Predicting Credit Card Fraud

August 27, 2024

0.0.1 Overview of the dataset

The dataset contains the following features:

distance_from_home - the distance from home where the transaction happened.

distance_from_last_transaction - the distance from last transaction happened.

ratio_to_median_purchase_price - Ratio of purchased price transaction to median purchase price.

repeat_retailer - Is the transaction happened from same retailer.

used_chip - Is the transaction through chip (credit card).

used_pin_number - Is the transaction happened by using PIN number.

online_order - Is the transaction an online order.

The dependent variable of the dataset is:

fraud - Is the transaction fraudulent.

Below I import all of the libraries I will be using throughout this project.

[ ]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTENC
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
from itertools import product
from sklearn.metrics import accuracy_score, precision_score, f1_score,␣
↪recall_score, confusion_matrix
```

Below I read the data into a dataframe called "df", and then take a look at the first few rows of the data.

[ ]:
```
df = pd.read_csv("/content/card_transdata.csv")
df.head()
```

[ ]:
```
   distance_from_home  distance_from_last_transaction  \
0           57.877857                        0.311140
```

```
1           10.829943                       0.175592
2            5.091079                       0.805153
3            2.247564                       5.600044
4           44.190936                       0.566486

   ratio_to_median_purchase_price  repeat_retailer  used_chip  \
0                        1.945940              1.0        1.0
1                        1.294219              1.0        0.0
2                        0.427715              1.0        0.0
3                        0.362663              1.0        1.0
4                        2.222767              1.0        1.0

   used_pin_number  online_order  fraud
0              0.0           0.0    0.0
1              0.0           0.0    0.0
2              0.0           1.0    0.0
3              0.0           1.0    0.0
4              0.0           1.0    0.0
```

I also check the dataset for missing values and observe that none of the columns has missing values.

[ ]:
```
df.isna().sum()
df.dropna(inplace = True)
```

To obtain a better understanding of the categorical and numeric features, I take a look at the number of unique instances of each feature.

The numerical features in this dataset are “distance_from_home” - the distance from home where the transaction happened, “distance_from_last_transaction” - the distance from last transaction happened, “ratio_to_median_purchase_price” - ratio of purchased price transaction to median purchase price.

The categorical features are “repeat_retailer” - Is the transaction happened from same retailer, “used_chip” - Is the transaction through chip (credit card), “used_pin_number” - Is the transaction happened by using PIN number, and “online_order” - Is the transaction an online order. Each categorical feature has 2 possible values (either 1 for the presence of that feature or 0 for the absence of that feature).

The dependent variable of the dataset, “fraud” - Is the transaction fraudulent also has two possible values: 0 if the transaction was not fraudulent and 1 if the transaction was fraudulent.

[ ]:
```
df.nunique()
```

[ ]:
```
distance_from_home                1000000
distance_from_last_transaction    1000000
ratio_to_median_purchase_price    1000000
repeat_retailer                         2
used_chip                               2
used_pin_number                         2
online_order                            2
```

```
fraud                                 2
dtype: int64
```

Below I check if the dataset is imbalanced.

[ ]:
```
frauds = len(df[df['fraud'] == 1])
non_frauds = len(df[df['fraud'] == 0])

print(f"The percentage of fraudulent transactions in the dataset is {frauds/
 ↪len(df)*100}, while the percentage of non-fraudulent transactions is␣
 ↪{non_frauds/len(df)*100}")
```

```
The percentage of fraudulent transactions in the dataset is 8.7403, while the
percentage of non-fraudulent transactions is 91.2597
```

Since the minority class of the dataset (observations where the transaction was fraudulent) only represent 8.74% of total observations, I conclude that the dataset is imbalanced.

0.0.2 Resampling the Data

SInce the dataset is imbalanced, I will resample it. Prior to doing so, I will first split the original data into training and testing subsets (where the test subset will have 20% of all original observations).

[ ]:
```
X = df.drop('fraud',axis=1).copy()
y = df[['fraud']]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪stratify = y, random_state = 123)
```

Then, I will resample the training subset to achieve a 1:1 ratio between 'fraud' and 'non-fraud' samples. To do this, I first store the indexes of my categorical columns in a list I call 'cat_ind'. This is a necessary step when using the SMOTENC command. Then, I oversample my training data. I store the oversampled features in a dataframe called "X_train_new" and the oversampled target in a dataframe called "y_train_new".

[ ]:
```
cat_cols = [col for col in X.columns if X[col].nunique() == 2]
cat_ind = [X.columns.get_loc(col) for col in cat_cols]

sm = SMOTENC(categorical_features = cat_ind, sampling_strategy = 1,␣
 ↪random_state = 1)
X_train_new, y_train_new = sm.fit_resample(X_train, y_train)

X_train_new_knn = X_train_new.copy()
X_test_new_knn = X_test.copy()
```

0.0.3 Fitting a Neural Network

Prior to fitting a Neural Netowrk to the training set, I will fit a scaler to the numerical features in the training ste. I will then use this scaler to transfrom both the numerical features in the training set and in the test set.

[ ]:
```
scaler = StandardScaler()

scaler.fit(X_train_new[['distance_from_home','distance_from_last_transaction',\
                        'ratio_to_median_purchase_price']])

X_train_new[['distance_from_home','distance_from_last_transaction',\
             'ratio_to_median_purchase_price']] = \
scaler.transform(X_train_new[['distance_from_home',\
                              ␣
 ↪'distance_from_last_transaction','ratio_to_median_purchase_price']])

X_test[['distance_from_home','distance_from_last_transaction',\
        'ratio_to_median_purchase_price']] = \
scaler.transform(X_test[['distance_from_home',\
                         ␣
 ↪'distance_from_last_transaction','ratio_to_median_purchase_price']])
```

Below I create a function which acts as a framework for the different densely connected neural networks that I am going to fit to the training data through cross-validation. This function:

-initializes a neural netowrk with an input layer, whose dimension is equal to the number of features (8). The activation function of the input layer is specified by the 'activation' argument.
-sequntially adds more layers to the neural network until the number of hidden layers specified by the "layer" argument is reached. The activation function of each hideen layer is specified by the 'activation' argument
-adds an output layer which uses a sgmoid activation function to produce the output
-compiles the model by using the "SGD" optimizer and 'binary crossentropy loss'

[ ]:
```
def create_model(input_dim, layers=1, neurons=10, learning_rate=0.01,␣
 ↪activation='relu', batch_size =100, epochs = 5):
    # Initialize the constructor
    model = tf.keras.Sequential()

    # Add input layer
    model.add(tf.keras.layers.Dense(neurons, input_dim=input_dim,␣
 ↪activation=activation))

    # Add hidden layers
    for _ in range(layers):
        model.add(tf.keras.layers.Dense(neurons, activation=activation))

    # Add output layer
    model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

    # Compile model
    optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    model.compile(loss='binary_crossentropy', optimizer=optimizer,␣
 ↪metrics=['accuracy'])
```

```
    return model
```

Below I define my parameter grid and create all possible combinations from this grid. I also create a list, "splits" containing all of the indices where the training set will be split.

[ ]:
```
param_grid = {
    'layers': [2, 3],
    'neurons': [50,80],
    'learning_rate': [0.01, 0.1],
    'activation': ['tanh','relu'],
    'batch_size': [50, 80],
    'epochs' : [2, 3]
}

param_combinations = list(product(*param_grid.values()))

kf = KFold(n_splits=3)
splits = list(kf.split(X_train_new))
```

The code below fits all possible combinations of our parameters on the training set thorugh cross-validation. It firs creates a list, 'results' which will contain the mean accuracy of every single model accross all folds. Then, it:

-goes through every single paramter combination from our grid
-it creates a dictionary where the keys are the parameters which are going to be used to construct the neural network, and the values are the specific values taken from a given parameter combination
-it initializes a list called 'accuracies' which will contain the accuracies on each validation fold
-it goes trhough every possible combination where 2/3 folds of the training set are actually going to be used for training and the last fold is going to be used for validation
-it fits a neural network model using the current parameter combination on the 2/3 folds which are going to be used for training. -it obtains the accuracy of this model on the validation fold and adds the value of this accuracy to the accuracies list
-after the accuracies list is completed, its mean value is calculated and added tothe results list along with the combination of parameters that led to that mean value
-it identifies the paramter combination which provided the highest mean accuracy accross all folds and prints this combination along with the mean accuracy score

[ ]:
```
results = []
for params in param_combinations:
    param_dict = dict(zip(param_grid.keys(), params))
    print(param_dict)
    accuracies = []
    for train_idx, val_idx in splits:
        X_train_fold, X_val_fold = X_train_new.iloc[train_idx], X_train_new.
 ↪iloc[val_idx]
        y_train_fold, y_val_fold = y_train_new.iloc[train_idx], y_train_new.
 ↪iloc[val_idx]
```

```
        print(f"Training fold size: {len(X_train_fold)}, Validation fold size:␣
↪{len(X_val_fold)}")

        # Extract batch_size and epochs from param_dict
        batch_size = param_dict['batch_size']
        epochs = param_dict['epochs']

        # Ensure the input_dim is correctly provided to the model
        model = create_model(input_dim=X_train_fold.shape[1], **param_dict)
        model.fit(X_train_fold, y_train_fold, epochs=epochs,␣
↪batch_size=batch_size, shuffle = True, verbose=2)
        _, accuracy = model.evaluate(X_val_fold, y_val_fold, batch_size =␣
↪batch_size, verbose=2)
        accuracies.append(accuracy)

    mean_accuracy = np.mean(accuracies)
    print(mean_accuracy)
    results.append((param_dict, mean_accuracy))

# Find the best parameter combination
best_params, best_score = max(results, key=lambda x: x[1])
print("Best Params:", best_params)
print("Best Score:", best_score)
```

Best Params: {'layers': 2, 'neurons': 80, 'learning_rate': 0.1, 'activation': 'tanh', 'batch_size': 50, 'epochs': 3}

```
Best Score: 0.9980536301930746
```

[ ]:
```
# Recreate the model using the best parameters
best_params = {'layers': 2, 'neurons': 80, 'learning_rate': 0.1, 'activation':␣
↪'tanh', 'batch_size': 50, 'epochs': 3}

best_model = create_model(input_dim=X_train_new.shape[1], **best_params)

epochs = best_params['epochs']
batch_size = best_params['batch_size']


# Train the model on the entire training dataset
best_model.fit(X_train_new, y_train_new, epochs=epochs, batch_size=batch_size,␣
↪verbose=2)
```

```
Epoch 1/3
29204/29204 - 69s - loss: 0.0192 - accuracy: 0.9933 - 69s/epoch - 2ms/step Epoch
2/3
29204/29204 - 70s - loss: 0.0070 - accuracy: 0.9974 - 70s/epoch - 2ms/step
```

```
Epoch 3/3
29204/29204 - 77s - loss: 0.0053 - accuracy: 0.9979 - 77s/epoch - 3ms/step
```

[ ]:
```
<keras.src.callbacks.History at 0x7eff0211f100>
```

[ ]:
```
# Evaluate the model on the test set
pred = best_model.predict(X_test)

pred = (pred>0.5).astype('int')

accuracy= accuracy_score(y_test, pred)
precision = precision_score(y_test, pred)
recall = recall_score(y_test, pred)
f1 = f1_score(y_test, pred)
#loss, accuracy = best_model.evaluate(X_test, y_test, verbose=2)
print(f"Test Accuracy: {accuracy}, Test precision: {precision}, Test Recall :␣
 ↪{recall}, Test F1 Score: {f1}")
```

```
6250/6250 [==============================] - 17s 3ms/step
Test Accuracy: 0.99813, Test precision: 0.9797520892927253, Test Recall :
0.9992563354499171, Test F1 Score: 0.9894080996884735
```

[ ]:
```
print(confusion_matrix(y_test, pred))
```

```
[[182158    361]
 [    13  17468]]
```

0.1 Using KNN

[ ]:
```
scaler_knn = StandardScaler()

X_train_new_knn = pd.DataFrame(scaler.fit_transform(X_train_new_knn), columns =␣
 ↪X_train_new_knn.columns)
X_test_new_knn = pd.DataFrame(scaler.transform(X_test_new_knn), columns =␣
 ↪X_test_new_knn.columns)
```

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier

neighbors = [3,5,8]
results_knn = []
for neighbor in neighbors:
  accuracies = []
  for train_idx, val_idx in splits:
    print(neighbor)
    X_train_fold, X_val_fold = X_train_new_knn.iloc[train_idx], X_train_new_knn.
 ↪iloc[val_idx]
    y_train_fold, y_val_fold = y_train_new.iloc[train_idx], y_train_new.
 ↪iloc[val_idx]
```

```
    model = KNeighborsClassifier(n_neighbors = neighbor)
    model.fit(X_train_fold, y_train_fold)
    accuracy = model.score(X_val_fold, y_val_fold)
    accuracies.append(accuracy)

  mean_accuracy = np.mean(accuracies)
  print(mean_accuracy)
  results_knn.append((neighbor, mean_accuracy))

best_params_knn, best_score_knn = max(results_knn, key=lambda x: x[1])
print("Best Number of Neighbors:", best_params_knn)
print("Best Score:", best_score_knn)
```

Best Number of Neighbors: 3
Best Score: 0.9987241095372829

[ ]:
```
# Evaluate the model on the test set
best_model_knn = KNeighborsClassifier(n_neighbors = best_params_knn)
best_model_knn.fit(X_train_new, y_train_new)
pred_knn = best_model_knn.predict(X_test_new_knn)

accuracy_knn = accuracy_score(y_test, pred_knn)
precision_knn = precision_score(y_test, pred_knn)
recall_knn = recall_score(y_test, pred_knn)
f1_knn = f1_score(y_test, pred_knn)

print(f"Test Accuracy: {accuracy_knn},Test precision: {precision_knn},Test␣
↪Recall : {recall_knn}, Test F1 Score: {f1_knn}")
```

```
Test Accuracy: 0.98117, Test precision: 0.8321627512714943, Test Recall :
0.9827813054173102, Test F1 Score: 0.9012222630226092
```

[ ]:
```
print(confusion_matrix(y_test, pred_knn))
```

```
[[179054   3465]
 [   301  17180]]
```

0.2 Using Logistic Regression

[ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix

logit = LogisticRegression()
logit.fit(X_train_new, y_train_new)

y_pred_logit = logit.predict(X_test)

print(accuracy_score(y_test, y_pred_logit))
print(precision_score(y_test, y_pred_logit))
print(recall_score(y_test, y_pred_logit))
print(f1_score(y_test, y_pred_logit))
```

```
0.933585
0.57257451075306
0.9473142268748928
0.7137469560157748
```

[ ]:
```
print(confusion_matrix(y_test, y_pred_logit))
```

```
[[170157  12362]
 [   921  16560]]
```