Credit Card Fraud Detection

Xinkai Zhao (xinkaiz2)
Yumeng Li (yumengl4)

Project Description and Summary

Electronic transactions nowadays are very common which benefits credit card companies greatly. But current fraud detection system is not perfect. So, it is essential to have an accurate fraud prevention system to be of better service to both customers and companies. The goal of our project is to identify as much credit card fraud as possible. To achieve this goal, we implemented three resampling methods to balance the outcome variable: random over sampler, SMOTE, and SMOTE+Tomek Links. And then we fit the KNN and Random Forest models on three transformed datasets respectively. In conclusion, the random over sampler with the Random Forest model gives the best result. The accuracy reaches 0.9825, the recall rate for being a fraud is 0.73, and the AUC score is 0.861.

Literature Review

Given the importance of detecting credit card fraud, there are a lot of techniques to solve this problem and increase the accuracy of capturing credit card fraud, such as the machine learning algorithms. However, in a real-world problem, it is extremely rare to have a balanced dataset to work with. Therefore, the resampling techniques have been introduced to address this issue.

Xuan, S. et al. [1] introduced Decision-Tree-Based Random Forest and CART-Based Random Forest in the paper, and implemented three experiments on a subset of full data, undersampled subset with different fraudulent versus not fraudulent ratios, and full data. The CART-Based Random Forest performed best in training, approaching 96.77% accuracy and 95.27% recall with 0.9601 F-measure. They also found that the best undersampling ratio of fraud and legal transactions is at 1:5 based on F-score. Despite the high accuracy (98.67%) when testing on the full data, the model performed poorly detecting fraud with 32.68% precision and 59.62% recall. To further investigate the best combination of machine learning algorithms and resampling techniques, Alfa and Fati [2] proposed an advanced approach. At first, they examined nine machine learning algorithms with their default parameters, including Logistic Regression, K-Nearest

Neighbors, Decision Tree, Naive Bayes, Random Forest, Gradient Boosting Machines, Light Gradient Boosting Machine, Extreme Gradient Boosting, and Category Boosting. And they choose three best model to perform resampling techniques. They analyzed 11 under sampling methods, 6 over sampling methods, and 2 combined methods. In conclusion, the CatBoost model combined with AIIKNN techniques outperformed others. The AUC score reaches 97.94%. The recall rate is 95.91%, and the F1-Score reaches 87.4%.

Dataset

The dataset used in this study comes from Vesta's e-commerce data [3]. There are 590,540 transactions and 394 columns. Due to the protection of the customer's privacy, the description of most features is masked. This dataset contains numerical features like transaction amount, transaction time delta, and 339 Vesta features. The categorical features include card type, email domain of purchaser and recipient, issue bank and issue country of the card, etc.

The outcome variable, "isFraud" in this dataset is highly unbalanced. A frequency plot illustrating this fact is shown in Figure 1.

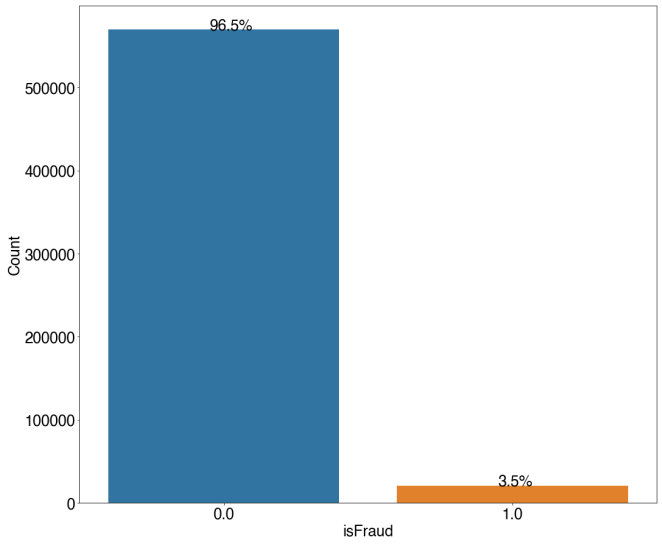

Figure 1

Preprocessing

1. Handle missing values
   We have a large proportion of missing data, especially among the Vesta features. There are 192 columns with more than 30% of missing values, and we decided to delete these columns. For the rest of the columns, we imputed the numerical with median and categorical with mode.

2. Regroup categorical variables
   Some categorical variables have highly unbalanced levels, which would have an impact on the accuracy of the Random Forest model. Also, some features like the purchaser email domain contain duplicates. For example, 'yahoo.com', 'yahoo.com.mx', 'yahoo.fr', 'yahoo.de', 'yahoo.es', 'yahoo.co.uk', and 'yahoo.co.jp' all refer to Yahoo email at different language servers. We re-coded these 7 domains into 'yahoo.com'. Then we renamed the 42 domains with less than 1% as "others" to avoid such impact. After regrouping, 6 levels left.

3. One-hot coding
   After regrouping the categorical variables, we transformed all the categorical variables into dummy variables for further analysis.

Magic Feature

Once a transaction was detected as fraudulent, all posterior transactions with linked information such as card information, email address, and billing address would be flagged as fraud. However, Vesta did not provide information to connect transactions with individual clients or cards. Thus, a user identification feature is needed to fulfill the task. Yakovlev [4] provided a method to utilize limited information to identify distinct cards.

The method uses features "card1", "addr1", "D1", and "TransactionDT". Yakovlev found that the two features, "card1" and "addr1" contained important information that could identify the majority of clients. After combining these two columns into a new "card1_addr1" feature, the method will evaluate each transaction in the same group.

"D1" is the days passed since the card began, and "TransactionDT" is how many seconds have elapsed since a specific timestamp for all the transactions in the dataset. After translating seconds to days, we can introduce a new column to calculate how many days have passed since the first transaction within the group. If the value matches the difference in "D1" feature, these transactions will be assigned to the same user. Figure 2 shows an example of the algorithm to artificially create the "uid" to identify each unique card.

	TransactionID	isFraud	TransactionDT	TransactionAmt	card1	card4	addr1	D1	uid	DTdiff	D1diff
1261	2988261	1	129512	160.5	11839	visa	420.0	395.0	2988261.0	0.0	0.0
1274	2988274	1	129834	280.0	11839	visa	420.0	395.0	2988261.0	0.0	0.0
1282	2988282	1	130050	117.0	11839	visa	420.0	395.0	2988261.0	0.0	0.0
127650	3114650	1	2537461	108.0	11839	visa	420.0	423.0	2988261.0	28.0	28.0
137995	3124995	1	2804429	171.0	11839	visa	420.0	426.0	2988261.0	31.0	31.0
230888	3217888	1	5474535	171.0	11839	visa	420.0	457.0	2988261.0	62.0	62.0
230893	3217893	1	5474733	100.0	11839	visa	420.0	457.0	2988261.0	62.0	62.0
316951	3303951	1	7889004	171.0	11839	visa	420.0	485.0	2988261.0	90.0	90.0
316955	3303955	1	7889277	117.0	11839	visa	420.0	485.0	2988261.0	90.0	90.0
341594	3328594	1	8429542	117.0	11839	visa	420.0	491.0	2988261.0	96.0	96.0
411332	3398346	1	10391596	117.0	11839	visa	420.0	514.0	2988261.0	119.0	119.0
411335	3398349	1	10391846	171.0	11839	visa	420.0	514.0	2988261.0	119.0	119.0
445894	3432916	1	11359563	117.0	11839	visa	420.0	525.0	2988261.0	130.0	130.0
479289	3466323	1	12438287	117.0	11839	visa	420.0	537.0	2988261.0	142.0	142.0
501966	3489000	1	13154560	171.0	11839	visa	420.0	546.0	2988261.0	151.0	151.0
501971	3489005	1	13154807	171.0	11839	visa	420.0	546.0	2988261.0	151.0	151.0

Figure 2

The user ID is not perfect without a full interpretation of the features, but it suffices in the basic step to proceed with our objective. Figure 3 shows the validation for the user ID. Among the uid created with two or more transactions, over 99.9% of the uid's contain either all fraudulent or all not fraudulent transactions. Less than 0.1% of the uid's contain transactions of mixed "isFraud" label, which is negligible.

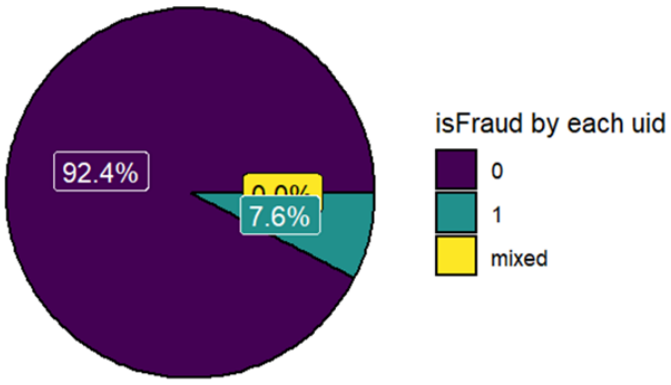

Figure 3

In order to implement this model to predict new customers in future prediction and prevent overfitting, we cannot use the uid column directly in our model. Instead, we generalized some group features to preserve the information contains inside the use ID column. Specifically, we aggregated the mean and standard deviation values for numerical attributes by each user ID, and aggregated the number of unique values for categorical attributes by each use ID. After aggregating, 389 additional columns were created and added to the dataset.

Resampling Methods

Due to the highly unbalanced nature of our outcome variable, we used three resampling methods to address this problem.

1. Random over sampler

   The first method we used is the random over sampler. The objective of this method is to oversample the minority class in the dataset. It will pick the samples at random with replacement. After implementing this method, our frequency of fraud and non-fraud becomes 50% to 50%. A comparison of the scatter plot from the original train set and data after resampling is shown in Figure 4.

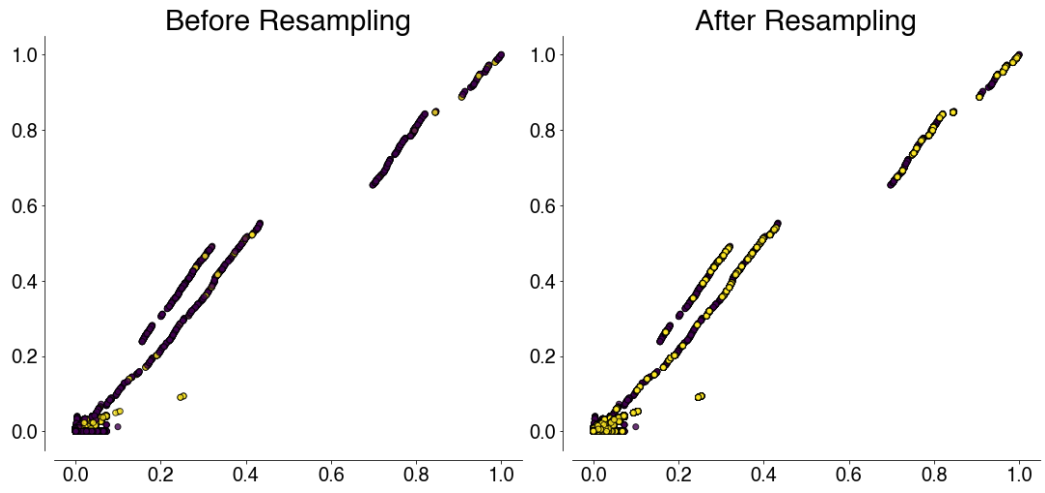

Figure 4

2. SMOTE

SMOTE is also an oversampling method. It uses a k-nearest neighbor algorithm to create synthetic data points. Specifically, the steps include identifying the minority class vector, computing a line between the minority data points and any of its neighbors, and placing a synthetic point, then repeating this process until balanced. A set of scatter plots describing this process is shown in Figure 5.

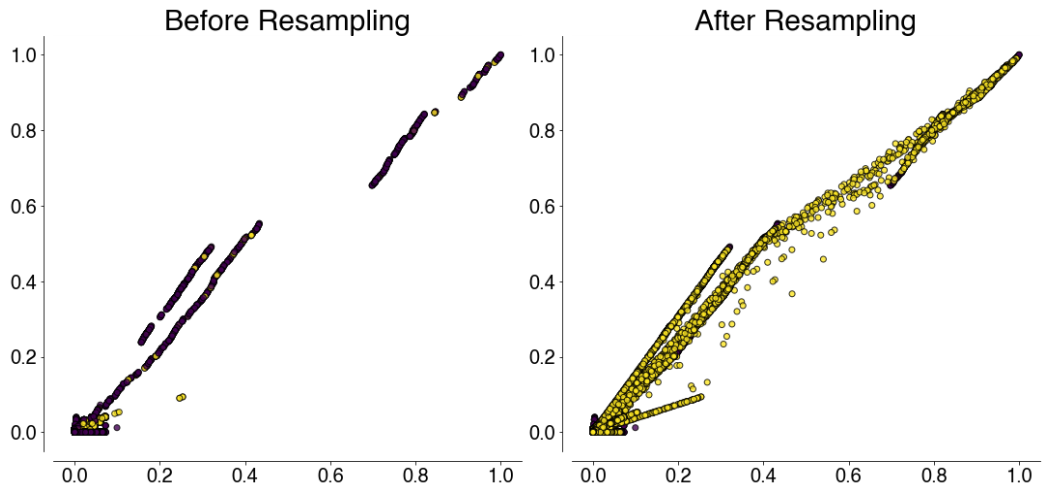

Figure 5

3. SMOTE + Tomek Links

   This is a combined method of oversampling technique and undersampling technique. SMOTE is applied to create new synthetic minority samples. Tomek Links is one of a modification from the Condensed Nearest Neighbors undersampling technique that is developed by Tomek (1976). Unlike the CNN method that only randomly selects the samples with its k nearest neighbors from the majority class that wants to be removed, the Tomek Links method uses the rule to select the pair of observation (say, a and b) that are fulfilled these properties:

   1) The observation a's nearest neighbor is b.
   2) The observation b's nearest neighbor is a.
   3) Observations a and b belong to different classes. That is, a and b belong to the minority and majority class (or vice versa), respectively.[1]

   By the nature of the Tomek links method, it could be used to remove the samples close to the boundary of the two classes to increase the separation. After implementing this method, the frequency of fraud and non-fraud becomes 50% to 50%. A set of scatter plots describing this combined method is shown in Figure 6.

[1] https://towardsdatascience.com/imbalanced-classification-in-python-smote-tomek-links-method-6e48dfe69bbc

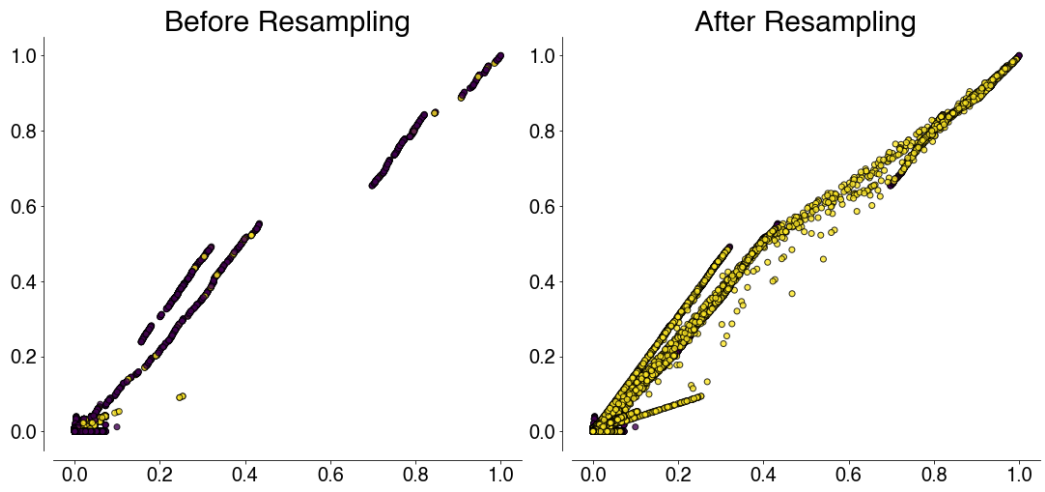

Figure 6

Models

1. KNN Model

   This is a classifier that is used for classification and regression problems. It works based on similarity measures. So before using it, all variables need to be scaled. The disadvantage of it is the prediction cost of KNN is too expensive, since every time when KNN is trying to predict a data point, it will go back and calculate the distance between every point in the training set. Another drawback of KNN is that it suffers from the curse of dimensionality, so it usually does not perform well with high-dimensional data.

2. Random Forests

   A random forest is an ensemble method that can utilize many decision trees, whose decisions it averages. One of the advantages of random forests is that it could reduce overfitting by bagging and choosing a random subset of features at each split.

Results

To evaluate the performance of the models, three evaluation metrics were used:

Overall accuracy rate is used to show the percentage of correctly classified observations.

The recall rate for fraud is to show the proportion of actual fraud detected. AUC illustrates how well the model separates two classes. To further compare the effects of implementing the resampling methods, the models using the original train and test sets are set as the baseline.

1. KNN Results Comparison
   To reduce the impact of high dimensionality, we applied the principal component analysis before fitting the model. We chose 95% of the variation explained as the threshold. As a result, the number of components of three resampling methods is shown in Table 1.

Method	RandomOverSampler	SMOTE	SMOTE+ Tomek Links
N_Comp	48	41	41

Table 1

Using a 5-fold cross-validation to tune the parameter k, then the results of the models are in Table 2.

Resampling Methods	K	Accuracy	Recall	AUC
None	4	0.9675	0.1037	0.5513
Random Over Sampler	15	0.844	0.54	0.698
SMOTE	87	0.7162	0.72	0.7178
SMOTE + Tomek Link	63	0.7285	0.71	0.7183

Table 2

The “None” method indicates the baseline model which uses the original train and

test sets. It has a very high accuracy of 0.96 but only gets 0.1 for the recall rate. Most of the predictions of actual fraud fall into the non-fraud class. Besides, the AUC score for the baseline model is only 0.5513, which also indicates the baseline model didn't do a good job of detecting fraud. After applying resampling methods, recall rates and AUC score increase a lot, while the accuracy decreases a little. Among these three resampling methods, SMOTE and SMOTE + Tomek Link are better than Random Over Sampler, which has a much lower recall rate than the other two methods.

2. Random Forests Results

   We used the resampled dataset without performing PCA transformation to fit the random forests model. After using 5-fold cross-validation to tune n_estimators and max_depth, the results are shown in Table 3.

Resampling Methods	max_depth	n_estimators	Accuracy	Recall	AUC
None	11	101	0.9742	0.29	0.644
Random Over Sampler	37	108	0.9825	0.73	0.861
SMOTE	35	108	0.9815	0.62	0.8088
SMOTE + Tomek Link	36	102	0.9815	0.62	0.8076

Table 3

From the comparison with the baseline model, all three evaluation metrics are improved, which are better than the results of KNN models. Among these three methods, Random Over Sampler outperformed the other two in accuracy rate, recall, and AUC.

Conclusion

This paper implemented the resampling methods and the machine learning techniques for credit card fraud detection problems. To get an efficient and accurate model to detect

fraud, balancing the data is necessary. In this paper, two different supervised techniques and three resampling techniques were applied and their performance was evaluated with accuracy, recall, and AUC score. We found that Random Forests performs much better than the KNN model. And there is no optimal resampling method applying to all cases. The effectiveness of a resampling method should be checked after fitting into a specific model. In our findings, SMOTE and SMOTE+ Tomek Links have similar performance and are better than Random Over Sampler in the KNN model. But in the Random Forests model, the Random Over Sampler seems to be the best choice to balance the outcome variable.

Limitations and Future Research

For privacy reasons, the majority of attribute meanings are masked by the data provider, which has limited our work on several aspects. If we could obtain more detailed information in the future, we could improve our model in the following ways.

First, we might keep some important variables even with a relatively large proportion of missing data. And we would impute missing values in a more meaningful way. Second, we could refine our data preprocessing by creating better encoding methods for categorical features and bins for numerical features. Third, we could reduce duplicates in feature engineering. If the original data contains features such as the number of addresses linked to this card, we could avoid performing the n-unique engineering on relevant addresses.

Reference

1. S. Xuan, G. Liu, Z. Li, L. Zheng, S. Wang and C. Jiang, "Random forest for credit card fraud detection," 2018 IEEE 15th International Conference on Networking, Sensing and Control (ICNSC), 2018, pp. 1-6, doi: 10.1109/ICNSC.2018.8361343.

2. Alfaiz NS, Fati SM. Enhanced Credit Card Fraud Detection Model Using Machine Learning. Electronics. 2022; 11(4):662. https://doi.org/10.3390/electronics11040662

3. https://www.kaggle.com/competitions/ieee-fraud-detection/data

4. https://www.kaggle.com/code/kyakovlev/ieee-uid-detection-v6

Code

```
import pandas as pd

import seaborn as sns

import dexplot as dxp

import sklearn_pandas

from sklearn.impute import SimpleImputer

import numpy as np

from sklearn.model_selection import GridSearchCV

from sklearn.neighbors import KNeighborsClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.linear_model import LogisticRegressionCV

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

import numpy as np

from sklearn.metrics import accuracy_score

from sklearn.metrics import classification_report

from sklearn.metrics import confusion_matrix

from sklearn.metrics import roc_auc_score

from imblearn.over_sampling import SMOTE

from imblearn.combine import SMOTETomek
```

```
from imblearn.over_sampling import RandomOverSampler
from sklearn.preprocessing import MinMaxScaler




# read in data
df = pd.read_csv("train_transaction.csv")
df.head(5)
df.shape

uid_df = pd.read_csv("train_uids_full_v3.csv")

df = pd.merge(df,uid_df.iloc[:,1:],on='TransactionID')
df.head(5)

# visualization

def without_hue(plot, feature):
    total = len(feature)
    for p in ax.patches:
        percentage = '{:.1f}%'.format(100 * p.get_height()/total)
```

```
        x = p.get_x() + p.get_width() / 2 - 0.05
        y = p.get_y() + p.get_height()
        ax.annotate(percentage, (x, y), size = 25)
    plt.show()

# frequency plot of isFraud
plt.figure(figsize = (17,15))
ax = sns.countplot('isFraud', data=df)
plt.xticks(size = 25)
plt.xlabel('isFraud', size = 25)
plt.yticks(size = 25)
plt.ylabel('Count', size = 25)
without_hue(ax, df.isFraud)



## delete NA columns
percent_missing = df.isnull().sum()*100/len(df)
missing_cols = list(percent_missing[percent_missing>30].index)
df = df.drop(columns=missing_cols)

## visualize remaining categorical variables
```

```
cat_list = ["ProductCD", "card1", "card2", "card3", "card4", "card5",
"card6","addr1","addr2","P_emaildomain","M6"]

dxp.count('isFraud', data=df, split='ProductCD', normalize='isFraud')

dxp.count('isFraud', data=df, split='card4', normalize='isFraud')

dxp.count('isFraud', data=df, split='card6', normalize='isFraud')

dxp.count('isFraud', data=df, split='P_emaildomain', normalize='isFraud')

dxp.count('isFraud', data=df, split='M6', normalize='isFraud')



## fill NA

df[cat_list] = df[cat_list].astype("object")

imputer = SimpleImputer(strategy="most_frequent")

df[cat_list] = imputer.fit_transform(df[cat_list])

imputer2 = SimpleImputer(strategy="median")

df_num = df.drop(cat_list, axis = 1)

df[df_num.columns]= imputer2.fit_transform(df[df_num.columns])


## feature engineering


# need to remap, then do one-hot coding
```

```
map_list = ["P_emaildomain"]


# delete after agg, same as the cat_list

del_list = ["card1", "card2", "card3","card5","addr1","addr2", "card6", "M6", "ProductCD",
"card4", "P_emaildomain"]


# directly use one-hot coding


one_hot = ["card6", "M6", "ProductCD", "card4", "P_emaildomain"]


# remap


len(df["P_emaildomain"].unique())

email_list = list(df["P_emaildomain"].unique())

email_list = email_list[1:]


[s for s in email_list if "gmail" in s]

df.loc[df["P_emaildomain"].isin(['gmail']),"P_emaildomain"] = "gmail.com"


yahood = [s for s in email_list if "yahoo" in s]

yahood
```

```
df.loc[df["P_emaildomain"].isin(yahood),"P_emaildomain"] = "yahoo.com"


hm = [s for s in email_list if "hotmail" in s]

hm

df.loc[df["P_emaildomain"].isin(hm),"P_emaildomain"] = "hotmail.com"


len(df["P_emaildomain"].unique())


df.loc[~df["P_emaildomain"].isin(['gmail.com',"yahoo.com","hotmail.com","anonymous.com","aol.com"]),"P_emaildomain"] = "Others"


len(df["P_emaildomain"].unique())


# One Hot Coding

from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder()

one_code_df = pd.DataFrame(encoder.fit_transform(df[one_hot]).toarray())

one_code_df.columns = encoder.get_feature_names(one_hot)

one_code_df

df3 = df.join(one_code_df)
```

```
# Aggregation

num_list= list(df3.columns[(~df3.columns.isin(cat_list)) &
(~df3.columns.isin(list(one_code_df.columns)))])

del num_list[0:2] # delete transaction id, isFraud

del num_list[189] # delete uid

num_agg = df3.groupby(['uid'])[num_list].agg(['mean', 'std'])

num_agg_col = list()

for i in num_list:

    num_agg_col.append(i+"_mean")

    num_agg_col.append(i+"_std")

num_agg.columns = num_agg_col

num_agg.reset_index(inplace = True)

num_agg



cat_agg = df3.groupby(['uid'])[cat_list].nunique()

cat_agg_col = list()

for i in cat_list:

    cat_agg_col.append(i+"_nunique")

cat_agg_col

cat_agg.columns = cat_agg_col
```

```
cat_agg.reset_index(inplace=True)

cat_agg


df4 = pd.merge(pd.merge(df3,cat_agg,on='uid'), num_agg, on = 'uid')

df4 = df4.drop(cat_list, axis = 1).fillna(0)


# Normalization


from sklearn.preprocessing import MinMaxScaler


norm_col = list(df4.columns[(~df4.columns.isin(["uid", "isFraud", "TransactionID"])) &
(~df4.columns.isin(list(one_code_df.columns)))])

scaler = MinMaxScaler()

normalized_arr = scaler.fit_transform(df4[norm_col])

scaled_df = pd.DataFrame(normalized_arr, columns=norm_col)

predictors_scaled_df = pd.concat([scaled_df, one_code_df], axis=1)


# train test split


X = predictors_scaled_df

y = df4['isFraud'].values
```

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1, stratify=y)

X_sub = X_train.iloc[:2000,:]

y_sub = y_train[:2000]


## Baseline Model


# KNN


#create new a knn model

knn = KNeighborsClassifier()

#create a dictionary of all values we want to test for n_neighbors

param_grid = {'n_neighbors': np.arange(1, 100)}

#use gridsearch to test all values for n_neighbors

knn_gscv = GridSearchCV(knn, param_grid, cv=5, verbose = 2)

#fit model to data

knn_gscv.fit(X_sub, y_sub)

#check top performing n_neighbors value

knn_gscv.best_params_

knn2 = KNeighborsClassifier(n_neighbors = knn_gscv.best_params_)
```

```
# Fit the classifier to the data

knn2.fit(X_train,y_train)

y_pred_k = knn2.predict(X_test)


confusion_matrix(y_test, y_pred_k)


print(accuracy_score(y_test, y_pred_k))


print(classification_report(y_test, y_pred_k))


print(roc_auc_score(y_test, y_pred_k))



# Random Forest


clf = RandomForestClassifier() #Initialize with whatever parameters you want to


param_grid = {
            'n_estimators': np.arange(90,110,1),
            'max_depth': np.arange(10,40,1)
        }
```

```
grid_clf = GridSearchCV(clf, param_grid, cv=5, verbose = 2)

grid_clf.fit(X_sub, y_sub)

grid_clf. best_params_

rf2 = RandomForestClassifier(max_depth= 11, n_estimators = 101)

# Fit the classifier to the data

rf2.fit(X_train,y_train)

y_pred_rf = rf2.predict(X_test)


confusion_matrix(y_test, y_pred_rf)


print(accuracy_score(y_test, y_pred_rf))


print(classification_report(y_test, y_pred_rf))


print(roc_auc_score(y_test, y_pred_rf))



## SMOTE+Tomek


smt = SMOTETomek(random_state = 42)

X_smt, y_smt = smt.fit_resample(X_train, y_train)
```

```
# visualize resampling process

def plot_resampling(X, y, ax, title=None):

    ax.scatter(X.iloc[:, 2], X.iloc[:, 5], c=y, alpha=0.8, edgecolor="k")
    if title is None:
        title = f"Resampling with {sampler.__class__.__name__}"
    ax.set_title(title, fontsize= 30)
    ax.tick_params(axis='x', labelsize=20)
    ax.tick_params(axis='y', labelsize=20)
    sns.color_palette("husl", 9)
    sns.despine(ax=ax, offset=10)

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 7))
plot_resampling(X_train, y_train, ax=axs[0], title="Before Resampling")
plot_resampling(X_smt, y_smt,ax=axs[1], title="After Resampling")
fig.tight_layout()

# PCA
pca = PCA(n_components = 0.95).fit(X_smt)
```

```
pca.n_components_

PC_values = np.arange(pca.n_components_) + 1

plt.plot(PC_values, pca.explained_variance_ratio_, 'o-', linewidth=2, color='red')

plt.title('Scree Plot')

plt.xlabel('Principal Component')

plt.ylabel('Variance Explained')

plt.show()


plt.rcParams["figure.figsize"] = (12,6)


fig, ax = plt.subplots()

xi = np.arange(1, 42, step=1)

yp = np.cumsum(pca.explained_variance_ratio_)


plt.ylim(0.0,1.1)

plt.plot(xi, yp, marker='o', linestyle='--', color='b')


plt.xlabel('Number of Components')

plt.xticks(np.arange(0, 42, step=1)) #change from 0-based array index to 1-based human-readable label

plt.ylabel('Cumulative variance (%)')
```

```
plt.title('The number of components needed to explain variance')


plt.axhline(y=0.95, color='r', linestyle='-')

plt.text(0.5, 0.85, '95% cut-off threshold', color = 'red', fontsize=16)


ax.grid(axis='x')

plt.show()


X_smt_pc = pca.transform(X_smt)

X_test_smt_pc = pca.transform(X_test)

pca_col =list()

for i in np.arange(1,42,1):

    pca_col.append("PC_{}".format(i))


X_smt_pc = pd.DataFrame(X_smt_pc, columns =pca_col )

X_test_smt_pc = pd.DataFrame(X_test_smt_pc, columns =pca_col)


import random

random.seed(10)

sub_index = random.sample(range(1, len(X_smt_pc)), 2000)

X_smt_sub = X_smt_pc.iloc[sub_index,:]
```

```
y_smt_sub = y_smt[sub_index]


# KNN
#create new a knn model
knn = KNeighborsClassifier()
#create a dictionary of all values we want to test for n_neighbors
param_grid = {'n_neighbors': np.arange(1, 100)}
#use gridsearch to test all values for n_neighbors
knn_gscv = GridSearchCV(knn, param_grid, cv=5, verbose = 2)
#fit model to data
knn_gscv.fit(X_smt_sub, y_smt_sub)
knn_gscv.best_params_
knn3 = KNeighborsClassifier(n_neighbors = knn_gscv.best_params_)
# Fit the classifier to the data
knn3.fit(X_smt_pc,y_smt)
y_pred_k_1 = knn3.predict(X_test_smt_pc)


confusion_matrix(y_test, y_pred_k_1)


print(accuracy_score(y_test, y_pred_k_1))
```

```
print(classification_report(y_test, y_pred_k_1))


print(roc_auc_score(y_test, y_pred_k_1))


# Random Forest
X_smt_sub2 = X_smt.iloc[sub_index,:]
clf = RandomForestClassifier() #Initialize with whatever parameters you want to


param_grid = {
            'n_estimators': np.arange(100,120,1),
            'max_depth': np.arange(30,40,1)
        }
grid_clf = GridSearchCV(clf, param_grid, cv=5, verbose = 2)
grid_clf.fit(X_smt_sub2,y_smt_sub)
grid_clf. best_params_
rf2 = RandomForestClassifier(max_depth= 36, n_estimators = 102)
# Fit the classifier to the data
rf2.fit(X_smt,y_smt)
y_pred_r1 = rf2.predict(X_test)


confusion_matrix(y_test, y_pred_r1)
```

```
print(accuracy_score(y_test, y_pred_r1))

print(classification_report(y_test, y_pred_r1))

print(roc_auc_score(y_test, y_pred_r1))



## Random Over Sampler

ros = RandomOverSampler(random_state=0)
X_ros, y_ros = ros.fit_resample(X_train, y_train)

# Visualization

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 7))
plot_resampling(X_train, y_train, ax=axs[0], title="Before Resampling")
plot_resampling(X_ros, y_ros,ax=axs[1], title="After Resampling")
fig.tight_layout()
```

```
# pca

pca = PCA(n_components = 0.95).fit(X_ros)

pca.n_components_

pca_col =list()

for i in np.arange(1,49,1):

    pca_col.append("PC_{}".format(i))

X_ros_pc = pca.transform(X_ros)

X_test_ros_pc = pca.transform(X_test)

X_ros_pc = pd.DataFrame(X_ros_pc, columns =pca_col )

X_test_ros_pc = pd.DataFrame(X_test_ros_pc, columns =pca_col)


X_ros_sub = X_ros_pc.iloc[sub_index,:]

y_ros_sub = y_ros[sub_index]


# KNN

#create new a knn model

knn = KNeighborsClassifier()

#create a dictionary of all values we want to test for n_neighbors

param_grid = {'n_neighbors': np.arange(1, 100)}

#use gridsearch to test all values for n_neighbors
```

```
knn_gscv = GridSearchCV(knn, param_grid, cv=5, verbose = 2)

#fit model to data

knn_gscv.fit(X_ros_sub, y_ros_sub)

knn_gscv.best_params_

knn4 = KNeighborsClassifier(n_neighbors = 15)

# Fit the classifier to the data

knn4.fit(X_ros_pc,y_ros)

y_pred_k2 = knn4.predict(X_test_ros_pc)


confusion_matrix(y_test, y_pred_k2)


print(accuracy_score(y_test, y_pred_k2))


print(classification_report(y_test, y_pred_k2))


print(roc_auc_score(y_test, y_pred_k2))


# Random Forest

X_ros_sub2 = X_ros.iloc[sub_index,:]

clf = RandomForestClassifier() #Initialize with whatever parameters you want to
```

```
param_grid = {
                'n_estimators': np.arange(100,120,1),
                'max_depth': np.arange(30,40,1)
            }
grid_clf = GridSearchCV(clf, param_grid, cv=5, verbose = 2)
grid_clf.fit(X_ros_sub2, y_ros_sub)
grid_clf.best_params_
rf2 = RandomForestClassifier(max_depth= 37, n_estimators = 108)
# Fit the classifier to the data
rf2.fit(X_ros,y_ros)
y_pred_r2 = rf2.predict(X_test)


confusion_matrix(y_test, y_pred_r2)


print(accuracy_score(y_test, y_pred_r2))


print(classification_report(y_test, y_pred_r2))


print(roc_auc_score(y_test, y_pred_r2))
```

```
## SMOTE


sm = SMOTE(random_state=42)

X_sm, y_sm = sm.fit_resample(X_train, y_train)


# visualization

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 7))

plot_resampling(X_train, y_train, ax=axs[0], title="Before Resampling")

plot_resampling(X_sm, y_sm,ax=axs[1], title="After Resampling")

fig.tight_layout()


# PCA

pca = PCA(n_components = 0.95).fit(X_sm)

pca.n_components_

X_sm_pc = pca.transform(X_sm)

X_test_sm_pc = pca.transform(X_test)

pca_col =list()


for i in np.arange(1,pca.n_components_+1,1):

    pca_col.append("PC_{}".format(i))
```

```
X_sm_pc = pd.DataFrame(X_sm_pc, columns =pca_col )

X_test_sm_pc = pd.DataFrame(X_test_sm_pc, columns =pca_col)

X_sm_sub = X_sm_pc.iloc[sub_index,:]

y_sm_sub = y_sm[sub_index]


# KNN

knn = KNeighborsClassifier()

#create a dictionary of all values we want to test for n_neighbors

param_grid = {'n_neighbors': np.arange(1, 100)}

#use gridsearch to test all values for n_neighbors

knn_gscv = GridSearchCV(knn, param_grid, cv=5, verbose = 2)

#fit model to data

knn_gscv.fit(X_sm_sub, y_sm_sub)

knn_gscv.best_params_

knn3 = KNeighborsClassifier(n_neighbors = 87)

# Fit the classifier to the data

knn3.fit(X_sm_pc,y_sm)


y_pred_k_3 = knn3.predict(X_test_sm_pc)


confusion_matrix(y_test, y_pred_k_3)
```

```
print(accuracy_score(y_test, y_pred_k_3))


print(classification_report(y_test, y_pred_k_3))


print(roc_auc_score(y_test, y_pred_k_3))


# Random Forest
X_sm_sub2 = X_sm.iloc[sub_index,:]
clf = RandomForestClassifier() #Initialize with whatever parameters you want to


param_grid = {
            'n_estimators': np.arange(100,120,1),
            'max_depth': np.arange(30,40,1)
        }
grid_clf = GridSearchCV(clf, param_grid, cv=5, verbose = 2)
grid_clf.fit(X_sm_sub2,y_sm_sub)
grid_clf. best_params_


rf2 = RandomForestClassifier(max_depth= 35, n_estimators = 108)
# Fit the classifier to the data
```

```
rf2.fit(X_sm,y_sm)

y_pred_r3 = rf2.predict(X_test)


confusion_matrix(y_test, y_pred_r3)


print(accuracy_score(y_test, y_pred_r3))


print(classification_report(y_test, y_pred_r3))


print(roc_auc_score(y_test, y_pred_r3))
```