Research Paper: Credit Card Fraud Detection Using Machine Learning

By Saqib Ali - PGD DS with AI Batch - 5 (DL Project)

Abstract

This research paper delves into building a machine learning model to detect credit card fraud. The data is preprocessed, balanced using undersampling and oversampling techniques, and multiple machine learning algorithms are applied to predict fraudulent transactions. The model is evaluated using various performance metrics, and the results demonstrate the efficacy of machine learning in detecting fraud.

Introduction

Credit card fraud is a significant concern in the financial sector. Machine learning can be used to create models that automatically detect fraudulent activities. This project utilizes a real-world dataset containing anonymized features and applies machine learning models to predict fraud with high accuracy. The data is imbalanced, as fraudulent transactions are rare compared to legitimate ones, so we apply techniques to address this imbalance before model building.

Data Loading and Exploration

```
import pandas as pd
data = pd.read_csv("creditcard.csv")
data.head()
```

Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.2
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.2
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.1
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.0

5 rows × 31 columns

Explanation: We first load the dataset using **pandas** and take a look at the first few rows to understand its structure.

```
pd.options.display.max_columns = None
data.tail()
```

Out[6]:

Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
2786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	4.356170
2787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	-0.975926
2788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	-0.484782
2788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	-0.399126
2792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	-0.915427

Explanation: Display all columns and inspect the last few rows to ensure the dataset is read correctly.

```
data.shape
```

```
print("Number of columns: {}".format(data.shape[1]))
```

```
print("Number of rows: {}".format(data.shape[0]))
```

```
In [7]: data.shape

Out[7]: (284807, 31)

In [9]: print("Number of columns: {}".format(data.shape[1]))
        print("Number of rows: {}".format(data.shape[0]))

Number of columns: 31
Number of rows: 284807
```

Explanation: Here, we check the shape of the dataset, which gives us an idea of how many rows and columns we are dealing with.

Data Preprocessing

1. **Missing Value Check**

```
data.isnull().sum()
```

Explanation: This step checks if any columns have missing values. Since this dataset contains no missing values, no further imputation is needed.

```
In [11]: data.isnull().sum()

Out[11]: Time      0
         V1        0
         V2        0
         V3        0
         V4        0
         V5        0
         V6        0
         V7        0
         V8        0
         V9        0
         V10       0
         V11       0
         V12       0
         V13       0
         V14       0
         V15       0
         V16       0
         V17       0
         V18       0
         V19       0
         V20       0
         V21       0
         V22       0
         V23       0
         V24       0
         V25       0
         V26       0
         V27       0
         V28       0
         Amount    0
```

2. **Standardizing the 'Amount' Column**

```
from sklearn.preprocessing import StandardScaler
```

```
sc = StandardScaler()
```

```
data['Amount'] = sc.fit_transform(pd.DataFrame(data['Amount']))
data.head()
```

```
In [12]: from sklearn.preprocessing import StandardScaler

In [13]: sc = StandardScaler()
         data['Amount'] = sc.fit_transform(pd.DataFrame(data['Amount']))

In [14]: data.head()
```

Out[14]:

V18	V19	V20	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
.025791	0.403993	0.251412	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	0.244964	0
.183361	-0.145783	-0.069083	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	-0.342475	0
.121359	-2.261857	0.524980	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	1.160686	0
.965775	-1.232622	-0.208038	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	0.140534	0
.038195	0.803487	0.408542	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	-0.073403	0

Explanation: The 'Amount' column is standardized using **StandardScaler** to ensure that the machine learning algorithms handle it appropriately.

3. **Dropping the 'Time' Column**

```
data = data.drop(['Time'], axis = 1)
data.head()
```

```
In [15]: data = data.drop(['Time'], axis =1)

In [16]: data.head()
```

Out[16]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.551600	-0.617801
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.065235
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.066084
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.178228
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.538196

Explanation: The 'Time' column is dropped as it is irrelevant for fraud detection

4. **Removing Duplicates**

```
data.duplicated().any()
data = data.drop_duplicates()
data.shape
```

```
In [17]: data.duplicated().any()

Out[17]: True

In [18]: data = data.drop_duplicates()

In [19]: data.shape

Out[19]: (275663, 30)
```

Explanation: We remove any duplicate rows to ensure cleaner data for the model.

Data Imbalance and Visualization

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.countplot(data['Class'])
plt.show()
```

Explanation: The **countplot** shows the imbalance between fraud and non-fraud transactions, highlighting the need for undersampling or oversampling techniques.

Model Building and Evaluation

1. **Data Splitting**

```
from sklearn.model_selection import train_test_split
X = data.drop('Class', axis=1)
y = data['Class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Explanation: The dataset is split into training and testing sets. We reserve 20% of the data for testing, while the rest is used to train the model.

2. **Logistic Regression and Decision Tree Classifiers**

```
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score


classifier = {
    "Logistic Regression": LogisticRegression(),
    "Decision Tree Classifier": DecisionTreeClassifier()
}


for name, clf in classifier.items():
    print(f"\n=========={name}===========")
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
    print(f"Precision: {precision_score(y_test, y_pred)}")
    print(f"Recall: {recall_score(y_test, y_pred)}")
    print(f"F1 Score: {f1_score(y_test, y_pred)}")
```

```
In [30]:
classifier = {
    "Logistic Regression": LogisticRegression(),
    "Decision Tree Classifier": DecisionTreeClassifier()
}

for name, clf in classifier.items():
    print(f"\n=========={name}===========")
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(f"\n Accuaracy: {accuracy_score(y_test, y_pred)}")
    print(f"\n Precision: {precision_score(y_test, y_pred)}")
    print(f"\n Recall: {recall_score(y_test, y_pred)}")
    print(f"\n F1 Score: {f1_score(y_test, y_pred)}")
```

```
==========Logistic Regression===========

 Accuaracy: 0.9992200678359603

 Precision: 0.8870967741935484

 Recall: 0.6043956043956044

 F1 Score: 0.718954248366013

==========Decision Tree Classifier===========

 Accuaracy: 0.9990386882629279

 Precision: 0.6862745098039216

 Recall: 0.7692307692307693

 F1 Score: 0.7253886010362693
```

Explanation: Two classifiers, **Logistic Regression** and **Decision Tree**, are trained and evaluated. We use metrics such as accuracy, precision, recall, and F1-score to measure performance. These metrics help evaluate how well the model performs in detecting fraud.

Handling Data Imbalance with Undersampling and Oversampling

1. **Undersampling: Reducing the majority class to balance the dataset.**

```
normal = data[data['Class']==0]
```

```
fraud = data[data['Class']==1]
```

```
normal_sample = normal.sample(n=473)
```

```
new_data = pd.concat([normal_sample, fraud], ignore_index=True)
```

```
In [31]: # Undersampling

In [32]: normal = data[data['Class']==0]
         fraud = data[data['Class']==1]

In [33]: normal.shape

Out[33]: (275190, 30)

In [34]: fraud.shape

Out[34]: (473, 30)

In [35]: normal_sample = normal.sample(n=473)

In [36]: normal_sample.shape

Out[36]: (473, 30)

In [37]: new_data = pd.concat([normal_sample,fraud], ignore_index=True)

In [38]: new_data.head()

Out[38]:       V1       V2       V3       V4       V5       V6       V7       V8       V9      V10      V11      V12
```

```
In [37]: new_data = pd.concat([normal_sample,fraud], ignore_index=True)

In [38]: new_data.head()
```

Out[38]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	
0	0.390664	1.973736	-1.945026	1.789083	0.647230	-1.744105	0.592489	0.198540	-0.687046	-1.482981	1.416687	0.234829	0.62
1	-0.643022	-1.571045	0.387174	0.346115	1.384183	0.010196	-0.806426	0.443410	1.084875	-0.531256	-0.261706	1.138270	0.01
2	-2.549290	1.528338	-1.206697	-1.737109	1.953693	-0.176467	1.864754	-2.206843	1.072224	2.215222	0.359018	-0.491519	-1.04
3	1.993656	0.124010	-1.549448	1.300218	0.347976	-0.903320	0.455070	-0.210637	0.111435	0.464175	0.605472	0.428546	-1.51
4	-0.209858	0.173233	1.232594	3.246544	3.453106	-1.080207	-4.346397	-1.262771	-0.039137	1.726905	-1.759217	0.216746	-0.01

```
In [39]: new_data['Class'].value_counts()

Out[39]: 0    473
         1    473
         Name: Class, dtype: int64

In [40]: X = new_data.drop('Class', axis = 1)
         y= new_data['Class']

In [41]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
```

```
In [42]: classifier = {
             "Logistic Regression": LogisticRegression(),
             "Decision Tree Classifier": DecisionTreeClassifier()
         }

         for name, clf in classifier.items():
             print(f"\n=========={name}==========")
             clf.fit(X_train, y_train)
             y_pred = clf.predict(X_test)
             print(f"\n Accuaracy: {accuracy_score(y_test, y_pred)}")
             print(f"\n Precision: {precision_score(y_test, y_pred)}")
             print(f"\n Recall: {recall_score(y_test, y_pred)}")
             print(f"\n F1 Score: {f1_score(y_test, y_pred)}")

==========Logistic Regression==========

 Accuaracy: 0.9263157894736842

 Precision: 0.9489795918367347

 Recall: 0.9117647058823529

 F1 Score: 0.9300000000000002

==========Decision Tree Classifier==========
```

Explanation: We create a balanced dataset by undersampling the majority class (normal transactions). The new dataset contains an equal number of fraud and normal transactions.

2. **Oversampling with SMOTE: Using SMOTE to generate synthetic samples.**

```
from imblearn.over_sampling import SMOTE

X_res, y_res = SMOTE().fit_resample(X, y)

X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test_size=0.2, random_state=42)
```

```
In [ ]: # OVERSAMPLING

In [49]: X = data.drop('Class', axis = 1)
         y= data['Class']

In [50]: X.shape

Out[50]: (275663, 29)

In [51]: y.shape

Out[51]: (275663,)

In [52]: from imblearn.over_sampling import SMOTE

In [53]: X_res, y_res = SMOTE().fit_resample(X,y)

In [54]: y_res.value_counts()

Out[54]: 0    275190
         1    275190
         Name: Class, dtype: int64

In [55]: X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test_size = 0.2, random_state = 42)
```

```
In [56]: classifier = {
             "Logistic Regression": LogisticRegression(),
             "Decision Tree Classifier": DecisionTreeClassifier()
         }

         for name, clf in classifier.items():
             print(f"\n=========={name}===========")
             clf.fit(X_train, y_train)
             y_pred = clf.predict(X_test)
             print(f"\n Accuaracy: {accuracy_score(y_test, y_pred)}")
             print(f"\n Precision: {precision_score(y_test, y_pred)}")
             print(f"\n Recall: {recall_score(y_test, y_pred)}")
             print(f"\n F1 Score: {f1_score(y_test, y_pred)}")

==========Logistic Regression===========

 Accuaracy: 0.9438115483847523

 Precision: 0.9729326513213982

 Recall: 0.9129502027162155

 F1 Score: 0.9419875252075225

==========Decision Tree Classifier===========

 Accuaracy: 0.9982012427777173

 Precision: 0.9976391537274131

 Recall: 0.9987637037979746

 F1 Score: 0.9982011120398299
```

Explanation: **SMOTE** is used to oversample the minority class by generating synthetic examples. This helps the model to learn better from an imbalanced dataset.

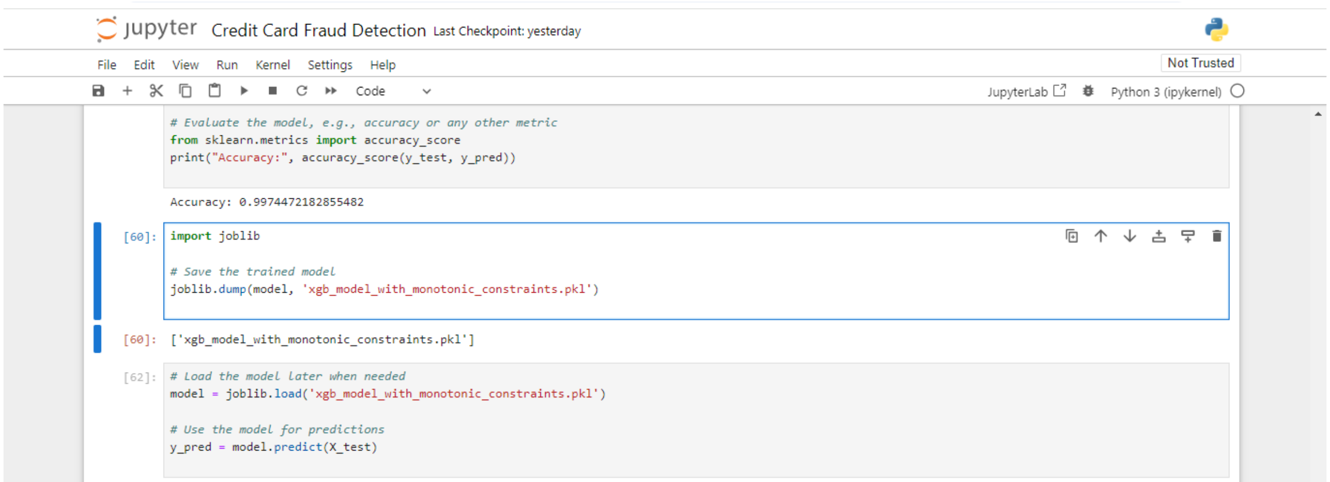
Model Saving and Deployment

```
import xgboost as xgb

# Define the model with monotonic constraints and additional parameters
model = xgb.XGBClassifier(
    monotone_constraints="(1, 0, -1)",
    learning_rate=0.1,  # learning rate (eta)
    n_estimators=100,  # number of trees (boosting rounds)
    max_depth=6,  # depth of the trees
    random_state=42  # for reproducibility
)

# Fit the model with training data
model.fit(X_train, y_train)

# We can use the model for predictions
y_pred = model.predict(X_test)

# Evaluate the model, e.g., accuracy or any other metric
from sklearn.metrics import accuracy_score
print("Accuracy:", accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9974472182855482
```

```
import joblib
# Save the trained model
joblib.dump(model, 'xgb_model_with_monotonic_constraints.pkl'
```

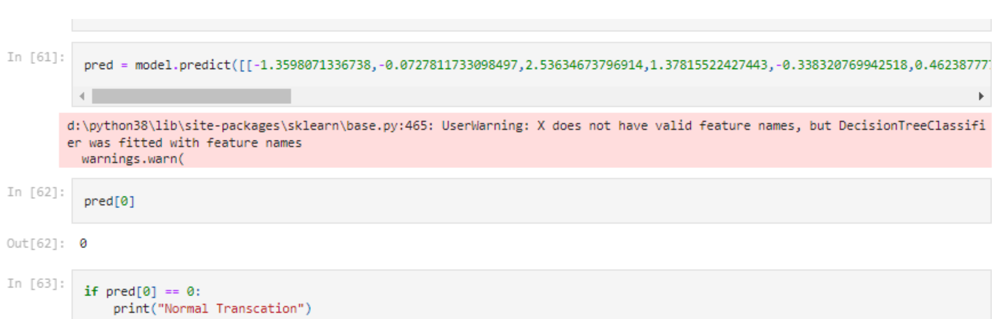
['xgb_model_with_monotonic_constraints.pkl']



Explanation: The trained model is saved using **joblib** for future use. The model can later be loaded and used to make predictions.

Predicting Fraudulent Transactions

```
pred = model.predict([[-1.3598071336738,-0.0727811733098497,2.53634673796914,1.37815522427443,-0.338320769942518,0.462387777762292,0.239598554061257,0.0986979012610507,0.363786969611213,0.0907941719789316,-0.551599533260813,-0.617800855762348,-0.991389847235408,-0.311169353699879,1.46817697209427,-0.470400525259478,0.207971241929242,0.0257905801985591,0.403992960255733,0.251412098239705,-0.018306777944153,0.277837575558899,-0.110473910188767,0.0669280749146731,0.128539358273528,-0.189114843888824,0.133558376740387,-0.0210530534538215,149.62]])

if pred[0] == 0:
    print("Normal Transaction")
else:
    print("Fraud Transaction")
```



Explanation: A specific transaction is tested, and the model predicts whether it's a normal or fraudulent transaction based on its features.

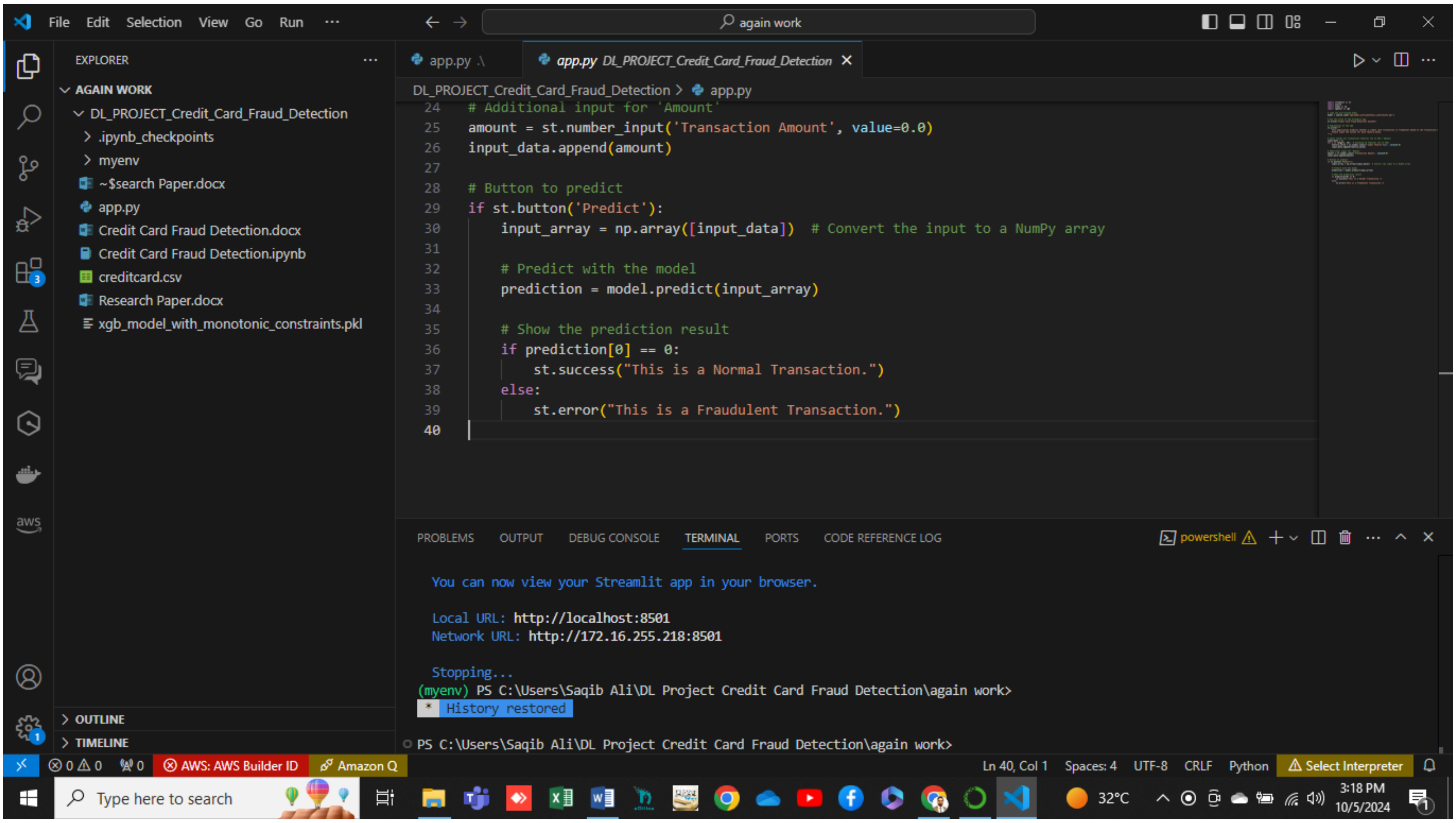
Streamlit App for UI

To make UI this model using **Streamlit**, we can create an interactive app that allows users to input transaction data and get predictions in real-time. Here is the basic code structure for the Streamlit app:

1. **Streamlit Script (File name app.py)**

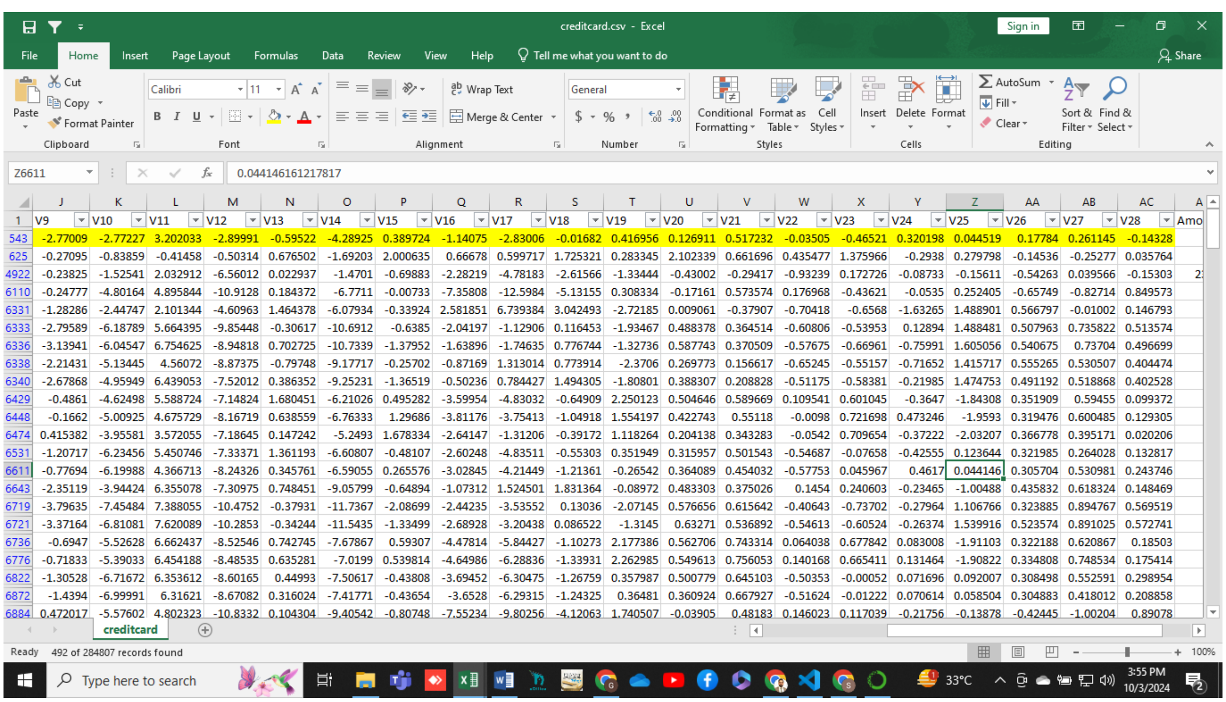
```
import streamlit as st
import joblib
import numpy as np
import xgboost as xgb

# Load the pre-trained model
model = joblib.load('xgb_model_with_monotonic_constraints.pkl')

# Set the title of the Streamlit app
st.title("Credit Card Fraud Detection System")

# Explanation of the app
st.write("""
    This application predicts whether a credit card transaction is fraudulent based on the transaction's features.
    Please input the values for each feature below.
""")

# Input fields for transaction features (V1 to V28 + Amount)
input_data = []
for i in range(1, 29):  # Assuming 28 features (V1 to V28)
    feature_value = st.number_input(f'Input feature V{i}', value=0.0)
    input_data.append(feature_value)

# Additional input for 'Amount'
amount = st.number_input('Transaction Amount', value=0.0)
input_data.append(amount)

# Button to predict
if st.button('Predict'):
    input_array = np.array([input_data])  # Convert the input to a NumPy array

    # Predict with the model
    prediction = model.predict(input_array)

    # Show the prediction result
    if prediction[0] == 0:
        st.success("This is a Normal Transaction.")
    else:
        st.error("This is a Fraudulent Transaction.")

```

Files in a Directory



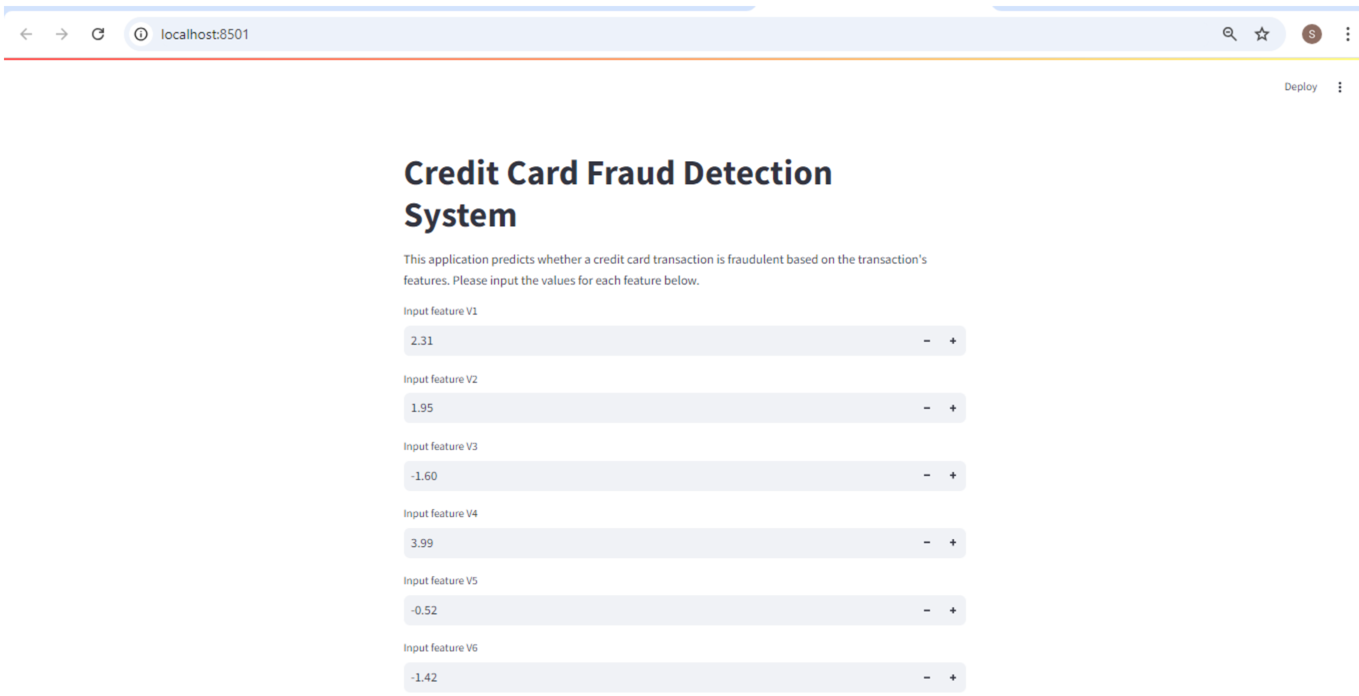
2 Run Streamlit

Manually input the below highlighted 28 features of a fraud transaction (Category 1 – Fraud Transacton)

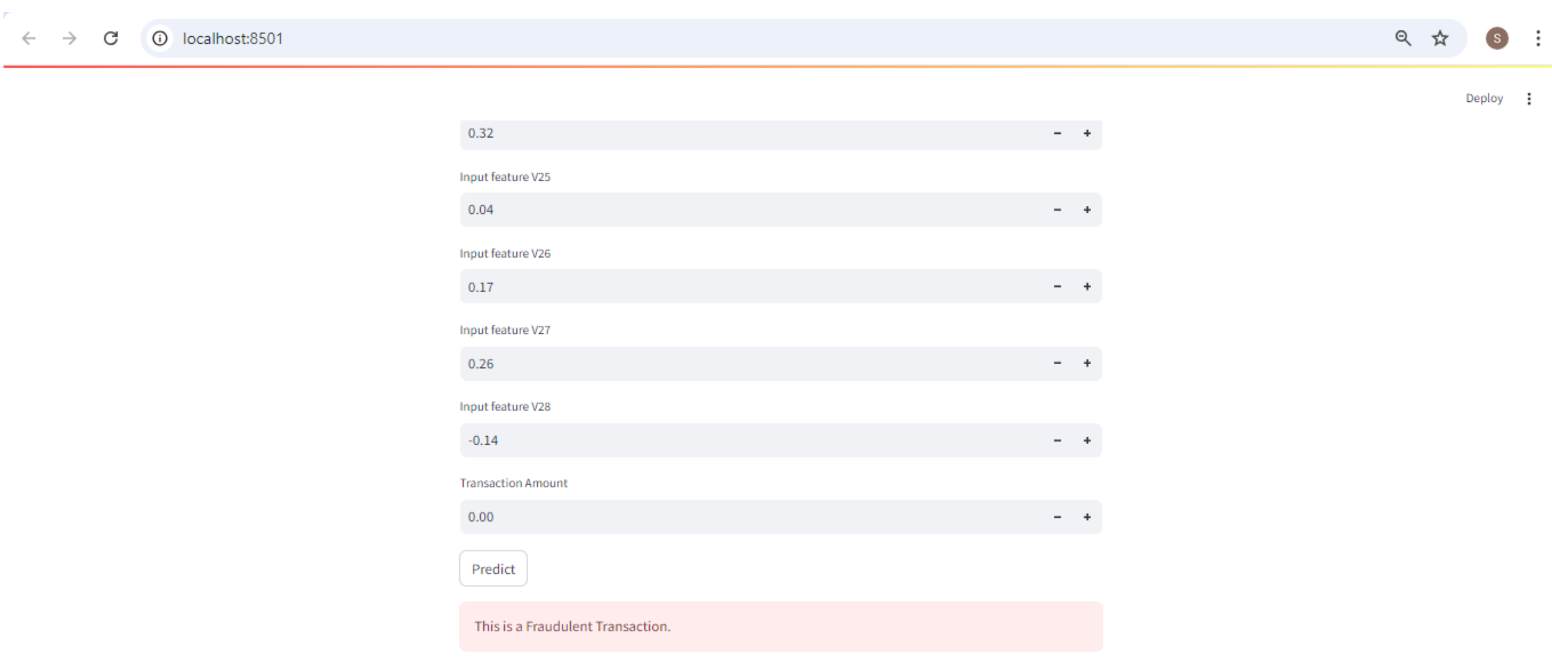


	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26	V27	V28
543	-2.77009	-2.77227	3.202033	-2.89991	-0.59522	-4.28925	0.389724	-1.14075	-2.83006	-0.01682	0.416956	0.126911	0.517232	-0.03505	-0.46521	0.320198	0.044519	0.17784	0.261145	-0.14328
625	-0.27095	-0.83859	-0.41458	-0.50314	0.676502	-1.69203	2.000635	0.66678	0.599717	1.725321	0.283345	2.102339	0.661696	0.435477	1.375966	-0.2938	0.279798	-0.14536	-0.25277	0.035764
4922	-0.23825	-1.52541	2.032912	-6.56012	0.022937	-1.4701	-0.69883	-2.28219	-4.78183	-2.61566	-1.33444	-0.43002	-0.29417	-0.93239	0.172726	-0.08733	-0.15611	-0.54263	0.039566	-0.15303
6110	-0.24777	-4.80164	4.895844	-10.9128	0.184372	-6.7711	-0.00733	-7.35808	-12.5984	-5.13155	0.308334	-0.17161	0.573574	0.176968	-0.43621	-0.0535	0.252405	-0.65749	-0.82714	0.849573
6331	-1.28286	-2.44747	2.101344	-4.60963	1.464378	-6.07934	-0.33924	2.581851	6.739384	3.042493	-2.72185	0.009061	-0.37907	-0.70418	-0.6568	-1.63265	1.488901	0.566797	-0.01002	0.146793
6333	-2.79589	-6.18789	5.664395	-9.85448	-0.30617	-10.6912	-0.6385	-2.04197	-1.12906	0.116453	-1.93467	0.488378	0.364514	-0.60806	-0.53953	0.12894	1.488481	0.507963	0.735822	0.513574
6336	-3.13941	-6.04547	6.754625	-8.94818	0.702725	-10.7339	-1.37952	-1.63896	-1.74635	0.776744	-1.32736	0.587743	0.370509	-0.57675	-0.66961	-0.75991	1.605056	0.540675	0.73704	0.496699
6338	-2.21431	-5.13445	4.56072	-8.87375	-0.79748	-9.17717	-0.25702	-0.87169	1.313014	0.773914	-2.3706	0.269773	0.156617	-0.65245	-0.55157	-0.71652	1.415717	0.555265	0.530507	0.404474
6340	-2.67868	-4.95949	6.439053	-7.52012	0.386352	-9.25231	-1.36519	-0.50236	0.784427	1.494305	-1.80801	0.388307	0.208828	-0.51175	-0.58381	-0.21985	1.474753	0.491192	0.518868	0.402528
6429	-0.4861	-4.62498	5.588724	-7.14824	1.680451	-6.21026	0.495282	-3.59954	-4.83032	-0.64909	2.250123	0.504646	0.589669	0.109541	0.601045	-0.3647	-1.84308	0.351909	0.59455	0.099372
6448	-0.1662	-5.00925	4.675729	-8.16719	0.638559	-6.76333	1.29686	-3.81176	-3.75413	-1.04918	1.554197	0.422743	0.55118	-0.0098	0.721698	0.473246	-1.9593	0.319476	0.600485	0.129305
6474	0.415382	-3.95581	3.572055	-7.18645	0.147242	-5.2493	1.678334	-2.64147	-1.31206	-0.39172	1.118264	0.204138	0.343283	-0.0542	0.709654	-0.37222	-2.03207	0.366778	0.395171	0.020206
6531	-1.20717	-6.23456	5.450746	-7.33371	1.361193	-6.60807	-0.48107	-2.60248	-4.83511	-0.55303	0.351949	0.315957	0.501543	-0.54687	-0.07658	-0.42555	0.123644	0.321985	0.264028	0.132817
6611	-0.77694	-6.19988	4.366713	-8.24326	0.345761	-6.59055	0.265576	-3.02845	-4.21449	-1.21361	-0.26542	0.364089	0.454032	-0.57753	0.045967	0.4617	0.044146	0.305704	0.530981	0.243746
6643	-2.35119	-3.94424	6.355078	-7.30975	0.748451	-9.05799	-0.64894	-1.07312	1.524501	1.831364	-0.08972	0.483303	0.375026	0.1454	0.240603	-0.23465	-1.00488	0.435832	0.618324	0.148469
6719	-3.79635	-7.45484	7.388055	-10.4752	-0.37931	-11.7367	-2.08699	-2.44235	-3.53552	0.13036	-2.07145	0.576656	0.615642	-0.40643	-0.73702	-0.27964	1.106766	0.323885	0.894767	0.569519
6721	-3.37164	-6.81081	7.620089	-10.2853	-0.34244	-11.5435	-1.33499	-2.68928	-3.20438	0.086522	-1.3145	0.63271	0.536892	-0.54613	-0.60524	-0.26374	1.539916	0.523574	0.891025	0.572741
6736	-0.6947	-5.52628	6.662437	-8.52546	0.742745	-7.67867	0.59307	-4.47814	-5.84427	-1.10273	2.177386	0.562706	0.743314	0.064038	0.677842	0.083008	-1.91103	0.322188	0.620867	0.18503
6776	-0.71833	-5.39033	6.454188	-8.48535	0.635281	-7.0199	0.539814	-4.64986	-6.28836	-1.33931	2.262985	0.549613	0.756053	0.140168	0.665411	0.131464	-1.90822	0.334808	0.748534	0.175414
6822	-1.30528	-6.71672	6.353612	-8.60165	0.44993	-7.50617	-0.43808	-3.69452	-6.30475	-1.26759	0.357987	0.500779	0.645103	-0.50353	-0.00052	0.071696	0.092007	0.308498	0.552591	0.298954
6872	-1.4394	-6.99991	6.31621	-8.67082	0.316024	-7.41771	-0.43654	-3.6528	-6.29315	-1.24325	0.36481	0.360924	0.667927	-0.51624	-0.01222	0.070614	0.058504	0.304883	0.418012	0.208858
6884	0.472017	-5.57602	4.802323	-10.8332	0.104304	-9.40542	-0.80748	-7.55234	-9.80256	-4.12063	1.740507	-0.03905	0.48183	0.146023	0.117039	-0.21756	-0.13878	-0.42445	-1.00204	0.89078

3. **UI**



And then predict the input and got the same output "This is a fraudulent Transaction"



Conclusion

This project showcases how machine learning can be used to detect fraudulent transactions in credit card data. The use of SMOTE to balance the data improves model performance and the app built with Streamlit allows for easy deployment of the model.