Fraud Detection

Dataset: `vardhansiramdasu/fraudulent-transactions-prediction`

Reading the data

For this task, I will use some basic analyze of data which I know to process EDA task.

```
# Necessary module for discover data
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
import warnings
warnings.filterwarnings("ignore")
```

First, we import the necessary module first.

```
# Read the dataset.
df = pd.read_csv("Fraud.csv")
df.head(10)
```

First we read the CSV file to dataframe

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.00	160296.36	M1979787155	0.0	0.00	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.00	19384.72	M2044282225	0.0	0.00	0	0
2	1	TRANSFER	181.00	C1305486145	181.00	0.00	C553264065	0.0	0.00	1	0
3	1	CASH_OUT	181.00	C840083671	181.00	0.00	C38997010	21182.0	0.00	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.00	29885.86	M1230701703	0.0	0.00	0	0
5	1	PAYMENT	7817.71	C90045638	53860.00	46042.29	M573487274	0.0	0.00	0	0
6	1	PAYMENT	7107.77	C154988899	183195.00	176087.23	M408069119	0.0	0.00	0	0
7	1	PAYMENT	7861.64	C1912850431	176087.23	168225.59	M633326333	0.0	0.00	0	0
8	1	PAYMENT	4024.36	C1265012928	2671.00	0.00	M1176932104	0.0	0.00	0	0
9	1	DEBIT	5337.77	C712410124	41720.00	36382.23	C195600860	41898.0	40348.79	0	0

Some columns

```
# List of the columns
list(df.columns)
df.info()
```

Drop the none value

```
df = df.dropna()
df
```

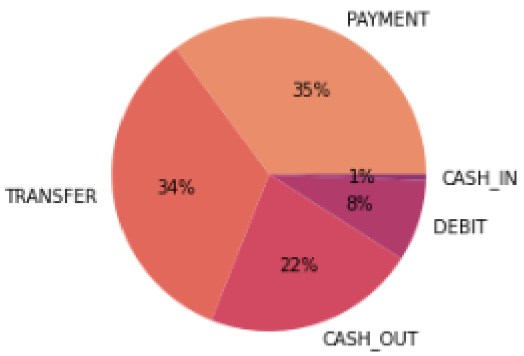
First we need to drop the "none" value in the dataframe by `df = df.dropna()` command, the output of table have the same size at input so this table does not have any "none"-value cell.

Visualize `type` percentages

We can see the numbers of different type of transactions by `df.type.unique_values()` and now I want to visualize the percentages of each type using the pie chart in `seaborn`

```
col_type = df.type
col_type_unique_value = df.type.unique().tolist()
percentages = (col_type.value_counts() / len(df)).tolist()
palette_color = sns.color_palette("flare")

# plotting data on chart
fig = plt.pie(percentages, labels=col_type_unique_value, colors=palette_color,
        autopct='%.0f%%')
```

The output:



Imbalance labeling dataset

We can see the different between the log of fraud-transactions and not-fraud-transactions by visualize the bar plot of its log.

```
# Value counts for the is_Fraud, we can see that the value of fraud
transactions ~ 0.129082044 % the data,
# which is kinds of imbalance - So if we use whole dataframe as input, we may
consider F1-score to choose the best model
df["isFraud"].value_counts()
fig, ax = plt.subplots()
#print(n)
ax.bar(height=np.log(df.isFraud.value_counts()), x=[0,1])
#plt.set_yticks([0.95, 0975.,1.0])
ax.set_xticks([0,1])
ax.set_xticklabels(["Fraud","isFraud"])
#ax.set_yticks()
ax.set_ylabel("Log of transactions")
fig.show()
```

Trying to find some pattern of data

We try with payment methods first

```
# Filter the type column to get whole payment ones
payment_df = df[df["type"] == "PAYMENT"]
payment_df[["type", "isFraud"]]["isFraud"].value_counts()
```

Output:

```
0    2151495
Name: isFraud, dtype: int64
```

So we can see that the `PAYMENT` type always is believable - not fraud.

The question was raise here is maybe there will be several other methods will always be not fraud too? Let try to find that.

```
def type_with_fraud(df, TYPE):
    print(f"Analyze for {TYPE}-transaction")
    x_df = df[df["type"] == TYPE]
    print(f"Number of {TYPE}-transaction is about {round(len(x_df)/ len(df), 2)
* 100} %")
    val_count = x_df[["isFraud"]].value_counts()
    print(val_count)
    if val_count[0] == len(x_df):
        return True
    return False
TYPES = list(df["type"].unique())
list_types = []
list_not = []
for t in TYPES:
    t_or_f = type_with_fraud(df, t)
    if t_or_f:
        list_types.append(t)
    else:
        list_not.append(t)
print(f"\n\n\nList of those types that not 100% not fraud {list_not}")
print(f"\n\n\nList of those types that 100% not fraud {list_types}")
```

Output:

```
Analyze for PAYMENT-transaction
Number of PAYMENT-transaction is about 34.0 %
isFraud
0          2151495
dtype: int64
Analyze for TRANSFER-transaction
Number of TRANSFER-transaction is about 8.0 %
isFraud
0          528812
1            4097
dtype: int64
Analyze for CASH_OUT-transaction
Number of CASH_OUT-transaction is about 35.0 %
isFraud
0          2233384
1             4116
dtype: int64
Analyze for DEBIT-transaction
Number of DEBIT-transaction is about 1.0 %
isFraud
0          41432
dtype: int64
Analyze for CASH_IN-transaction
Number of CASH_IN-transaction is about 22.0 %
isFraud
0          1399284
dtype: int64




List of those types that not 100% not fraud ['TRANSFER', 'CASH_OUT']




List of those types that 100% not fraud ['PAYMENT', 'DEBIT', 'CASH_IN']
```

So we can see the believable methods is **[PAYMENT, DEBIT, CASH_IN]**
Now we drop those methods and make model and predictions on the remain twos.

```
#m_df will be the main dataframe that we use to predict between fraud or non-fraud transactions
# We can see that this dataframe contain ~ 43.54 % the original dataframe, so when we use this dataframe at main dataset to predict, it will be more efficient.
m_df = df.loc[df["type"].isin(list_not)]
m_df
```

Now my mind raise a pattern that, may be some fixed destination and fixed Orig may cause the fraud transactions? I tried to verify the idea

```
name_m_df = m_df[["nameOrig","nameDest","isFraud", "isFlaggedFraud"]]
nameOrig_and_labels = name_m_df[["nameOrig", "isFraud"]]
nameOrig_and_labels_val_count = nameOrig_and_labels[name_m_df["isFraud"] ==
1].value_counts()
nameOrig_and_labels_val_count
nameDest_and_labels = name_m_df[["nameDest", "isFraud"]]
nameDest_and_labels_val_count = nameDest_and_labels[name_m_df["isFraud"] ==
1].value_counts()
nameDest_and_labels_val_count
```

And the output:

```
nameOrig      isFraud
C1000036340   1          1
C334503836    1          1
C357089378    1          1
C356905617    1          1
C356781229    1          1
                        ..
C1629072698   1          1
C162879753    1          1
C1628562361   1          1
C162812306    1          1
C99979309     1          1
Length: 8213, dtype: int64

nameDest      isFraud
C2020337583   1          2
C650699445    1          2
C475338087    1          2
C505532836    1          2
C1185292292   1          2
                        ..
C1661119285   1          1
C1661087818   1          1
C1660826618   1          1
C1660783549   1          1
C999955448    1          1
Length: 8169, dtype: int64
```

After some naive analyze I have not see any pattern in the context of dataset.

- I **think** the `nameOrig` and the `nameDest` does look like the kind of `id` which does not generalize our solution if we have more transaction, so I will drop those field
- I does not drop the `oldbalanceDest` and `newBalanceDest` althrough about ~30% of data will do not affected but our consideration is on 2 type `TRANSFER` and `CASH_OUT`
- But we will not use both features because I think our model can learn relationship between `newbalanceOrg` and `oldbalanceDest` base on `type`

```
drop_m_df = m_df.drop(columns = ["isFraud", "newbalanceDest","nameDest",
"nameOrig"])
drop_m_df
```

I want to see the correlation between the `isFraud` and `isFlaggedFraud` to see if the transactions was marked as Fraud so will it high probability is a `fraud` too.

```
fraud_and_flagged = m_df[["isFraud", "isFlaggedFraud"]]
corr = fraud_and_flagged.corr()
corr
```

```
                isFraud     isFlaggedFraud
isFraud         1.000000    0.044072
isFlaggedFraud  0.044072    1.000000
```

So this too columns not affect to much to each others.

Try to build the model to detect the Fraud detection

Preprocessing data

I tried to transform the different scaler for different column.

```
from sklearn.preprocessing import LabelEncoder, MinMaxScaler, Normalizer,
StandardScaler, MinMaxScaler
labelize = LabelEncoder()
standard_scaler = StandardScaler()
normalizer = Normalizer(norm="l2")
minmax = MinMaxScaler()
drop_m_df.type = labelize.fit_transform(drop_m_df.type) # Transform the type
drop_m_df.step = minmax.fit_transform(drop_m_df.step.to_numpy().reshape(-1,1))
#Scale the time to 0 - 1

drop_m_df[["amount", "oldbalanceOrg", "oldbalanceDest"]] =
standard_scaler.fit_transform(drop_m_df[["amount", "oldbalanceOrg",
"oldbalanceDest"]].to_numpy().reshape(-1,3))
drop_m_df
```

I also apply the one-hot encoding to the label

```
# split train-test
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder()

#y = enc.fit_transform(m_df[['isFraud']]).toarray()
y = m_df["isFraud"].to_numpy().reshape(-1,1)
X = drop_m_df.to_numpy()
#y = onehot.fit_transform(m_df["isFraud"].to_numpy().reshape(-1,1))
#y = enc.transform(m_df["isFraud"].to_numpy().reshape(-1,1))
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.4)
```

Now I tried to use different model to training and predict the result

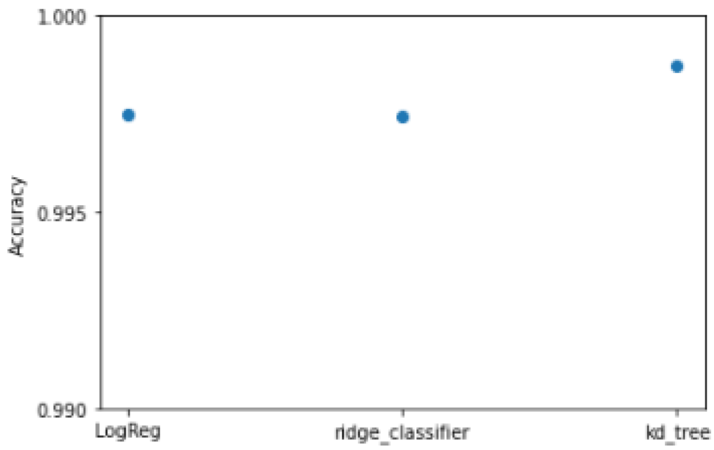
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression, RidgeClassifier
from sklearn import linear_model
from sklearn.metrics import f1_score, accuracy_score
from sklearn.ensemble import GradientBoostingClassifier
```

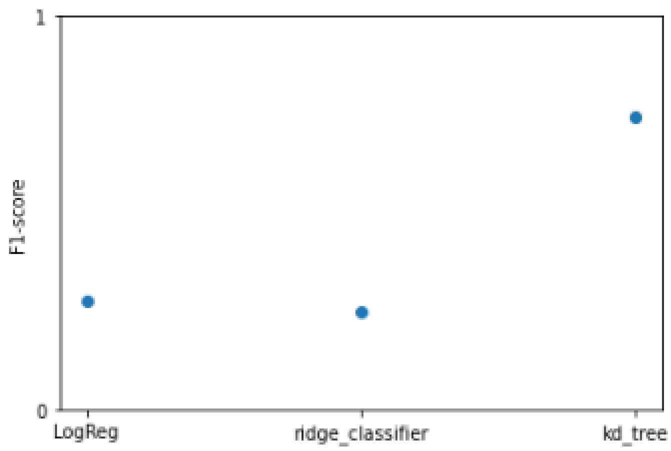
```
from sklearn import svm


result_list = []
def get_score_with_specify_model(name, model,X_train, y_train, X_test, y_test,
result_dict, num_run):
    print(f"Fitting model {name}  ...")
    model.fit(X_train, y_train)
    print("Finishing model! ")
    print("Predict the result and getting the results.")
    test_pred = model.predict(X_test)

    result_dict.update({name: {
        'acc': accuracy_score(test_pred, y_test),
        'f1': f1_score(test_pred, y_test)
    }})
    print(f"Finishing model {name}\n\n\n\n")
    return result_dict
models = {'LogReg': LogisticRegression(),
          'ridge_classifier': RidgeClassifier(),
          "kd_tree": KNeighborsClassifier(n_neighbors=10, algorithm='kd_tree'),
          #"mlp": MLPClassifier(hidden_layer_sizes=(20,10,),
activation='relu',solver='adam', alpha=0.001, batch_size="auto",
learning_rate="adaptive", early_stopping=True, max_iter=250),
          #"grad_boost": GradientBoostingClassifier(n_estimators=100,
learning_rate=0.5, max_depth=3),
          #"rbf_svm": svm.SVC(kernel="rbf"),
          #"linear_svm": svm.SVC(kernel="linear")
         }


for num_run in range(1):
    result = {}
    for k,v in models.items():
        result = get_score_with_specify_model(k,v,X_train, y_train, X_test,
y_test,result,num_run)
    result_list.append(result)
result_list
```

After few runs, I decied to omit the Multi-layer Perceptrons, Gradient Boosting, Radius Basis Function Support Vector Machine and Linear Support Vector Machine because it took very long time to produce the trained model.

```
fig, ax = plt.subplots()

ax.scatter(y = [i['acc'] for i in result_list[0].values()],x = [1,2,3])
#plt.set_yticks([0.95, 0975.,1.0])
ax.set_xticks([1,2,3])
ax.set_xticklabels(result_list[0].keys())
ax.set_yticks([0.99, 0.995,1.0])
ax.set_ylabel("Accuracy")
fig.show()

fig, ax1 = plt.subplots()

ax1.scatter(y = [i['f1'] for i in result_list[0].values()],x = [1,2,3])
#plt.set_yticks([0.95, 0975.,1.0])
```

```
ax1.set_xticks([1,2,3])
ax1.set_xticklabels(result_list[0].keys())
ax1.set_yticks([0.0, 1.0])
ax1.set_ylabel("F1-score")
fig.show()
```





The Accuracy and F1 score was visualize like above, because this is an imbalance dataset so we can choose the model with best f1-score.

Final

Thats all my basic analyze for this problem.