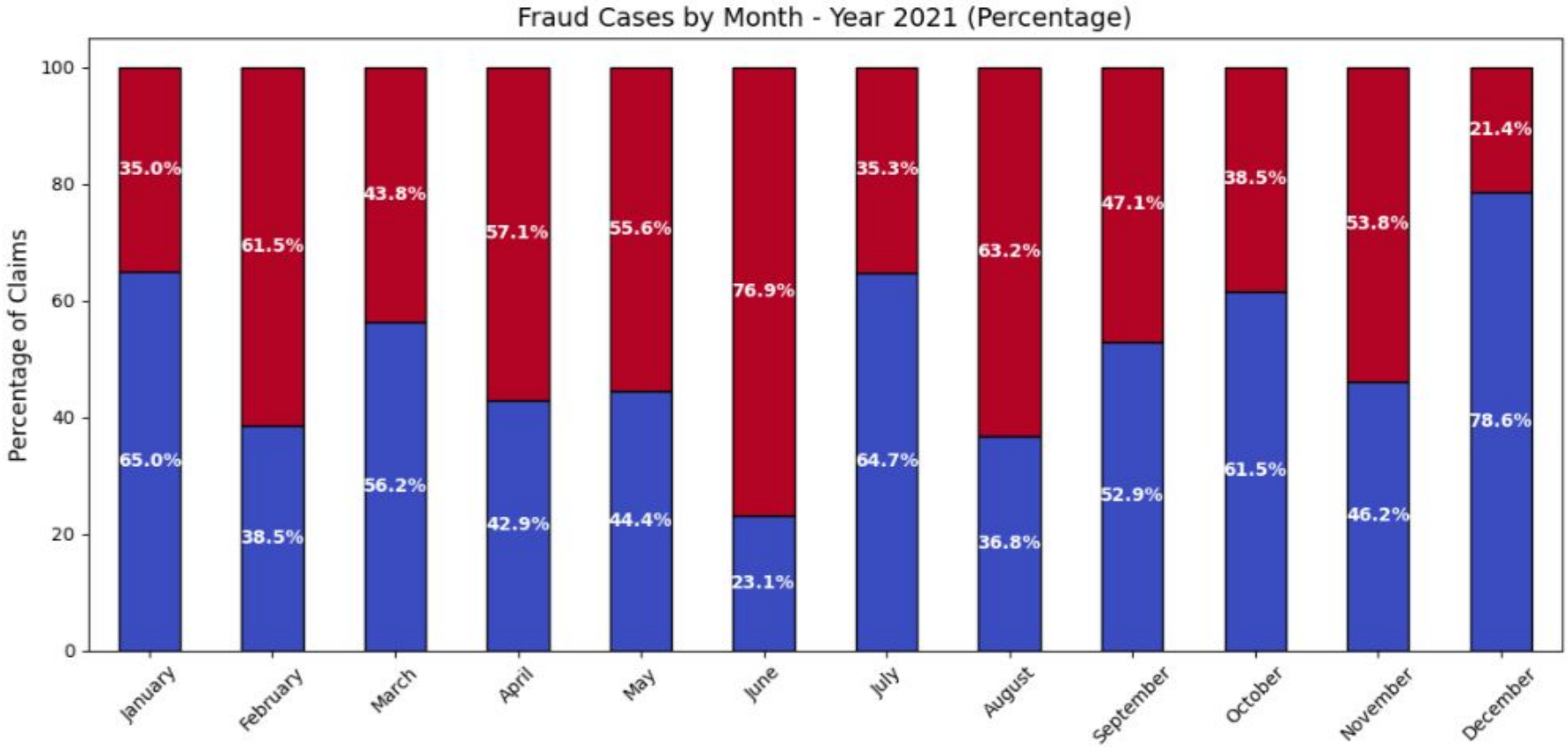
2) Year, Month_name vs. Fraud_Label

Bar Chart: To analyze fraud occurrences per month.



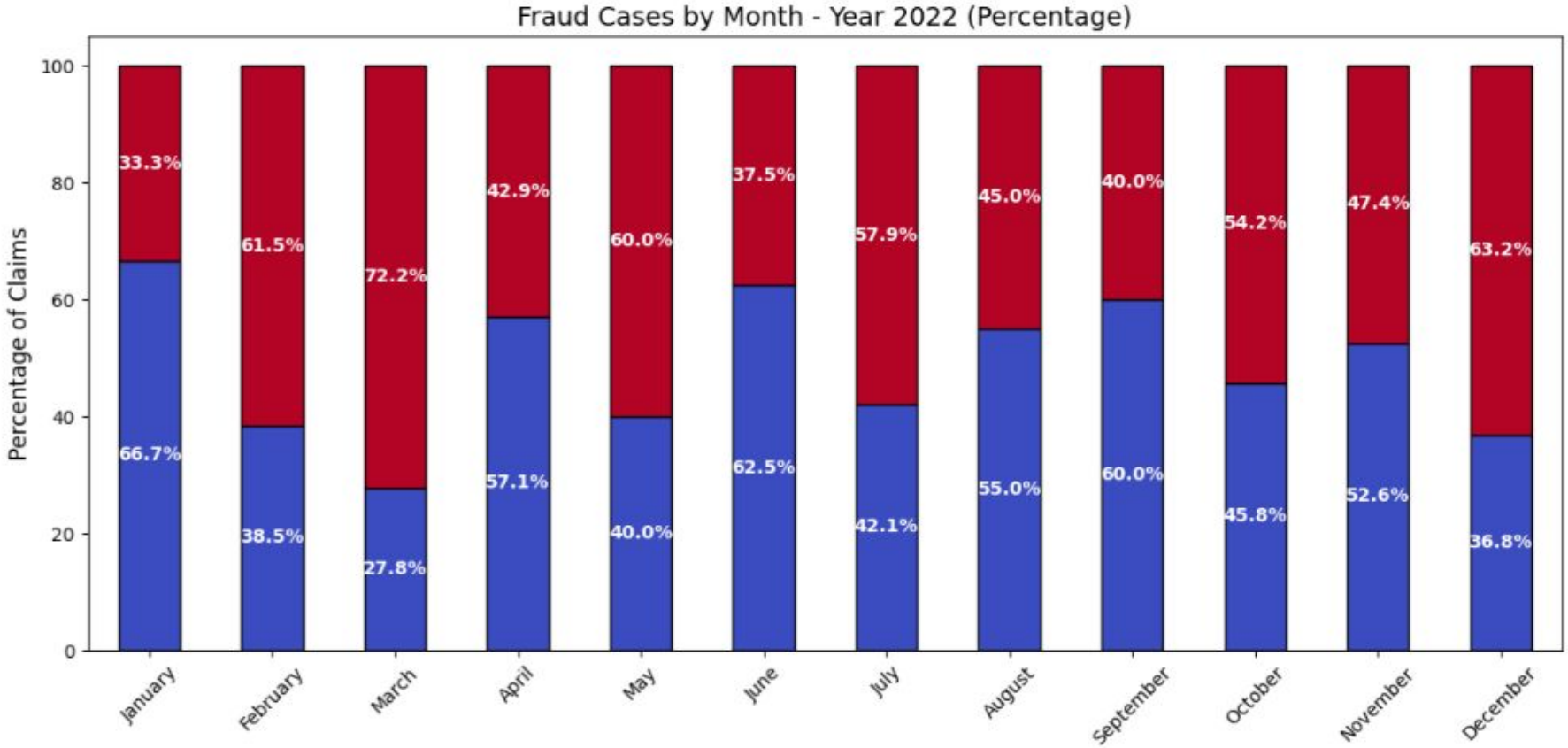
In 2020 More than 50% Fraud Happens on

1) May - 79% Fraud Claim
2) September - 56% Fraud Claim
3) November - 50% Fraud Claim
4) December - 58% Fraud Claim


Fraud Cases by Month - Year 2021 (Percentage)
Percentage of Claims
0
20
40
60
80
100
35.0%
65.0%
61.5%
38.5%
43.8%
56.2%
57.1%
42.9%
55.6%
44.4%
76.9%
23.1%
35.3%
64.7%
63.2%
36.8%
47.1%
52.9%
38.5%
61.5%
53.8%
46.2%
21.4%
78.6%
January
February
March
April
May
June
July
August
September
October
November
December

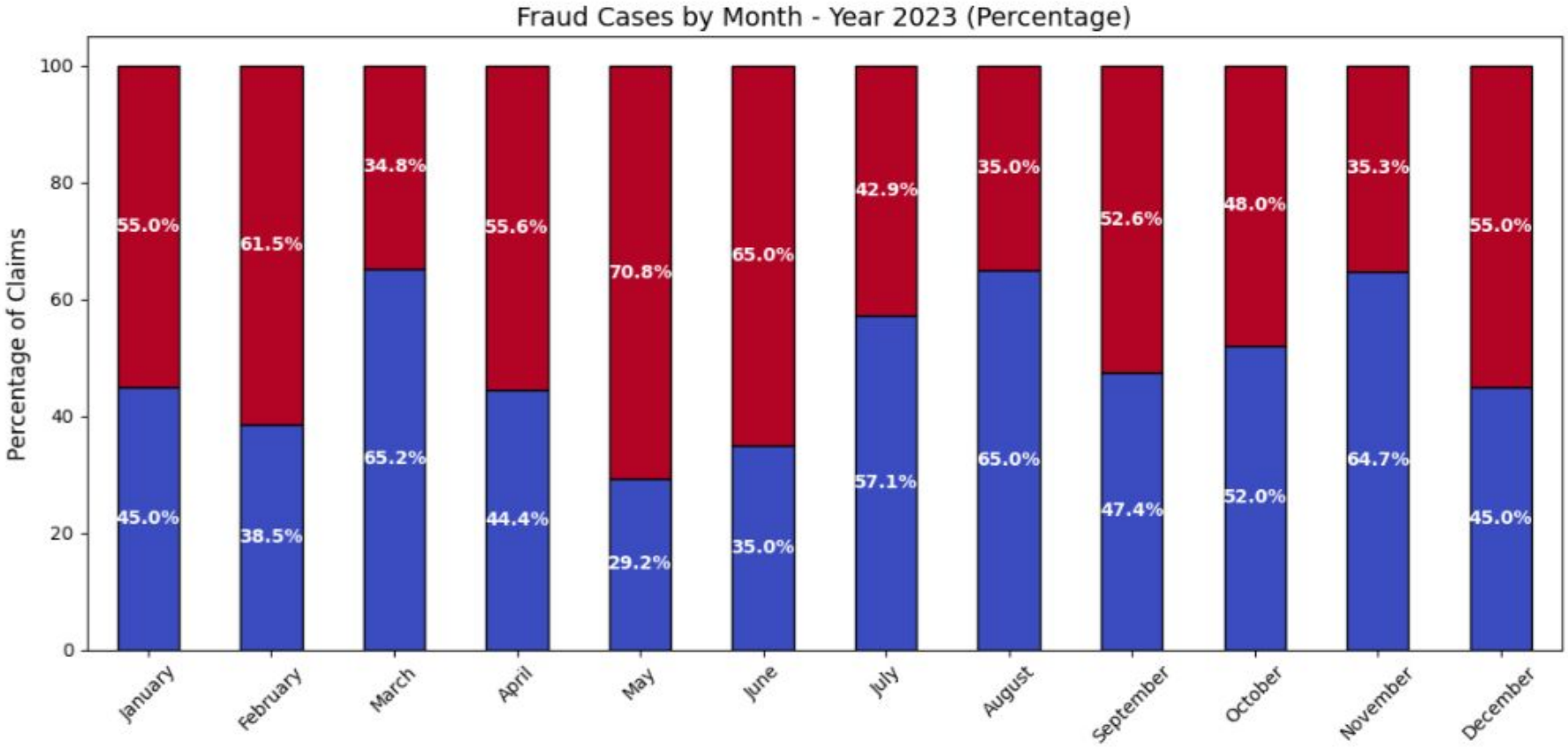
In 2021 More than 50% Fraud Claims on

1) February - 62% Fraud Claims
2) April - 51% Fraud Claims
3) May - 55% Fraud Claims
4) June - 77% Fraud Claims
5) August - 63% Fraud Claims
6) November - 54% Fraud Claims


Fraud Cases by Month - Year 2022 (Percentage)
Percentage of Claims
0
20
40
60
80
100
33.3%
66.7%
61.5%
38.5%
72.2%
27.8%
42.9%
57.1%
60.0%
40.0%
37.5%
62.5%
57.9%
42.1%
45.0%
55.0%
40.0%
60.0%
54.2%
45.8%
47.4%
52.6%
63.2%
36.8%
January
February
March
April
May
June
July
August
September
October
November
December

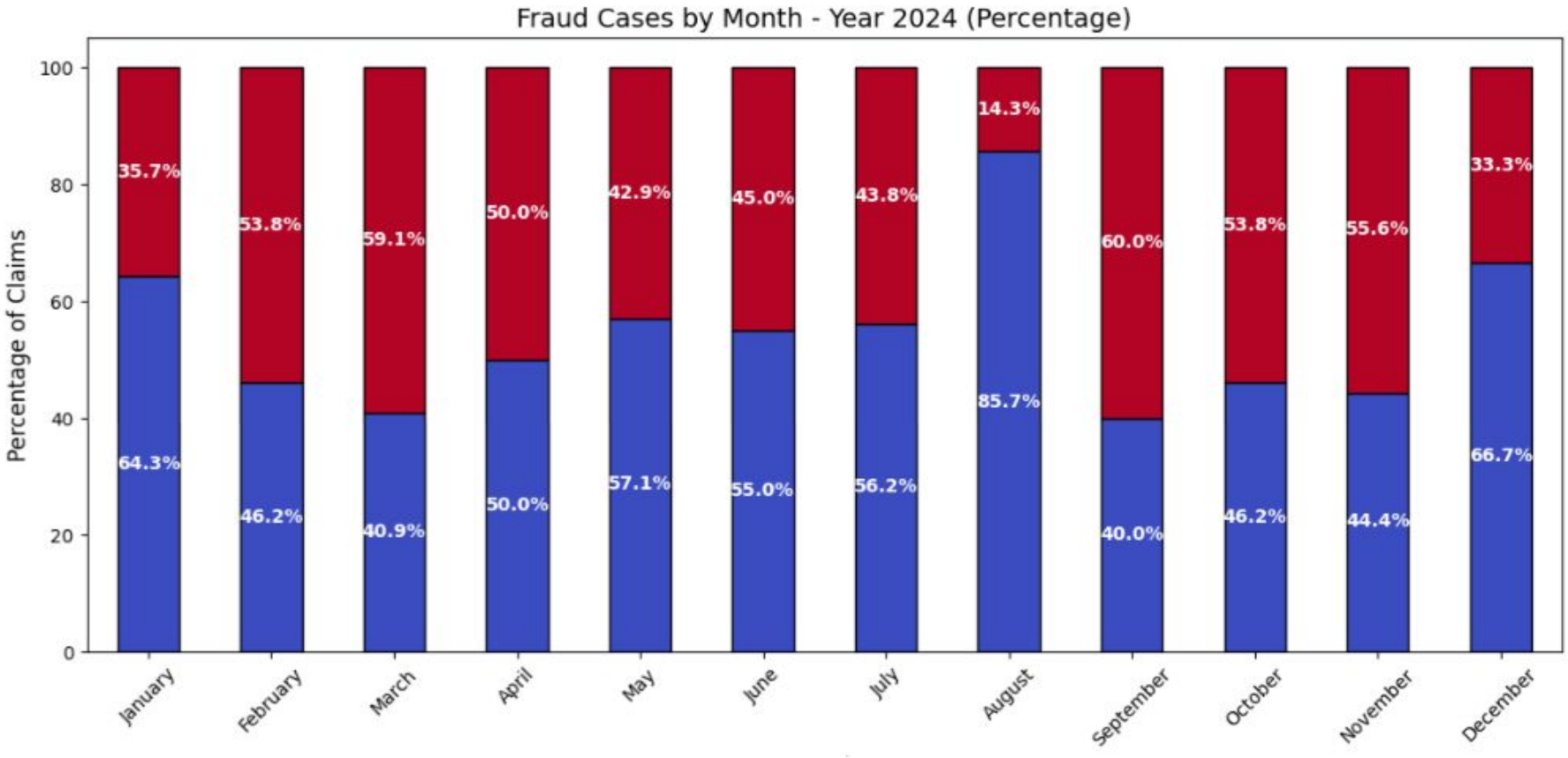
In 2022 More than 50% Fraud Claims on

1) February - 61% Fraud Claims
2) March - 72% Fraud Claims
3) May - 60% Fraud Claims
4) July - 58% Fraud Claims
5) October - 54% Fraud Claims
6) December - 63% Fraud Claims


Fraud Cases by Month - Year 2023 (Percentage)
Percentage of Claims
0
20
40
60
80
100
55.0%
45.0%
61.5%
38.5%
34.8%
65.2%
55.6%
44.4%
70.8%
29.2%
65.0%
35.0%
42.9%
57.1%
35.0%
65.0%
52.6%
47.4%
48.0%
52.0%
35.3%
64.7%
55.0%
45.0%
January
February
March
April
May
June
July
August
September
October
November
December

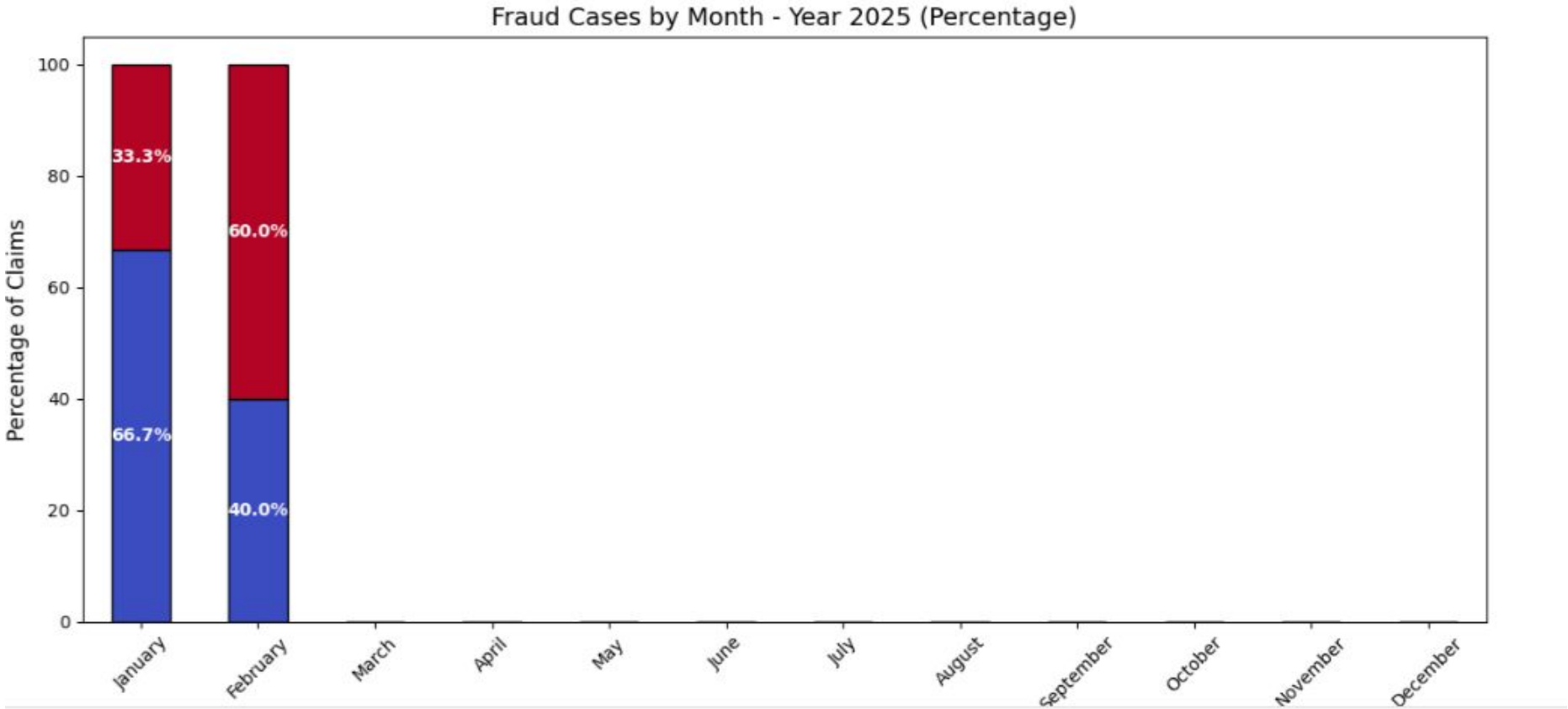
In 2023 More than 50% Fraud Claims on

1) January - 55%
2) February - 61%
3) April - 55%
4) May - 70%
5) June - 65%
6) September - 52%
7) December -55%


Fraud Cases by Month - Year 2024 (Percentage)
Percentage of Claims
0
20
40
60
80
100
35.7%
64.3%
53.8%
46.2%
59.1%
40.9%
50.0%
50.0%
42.9%
57.1%
45.0%
55.0%
43.8%
56.2%
14.3%
85.7%
60.0%
40.0%
53.8%
46.2%
55.6%
44.4%
33.3%
66.7%
January
February
March
April
May
June
July
August
September
October
November
December

In 2024 More than 50% Fraud Claims on

1) February - 53%
2) March - 59%
3) April - 50%
4) September - 60%
5) October - 54%
6) November - 55%


Fraud Cases by Month - Year 2025 (Percentage)
Percentage of Claims
100
80
60
40
20
0
33.3%
66.7%
60.0%
40.0%
January
February
March
April
May
June
July
August
September
October
November
December

In 2025 Currently 2 Months Data was taken

February Month - 60% Fraud Claim Happens

How to Detect Frauds ?

1) Highest Number of Claims
2) When Insurance Warrant Finish they Claim any artificial Reason
3) Claim Reason is inappropriate
4) Low Salary Persons Pay Premium Amount
5) Highest Scam Happens on - Medical & Home Damage

3) Claim_Amount vs. Fraud_Label

Box Plot: To check if fraudulent claims have different amount distributions.

Fraudulent claims tend to be for larger amounts of money compared to legitimate claims.

The amounts of fraudulent claims are also more unpredictable and vary more widely.

While claim amount can be a clue, it shouldn't be the only factor used to detect fraud, as there's some overlap with legitimate claim amounts.

Risk Assessment:

These findings can inform risk assessment strategies. **Claims for larger amounts might warrant closer** scrutiny for potential fraud.

Fraud Detection Models:

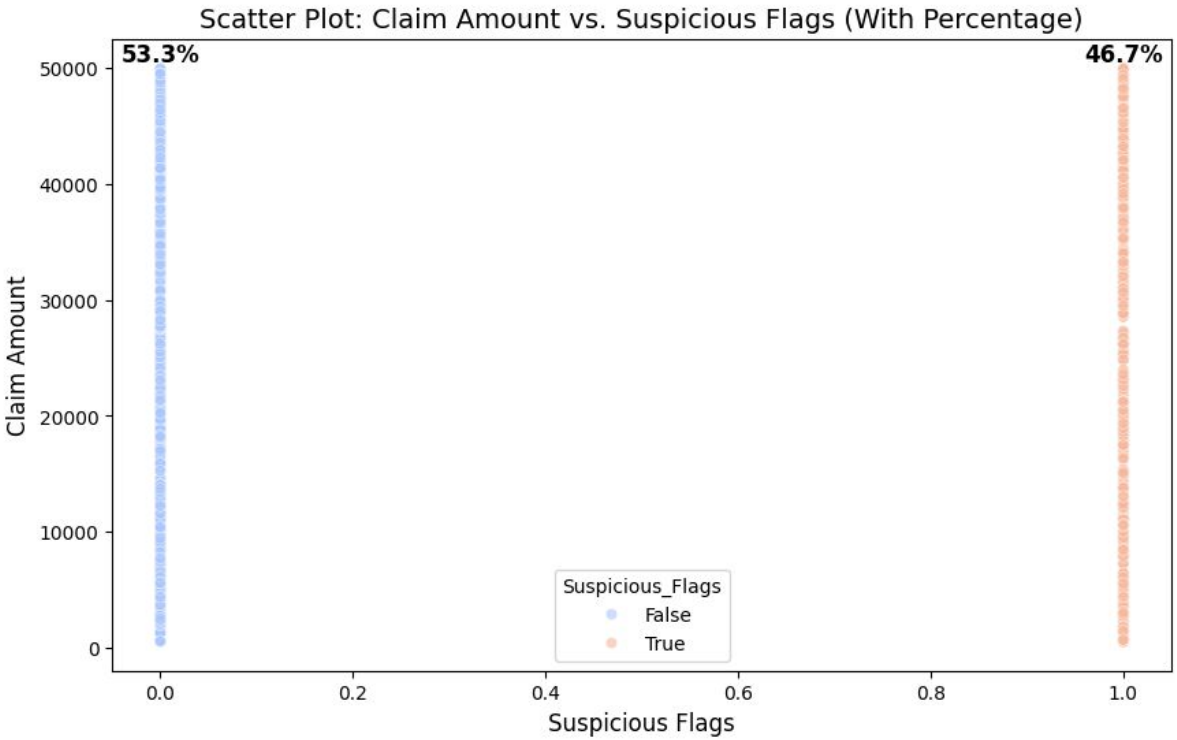
Claim amount could be a useful feature **in fraud detection models, especially when combined with other relevant variables**.

Resource Allocation:

Investigating potentially fraudulent claims, particularly those for larger amounts, might be a more efficient use of resources.

4) Claim_Amount vs. Suspicious_Flags

Scatter Plot: To analyze correlation between flagged claims and amount.



True (1) -> Fraud

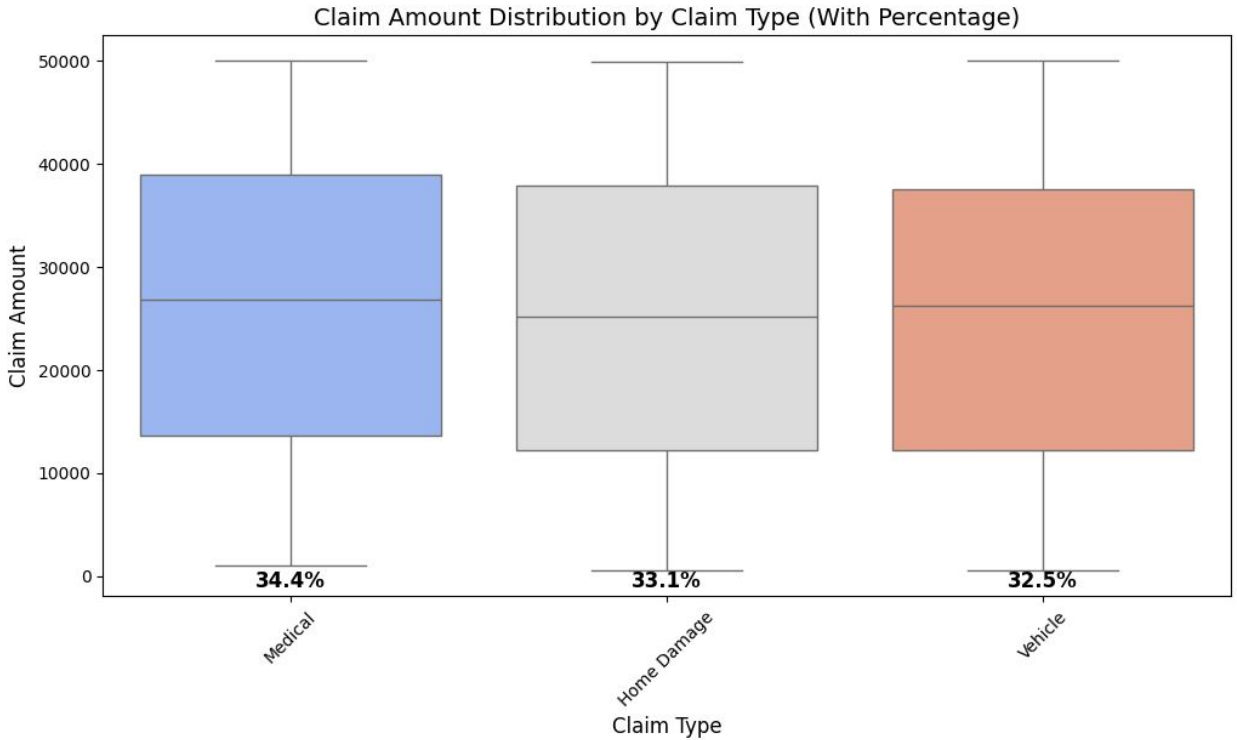
False (0) -> Genuine

High claim amounts correlate with suspicious flags: Larger claims are more likely to have suspicious flags.

Low claim amounts mostly have no flags: Smaller claims typically have no suspicious flags.

5) Claim_Amount vs. Claim_Type

Box Plot: To check if claim type influences the amount.



Medical claims show the highest Claim amount.

Home damage claims have the Medium range of amounts.

Vehicle claim amounts are the Low Claim consistent.

SQL

Basic Queries

1️⃣ Retrieve all claim details

2️⃣ Find the total number of claims

3️⃣ Find the number of fraud vs. non-fraud claims

Claim Amount Analysis

4️⃣ Find the average claim amount per claim type

5️⃣ Find the highest and lowest claim amounts

6️⃣ Find claims with an amount greater than $50,000

Fraud Detection & Suspicious Flags

7️⃣ Find the total number of flagged claims

8️⃣ Find the most common claim type for fraudulent claims

9️⃣ Find the percentage of fraud claims per claim type

Time-Based Analysis

🔟 Find the number of claims per year

1️⃣1️⃣ Find the month with the highest number of claims

1️⃣2️⃣ Find the number of fraud claims per year

Policyholder Insights

1️⃣3️⃣ Find the total number of unique policyholders

1️⃣4️⃣ Find the policyholder with the highest total claim amount

```
[ ]
```

```python
import sqlite3
import pandas as pd

# File name of the uploaded CSV
csv_filename = "/content/df3-date-modified.csv"  # Replace this with the uploaded file name

# Load CSV into a pandas DataFrame
df = pd.read_csv(csv_filename)

# Connect to SQLite database (or create a new one)
conn = sqlite3.connect("example.db")
cursor = conn.cursor()

# Write DataFrame to SQLite table
table_name = "df3"  # Specify your table name
df.to_sql(table_name, conn, if_exists="replace", index=False)

print(f"Table '{table_name}' created in SQLite database.")

```

```
Table 'df3' created in SQLite database.
```

1) Retrieve all claim details

```
query = f"SELECT * from df3;"
result = pd.read_sql_query(query, conn)

# Display the results
result
```

	Unnamed: 0	Claim_ID	Policyholder_ID	Claim_Amount	Claim_Type	Suspicious_Flags	Fraud_Label	Year	Month	Day	Month_name
0	0	bdd640fb-0667-4ad1-9c80-317fa3b1799d	1a3d1fa7-bc89-40a9-a3b8-c1e9392456de	32151.63	Medical	0	0	2021	7	4	July
1	1	8b9d2434-e465-4150-bd9c-66b3ad3c2d6d	17fc695a-07a0-4a6e-8822-e8f36c031199	11548.93	Home Damage	1	0	2020	7	25	July
2	2	9a1de644-815e-46d1-bb8f-aa1837f8a88b	b38a088c-a65e-4389-b74d-0fb132e70629	29729.38	Medical	1	0	2020	4	6	April
3	3	72ff5d2a-386e-4be0-ab65-a6a48b8148f6	c241330b-01a9-471f-9e8a-774bcf36d58b	11322.58	Home Damage	1	0	2023	1	29	January
4	4	6c307511-b2b9-437a-a8df-6ec4ce4a2bbd	c37459ee-f50b-4a63-b71e-cd7b27cd8130	35942.97	Home Damage	0	0	2021	10	31	October

2) Find the total number of claims

```
[ ]  1 query = f"SELECT count(Claim_ID) from df3"
     2 result = pd.read_sql_query(query, conn)
     3
     4 # Display the results
     5 result
```

	count(Claim_ID)
0	1000

3) Find the number of fraud vs. non-fraud claims

```
[ ]  1 query = f"SELECT DISTINCT Fraud_Label,count(Fraud_Label) from df3 GROUP BY Fraud_Label"
     2 result = pd.read_sql_query(query, conn)
     3
     4 # Display the results
     5 result
```

	Fraud_Label	count(Fraud_Label)
0	0	503
1	1	497

Fraud_Label -> (0 - Genuine, 1 - Fraud)

Claim Amount Analysis

4) Find the average claim amount per claim type

```
query="Select Claim_Type,avg(Claim_Amount) as AVG_CLAIM from df3 GROUP BY Claim_Type ORDER by AVG_CLAIM DESC"
result=pd.read_sql_query(query,conn)

result
```

	Claim_Type	AVG_CLAIM
0	Medical	26190.093354
1	Vehicle	25351.762447
2	Home Damage	25105.985930

Medical as a Highest Average Claim Amount

Vehicle as Medium Average Claim

Home Damage is Least Average Claim

5) Find the highest and lowest claim amounts

```
[ ]  1 query="Select max(Claim_Amount) as Highest_Claim_Amount,min(Claim_Amount) as Lowest_Claim_Amount from df3"
     2
     3 result=pd.read_sql_query(query,conn)
     4 result
```

	Highest_Claim_Amount	Lowest_Claim_Amount
0	49998.84	533.46

highest and lowest claim amounts based on Claim Type

```
Select Claim_Type,max(Claim_Amount) as
Highest_Claim_Amount,min(Claim_Amount) as Lowest_Claim_Amount from
df3 GROUP BY Claim_Type;
```

	Claim_Type	Highest_Claim_Amount	Lowest_Claim_Amount
0	Home Damage	49864.83	533.83
1	Medical	49998.84	1064.08
2	Vehicle	49968.99	533.46

6) Find claims with an amount greater than $25,000

```
query="Select * From df3 where Claim_Amount > 25000;"

result=pd.read_sql_query(query,conn)
result
```

	Unnamed: 0	Claim_ID	Policyholder_ID	Claim_Amount	Claim_Type	Suspicious_Flags	Fraud_Label	Year	Month	Day	Month_name
0	0	bdd640fb-0667-4ad1-9c80-317fa3b1799d	1a3d1fa7-bc89-40a9-a3b8-c1e9392456de	32151.63	Medical	0	0	2021	7	4	July
1	2	9a1de644-815e-46d1-bb8f-aa1837f8a88b	b38a088c-a65e-4389-b74d-0fb132e70629	29729.38	Medical	1	0	2020	4	6	April
2	4	6c307511-b2b9-437a-a8df-6ec4ce4a2bbd	c37459ee-f50b-4a63-b71e-cd7b27cd8130	35942.97	Home Damage	0	0	2021	10	31	October
3	6	759cde66-bacf-43d0-8b1f-9163ce9ff57f	142c3fe8-60e7-4113-ac1b-8ca1f91e1d4c	35057.90	Vehicle	0	0	2022	10	24	October
4	9	614ff3d7-19db-4ad0-9dd1-dfb23b982ef8	29a3b2e9-5d65-4441-9588-42dea2bc372f	40452.85	Home Damage	0	0	2021	7	9	July

Total 522 Claims Happen

Count of claims with an amount greater than $25,000 on Each Claim Types

```
query="Select Claim_Type,count(Claim_Amount) as Count_Claim_Amount From df3 WHERE Claim_Amount >= 25000 GROUP BY Claim_Type"

result=pd.read_sql_query(query,conn)
result
```

	Claim_Type	Count_Claim_Amount
0	Home Damage	173
1	Medical	174
2	Vehicle	175

Fraud Detection & Suspicious Flags

7) Find the total number of flagged claims

```
query="Select count(Suspicious_Flags) as Flagged_Claims From df3 where Suspicious_Flags=1;"

result=pd.read_sql_query(query,conn)
result
```

	Flagged_Claims
0	467

✅ True (1) → The claim is flagged as suspicious (potential fraud or anomaly detected).

❌ False (0) → The claim is not suspicious (no immediate fraud indicators).

8) Find the most common claim type for fraudulent claims

Genuine = 1 & Fraud = 0

```
Select Claim_Type,Count(Fraud_Label) as Fraudlent_Claims From df3
where Fraud_Label=1 GROUP BY Claim_Type ORDER BY Fraudlent_Claims
DESC;
```

	Claim_Type	Fraudlent_Claims
0	Vehicle	168
1	Medical	166
2	Home Damage	163

Vehicle Claim Type Most Fraudlent Happens

9) Find the percentage of fraud claims per claim type

```
Select Claim_Type,count(CASE when Fraud_Label=1 Then 1
END)*100/count(*) as Fraud_Percentage From df3 Group By Claim_Type
ORDER BY Fraud_Percentage DESC;
```

	Claim_Type	Fraud_Percentage
0	Medical	51
1	Vehicle	50
2	Home Damage	47

Time-Based Analysis

10) Find the number of claims per year

```
query="Select Year,count(Claim_ID) as Claims From df3 GROUP BY Year"

result=pd.read_sql_query(query,conn)
result
```

	Year	Claims
0	2020	168
1	2021	187
2	2022	193
3	2023	240
4	2024	181
5	2025	31

11) Find the number of Fraud Claims per Year

0 = Genuine, 1 = Fraud

```
query="Select Year,count(Claim_ID) as Claims From df3 where Fraud_Label=1 GROUP BY Year"

result=pd.read_sql_query(query,conn)
result
```

	Year	Claims
0	2020	82
1	2021	91
2	2022	102
3	2023	122
4	2024	87
5	2025	13

2023 has highest Fraud Claims

12) Find the month with the highest number of claims

```
Select month_name,Year,count(Claim_Amount) as Total_claims,sum(Claim_Amount) as Sum_Claims From df3 GROUP BY month_name order by Total_claims DESC
```

	Month_name	Year	Total_claims	Sum_Claims
0	March	2023	94	2268040.56
1	May	2021	90	2476327.28
2	October	2021	88	2157146.90
3	July	2021	88	2268850.09
4	August	2020	87	2317908.86
5	January	2023	84	1922369.47

Policyholder Insights

13) Find the total number of unique policyholders

```
[20]  1 query="Select DISTINCT Policyholder_ID,count(Policyholder_ID)  From df3 Group By Policyholder_ID"  #Policyholder_ID
      2
      3 result=pd.read_sql_query(query,conn)
      4 result
```

	Policyholder_ID	count(Policyholder_ID)
0	004b6fab-fcf5-4188-932e-6dcd83bc9478	1
1	004ea81a-d3d8-47a8-8a0e-a57a9a7d509d	1
2	00a81de9-d20f-47d0-8465-6d6b81fb18b3	1
3	00a85473-8242-4912-bda3-c1623a78b64a	1
4	00c59f30-d103-4424-8124-1bb4655ce508	1

14) Find the policyholder with the highest total claim amount

```
Select Policyholder_ID,Count(Claim_Amount),max(Claim_Amount) as Max_Claims,sum(Claim_Amount) as Total_Amount From df3 Group By Policyholder_ID ORDER BY Total_Amount DESC
```

	Policyholder_ID	Count(Claim_Amount)	Max_Claims	Total_Amount
0	4588fc1b-b238-4849-8fe0-69b6eedaa802	1	49998.84	49998.84
1	f54e2019-ba35-444e-99e1-ac095970a859	1	49968.99	49968.99
2	243012da-b185-4eb8-9f90-fc872edc017e	1	49907.08	49907.08
3	5820b3f2-5989-4d6c-8d24-34f9963dc8bf	1	49886.28	49886.28
4	99546eb4-0025-4ad1-ab22-63dd87c5421e	1	49867.63	49867.63

Preprocessing

No Missing Values Found

If Missing Values have this Datasets.

Step 1:

check % of Missing Values if more than 70% Missing Values -> Delete that Columns

Step 2:

Check this Columns have Normal Distribution and No Outliers - Use Mean Imputation

Else use Median Imputation

Step 3:

Check that Columns Have Categorical Columns Use Mode

Step 4:

If That Dataset have Times series Use Forward fill or Backward Fill

Step 5:

If that Missing Values was Voluntarily Happen Based On Confidential else any other Situation

Client or SME or BA Give us Custom values fill us Cutom method to fill Missing Values

2) Outliers

No Outlier Found in this Datasets

If Outlier Found in this Datasets:-

Step 1: Understand the Outliers

Check Data Entry Errors: Typos or incorrect values? Fix them if possible.

Assess Context: Are these extreme values realistic or expected in your data domain?

Step 2: Choose an Outlier Treatment Approach:

1) Removal Methods:

Delete Outliers: If they're definitely errors or irrelevant.

```
df_clean = df1[~((df1['A'] < lower_bound) | (df1['A'] > upper_bound))]
```

2) Imputation Methods:

Replace with Mean/Median: Works well when outliers aren't extreme.

median = df1['A'].median() df1['A'] = np.where((df1['A'] < lower_bound) | (df1['A'] > upper_bound), median, df1['A'])

Use Mode: For categorical data with outliers

3) Transformation Methods:

Log Transformation: Reduces effect of right-skewed outliers.

df1['A_log'] = np.log1p(df1['A'])

Square Root or Box-Cox: For data with different types of skew.

4) Capping or Clipping

Winsorization: Limit extreme values to percentiles.

from scipy.stats.mstats import winsorize df1['A_winsorized'] = winsorize(df1['A'], limits=[0.05, 0.05]) # Caps at 5th & 95th percentile

Cap at Boundaries: Replace outliers with IQR bounds.

df1['A'] = np.clip(df1['A'], lower_bound, upper_bound)

3) Feature Engineering

(i) Generate Claim-to-Income Ratio (Claim_Amount / Annual_Income) as a new feature.

```
# Generating Claim-to-Income Ratio as a new feature
df3['Claim_to_Income_Ratio'] = df3['Claim_Amount'] / df3['Annual_Income']
```

What does it represent?

The `Claim_to_Income_Ratio` shows how large the insurance claim is compared to the person's income. For example:

- If `Claim_Amount` = $10,000 and `Annual_Income` = $50,000 → Ratio = 0.2 (20% of their annual income)
- Higher ratios might indicate more suspicious or risky claims.

This feature is often useful for fraud detection because unusually high claim amounts relative to income can be a red flag

Claim-to-Income Ratio Columns Created

(ii) Identify claims filed within an unusually short period after policy issuance.

create extra columns for

1) Policy Issues Date vs

2) claim date

3) difference of policy issues date vs claim data find (short period time True or False type)

eg: claims happens before of 365 days b/w of policy isses and claim date its True otherwise its false

```python
import pandas as pd
import numpy as np

# Randomly generating Policy Issuance Date with some realistic variation
df3['Policy_Issuance_Date'] = pd.to_datetime(df3['Claim_Date']) - pd.to_timedelta(np.random.rand

# Convert 'Claim_Date' to datetime objects before subtraction
df3['Claim_Date'] = pd.to_datetime(df3['Claim_Date'])

# Calculating days between policy issuance and claim date
df3['Days_Since_Issuance'] = (df3['Claim_Date'] - df3['Policy_Issuance_Date']).dt.days

# Flagging unusually short periods (e.g., within 30 days)
df3['Short_Period_Claim'] = df3['Days_Since_Issuance'] <= 365
```

Short_Period_Claim Consider as a Scam otherwise Genuine

```
df3['Short_Period_Claim'].value_counts()
```

Short_Period_Claim	count
False	651
True	349

dtype: int64

```
df3['Days_Since_Issuance'].value_counts()
```

Days_Since_Issuance	count
444	5
348	5
866	5
802	4
276	4
...	...
275	1
785	1
281	1
485	1
851	1

629 rows × 1 columns

Anomaly Detection

✅ Outlier Removal & Anomaly Detection

- Use **Elliptic Envelope, Isolation Forest, and Local Outlier Factor (LOF)** to tag suspicious claims.

Which One is Best?

- For **large, high-dimensional datasets**: **Isolation Forest** — scalable and flexible, well-suited for fraud detection.
- For **normally distributed data**: **Elliptic Envelope** — simple and efficient when assumptions hold.
- For **small or clustered datasets**: **Local Outlier Factor (LOF)** — captures local density changes well.

In your case (insurance fraud detection):

Since your insurance fraud dataset likely has high-dimensional, complex data patterns — **Isolation Forest** would usually be the best choice. It's scalable and captures outliers without strong distributional assumptions.

I am trying those three

In Model Training i am using Isolation Forest

Reason 1: My Dataset is Not a Normal Distribution
Frauds 20% and 80% Genuine

Reason 2: I am Choosing Synthetic Dataset High Complexity Issues

```
# Selecting relevant numerical columns for anomaly detection
X = df3[['Claim_Amount', 'Claim_Year', 'Claim_Month', 'Claim_Day', 'Claim_to_Income_Ratio', 'Days_Since_Issuance']]
```

Using Based on these Columns to find Anomalies Detection

```
from sklearn.covariance import EllipticEnvelope
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
```

- `EllipticEnvelope` → from `sklearn.covariance`
- `IsolationForest` → from `sklearn.ensemble`
- `LocalOutlierFactor (LOF)` → from `sklearn.neighbors`

1. `sklearn.covariance`
    - Deals with **covariance estimation**, which measures how much two variables change together. It's used to understand the structure of your data and detect outliers.
    - Key models:
        - `EllipticEnvelope`: Fits a multivariate Gaussian distribution and identifies outliers.
        - `EmpiricalCovariance`: Estimates the covariance matrix directly from the data.
        - `MinCovDet` **(Minimum Covariance Determinant)**: A robust method for estimating covariance, less sensitive to outliers.
    - **When to use**: When you assume your data has a Gaussian distribution and you want to identify outliers or understand data relationships.

2. `sklearn.ensemble`
    - Implements **ensemble methods**, which combine multiple models to improve prediction accuracy and robustness.
    - Key models:
        - `RandomForestClassifier` / `RandomForestRegressor`: A collection of decision trees to improve accuracy and reduce overfitting. (You're already using this!)
        - `IsolationForest`: Anomaly detection method that isolates outliers through random partitioning.
        - `GradientBoosting`, `AdaBoost`, `Bagging`: Other methods combining weak learners to create strong models.
    - **When to use**: When you want strong predictive performance, especially when your data is complex or high-dimensional.

3. `sklearn.neighbors`
    - Implements **nearest neighbor algorithms**, which classify or detect anomalies based on proximity to other data points.
    - Key models:
        - `KNeighborsClassifier` / `KNeighborsRegressor`: Classifies or predicts based on the majority class/average value of nearby points.
        - `LocalOutlierFactor (LOF)`: Detects outliers by measuring local density deviations from neighbors.
        - `NearestNeighbors`: Finds the nearest points without making predictions — useful for clustering and anomaly detection.
    - **When to use**: When local data relationships matter, like clustering or identifying anomalies in dense regions.

```python
# Elliptic Envelope
elliptic = EllipticEnvelope(contamination=0.20)
df3['Elliptic_Anomaly'] = elliptic.fit_predict(X_scaled)

# Isolation Forest
iso_forest = IsolationForest(contamination=0.20, random_state=42)
df3['Isolation_Anomaly'] = iso_forest.fit_predict(X_scaled)

# Local Outlier Factor (LOF)
lof = LocalOutlierFactor(n_neighbors=20, contamination=0.20)
df3['LOF_Anomaly'] = lof.fit_predict(X_scaled)

# Mapping results for better readability
anomaly_map = {-1: 'Anomaly', 1: 'Normal'}
df3['Elliptic_Anomaly'] = df3['Elliptic_Anomaly'].map(anomaly_map)
df3['Isolation_Anomaly'] = df3['Isolation_Anomaly'].map(anomaly_map)
df3['LOF_Anomaly'] = df3['LOF_Anomaly'].map(anomaly_map)
```

Using Contamination rate as 0.20 - 20% based on Fraud_Label

My Dataset have Labeled Column (Target)

```
[ ]
1 import pandas as pd
2
3 # Assuming df is your DataFrame
4 contamination_rate = df3['Fraud_Label'].mean()  # Assuming 1 is fraud and 0 is not
5 print(f"Contamination rate: {contamination_rate:.2f}")
6
```

```
Contamination rate: 0.20
```

Contamination rate: 0.20

Incase My Dataset have UnLabeled (No Target Column)

If No Labeled column in my dataset only use that time

```
[ ]
1 from scipy.stats import zscore
2
3 numeric_cols = ['Claim_Amount', 'Claim_to_Income_Ratio', 'Days_Since_Issuance']
4 z_scores = np.abs(zscore(df3[numeric_cols]))
5
6 outliers = (z_scores > 3).any(axis=1)  # Threshold of 3 std deviations
7 contamination_rate = np.mean(outliers)
8 print(f"Estimated contamination rate: {contamination_rate:.2f}")
9
```

```
1 df3['LOF_Anomaly'].value_counts()
```

LOF_Anomaly	count
Normal	800
Anomaly	200

dtype: int64

Method	Works Best For	Assumptions	Strength	Weakness
EllipticEnvelope	Normally distributed data	Gaussian shape	Fast & simple	Struggles with non-normal data
IsolationForest	High-dimensional data	None	Scalable, works on any distribution	Can overflag outliers
LOF	Local density variations	Points close but different from neighbors	Captures local outliers	Sensitive to parameter choice

Fraud Score Calculation

- Assign a **fraud probability score (0-1) using ensemble models (Random Forest + Neural Networks)**.

```
import pandas as pd
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.ensemble import IsolationForest, RandomForestClassifier
from sklearn.neighbors import LocalOutlierFactor
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import roc_auc_score
```

3. `scikit-learn` (`sklearn`):
    - Provides machine learning models and utilities. We use:
        - `EllipticEnvelope` (from `covariance`): For anomaly detection using Gaussian distribution (not actually used here).
        - `IsolationForest` (from `ensemble`): Tree-based anomaly detection (also not used here).
        - `RandomForestClassifier`: A tree-based classification model (used here).
        - `LocalOutlierFactor` (from `neighbors`): Density-based anomaly detection (also not used here).
        - `StandardScaler` (from `preprocessing`): Standardizes data by scaling to zero mean and unit variance.
        - `train_test_split` (from `model_selection`): Splits the data into training and test sets.
        - `MLPClassifier` (from `neural_network`): A multi-layer perceptron (feedforward neural network).

What the code does:

1. **Feature selection (`X`):**
   - Picks numerical features for fraud detection:
     - `Claim_Amount`, `Claim_Year`, `Claim_Month`, `Claim_Day`
     - `Claim_to_Income_Ratio`, `Days_Since_Issuance`
2. **Standardizing the data (`StandardScaler`):**
   - Scales all features to have **mean = 0** and **standard deviation = 1**.
   - Important for models like neural networks that are sensitive to feature scale.

```
# Selecting relevant numerical columns for anomaly detection
X = df3[['Claim_Amount', 'Claim_Year', 'Claim_Month', 'Claim_Day', 'Claim_to_Income_Ratio', 'Days_Since_Issuance']]

# Standardizing the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

3. **Splitting the data (`train_test_split`)**:
    - Divides the dataset into:
        - `X_train`, `X_test`: Scaled features.
        - `y_train`, `y_test`: Labels (`Fraud_Label` — whether the claim is fraud or not).
        - `train_index`, `test_index`: Row indices, useful for mapping predictions back to the original DataFrame.
4. **Random Forest Classifier (`rf_model`)**:
    - A bagging ensemble method combining multiple decision trees.
    - `n_estimators=100`: Uses 100 trees.
    - `random_state=42`: Ensures results are reproducible.
    - **Fits on training data** and **predicts probabilities** of fraud for the test set (`rf_probs`).

```
# Splitting the data
X_train, X_test, y_train, y_test, train_index, test_index = train_test_split(X_scaled, df3['Fraud_Label'],
                                                                             df3.index, test_size=0.2, random_state=42)

# Random Forest Model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
rf_probs = rf_model.predict_proba(X_test)[:, 1]
```

5. **Neural Network Model (`nn_model`)**:
    - A **Multi-Layer Perceptron (MLP)** with 2 hidden layers:
        - **64 neurons → 32 neurons**
    - **ReLU activation**: Rectified Linear Unit, a common non-linearity.
    - `max_iter=500`: Trains up to 500 epochs.
    - **Fits on training data** and **predicts probabilities** (`nn_probs`).

```
# Neural Network Model
nn_model = MLPClassifier(hidden_layer_sizes=(64, 32), activation='relu', max_iter=500, random_state=42)
nn_model.fit(X_train, y_train)
nn_probs = nn_model.predict_proba(X_test)[:, 1]
```

6. **Averaging predictions**:

   - Combines the predictions from Random Forest and Neural Network:

$$\text{Fraud Probability Score} = \frac{\text{rf\_probs} + \text{nn\_probs}}{2}$$

   - Ensemble methods often improve performance by reducing individual model weaknesses.

```
# Averaging predictions to get an ensemble fraud probability score
df3.loc[test_index, 'Fraud_Probability_Score'] = (rf_probs + nn_probs) / 2

# Filling NaN values with 0 for rows not in the test set
```

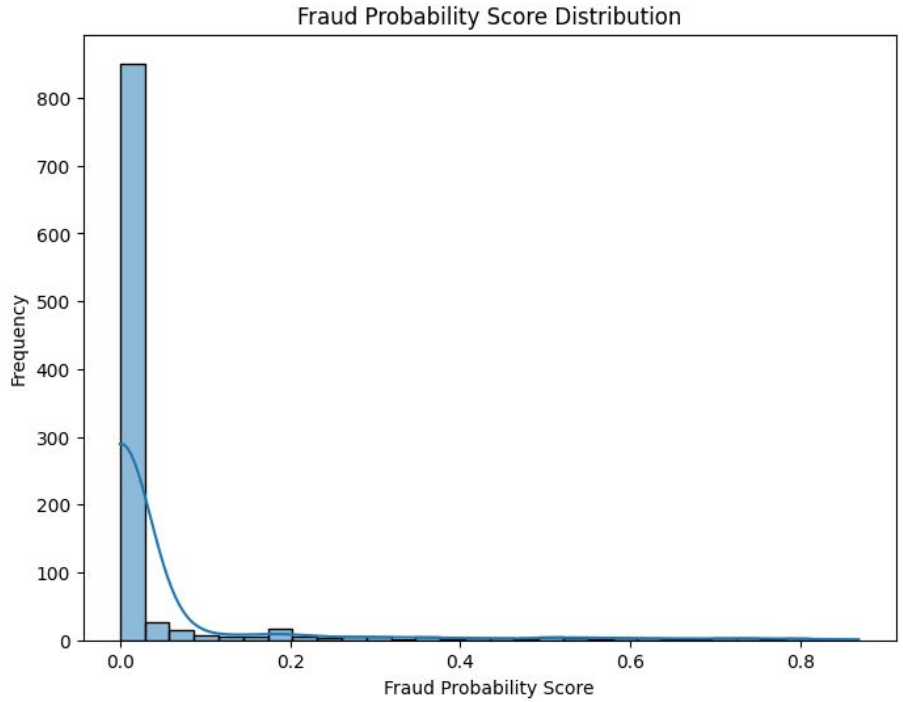
7. **Handling missing values**:

   - Rows not in the test set get `NaN`, so it fills those with `0` for a complete column.

```
# Filling NaN values with 0 for rows not in the test set
df3['Fraud_Probability_Score'].fillna(0, inplace=True)
```

8. **Visualizing Fraud Probability Distribution**:

- Uses a **Seaborn histogram with KDE (Kernel Density Estimate)**.
- Shows how fraud probability scores are distributed across claims.



Overall goal of the code:

- Trains **two models (Random Forest + Neural Network)** for fraud detection.
- Averages their predicted fraud probabilities to create a **Fraud Probability Score**.
- Visualizes the distribution of this fraud risk score across the dataset.

```
1 df3['Fraud_Probability_Score']
```

	Fraud_Probability_Score
0	0.000000
1	0.000000
2	0.000000
3	0.000000
4	0.000000
...	...
995	0.148099
996	0.000000
997	0.000000
998	0.614191
999	0.000000

```
1 df3['Fraud_Probability_Score'].value_counts()
```

Fraud_Probability_Score	count
0.000000	800
0.727842	1
0.513845	1
0.135467	1
0.106245	1
...	...
0.107329	1
0.361100	1
0.197159	1
0.175828	1
0.195443	1

Whatis Ensemble ?

a machine learning technique that combines multiple models to make better predictions

Combine models: Multiple models, such as neural networks or regression models, are combined to create an ensemble model.

Improve predictions: The ensemble model produces more accurate predictions than a single model.

Learn from mistakes: Some techniques, like boosting, learn from previous mistakes to improve predictions.

1st - Attempt

Feature Selection

df3 - Model Training

Fraudulent Claims Dataset

```
[ ]  1 ########################################################################
```

This dataset is specifically used to detect fraudulent claims using anomaly detection and classification models.

Extract - Year, Month & Day From Date

Extract Dates From Policy Issues Date

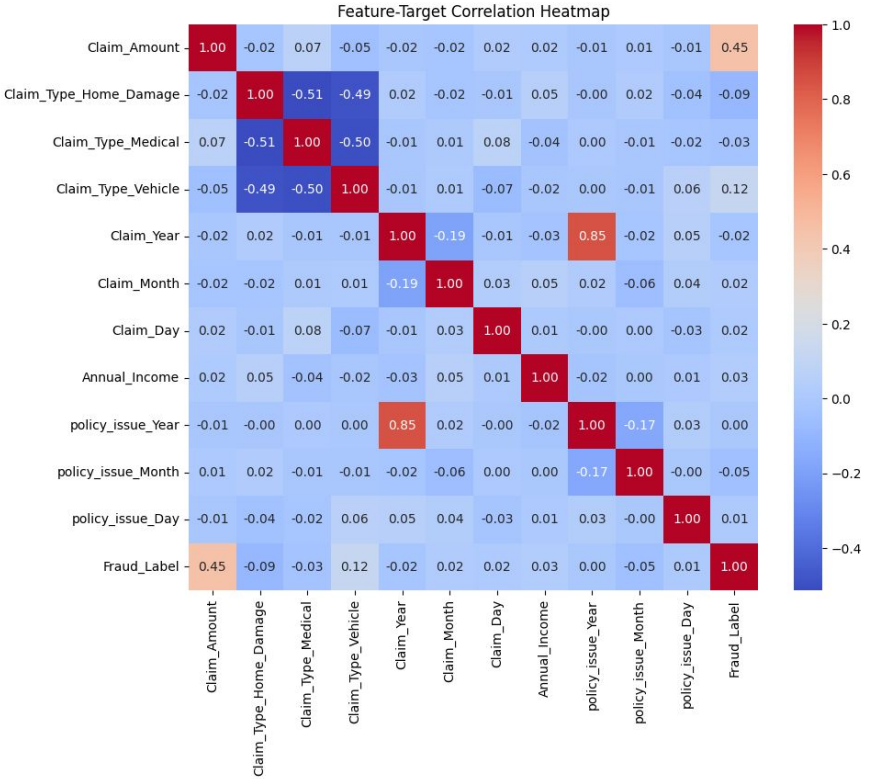
```
[ ]
1 # Extract date parts from Claim_Date
2 df3['Policy_Issuance_Date'] = pd.to_datetime(df3['Policy_Issuance_Date'], errors='coerce')
3 df3['policy_issue_Year'] = df3['Policy_Issuance_Date'].dt.year
4 df3['policy_issue_Month'] = df3['Policy_Issuance_Date'].dt.month
5 df3['policy_issue_Day'] = df3['Policy_Issuance_Date'].dt.day
```

Important Columns for Detect fraudlent Claim - Insurance Domain

```
[ ]  1 feature=df3_feature[['Claim_Amount','Claim_Type_Home_Damage','Claim_Type_Medical','Claim_Type_Vehicle','Claim_Year',
     2                      'Claim_Month', 'Claim_Day', 'Annual_Income','policy_issue_Year','policy_issue_Month', 'policy_issue_Day']]
     3 target=df3_feature['Fraud_Label']
```

Filter Method Using Heatmap To Find to take feature Columns Based on Highly Correlate - Positive or Negative Correlated

No Correlated Column near to Zero Columns Avoid it



All Columns are Slightly Corelated both posative and negative

Policy Issues year and Claim year highly Posative Correlated

Fraud and Claim Amount Highly Correlated

```
[ ]    1 feature.columns
```

```
Index(['Claim_Amount', 'Claim_Type_Home_Damage', 'Claim_Type_Medical',
       'Claim_Type_Vehicle', 'Claim_Year', 'Claim_Month', 'Claim_Day',
       'Annual_Income', 'policy_issue_Year', 'policy_issue_Month',
       'policy_issue_Day'],
      dtype='object')
```

Feature Engineering Columns Uses:

1) Using Claim_Amount and Annual_Income -

```
Backend Work diffence of Claim Amount and Annual Income  to
[ Claim Income Ratio ]

Result:

High Claim Income Ratio is Geniune and Low Claim Income Ratio Suspected as Fraud

Eg:

Claim Amount Lower than 10% of Annual Income is Genuine Claim Almost but - Claim Amount is More than 25% or 30% is Almost a Fraud
```

2) Claim_Date and Policy_Issuence_Date

```
Policy Issue Date and Claim Date is Less than a year means. suspect as a Fraud more than a year means Genuine

Result:

calculate the difference of claim date and policy issues date - value greater than 365 is false suspecius
and value lesser than 365 is True Suspecius
```

1st - Attempt

Model Training

```
1 target.value_counts()
```

Fraud_Label	count
0	797
1	203

dtype: int64

Target have only 2 Unique - so its a Binary Classification Type

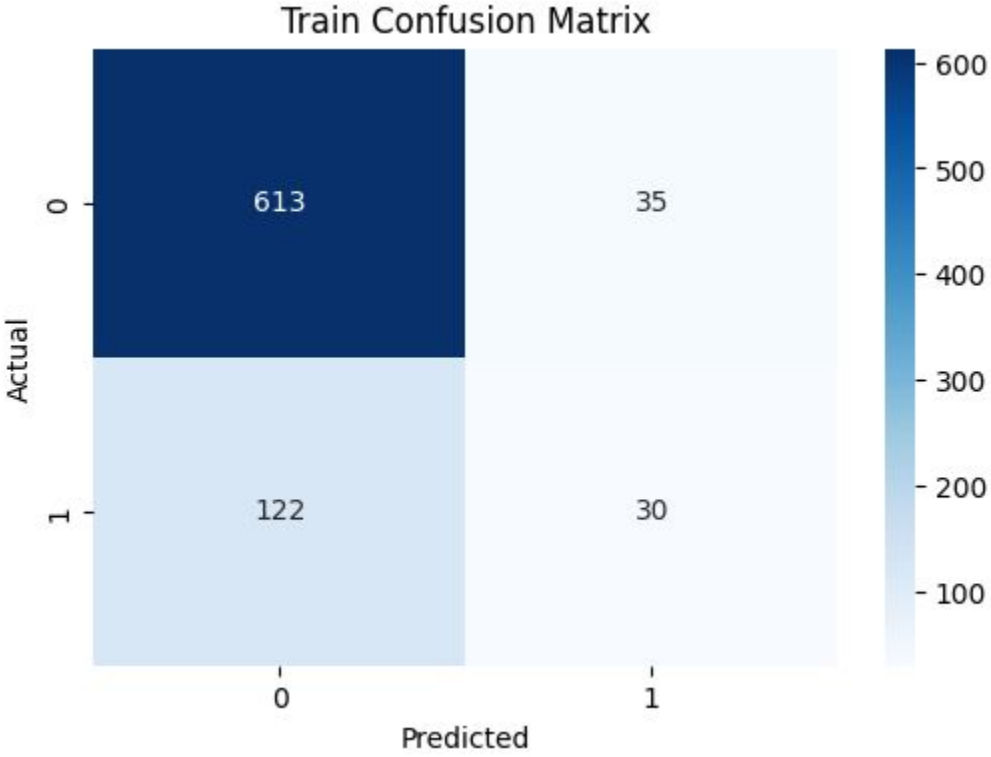
1) Try Logistic Regression

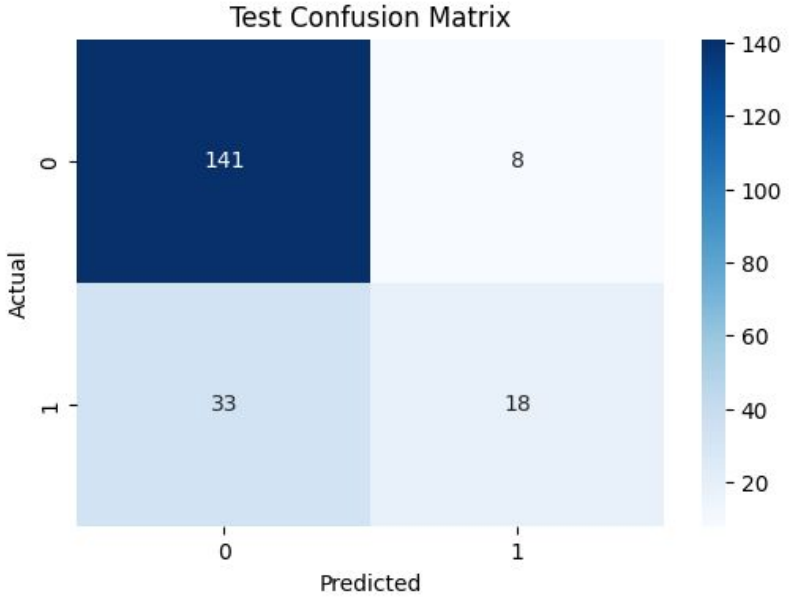
```
[ ]
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Logistic Regression model
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)

```

```python
# Predictions
y_train_pred = log_reg.predict(X_train)
y_test_pred = log_reg.predict(X_test)

# Confusion matrices
train_cm = confusion_matrix(y_train, y_train_pred)
test_cm = confusion_matrix(y_test, y_test_pred)

# Function to plot confusion matrix
def plot_confusion_matrix(cm, title):
    plt.figure(figsize=(6, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(title)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

# Plot confusion matrices
plot_confusion_matrix(train_cm, 'Train Confusion Matrix')
plot_confusion_matrix(test_cm, 'Test Confusion Matrix')

# Classification reports
print("Train Classification Report:\n", classification_report(y_train, y_train_pred))
print("Test Classification Report:\n", classification_report(y_test, y_test_pred))
```

```
47
48 train_acc = accuracy_score(y_train, y_train_pred)
49 test_acc = accuracy_score(y_test, y_test_pred)
50
51
52 if abs(train_acc - test_acc) < 0.05 and test_acc > 0.75:
53     print("The model generalizes well!")
54 elif train_acc > test_acc + 0.1:
55     print("The model is overfitting.")
56 elif train_acc < 0.75 and test_acc < 0.75:
57     print("The model is underfitting.")
58 else:
59     print("The model's performance needs closer inspection.")
60
```


Train Confusion Matrix
613
35
122
30
0
1
Actual
Predicted
600
500
400
300
200
100


Test Confusion Matrix
141
8
33
18
0
1
Actual
Predicted
140
120
100
80
60
40
20

```
Train Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.95      0.89       648
           1       0.46      0.20      0.28       152

    accuracy                           0.80       800
   macro avg       0.65      0.57      0.58       800
weighted avg       0.76      0.80      0.77       800

Test Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.95      0.87       149
           1       0.69      0.35      0.47        51

    accuracy                           0.80       200
   macro avg       0.75      0.65      0.67       200
weighted avg       0.78      0.80      0.77       200

Train Accuracy: 0.80375
Test Accuracy: 0.795
The model generalizes well!
```

The model generalizes well!

But Need to Increase a Test Accuracy

```python
# Extract TP, FP, TN, FN from confusion matrix
tn, fp, fn, tp = confusion_matrix(y_test, y_test_pred).ravel()

print(f"True Positives: {tp}")
print(f"False Positives: {fp}")
print(f"True Negatives: {tn}")
print(f"False Negatives: {fn}")

precision = tp / (tp + fp) if (tp + fp) != 0 else 0
recall = tp / (tp + fn) if (tp + fn) != 0 else 0
f1_score = 2 * (precision * recall) / (precision + recall) if (precision + recall) != 0 else 0

print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1_score:.2f}")

```

```
True Positives: 18
False Positives: 8
True Negatives: 141
False Negatives: 33
Precision: 0.69
Recall: 0.35
F1 Score: 0.47
```

Formulas:

- **Precision** = $\frac{TP}{TP+FP}$ → How many of the predicted positives were actually positive
- **Recall (Sensitivity)** = $\frac{TP}{TP+FN}$ → How many actual positives were correctly identified
- **F1 Score** = $2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$ → Harmonic mean of precision and recall

- **Recall (Sensitivity)**: Measures how well the model captures actual positive cases.

$$\text{Recall} = \frac{TP}{TP + FN}$$

  - TP (True Positive): Actual positive and correctly predicted as positive.
  - FN (False Negative): Actual positive, but predicted as negative (missed positive).

So **recall** answers: *"Out of all the actual positive cases, how many did the model catch?"*

- Actual Positives = TP + FN
- Predicted Positives = TP + FP

Recall - Actual Positive

Precision - Actual + Predictive Positive

To sum up:

FP is about mistakenly calling something positive when it's actually negative.

Recall is about how well we identify the actual positives in the dataset.

Precision focuses on how accurate our positive predictions are.

When detecting fraud, the confusion matrix works like this:

Actual / Predicted	Fraud (Positive)	Not Fraud (Negative)
Fraud (Positive)	**TP** (True Positive)	**FN** (False Negative)
Not Fraud (Negative)	**FP** (False Positive)	**TN** (True Negative)

Breaking it down:

- **TP (True Positive)** → Actual is fraud, predicted is fraud (Correct detection)
- **FP (False Positive)** → Actual is not fraud, predicted as fraud (False alarm)
- **TN (True Negative)** → Actual is not fraud, predicted as not fraud (Correct rejection)
- **FN (False Negative)** → Actual is fraud, predicted as not fraud (Missed fraud)

When detecting fraud, the confusion matrix works like this:

Actual / Predicted	Fraud (Positive)	Not Fraud (Negative)
Fraud (Positive)	**TP** (True Positive)	**FN** (False Negative)
Not Fraud (Negative)	**FP** (False Positive)	**TN** (True Negative)

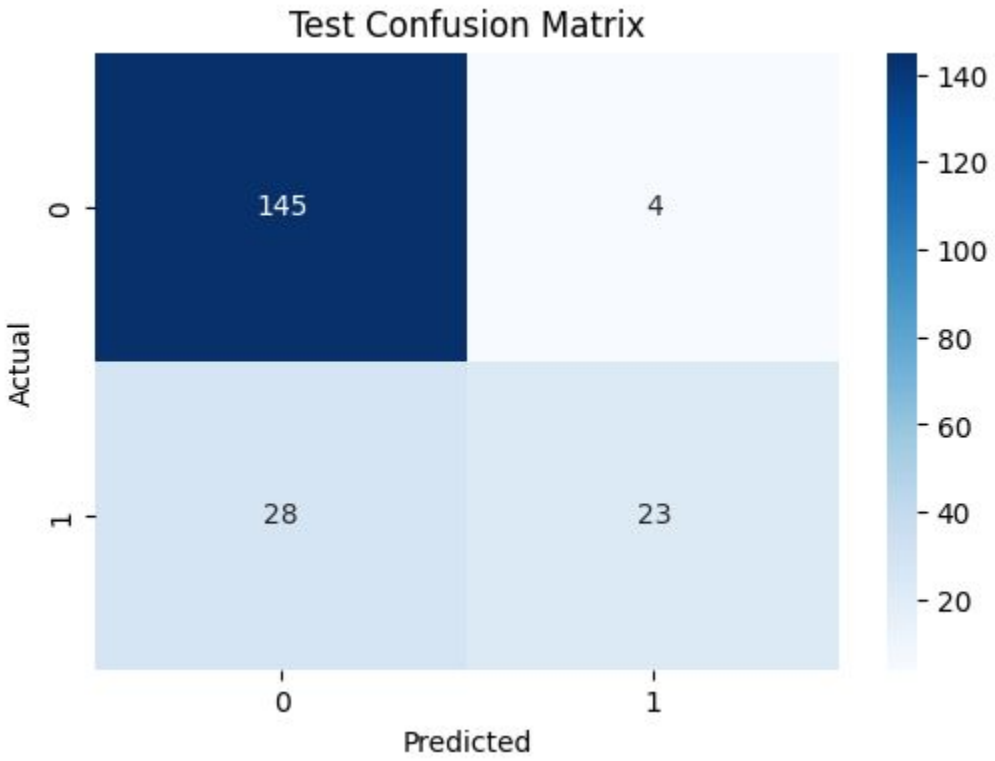
Breaking it down:

- **TP (True Positive)** → Actual is fraud, predicted is fraud (Correct detection)
- **FP (False Positive)** → Actual is not fraud, predicted as fraud (False alarm)
- **TN (True Negative)** → Actual is not fraud, predicted as not fraud (Correct rejection)
- **FN (False Negative)** → Actual is fraud, predicted as not fraud (Missed fraud)

2) Try Random Forest Classifier Algorithm

```
[ ]
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

```




```
Test Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.97      0.90       149
           1       0.85      0.45      0.59        51

    accuracy                           0.84       200
   macro avg       0.85      0.71      0.75       200
weighted avg       0.84      0.84      0.82       200

Train Accuracy: 1.0
Test Accuracy: 0.84
The model is overfitting.
```

```
True Positives: 23
False Positives: 4
True Negatives: 145
False Negatives: 28
Precision: 0.85
Recall: 0.45
F1 Score: 0.59
```

Train Accuracy: 1.0

Test Accuracy: 0.84

Model is Overfit

True Positives: 23, False Positives: 4, True Negatives: 145, False Negatives: 28

Precision: 0.85 Recall: 0.45

F1 Score: 0.59

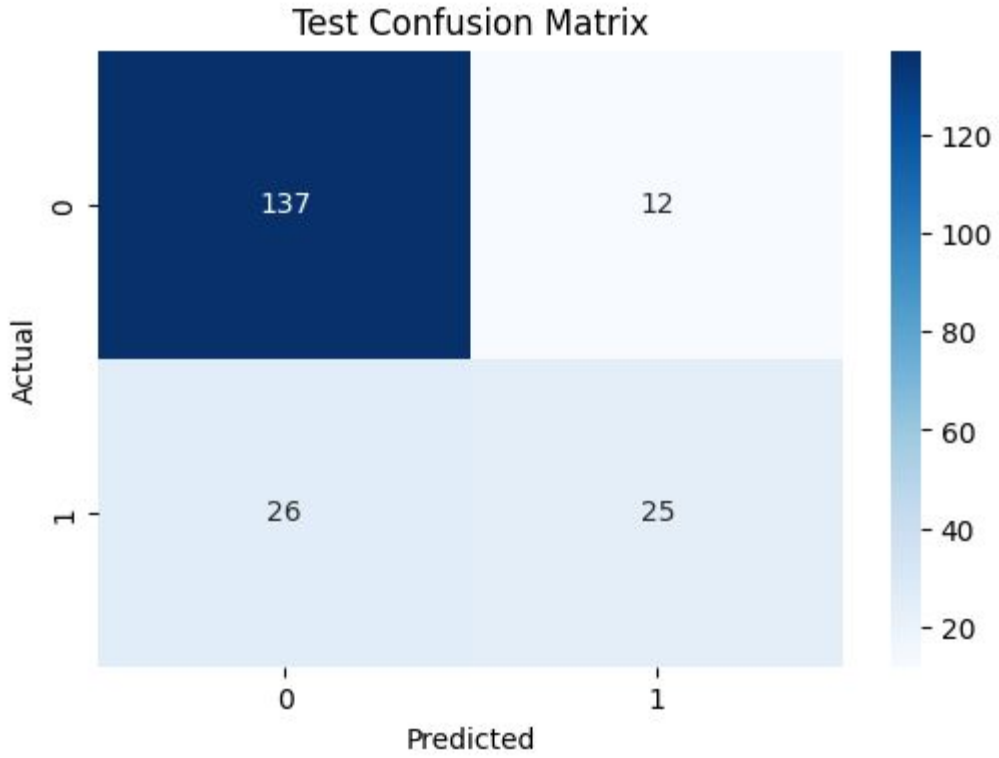
F1 only stays high when both precision and recall are balanced and strong.

- **Precision (0.85)**: When the model predicts positive (like fraud), **85% of those predictions are actually positive**.
    - So the model is **good at avoiding false positives** — when it calls something fraud, it's usually right.
- **Recall (0.45)**: Out of all the **actual positive cases**, the model only identifies **45% of them**.
    - This tells us the model **misses a lot of actual positive cases** — hence why your **false negatives are high** (28 FN).

3) Try XG Boosting

```
[ ]
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from xgboost import XGBClassifier
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# XGBoost model
xgb_model = XGBClassifier(n_estimators=100, random_state=42, use_label_encoder=False, eval_metric='logloss')
xgb_model.fit(X_train, y_train)

# Predictions
y_train_pred = xgb_model.predict(X_train)
y_test_pred = xgb_model.predict(X_test)
```


Test Confusion Matrix
137
12
26
25
Actual
0
1
Predicted
0
1
120
100
80
60
40
20

```
Test Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.92      0.88       149
           1       0.68      0.49      0.57        51

    accuracy                           0.81       200
   macro avg       0.76      0.70      0.72       200
weighted avg       0.80      0.81      0.80       200

Train Accuracy: 1.0
Test Accuracy: 0.81
The model is overfitting.
```

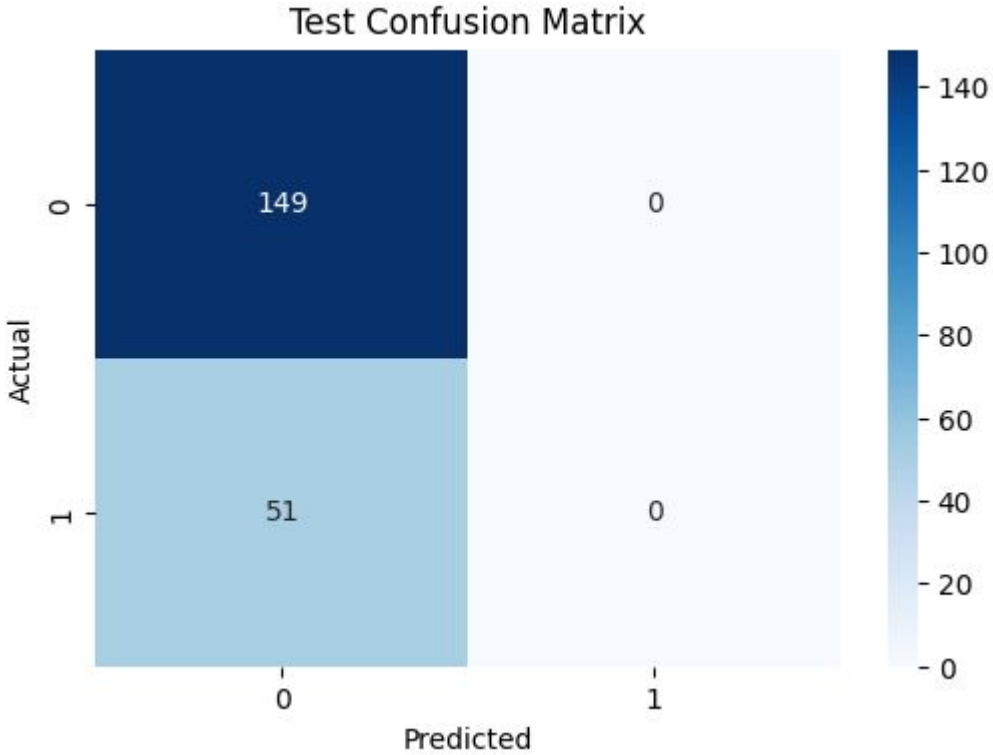
```
True Positives: 25
False Positives: 12
True Negatives: 137
False Negatives: 26
Precision: 0.68
Recall: 0.49
F1 Score: 0.57
```

Combare to Randomforest xg boosting show low accuracy and low recall shows

4) Try Support Vector Machine (SVM)

```
[ ]
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Support Vector Machine model
svm_model = SVC(kernel='rbf', random_state=42, probability=True)
svm_model.fit(X_train, y_train)

# Predictions
y_train_pred = svm_model.predict(X_train)
y_test_pred = svm_model.predict(X_test)
```


Test Confusion Matrix
149
0
51
0
Actual
0
1
0
1
Predicted
140
120
100
80
60
40
20
0

```
Test Classification Report:
              precision    recall  f1-score   support

           0       0.74      1.00      0.85       149
           1       0.00      0.00      0.00        51

    accuracy                           0.74       200
   macro avg       0.37      0.50      0.43       200
weighted avg       0.56      0.74      0.64       200

Train Accuracy: 0.81
Test Accuracy: 0.745
The model's performance needs closer inspection.
```

```
True Positives: 0
False Positives: 0
True Negatives: 149
False Negatives: 51
Precision: 0.00
Recall: 0.00
F1 Score: 0.00
```

Train Accuracy: 0.81

Test Accuracy: 0.74

Precision: 0.00

Recall: 0.00

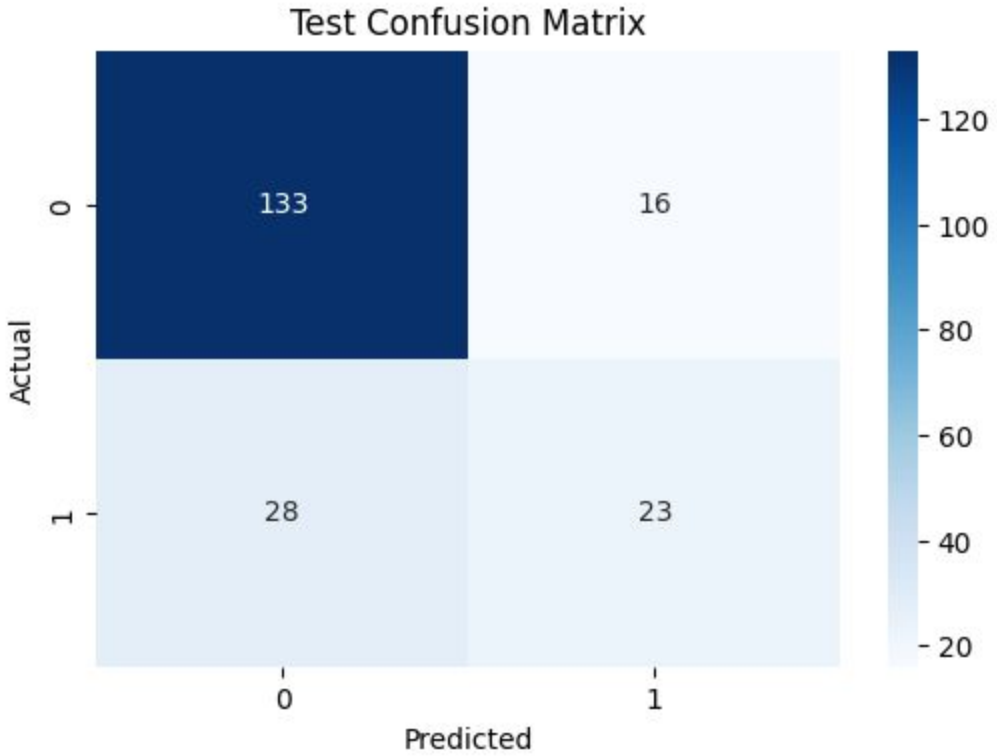
F1 Score: 0.00

Feature scaling: SVMs are sensitive to feature scale — try normalizing or standardizing your features (like using StandardScaler).

5) Try KNN (K-Nearest Neighbour) Classifier

[ ]

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# K-Nearest Neighbors model
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train, y_train)

# Predictions
y_train_pred = knn_model.predict(X_train)
y_test_pred = knn_model.predict(X_test)
```


Test Confusion Matrix
133
16
28
23
Actual
0
1
Predicted
0
1
120
100
80
60
40
20

```
Test Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.89      0.86       149
           1       0.59      0.45      0.51        51

    accuracy                           0.78       200
   macro avg       0.71      0.67      0.68       200
weighted avg       0.77      0.78      0.77       200

Train Accuracy: 0.86625
Test Accuracy: 0.78
The model's performance needs closer inspection.
```

```
True Positives: 23
False Positives: 16
True Negatives: 133
False Negatives: 28
Precision: 0.59
Recall: 0.45
F1 Score: 0.51
```

Train Accuracy: 0.86

Test Accuracy: 0.78

but precision, recall, f1 score value was very low

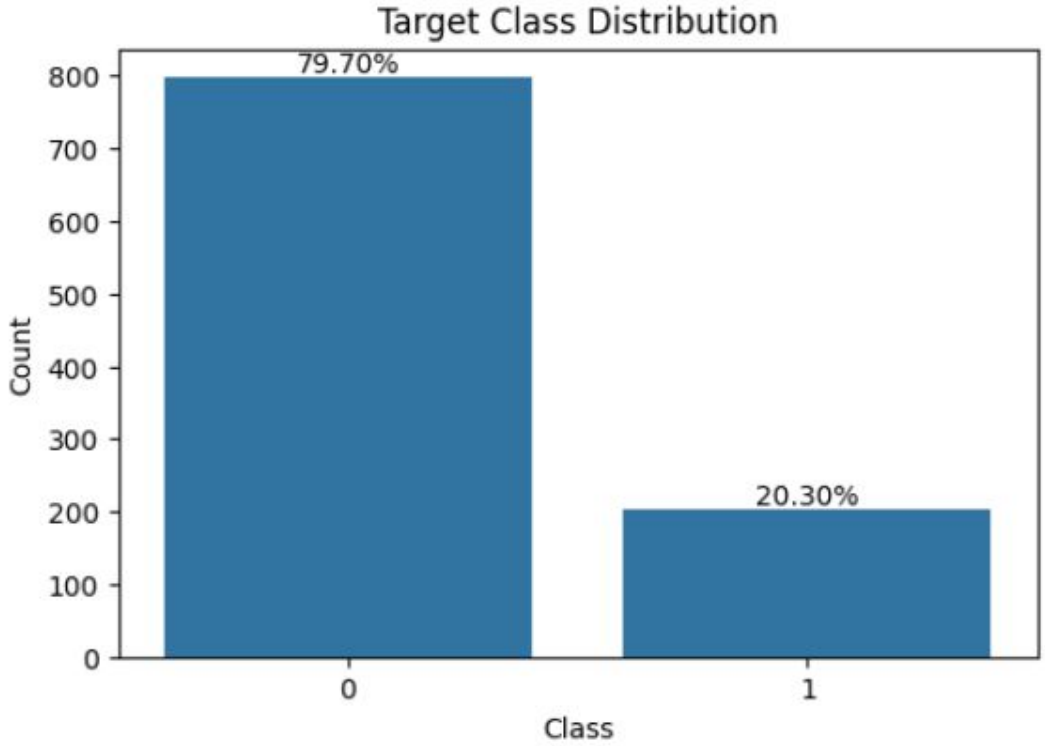
Precision, Recall, F1_Score Very Low

6) Now Check Dataset is Balanced or Not

```
[ ]
```

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame and 'target' is your target column
class_distribution = target.value_counts()
class_percentage = (class_distribution / class_distribution.sum()) * 100

# Printing class counts and percentages
print(pd.concat([class_distribution, class_percentage.round(2)], axis=1, keys=['Count', 'Percentage']))

# Checking the balance percentage
balance_ratio = class_distribution.min() / class_distribution.max()
print(f"Balance Ratio: {balance_ratio:.2f}")

if balance_ratio > 0.8:  # You can adjust this threshold
    print("The dataset is balanced.")
else:
    print("The dataset is imbalanced.")

```

```
              Count  Percentage
Fraud_Label
0               797        79.7
1               203        20.3
Balance Ratio: 0.25
The dataset is imbalanced.
```



Dataset is Imbalanced

```
[ ]  1 ##########################################################################################################
```

using SMOTE -> Synthetic Oversambling for Minority Class

Negative:

its add Synthtic Data on oversampling

##

so i am trying cross validation - **Stratified K-Fold to Concentrate Particular Class Based on Percentage**

7) Random Forest - Stratified K Fold

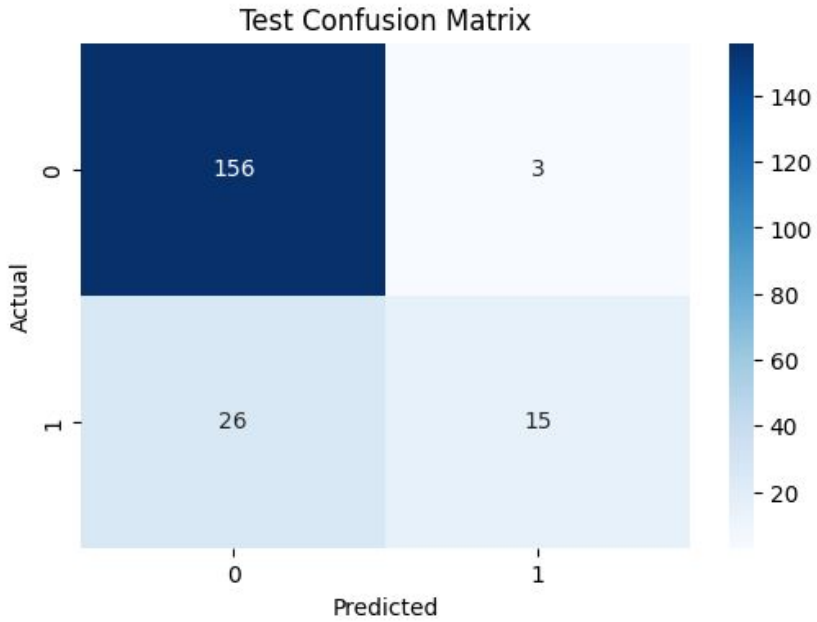
Using Folds - 2 & Focusing Minority Class is 21% and Majority Class is 79%

```
[ ]
```

```python
import pandas as pd
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Class distribution focus: 21% class 0, 79% class 1
class_distribution = {0: 0.21, 1: 0.79}

# Stratified K-Fold cross-validation
skf = StratifiedKFold(n_splits=2, shuffle=True, random_state=42)
```

```python

# Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42, class_weight=class_distribution)

# Cross-validation accuracy
cv_scores = cross_val_score(rf_model, X, y, cv=skf, scoring='accuracy')
print(f'Cross-Validation Accuracy Scores: {cv_scores}')
print(f'Mean CV Accuracy: {cv_scores.mean():.2f}')

```

```
Cross-Validation Accuracy Scores: [0.838 0.838]
Mean CV Accuracy: 0.84

Test Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.98      0.91       159
           1       0.83      0.37      0.51        41

    accuracy                           0.85       200
   macro avg       0.85      0.67      0.71       200
weighted avg       0.85      0.85      0.83       200

Train Accuracy: 1.0
Test Accuracy: 0.855
The model is overfitting.
```



Model is Overfit But Precision & Recall Score Very Low

```
True Positives: 15
False Positives: 3
True Negatives: 156
False Negatives: 26
Precision: 0.83
Recall: 0.37
F1 Score: 0.51
```

8) Use Hyperparameter Tuning for Random Forest

```
[ ]
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt
```

```python
# Hyperparameter grid
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

Best Parameters: {'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 150}

```
15 # Hyperparameter grid
16 param_grid = {
17     'n_estimators': [50, 100, 150],
18     'max_depth': [None, 10, 20],
19     'min_samples_split': [2, 5, 10],
20     'min_samples_leaf': [1, 2, 4]
21 }
22
```

After Using Hyperparameter Tuning

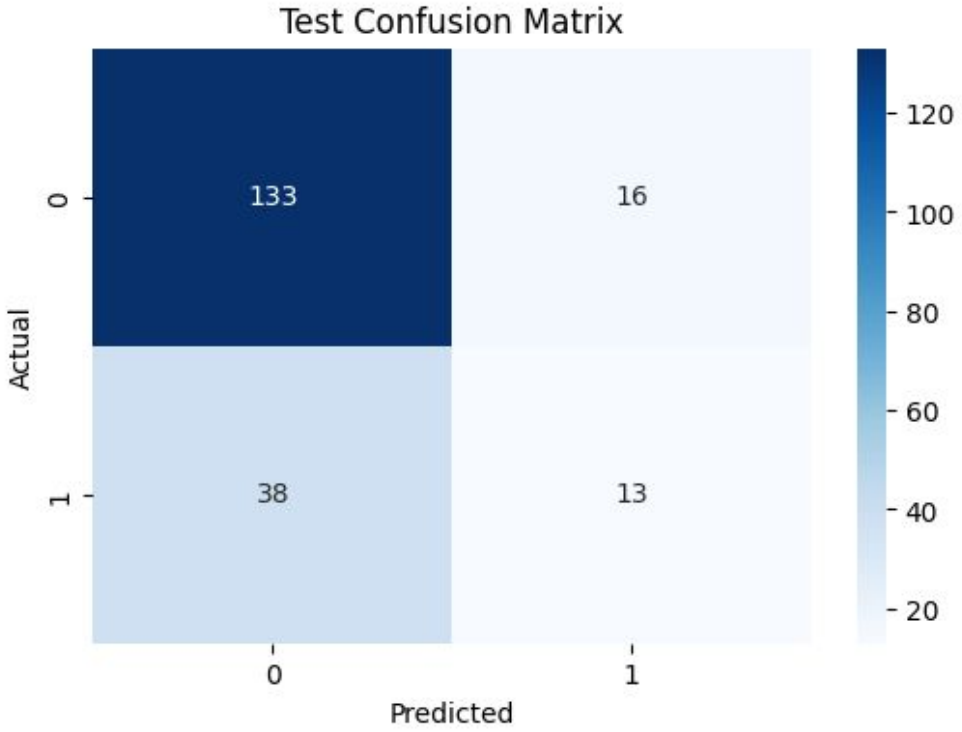
```
Test Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.99      0.92       159
           1       0.88      0.34      0.49        41

    accuracy                           0.85       200
   macro avg       0.86      0.66      0.70       200
weighted avg       0.86      0.85      0.83       200

Train Accuracy: 0.92375
Test Accuracy: 0.855
The model's performance needs closer inspection.
```

9) Use Hyperparameter Tuning for KNN Classifier

```
[ ]
```

```python
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# KNN hyperparameter grid
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}
```

Best Parameters: {'metric': 'manhattan', 'n_neighbors': 11, 'weights': 'distance'}



Test Classification Report:

	precision	recall	f1-score	support
0	0.78	0.89	0.83	149
1	0.45	0.25	0.33	51
accuracy			0.73	200
macro avg	0.61	0.57	0.58	200
weighted avg	0.69	0.73	0.70	200

Train Accuracy: 1.0
Test Accuracy: 0.73
The model is overfitting.

Not Get Accuracy / Generalized Well Using Machine Learning

Deep Learning is Sensitive for Unscalled Date - So Scaled Claim amount and Annual Income

```python
from sklearn.preprocessing import MinMaxScaler

# Selecting the features
features_to_scale = feature[['Claim_Amount', 'Annual_Income']]

# Initializing and applying MinMaxScaler
scaler = MinMaxScaler()
features_scaled = scaler.fit_transform(features_to_scale)

# Replacing the original columns with scaled values
feature[['Claim_Amount', 'Annual_Income']] = features_scaled

# Checking the result
print(feature[['Claim_Amount', 'Annual_Income']].head())
```

```
   Claim_Amount  Annual_Income
0      0.639475       0.567695
1      0.138698       0.906766
2      0.086005       0.308666
3      0.231986       0.913694
4      0.649951       0.258611
```

10) Pytorch - Deep Learning

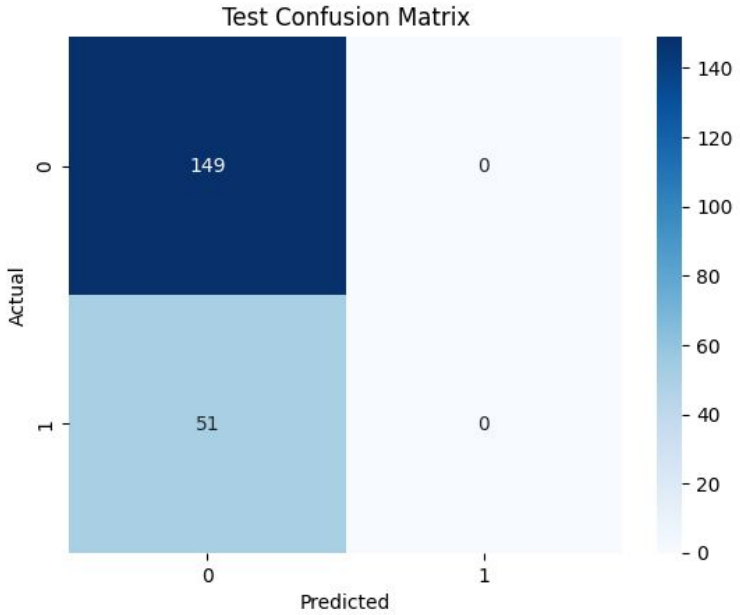
```
]
```

```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df is your DataFrame and 'target' is your target column
X = feature
y = target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train = torch.tensor(X_train.values, dtype=torch.float32)
X_test = torch.tensor(X_test.values, dtype=torch.float32)
y_train = torch.tensor(y_train.values, dtype=torch.long)
y_test = torch.tensor(y_test.values, dtype=torch.long)
```

```python
# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self, input_size):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 2)  # Assuming binary classification

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Model setup
input_size = X_train.shape[1]
model = SimpleNN(input_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

```

```python
# Training loop
epochs = 20
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()
    print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

# Evaluation
model.eval()
with torch.no_grad():
    train_preds = model(X_train).argmax(dim=1)
    test_preds = model(X_test).argmax(dim=1)

# Metrics
train_acc = accuracy_score(y_train, train_preds)
test_acc = accuracy_score(y_test, test_preds)
print(f"Train Accuracy: {train_acc:.2f}")
print(f"Test Accuracy: {test_acc:.2f}")

print("Test Classification Report:\n", classification_report(y_test, test_preds))

```

```
Epoch [11/20], Loss: 0.8225
Epoch [12/20], Loss: 1.3909
Epoch [13/20], Loss: 1.5441
Epoch [14/20], Loss: 1.3207
Epoch [15/20], Loss: 0.7850
Epoch [16/20], Loss: 0.8226
Epoch [17/20], Loss: 0.8857
Epoch [18/20], Loss: 0.6529
Epoch [19/20], Loss: 0.9810
Epoch [20/20], Loss: 0.9930
Train Accuracy: 0.81
Test Accuracy: 0.74
Test Classification Report:
              precision    recall  f1-score   support

           0       0.74      1.00      0.85       149
           1       0.00      0.00      0.00        51

    accuracy                           0.74       200
   macro avg       0.37      0.50      0.43       200
weighted avg       0.56      0.74      0.64       200
```



test accuracy higher:

Tuning the Neural Network Architecture:

Add more layers or neurons.

Try different activation functions like ReLU, LeakyReLU, or GELU.

Experiment with batch normalization or dropout to prevent overfitting.

Optimizers and Learning Rate:

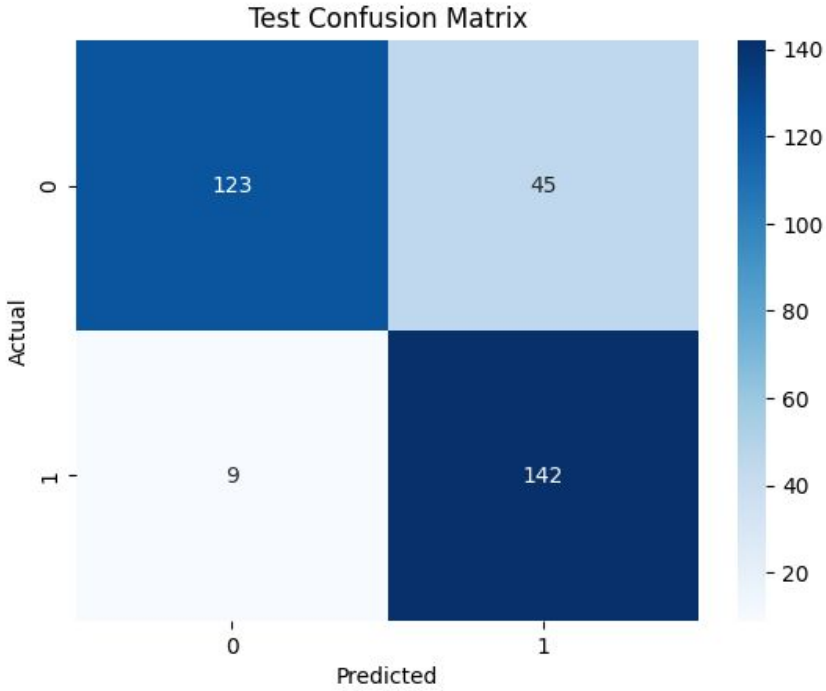
Try different optimizers like AdamW, RMSprop, or SGD.

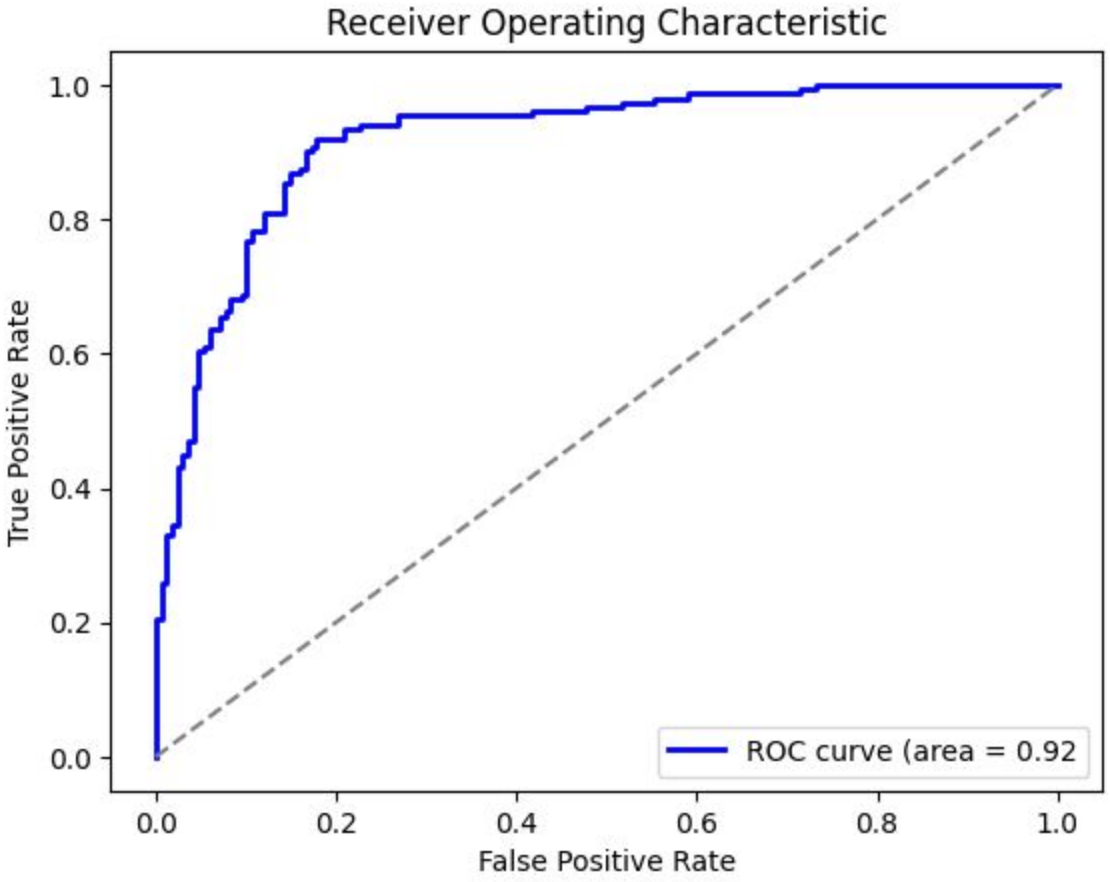
Tune the learning rate — sometimes lowering it helps generalization.

```python
# Define an enhanced neural network with more capacity
class EnhancedNN(nn.Module):
    def __init__(self, input_size):
        super(EnhancedNN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(),
            nn.Dropout(0.2),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(),
            nn.Dropout(0.2),
            nn.Linear(256, 128),
            nn.LeakyReLU(),
            nn.Linear(128, 64),
            nn.LeakyReLU(),
            nn.Linear(64, 2)
        )

    def forward(self, x):
        return self.net(x)
```

```
# Model setup
input_size = X_train.shape[1]
model = EnhancedNN(input_size)
criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=

# Training loop
epochs = 100
best_test_acc = 0
patience, patience_counter = 15, 0
min_target_acc = 0.85

for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()
    scheduler.step()
```

```
Epoch [98/100], Loss: 0.2239, Test Accuracy: 0.8245
Epoch [99/100], Loss: 0.2165, Test Accuracy: 0.8339
Epoch [100/100], Loss: 0.2097, Test Accuracy: 0.8307
Final Train Accuracy: 0.89
Final Test Accuracy: 0.83
Test Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.73      0.82       168
           1       0.76      0.94      0.84       151

    accuracy                           0.83       319
   macro avg       0.85      0.84      0.83       319
weighted avg       0.85      0.83      0.83       319
```




Receiver Operating Characteristic
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.92

```
True Positives (TP): 142 - Correctly predicted positive cases
True Negatives (TN): 123 - Correctly predicted negative cases
False Positives (FP): 45 - Incorrectly predicted positive cases (actual negative)
False Negatives (FN): 9 - Incorrectly predicted negative cases (actual positive)
```

True Positive (TP): Predicted 1, Actual 1

True Negative (TN): Predicted 0, Actual 0

False Positive (FP): Predicted 1, Actual 0

False Negative (FN): Predicted 0, Actual 1

ROC Curve Area is 0.93 is near to 1 so its very Good Score: - Becaue its Increase True Posative Rate from False Posative

Find a Best Threshold for my ROC Curve Area

```python
from sklearn.metrics import roc_curve
import numpy as np

# Get FPR, TPR, and thresholds
fpr, tpr, thresholds = roc_curve(y_test, probs)

# Youden's J statistic: TPR - FPR
youden_index = tpr - fpr

# Best threshold is where J is max
best_threshold = thresholds[np.argmax(youden_index)]

print(f"Best threshold: {best_threshold:.2f}")

```

```
Best threshold: 0.80
```

Other methods for choosing thresholds:

- **Maximizing F1-score**: Balances precision and recall.
- **Minimizing classification error**: Picks the threshold where predictions are most accurate.
- **Optimizing for business needs**: In fraud detection, you may want to minimize false negatives even at the cost of more false positives.

In the ROC curve:

- **Threshold** is the decision boundary for turning predicted probabilities into class labels (like fraud or not fraud).
- By changing the threshold, you adjust the **tradeoff between sensitivity (True Positive Rate)** and **specificity (False Positive Rate)**.

For binary classification, your model gives **probabilities** — like how likely a claim is fraudulent. A common threshold is `0.5`:

- If the probability ≥ 0.5 → Predict class 1 (fraud)
- If the probability < 0.5 → Predict class 0 (not fraud)

How to Pick Best Threshold

But **0.5 isn't always optimal**! The ROC curve helps you find a better threshold based on your goals — like maximizing accuracy, minimizing false positives, or balancing recall and precision.

How to pick the best threshold?

One popular method is the **Youden's J statistic**, which maximizes the difference between TPR and FPR:

This code builds and trains an enhanced neural network using PyTorch for a binary classification task on insurance claims data. Let's break down the process step by step:

Process:

1. **Feature Engineering:** A new feature, `Claim_Income_Ratio`, is created.
2. **Scaling:** MinMaxScaler scales the features.
3. **Handling Imbalance:** SMOTE is used to oversample the minority class.
4. **Class Weights:** Balanced class weights are computed to handle class imbalance in the loss function.
5. **Train-Test Split:** The dataset is split into training and testing sets.

6. **Neural Network:** The `EnhancedNN` model is defined with multiple layers.
7. **Loss & Optimizer:** CrossEntropyLoss (with class weights) and AdamW optimizer are used.
8. **Learning Rate Scheduler:** CosineAnnealingLR adjusts the learning rate over training.
9. **Training:** Early stopping is used to prevent overfitting based on validation performance.
10. **Evaluation:** Accuracy, classification report, confusion matrix, and ROC curve are plotted.
11. **Overfitting Check:** The model checks for signs of overfitting or underfitting.
12. **Model Saving:** The trained model is saved for future use.

Libraries Used:

- **PyTorch:** For building and training the neural network.
- **Scikit-learn:** For data preprocessing, evaluation metrics, and train-test split.
- **Imbalanced-learn:** For SMOTE to address class imbalance.
- **Seaborn & Matplotlib:** For visualizations like the confusion matrix and ROC curve.

Hyperparameters:

- **Learning Rate:** 0.001
- **Epochs:** 100
- **Early Stopping Patience:** 15 epochs
- **Minimum Target Accuracy:** 85%
- **Hidden Layers:** 4
- **Neurons per Layer:** 512 → 256 → 128 → 64
- **Dropout:** 0.2 for regularization
- **Activation Functions:** LeakyReLU for faster and more stable training

Dropout with a rate of 0.2 is used for **regularization** to help prevent **overfitting**. Let's break down how and why:

- **What is Dropout?**
  Dropout randomly "drops" (sets to zero) a fraction of neurons during each training step. In this case, **20% of neurons** (because of `0.2`) are turned off in each forward pass. This forces the network to **not rely too heavily on specific neurons** and encourages it to **learn more robust and general features**.

- **Why use Dropout?**
  Without regularization, deep networks can easily **memorize training data**, leading to **overfitting** — where the model performs well on training data but struggles on unseen data. Dropout helps by:
  - Reducing **co-adaptation** between neurons (where some neurons become dependent on others).
  - Forcing the network to **distribute learning** across different parts of the architecture.
  - Acting like **ensemble learning**, as different subsets of the network train during different passes.

- **Why 0.2?**

  A dropout rate of 0.2 is a **balanced choice** — it **removes enough neurons** to reduce overfitting but **keeps enough active neurons** to preserve learning capacity. Too high a rate (like 0.5) could lead to **underfitting**, and too low (like 0.05) may **not regularize enough**.

In this model, Dropout works alongside **Batch Normalization** and **LeakyReLU** to **stabilize and generalize** training — making sure the model **doesn't just memorize training data** but **performs well on new data**.

False Positive Rate (FPR) measures how many **actual negative cases** are incorrectly classified as positive by the model.

Formula:

$$\text{FPR} = \frac{\text{False Positives (FP)}}{\text{False Positives (FP)} + \text{True Negatives (TN)}}$$

Why a high False Positive Rate is not good:

Let's say you're predicting insurance fraud — a positive prediction means "fraud" and a negative prediction means "not fraud."

- **False Positive:** Predicting "fraud" when it's actually not fraud
- **False Negative:** Predicting "not fraud" when it **is** fraud

If the **False Positive Rate is high**, it means you're frequently accusing legitimate claims of being fraudulent. This can lead to:

- Unnecessary investigations, wasting time and money
- Frustrated customers whose legitimate claims get delayed
- Loss of trust in your system

That's why we try to **keep the FPR low** — we don't want too many false alarms.

Of course, the balance depends on the use case. In fraud detection, we often prioritize **catching actual fraud (True Positives)**, but not at the cost of wrongly accusing too many innocent claims. That's why metrics like **AUC-ROC** are useful — they balance both the True Positive Rate and False Positive Rate.

So, in short: **High False Positive Rate is usually not good** — because it means too many false alarms. Keeping it low means your model is better at distinguishing real fraud from legitimate claims.

Epoch [99/100], Loss: 0.1990, Test Accuracy: 0.8339 Early stopping triggered

Final Train Accuracy: 0.90

Final Test Accuracy: 0.83

loss [Cost Function is Loss: 0.1990]

99th Epoch out of total 100 Epoch attain Accuracy

```
[ ]  1 print("Test Classification Report:\n", classification_report(y_test, test_preds))
```

```
Test Classification Report:
                precision    recall  f1-score   support

           0       0.93      0.73      0.82       168
           1       0.76      0.94      0.84       151

    accuracy                           0.83       319
   macro avg       0.85      0.84      0.83       319
weighted avg       0.85      0.83      0.83       319
```

Overall Metrics:

Accuracy: 0.83

Macro Average:

Precision: 0.76

Recall: 0.71

F1-Score: 0.73

Weighted Average:

Precision: 0.85

Recall: 0.83

F1-Score: 0.83

In your report:

Macro Average (Precision: 0.76, Recall: 0.71, F1: 0.73)

— shows how well the model performs across both classes **equally**, even though Class 1 has fewer samples.

Weighted Average (Precision: 0.80, Recall: 0.81, F1: 0.81)

— **gives more importance** to Class 0's performance because it has more samples, making it a more realistic reflection of overall performance.

```
[ ]    1 ##########################################################################################
```

recall is often more important, especially in situations like insurance fraud detection or legal claims where missing a positive case (like a fraudulent claim) could lead to bigger consequences

1st - Attempt

Future Prediction

df3 - Future Prediction

```
[ ]  1 ##########################################################################
```

```
[ ]  1 import torch
     2 import torch.nn as nn
     3
     4 class EnhancedNN(nn.Module):
     5     def __init__(self, input_size):
     6         super(EnhancedNN, self).__init__()
     7         self.fc1 = nn.Linear(input_size, 512)
     8         self.relu = nn.ReLU()
     9         self.fc2 = nn.Linear(512, 2)  # Assuming binary classification
    10
    11     def forward(self, x):
    12         x = self.relu(self.fc1(x))
    13         x = self.fc2(x)
    14         return x
    15
```

```
[ ]
```

```python
# Load the entire model
model = torch.load('/content/enhanced_nn_model.pth')
model.eval()

print("Model loaded successfully!")

```

```
Model loaded successfully!
<ipython-input-3-efb30f7ad99a>:2: FutureWarning: You are using `torch.load` with `weights_only=False`
  model = torch.load('/content/enhanced_nn_model.pth')
```

```
[ ]  1 checkpoint = torch.load('enhanced_nn_model.pth')
     2 print(checkpoint)
```

```
EnhancedNN(
  (net): Sequential(
    (0): Linear(in_features=12, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01)
    (3): Dropout(p=0.2, inplace=False)
    (4): Linear(in_features=512, out_features=256, bias=True)
    (5): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.01)
    (7): Dropout(p=0.2, inplace=False)
    (8): Linear(in_features=256, out_features=128, bias=True)
    (9): LeakyReLU(negative_slope=0.01)
    (10): Linear(in_features=128, out_features=64, bias=True)
    (11): LeakyReLU(negative_slope=0.01)
    (12): Linear(in_features=64, out_features=2, bias=True)
  )
)
<ipython-input-4-e77816a0367f>:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the
  checkpoint = torch.load('enhanced_nn_model.pth')
```

This is a PyTorch neural network architecture defined using `nn.Sequential()`, which stacks layers in a simple, ordered way. Let's break it down step by step:

1. `(0): Linear(in_features=12, out_features=512, bias=True)`
    - This is a fully connected (dense) layer.
    - `in_features=12`: The input has 12 features — likely your scaled, reduced feature set for fraud prediction.
    - `out_features=512`: This layer outputs 512 features. It's a big layer — lots of learnable parameters.
    - `bias=True`: Adds a learnable bias term to the output of the layer.

2. `(1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)`
    - Applies batch normalization to the 512 outputs of the previous layer.
    - Batch norm helps stabilize training by normalizing input across the batch — it can help prevent overfitting and speed up training.
    - `eps=1e-05`: Small value added to avoid division by zero.
    - `momentum=0.1`: Controls the running mean and variance's update speed.
    - `affine=True`: Allows scale and shift after normalization (two more learnable parameters).
    - `track_running_stats=True`: Tracks mean and variance during training to use in evaluation.

3. `(2): LeakyReLU(negative_slope=0.01)`
    - Activation function. It introduces non-linearity, allowing the network to learn complex patterns.
    - `negative_slope=0.01`: Controls how much "leak" there is for negative input values (so small negative values don’t get completely zeroed out).
4. `(3): Dropout(p=0.2, inplace=False)`
    - Regularization technique to prevent overfitting by randomly dropping (zeroing out) 20% of the neurons’ outputs during training.
    - `p=0.2`: 20% dropout rate.
    - `inplace=False`: Doesn’t modify the input directly, creating a new tensor instead.

5. `(4): Linear(in_features=512, out_features=256, bias=True)`
    - Another fully connected layer, reducing the size of the feature space from 512 to 256.
6. `(5): BatchNorm1d(256, ...)`
    - Batch normalization again, now for the 256 outputs of the previous layer.
7. `(6): LeakyReLU(negative_slope=0.01)`
    - Same activation function as earlier, introducing non-linearity.
8. `(7): Dropout(p=0.2, inplace=False)`
    - Another dropout layer to further reduce overfitting.
9. `(8): Linear(in_features=256, out_features=128, bias=True)`
    - Fully connected layer reducing output from 256 to 128 features.
10. `(9): LeakyReLU(negative_slope=0.01)`
    - Non-linearity again. This layer doesn't use batch norm or dropout, possibly keeping things simpler now that the feature space is smaller.
11. `(10): Linear(in_features=128, out_features=64, bias=True)`
    - Another reduction in feature space, from 128 to 64.
12. `(11): LeakyReLU(negative_slope=0.01)`
    - Same activation function again.

13. `(12): Linear(in_features=64, out_features=2, bias=True)`

    - The final fully connected layer. It reduces output from 64 to 2 — likely for a binary classification task (like fraud detection: fraud/not fraud).
    - No activation here because the next step is usually applying something like `nn.Softmax()` or `nn.LogSoftmax()` (if using cross-entropy) or `nn.Sigmoid()` for binary classification.

In short, this network:

- Takes 12 input features.
- Expands them through larger layers (512, 256) and then reduces them step by step (128, 64) while applying batch normalization, activation functions, and dropout for regularization.
- Outputs 2 logits — likely for binary classification.

```python
import torch
import pandas as pd
import numpy as np

# Load the entire PyTorch model
model = torch.load('/content/enhanced_nn_model.pth')
model.eval()

print("Model loaded successfully!")

# Required columns (including engineered feature 'Claim_Income_Ratio')
required_columns = ['Claim_Amount','Claim_Type_Home_Damage','Claim_Type_Medical','Claim_Type_Vehicle',
                    'Claim_Year','Claim_Month','Claim_Day','Annual_Income',
                    'policy_issue_Year','policy_issue_Month','policy_issue_Day', 'Claim_Income_Ratio']

```

```
# Example: New data for prediction (make sure it's scaled the same way as training data)
new_data = pd.DataFrame({
    'Claim_Amount': [0.102079830222524],
    'Claim_Type_Home_Damage': [0],
    'Claim_Type_Medical': [1],
    'Claim_Type_Vehicle': [0],
    'Claim_Year': [2023],
    'Claim_Month': [5],
    'Claim_Day': [12],
    'Annual_Income': [0.139544916784529],
    'policy_issue_Year': [2020],
    'policy_issue_Month': [8],
    'policy_issue_Day': [15]
})

# Add the missing feature: Claim_Income_Ratio
new_data['Claim_Income_Ratio'] = new_data['Claim_Amount'] / (new_data['Annual_Income'] + 1e-5)

# Ensure column order matches the model’s training data
new_data = new_data[required_columns]

# Convert to PyTorch tensor and match dtype
new_data_tensor = torch.tensor(new_data.values, dtype=torch.float32)

```

```
# Convert to PyTorch tensor and match dtype
new_data_tensor = torch.tensor(new_data.values, dtype=torch.float32)

# Forward pass on the model
with torch.no_grad():
    output = model.net(new_data_tensor)  # <-- Notice the ".net" here


# Apply softmax to get class probabilities
probs = torch.softmax(output, dim=1).cpu().numpy()

# Get the predicted class (0 or 1)
predicted_class = np.argmax(probs, axis=1)

print(f"Predicted class: {predicted_class[0]}")
print(f"Class probabilities: {probs}")

```

```
Model loaded successfully!
Predicted class: 0
Class probabilities: [[1. 0.]]
<ipython-input-21-e82d52c51dcf>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current de
  model = torch.load('/content/enhanced_nn_model.pth')
```

Prediction Failure:

I am Checking Prediction Multiple Times Showing Prediction Genuine - Show Only One Class Based Result not Shows Fraud - so Model was Trained as Bias

Because - Genuine as 97% and 3% Only Fraud - So using **Smote**, - to Balanced a Dataset and **Cross-Validation** - Stratified K-Fold to focusing a classes based run

And using **Hyperparameter Tuning** to find a Best Parameter for Particular Algorithm and Finally Used **Pytorch Neural Network** to get Accuracy Model is Generalized well

Reason for Prediction Failure

Does not give Enough feature to Learn Machine Properly to Get Accuracy But Information Not Enough That’s the Main Reason for Prediction Failure

2nd - Attempt

Feature Selection

Increasing a Feature:

Feature Engineered New Feature Added on to Train a Model For better Machine Understanding

Using SQL to Analyse a Dataset to Add a Best Feature

SQL

```
[ ]
```

```python
import sqlite3
import pandas as pd

# File name of the uploaded CSV
csv_filename = "/content/df3_feature.csv"  # Replace this with the uploaded file name

# Load CSV into a pandas DataFrame
df = pd.read_csv(csv_filename)

# Connect to SQLite database (or create a new one)
conn = sqlite3.connect("example.db")
cursor = conn.cursor()

# Write DataFrame to SQLite table
table_name = "df3"  # Specify your table name
df.to_sql(table_name, conn, if_exists="replace", index=False)

print(f"Table '{table_name}' created in SQLite database.")

```

```
Table 'df3' created in SQLite database.
```

1) Find Short Period Claim True and Claim Income Ratio More Than 0.5

```
query = f"""SELECT Fraud_Label,Suspicious_Flags,Claim_to_Income_Ratio,Short_Period_Claim,Isolation_Anomaly
from df3 where Short_Period_Claim=True and Claim_to_Income_Ratio>=0.5  Order by Claim_to_Income_Ratio DESC;"""

result = pd.read_sql_query(query, conn)

# Display the results
result
```

	Fraud_Label	Suspicious_Flags	Claim_to_Income_Ratio	Short_Period_Claim	Isolation_Anomaly
0	1	1	1.459706	1	1
1	0	0	1.442398	1	1
2	1	1	1.367275	1	1
3	0	0	1.244962	1	1
4	0	0	1.244078	1	1
5	0	0	1.242741	1	1
6	0	0	1.200272	1	1

2) Find the Short Period Claim is False and Claim Income Ratio is Above 0.5

```
query = f"SELECT Fraud_Label,Suspicious_Flags,Claim_to_Income_Ratio,Short_Period_Claim,Isolation_Anomaly,Elliptic_Anomaly    f
result = pd.read_sql_query(query, conn)

# Display the results
result
```

	Fraud_Label	Suspicious_Flags	Claim_to_Income_Ratio	Short_Period_Claim	Isolation_Anomaly	Elliptic_Anomaly
0	1	1	1.613079	0	1	Anomaly
1	0	0	1.527923	0	1	Anomaly
2	1	1	1.463065	0	1	Anomaly
3	1	0	1.412514	0	1	Anomaly
4	0	0	1.386726	0	1	Anomaly
...	...	...	...	...	...	...
93	0	0	0.511780	0	0	Anomaly

3) Claim Income Ratio is above 0.5 and Short Period Claim is True and Isolation Forest is 1(Anomaly)

```
[ ]  1 query = f"""SELECT Fraud_Label,Suspicious_Flags,Claim_to_Income_Ratio,Short_Period_Claim,Isolation_Anomaly
     2 from df3 where Short_Period_Claim=True and Claim_to_Income_Ratio>=0.5 and Isolation_Anomaly=1
     3 Order by Claim_to_Income_Ratio DESC;"""
     4
     5 result = pd.read_sql_query(query, conn)
     6
     7 # Display the results
     8 result
```

6	0	0	1.200272	1	1
7	1	0	1.114620	1	1
8	1	1	1.066805	1	1
9	1	0	1.032081	1	1
10	0	0	1.000603	1	1
11	0	0	0.999121	1	1
12	0	0	0.993055	1	1

4) Claim Income Ratio is above 0.5 and Short Period Claim is False and Isolation Forest is 1(Anomaly)

```
[ ]  1 query = f"SELECT Fraud_Label,Suspicious_Flags,Claim_to_Income_Ratio,Short_Period_Claim,Isolation_Anomaly  from df3 where Short
     2 result = pd.read_sql_query(query, conn)
     3
     4 # Display the results
     5 result
```

23	1	1	0.913566	0	1
24	1	1	0.905368	0	1
25	1	1	0.894064	0	1
26	0	0	0.866729	0	1
27	0	0	0.855805	0	1
28	0	0	0.836727	0	1
29	0	0	0.834503	0	1
30	0	0	0.809480	0	1

5) For Suspecius Flag Find

Claim Income ratio is more than 0.5

```
[ ]  1 query = f"""SELECT Fraud_Label,Suspicious_Flags,Claim_to_Income_Ratio,Short_Period_Claim,Isolation_Anomaly from df3
     2 where Claim_to_Income_Ratio>=0.5 Order by Claim_to_Income_Ratio DESC;"""
     3
     4 result = pd.read_sql_query(query, conn)
     5
     6 # Display the results
     7 result
```

	Fraud_Label	Suspicious_Flags	Claim_to_Income_Ratio	Short_Period_Claim	Isolation_Anomaly
0	1	1	1.613079	0	1
1	0	0	1.527923	0	1
2	1	1	1.463065	0	1
3	1	1	1.459706	1	1
4	0	0	1.442398	1	1

6) For Fraudlend Claim Find Using

Short period Claim is True and Claim Ratio above 0.7 and Isolation Anomaly is 1 (Anomaly)

```
query = f"SELECT Fraud_Label,Suspicious_Flags,Claim_to_Income_Ratio,Short_Period_Claim,Isolation_Anomaly   from df3 where Shor
result = pd.read_sql_query(query, conn)

# Display the results
result
```

	Fraud_Label	Suspicious_Flags	Claim_to_Income_Ratio	Short_Period_Claim	Isolation_Anomaly
0	1	1	1.459706	1	1
1	0	0	1.442398	1	1
2	1	1	1.367275	1	1
3	0	0	1.244962	1	1
4	0	0	1.244078	1	1
5	0	0	1.242741	1	1
6	0	0	1.200272	1	1

🔍 **Suspicious_Flags (based on quick, rule-based checks):**

This flag identifies claims that look suspicious but aren't necessarily confirmed as fraud. It's triggered when any of the following conditions apply:

- **High Claim-to-Income Ratio (`Claim_to_Income_Ratio` > 0.8):** A claim amount that's more than 80% of the policyholder's annual income can be a red flag for exaggerated claims.
- **Short Period Claim (`Short_Period_Claim` == 1):** Claims made within 90 days of policy issuance are often seen as suspicious because some fraudsters buy policies just to make quick claims.

⚠️ **Fraud_Label (based on more sophisticated rules):**

This label represents actual confirmed fraud based on stricter and more specific criteria:

- **High Claim Amount:** If the `Claim_Amount` exceeds 90% of the policyholder's `Annual_Income`, the claim is suspiciously large compared to their earnings.
- **Short Period Claim:** Again, quick claims after policy purchase are a well-known fraud indicator.
- **Anomaly Detection:**
  - Either `Elliptic_Anomaly` or `Isolation_Anomaly` == 1 confirms unusual behavior.

Using Synthetic Dataset - So Values are Random

1) Fraud_Label

```
[ ]
```

```python
import pandas as pd

# Update Fraud_Label where conditions are met
query1 = """
-- Set Fraud_Label to 1 for high-risk claims
UPDATE df3
SET Fraud_Label = 1
WHERE Short_Period_Claim = 1
  AND Claim_to_Income_Ratio > 0.7
  AND Isolation_Anomaly = 1;
"""

# Set all other Fraud_Label values to 0
query2 = """
-- Set Fraud_Label to 0 for all other claims
UPDATE df3
SET Fraud_Label = 0
WHERE NOT (Short_Period_Claim = 1
           AND Claim_to_Income_Ratio > 0.7
           AND Isolation_Anomaly = 1);
"""
```

1000 rows × 4 columns

```
[ ]
query = f"SELECT count(Fraud_Label) from df3 where Fraud_Label=1;"
result = pd.read_sql_query(query, conn)

# Display the results
result
```

	count(Fraud_Label)
0	26

```
[ ]
query = f"SELECT count(Fraud_Label) from df3 where Fraud_Label=0;"
result = pd.read_sql_query(query, conn)

# Display the results
result
```

	count(Fraud_Label)
0	974

```
[ ]
########################################################################
```

2) Suspicious Flag

```
[ ]
```

```python
import pandas as pd

# Update Suspicious_Flags where Claim_to_Income_Ratio is >= 0.5
query1 = """
-- Set Suspicious_Flags to 1 for claims with high Claim-to-Income Ratio
UPDATE df3
SET Suspicious_Flags = 1
WHERE Claim_to_Income_Ratio >= 0.5;
"""

# Set all other Suspicious_Flags to 0
query2 = """
-- Set Suspicious_Flags to 0 for all other claims
UPDATE df3
SET Suspicious_Flags = 0
WHERE Claim_to_Income_Ratio < 0.5;
"""

# Execute both queries and commit changes
conn.execute(query1)
conn.execute(query2)
```

```
[ ]
query = f"SELECT count(Suspicious_Flags) from df3 where Suspicious_Flags=1;"
result = pd.read_sql_query(query, conn)

# Display the results
result
```

	count(Suspicious_Flags)
0	150

```
[ ]
query = f"SELECT count(Suspicious_Flags) from df3 where Suspicious_Flags=0;"
result = pd.read_sql_query(query, conn)

# Display the results
result
```

	count(Suspicious_Flags)
0	850

Suspicious Flag vs Fraud Label

Finding Suspicious Flag Claim Income Ratio (Claim Insurance Lower than Annual Income) - If Claim Income Ratio Above 0.5 means its marked as a Suspicious

In Dataset Marked - 150 Users are Suspicious Out of 1000 Users

Finding Fraud Label - Claim Income Ratio Above 0.7 AND Day since Issuance (Diff of Policy Issue Date & Claim Date)- below 365 Days AND

Isolation Anomaly marked as a Anomaly Means Marked as a Fraud

In Dataset - Out of 150 Suspicious Users 26 Marked as a Fraud

Feature Selection

Essential features for detecting fraud:

- **Financial behavior:**
  - `'Claim_Amount'` — High or unusual claim amounts can signal fraud.
  - `'Annual_Income'` — Comparing income with claims helps spot inconsistencies.
  - `'Claim_to_Income_Ratio'` — A direct indicator of how large the claim is relative to the policyholder's income. High ratios often correlate with suspicious behavior.

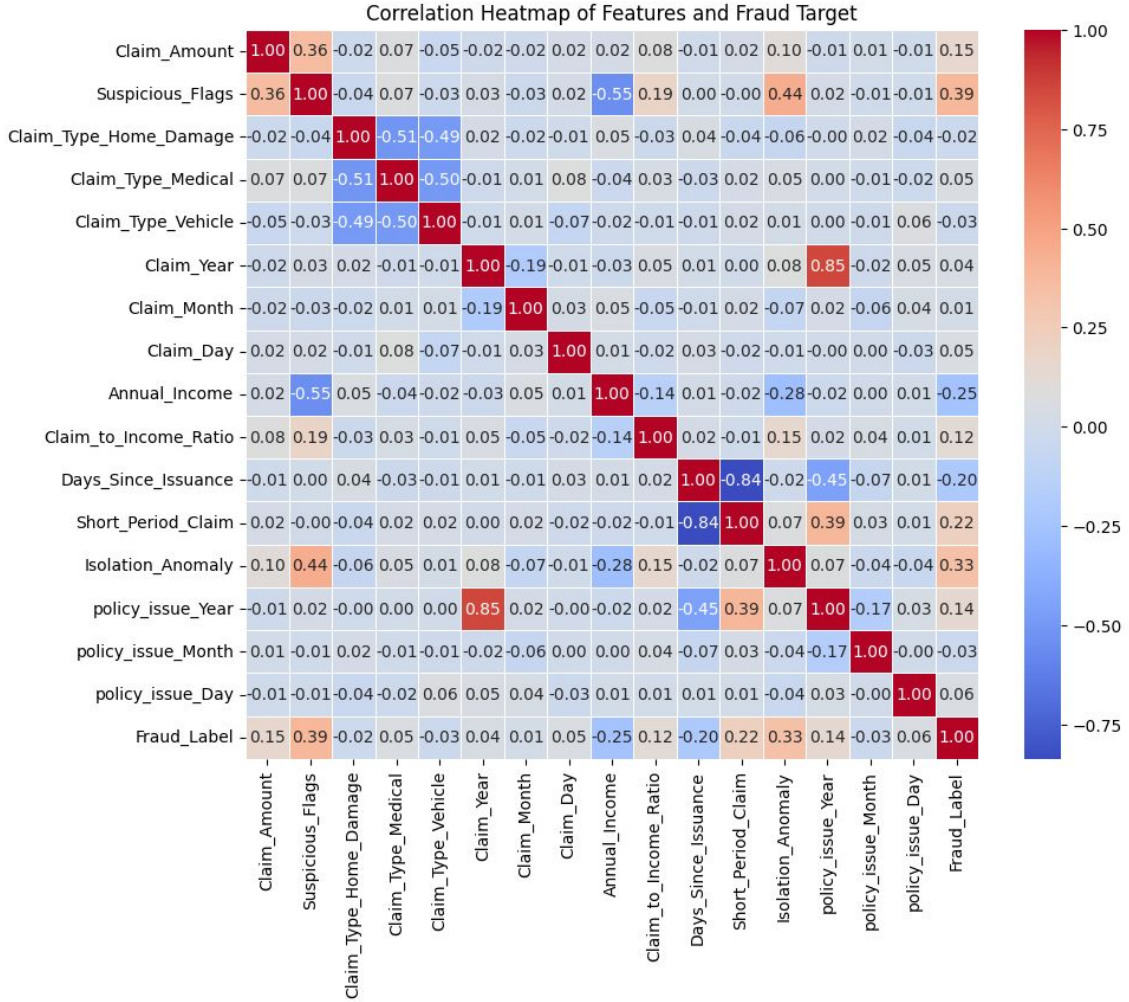
Timing and policy details:

- `'Claim_Year'`, `'Claim_Month'`, `'Claim_Day'` — Timing patterns can reveal fraud trends (like end-of-year or high-claim periods).
- `'Policy_Issuance_Date'` — Used to calculate the gap between policy start and claim date.
- `'Days_Since_Issuance'` — Early claims after policy issuance can be a major red flag.
- `'Short_Period_Claim'` — Indicates whether the claim happened soon after the policy was issued.
- `'policy_issue_Year'`, `'policy_issue_Month'`, `'policy_issue_Day'` — Useful if you want to see patterns based on when policies were issued.

Anomaly detection and flags:

- `'Suspicious_Flags'` — Directly highlights potentially suspicious activity.
- `'Elliptic_Anomaly'` — Outlier detection from one anomaly detection method.
- `'Isolation_Anomaly'` — Another outlier detection feature capturing rare or isolated behavior.

```
feature = df3_upd[['Claim_Amount', 'Suspicious_Flags', 'Claim_Type_Home_Damage', 'Claim_Type_Medical', 'Claim_Type_Vehicle',
 'Claim_Year', 'Claim_Month', 'Claim_Day', 'Annual_Income', 'Claim_to_Income_Ratio',
 'Days_Since_Issuance', 'Short_Period_Claim', 'Isolation_Anomaly',
 'policy_issue_Year', 'policy_issue_Month', 'policy_issue_Day']]

target = df3_upd['Fraud_Label']
```

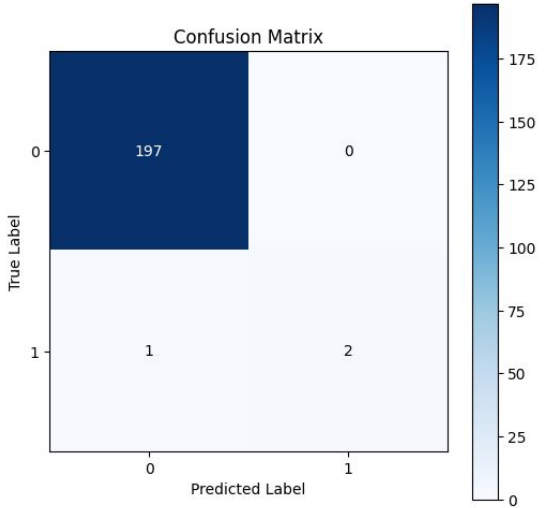

Correlation Heatmap of Features and Fraud Target
Claim_Amount 1.00 0.36 -0.02 0.07 -0.05 -0.02 -0.02 0.02 0.02 0.08 -0.01 0.02 0.10 -0.01 0.01 -0.01 0.15
Suspicious_Flags 0.36 1.00 -0.04 0.07 -0.03 0.03 -0.03 0.02 -0.55 0.19 0.00 -0.00 0.44 0.02 -0.01 -0.01 0.39
Claim_Type_Home_Damage -0.02 -0.04 1.00 -0.51 -0.49 0.02 -0.02 -0.01 0.05 -0.03 0.04 -0.04 -0.06 -0.00 0.02 -0.04 -0.02
Claim_Type_Medical 0.07 0.07 -0.51 1.00 -0.50 -0.01 0.01 0.08 -0.04 0.03 -0.03 0.02 0.05 0.00 -0.01 -0.02 0.05
Claim_Type_Vehicle -0.05 -0.03 -0.49 -0.50 1.00 -0.01 0.01 -0.07 -0.02 -0.01 -0.01 0.02 0.01 0.00 -0.01 0.06 -0.03
Claim_Year -0.02 0.03 0.02 -0.01 -0.01 1.00 -0.19 -0.01 -0.03 0.05 0.01 0.00 0.08 0.85 -0.02 0.05 0.04
Claim_Month -0.02 -0.03 -0.02 0.01 0.01 -0.19 1.00 0.03 0.05 -0.05 -0.01 0.02 -0.07 0.02 -0.06 0.04 0.01
Claim_Day 0.02 0.02 -0.01 0.08 -0.07 -0.01 0.03 1.00 0.01 -0.02 0.03 -0.02 -0.01 -0.00 0.00 -0.03 0.05
Annual_Income 0.02 -0.55 0.05 -0.04 -0.02 -0.03 0.05 0.01 1.00 -0.14 0.01 -0.02 -0.28 -0.02 0.00 0.01 -0.25
Claim_to_Income_Ratio 0.08 0.19 -0.03 0.03 -0.01 0.05 -0.05 -0.02 -0.14 1.00 0.02 -0.01 0.15 0.02 0.04 0.01 0.12
Days_Since_Issuance -0.01 0.00 0.04 -0.03 -0.01 0.01 -0.01 0.03 0.01 0.02 1.00 -0.84 -0.02 -0.45 -0.07 0.01 -0.20
Short_Period_Claim 0.02 -0.00 -0.04 0.02 0.02 0.00 0.02 -0.02 -0.02 -0.01 -0.84 1.00 0.07 0.39 0.03 0.01 0.22
Isolation_Anomaly 0.10 0.44 -0.06 0.05 0.01 0.08 -0.07 -0.01 -0.28 0.15 -0.02 0.07 1.00 0.07 -0.04 -0.04 0.33
policy_issue_Year -0.01 0.02 -0.00 0.00 0.00 0.85 0.02 -0.00 -0.02 0.02 -0.45 0.39 0.07 1.00 -0.17 0.03 0.14
policy_issue_Month 0.01 -0.01 0.02 -0.01 -0.01 -0.02 -0.06 0.00 0.00 0.04 -0.07 0.03 -0.04 -0.17 1.00 -0.00 -0.03
policy_issue_Day -0.01 -0.01 -0.04 -0.02 0.06 0.05 0.04 -0.03 0.01 0.01 0.01 0.01 -0.04 0.03 -0.00 1.00 0.06
Fraud_Label 0.15 0.39 -0.02 0.05 -0.03 0.04 0.01 0.05 -0.25 0.12 -0.20 0.22 0.33 0.14 -0.03 0.06 1.00
Claim_Amount
Suspicious_Flags
Claim_Type_Home_Damage
Claim_Type_Medical
Claim_Type_Vehicle
Claim_Year
Claim_Month
Claim_Day
Annual_Income
Claim_to_Income_Ratio
Days_Since_Issuance
Short_Period_Claim
Isolation_Anomaly
policy_issue_Year
policy_issue_Month
policy_issue_Day
Fraud_Label
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

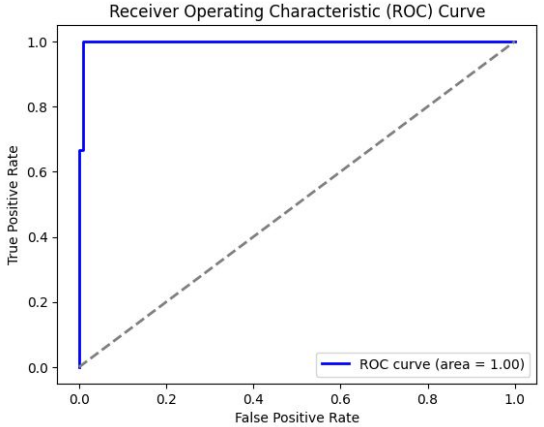
High Contribute Columns

1. **Claim_to_Income_Ratio** (Correlation: 0.45)
2. **Suspicious_Flags** (Correlation: 0.39)
3. **Isolation_Anomaly** (Correlation: 0.33)
4. **Short_Period_Claim** (Correlation: 0.25)
5. **Claim_Amount** (Correlation: 0.15)

1) Logistic Regression

```
Training Accuracy: 0.9888
Test Accuracy: 0.9950
Confusion Matrix:
[[197   0]
 [  1   2]]
True Positives: 2
True Negatives: 197
False Positives: 0
False Negatives: 1
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       197
           1       1.00      0.67      0.80         3

    accuracy                           0.99       200
   macro avg       1.00      0.83      0.90       200
weighted avg       1.00      0.99      0.99       200
```





Training Accuracy: 0.9213

Test Accuracy: 0.8950

Model is Genraized Well

True Positives: 32,

True Negatives: 147,

False Positives: 2,

False Negatives: 19

False Positive Rate is Low its Actually Good because its does not mark Genuine claim as Fraud its Create a Issues,

macro avg - taken overall equally

Precesion = 0.91

Recall = 0.81

F1_Score = 0.84

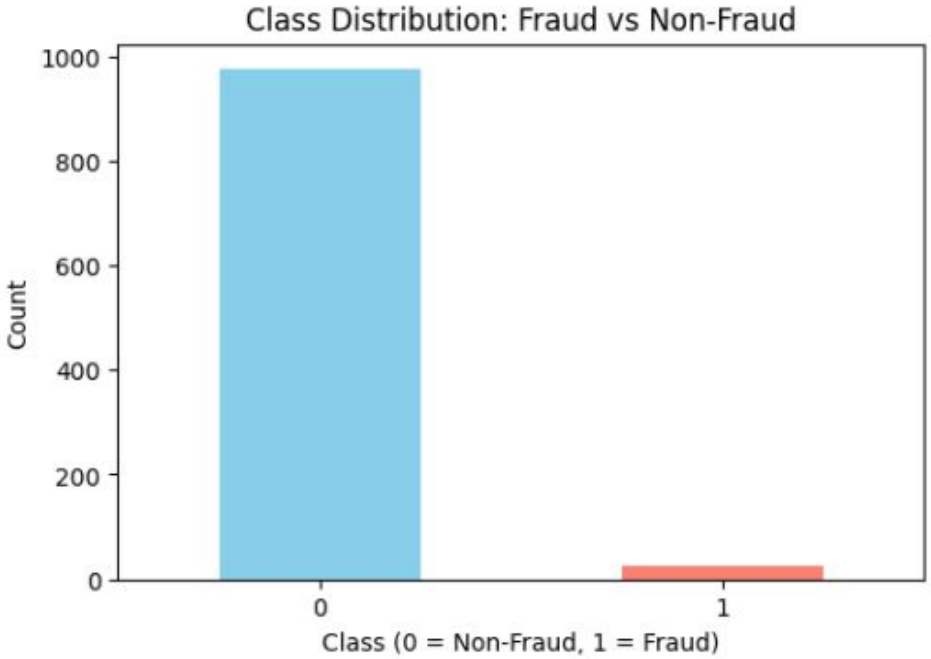
weighted avg - Concentrate Based on Majority or Minority to focused

Precesion = 0.90

Recall = 0.90

F1_Score = 0.89

```
Class Counts:
Fraud_Label
0    974
1     26
Name: count, dtype: int64

Class Percentages:
Fraud_Label
0    97.4
1     2.6
Name: proportion, dtype: float64
```



For Using SMOTE to Focus on Correct Prediction - to avodi Data Leakage or Data Memorize but SMOTE using Synthetic Dataset for Oversampling so am using Cross Validation

Cross Validation

Focusing

Class -0 is 79.7

Class -1 is 20.3

```
Class Counts:
Fraud_Label
0    974
1     26
Name: count, dtype: int64

Class Percentages:
Fraud_Label
0    97.4
1     2.6
Name: proportion, dtype: float64
```

```
Training Accuracy: 0.9900
Test Accuracy: 0.9850
Confusion Matrix:
[[194   1]
 [  2   3]]
True Positives: 3
True Negatives: 194
False Positives: 1
False Negatives: 2
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       195
           1       0.75      0.60      0.67         5

    accuracy                           0.98       200
   macro avg       0.87      0.80      0.83       200
weighted avg       0.98      0.98      0.98       200
```

Logistic Regression is Simple so after get Good Accuracy Some Prediction Shows Wrong

2) SO Try Random Forest Classifier

```
Training Accuracy: 1.0000
Test Accuracy: 1.0000
Confusion Matrix:
[[195   0]
 [  0   5]]
True Positives: 5
True Negatives: 195
False Positives: 0
False Negatives: 0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       195
           1       1.00      1.00      1.00         5

    accuracy                           1.00       200
   macro avg       1.00      1.00      1.00       200
weighted avg       1.00      1.00      1.00       200
```

Model is Overfit so use Hyperparameter Tuning for Random Forest

```python
# Define the parameter grid for RandomForest
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 7, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 3, 5],
    'max_features': ['sqrt', 'log2'],
    'class_weight': [{0: 0.3, 1: 0.7}, 'balanced']
}
```

Find a Best Parameter For out Model Using GridSearchCV

Fitting 5 folds for each of 216 candidates, totalling 1080 fits

```
Best Parameters: {'class weight': {0: 0.3, 1: 0.7}, 'max depth': 5,
'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2,
'n_estimators': 100}
```

Still Overfit after Using Hyper Parameter Tuning
And SMOTE Bethod
After Trying :- SVM, KNN, PyTorch Still Overfit

Comparing Logistic Regression is Good So Using Logistic Regression

Save Model

```
[ ]  1 # Save the trained model
     2 joblib.dump(log_reg, 'logistic_regression_fraud_model.pkl')
     3 print("Model saved as logistic_regression_fraud_model.pkl")
```

```
Model saved as logistic_regression_fraud_model.pkl
```

Now Prediction Work Perfectly

Streamlit

Reduce Customer FIlling Option

```
# Feature list used during training
training_features = ['Claim_Amount', 'Suspicious_Flags', 'Claim_Type_Home_Damage', 'Claim_Type_Medical',
                     'Claim_Type_Vehicle', 'Claim_Year', 'Claim_Month', 'Claim_Day', 'Annual_Income',
                     'Claim_to_Income_Ratio', 'Days_Since_Issuance', 'Short_Period_Claim', 'Isolation_Anoma
                     'policy_issue_Year', 'policy_issue_Month', 'policy_issue_Day']

# Function to calculate days since policy issuance
def calculate_days_since_issuance(claim_date, policy_issue_date):
    delta = claim_date - policy_issue_date
    return delta.days

# Function to prepare input data
def prepare_input_data(user_input):
    # Scale Claim_Amount and Annual_Income
    scaled_values = scaler.transform([[user_input['Annual_Income'], user_input['Claim_Amount']]])
    user_input['Annual_Income'], user_input['Claim_Amount'] = scaled_values[0]

    # Map Claim_Type to one-hot encoded columns
    claim_type_mapping = {
        'Home Damage': [1, 0, 0],
        'Medical': [0, 1, 0],
        'Vehicle': [0, 0, 1]
    }
```

```
user_input['Claim_Type_Home_Damage'], user_input['Claim_Type_Medical'], user_input['Claim_Type_Vehicle'] =
claim_type_mapping[user_input['Claim_Type']]

# Calculate engineered features
user_input['Claim_to_Income_Ratio'] = user_input['Claim_Amount'] / user_input['Annual_Income']
user_input['Suspicious_Flags'] = 1 if user_input['Claim_to_Income_Ratio'] > 0.5 else 0

# Calculate days since policy issuance
claim_date = date(user_input['Claim_Year'], user_input['Claim_Month'], user_input['Claim_Day'])
policy_issue_date = date(user_input['policy_issue_Year'], user_input['policy_issue_Month'], user_input['policy_issue_Day'])
user_input['Days_Since_Issuance'] = calculate_days_since_issuance(claim_date, policy_issue_date)

# Determine if it's a short-period claim
user_input['Short_Period_Claim'] = 1 if user_input['Days_Since_Issuance'] < 365 else 0
```

```
# Updated Isolation Anomaly detection using refit approach
sample_data = pd.DataFrame([[user_input['Claim_Amount'], user_input['Claim_Year'], user_input['Claim_Month'],
                             user_input['Claim_Day'], user_input['Claim_to_Income_Ratio'], user_input['Days_Since_Issuance']]])
iso_forest = IsolationForest(contamination=0.20, random_state=42)
user_input['Isolation_Anomaly'] = iso_forest.fit_predict(sample_data)[0]

# Align feature names and order
input_data = pd.DataFrame([user_input])[training_features]

return input_data, user_input, claim_date, policy_issue_date
```

```
st.write(f"Claim Date: {claim_date}")
st.write(f"Policy Issue Date: {policy_issue_date}")
st.write(f"Days Since Issuance: {user_input_features['Days_Since_Issuance']}")

# Show calculated feature values
st.write(f"Claim to Income Ratio: {user_input_features['Claim_to_Income_Ratio']}")
st.write(f"Suspicious Flags: {'True' if user_input_features['Suspicious_Flags'] == 1 else 'False'}")
st.write(f"Short Period Claim: {'True' if user_input_features['Short_Period_Claim'] == 1 else 'False'}")
st.write(f"Isolation Anomaly: {'Anomaly' if user_input_features['Isolation_Anomaly'] == -1 else 'Normal'}")
st.write(f"Scaled Annual Income: {user_input_features['Annual_Income']}")
st.write(f"Scaled Claim Amount: {user_input_features['Claim_Amount']}")
```

Predicted class: FRAUD

Claim Date: 2001-07-01

Policy Issue Date: 2001-01-01

Days Since Issuance: 181

Claim to Income Ratio: 14.095162951949549

Suspicious Flags: True

Short Period Claim: True

Isolation Anomaly: Normal

Scaled Annual Income: 0.2586108445721216

Scaled Claim Amount: 3.645161995385351

Predicted class: GENUINE

Claim Date: 2001-07-01

Policy Issue Date: 2001-01-01

Days Since Issuance: 181

Claim to Income Ratio: 6.269538744489819

Suspicious Flags: True

Short Period Claim: True

Isolation Anomaly: Normal

Scaled Annual Income: 0.2586108445721216

Scaled Claim Amount: 1.6213707097901509

Conclusion

This project successfully builds an insurance fraud detection app using Streamlit and a logistic regression model. Through feature engineering (like Claim-to-Income ratio and days since policy issuance) and anomaly detection (Isolation Forest), it enhances the model’s ability to detect suspicious claims. The app scales user input, aligns features, and provides detailed prediction insights, offering a clear and user-friendly experience.

Hands-on Experience

Building this app provided deep exposure to feature engineering, anomaly detection, and model deployment. I gained practical skills in integrating data preprocessing, It also strengthened my understanding of scaling, one-hot encoding, and real-time prediction workflows.

Thank You