Author: Elaine Ng

Date: December 7, 2020

Final Project: Standard Project - Credit Card Transaction

Problem Statement:

For this project, I will look at a large set of credit card transaction data in an attempt to build a model that can predict whether an activity is fraudulent or not. I will first grab the data set, clean and prepare it for modeling. Afterwards, I will try four different models: Binary Logistic Regression, Random Forest Classifier, AdaBoost, and XGBoost. Lastly, I will evaluate these models based on their F1 Score, AUROC, and Confusion Matrices.

Variable Description:

1) accountNumber - The account number.

2) customerId - The customer's ID.

3) creditLimit - The maximum outstanding balance one can have on the credit card without being penalized.

4) availableMoney - Money available on the credit card.

5) transactionDateTime - The time and day this transaction occurred.

6) transactionAmount - Transaction Amount.

7) merchantName - Name/Company name of the seller.

8) acqCountry - The country of the bank that processes and settles the credit card transaction.

9) merchantCountryCode - The country where the merchant is from.

10) posEntryMode - It is a code that tells the processer how the transaction was captured.

```
02: Magnetic stripe read.
05: Integrated circuit card read.
09: PAN entry via electronic commerse, including chip.
80: Chip card was unable to process/magnetic stripe read default.
90: Magnetic stripe read (CVV check is possible).
```

11) posConditionCode - A code identifying transaction condition at the point of sale or point of service.

```
01: Cardholder not present.
08: Mail/Telephone order.
99: N/A
```

12) merchantCategoryCode - Code for the type of business or service the company offers.

13) currentExpDate - Expiration date of the credit card.

14) accountOpenDate - The date the credit card account was opened.

15) dateOfLastAddressChange - The date they last changed their address.

16) cardCVV - A three or four digit number on a credit card that adds an extra layer of security.

17) enteredCVV - The CVV entered for the transaction.

18) transactionType - The type of transaction:

```
1) Purchase
2) Address Verification
3) Reversal
```

19) isFraud - States whether a transaction was a fraud or not.

20) echoBuffer - Echo Buffer.

21) currentBalance - The amount owed on your account.

22) merchantCity - City of the merchant.

23) merchantState - State of the merchant.

24) merchantZip - Zip Code of the merchant.

25) cardPresent - Card present at the time of the transaction.

26) posOnPremises - POS system were on premise, meaning they used an on-site server and could only run in a specific area in a merchant's store.

27) recurringAuthInd - Recurring something.

28) expirationDateKeyInMatch - Expiration date key in match.

In [1]:

```
#Importing all the necessary libraries

#The basics
import json
import pandas as pd
import statistics
import numpy as np
from numpy import array
from scipy import stats
import datetime
pd.set_option('display.max_columns', None)

#Visualizations
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", color_codes=True)

#Modelling
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import AdaBoostClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold

#Evaluations
from sklearn import metrics
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.model_selection import cross_val_score
from sklearn.metrics import roc_auc_score

#Resampling
from imblearn.under_sampling import TomekLinks
from imblearn.under_sampling import OneSidedSelection
from imblearn.under_sampling import CondensedNearestNeighbour
from imblearn.under_sampling import EditedNearestNeighbours
from imblearn.under_sampling import NearMiss
```

PART 1: Data Preparation

I will first, be loading the data set using the starter code provided in the zip file.

In [2]:

```
#Loading the data using the starter code included in the zip file

def load_data(data):
    parse_data=data[0].keys()
    trans_data={}
    for i in parse_data:
        trans_data[i]=list()
    for row in data:
        for keys,values in row.items():
            if values=='':
                trans_data[keys].append(float("NaN")) #missing datapoints are assigned w
ith 'NaN'
            else:
                trans_data[keys].append(values)
    return pd.DataFrame.from_dict(trans_data)

#Opening the file
contents = open("transactions.txt", "r").read()
list_dict = [json.loads(str(item)) for item in contents.strip().split('\n')]
transaction_df=load_data(list_dict)

#Transaction data Data Frame
display(transaction_df)
```

	accountNumber	customerId	creditLimit	availableMoney	transactionDateTime	transactionAmount	merchantName	ac
0	733493772	733493772	5000.0	5000.00	2016-01-08T19:04:50	111.33	Lyft	
1	733493772	733493772	5000.0	4888.67	2016-01-09T22:32:39	24.75	Uber	
2	733493772	733493772	5000.0	4863.92	2016-01-11T13:36:55	187.40	Lyft	
3	733493772	733493772	5000.0	4676.52	2016-01-11T22:47:46	227.34	Lyft	
4	733493772	733493772	5000.0	4449.18	2016-01-16T01:41:11	0.00	Lyft	
...	...	...	...	...	...	...	...	
641909	186770399	186770399	7500.0	2574.02	2016-12-04T12:29:21	5.37	Apple iTunes	
641910	186770399	186770399	7500.0	2568.65	2016-12-09T04:20:35	223.70	Blue Mountain eCards	
641911	186770399	186770399	7500.0	2344.95	2016-12-16T07:58:23	138.42	Fresh Flowers	
641912	186770399	186770399	7500.0	2206.53	2016-12-19T02:30:35	16.31	abc.com	
641913	186770399	186770399	7500.0	2190.22	2016-12-28T11:14:14	32.53	Next Day Online Services	

641914 rows × 29 columns

In [3]:

```
#Giving the missing value percentages for each column
missingvalues = []

#Getting the total number of activities/number of rows
total = len(transaction_df.index)

#Looping through all the columns to count and calculate how many are NaN in each column
for i in range(len(transaction_df.columns)):
    missingvalues.append(transaction_df[transaction_df.columns[i]].isnull().sum() / tota
l * 100)

#Creating a dataframe to display the missing value percentages for each column
missingvalues_df = pd.DataFrame(missingvalues)
missingvalues_df = missingvalues_df.T
missingvalues_df.columns = transaction_df.keys()
missingvalues_df = missingvalues_df.T
```

```
missingvalues_df.columns = ['Missing Value Percentages']
display(missingvalues_df)
```

	Missing Value Percentages
accountNumber	0.000000
customerId	0.000000
creditLimit	0.000000
availableMoney	0.000000
transactionDateTime	0.000000
transactionAmount	0.000000
merchantName	0.000000
acqCountry	0.609583
merchantCountryCode	0.097209
posEntryMode	0.521098
posConditionCode	0.044710
merchantCategoryCode	0.000000
currentExpDate	0.000000
accountOpenDate	0.000000
dateOfLastAddressChange	0.000000
cardCVV	0.000000
enteredCVV	0.000000
cardLast4Digits	0.000000
transactionType	0.091757
isFraud	0.000000
echoBuffer	100.000000
currentBalance	0.000000
merchantCity	100.000000
merchantState	100.000000
merchantZip	100.000000
cardPresent	0.000000
posOnPremises	100.000000
recurringAuthInd	100.000000
expirationDateKeyInMatch	0.000000

In the dataframe above, we can see that the columns echoBuffer, merchantCity, merchantState, merchantZip, posOnPremises, and recurringAuthInd has a 100% missing values. Since there is not even a single input for those features, I will be removing them and not considering them for modeling.

In [4]:

```
#Removing columns with 100% missing values

#echoBuffer
transaction_df = transaction_df.drop('echoBuffer', 1)

#merchantCity
transaction_df = transaction_df.drop('merchantCity', 1)

#merchantState
transaction_df = transaction_df.drop('merchantState', 1)

#merchantZip
```

```
transaction_df = transaction_df.drop('merchantZip', 1)

#posOnPremises
transaction_df = transaction_df.drop('posOnPremises', 1)

#recurringAuthInd
transaction_df = transaction_df.drop('recurringAuthInd', 1)
```

For the variables that do have some missing values: acqCountry, merchantCountryCode, posEntryMode, posConditionCode, and transactionType. I will be filling those NaN values with the column's mode. I made this decision due to noticing that the values for these columns are categorical, so it made more sense to use the mode instead of the mean. First, I will find their mode and then replace the NaN values with the mode.

In [5]:

```
#Finding the mode for the variables that have missing values

#acqCountry
print('acqCountry mode:', statistics.mode(transaction_df[transaction_df.columns[7]]))

#merchantCountryCode
print('merchantCountryCode mode:', statistics.mode(transaction_df[transaction_df.columns[
8]]))

#posEntryMode
print('posEntryMode mode:', statistics.mode(transaction_df[transaction_df.columns[9]]))

#posConditionCode
print('posConditionCode mode:', statistics.mode(transaction_df[transaction_df.columns[10
]]))

#transactionType
print('transactionType mode:', statistics.mode(transaction_df[transaction_df.columns[18]
]))
```

```
acqCountry mode: US
merchantCountryCode mode: US
posEntryMode mode: 05
posConditionCode mode: 01
transactionType mode: PURCHASE
```

In [6]:

```
#Replacing all the NaN values for these variables with their mode

transaction_df['acqCountry'] = transaction_df['acqCountry'].fillna('US')
transaction_df['merchantCountryCode'] = transaction_df['merchantCountryCode'].fillna('US'
)
transaction_df['posEntryMode'] = transaction_df['posEntryMode'].fillna('05')
transaction_df['posConditionCode'] = transaction_df['posConditionCode'].fillna('01')
transaction_df['transactionType'] = transaction_df['transactionType'].fillna('PURCHASE')
```

In the process of cleaning and preparing my data set, I noticed that there are a lot of date features. Since, the values in these columns are string types, I had to convert it into a datetime. But, I also looked at all the dates columns and made the decision to only pay attention to the transactionDateTime, because I did not think that the currentExpDate, accountOpenDate, and dateOfLastAddressChange would have much of an impact on whether an activity is fraudulent or not. For the transactionDateTime, I split it into Year, Month, and Day. Then I realized all the observations are from 2016, so the Year does not tell me anything, hence I decided to only keep the Month and Day, which I would treat as categorical, because I did not want my model to think these were continuous variables. But to encode all the Months and Days would give me a lot of dummy variables, that is why I decided to divide the months into seasons; spring (March-May), summer (June-August), autumn (September-November), and winter (December-February). Then for the days, I split them into early (1-10), mid (11-20), and late month (21-30/31). Afterwards, I added these two new categorical columns into my transaction_df.

In [7]:

```
#Dealing with dates
```

```
#Dealing with dates

date = []
transactionDay = []
transactionDayCat = []
transactionMonth = []
transactionMonthCat = []

#Changing the string into datetime type
for i in range(len(transaction_df.index)):
    string = transaction_df['transactionDateTime'][i]
    date.append(datetime.datetime.strptime(string, "%Y-%m-%dT%H:%M:%S"))


#Getting the days
for j in range(len(date)):
    transactionDay.append(date[j].day)

#Sorting them into early, mid, and late month
for k in range(len(transactionDay)):
    if 1 <= transactionDay[k] <= 10:
        transactionDayCat.append('early')
    elif 11 <= transactionDay[k] <= 20:
        transactionDayCat.append('mid')
    else:
        transactionDayCat.append('late')

#Getting the months
for l in range(len(date)):
    transactionMonth.append(date[l].month)

#Sorting them into winter, spring, summer, and autumn
for m in range(len(transactionDay)):
    if transactionMonth[m] in [12, 1, 2]:
        transactionMonthCat.append('winter')
    elif transactionMonth[m] in [3, 4, 5]:
        transactionMonthCat.append('spring')
    elif transactionMonth[m] in [6, 7, 8]:
        transactionMonthCat.append('summer')
    elif transactionMonth[m] in [9, 10, 11]:
        transactionMonthCat.append('autumn')

#Turning the arrays into dataframes and adding them to the end of transaction_df
transactionDayCat_df = pd.DataFrame(transactionDayCat)
transactionDayCat_df.columns = ['transactionDayCat']
transactionMonthCat_df = pd.DataFrame(transactionMonthCat)
transactionMonthCat_df.columns = ['transactionMonthCat']
transaction_df = transaction_df.join(transactionDayCat_df)
transaction_df = transaction_df.join(transactionMonthCat_df)
```

What I also noticed while preparing my data set, is that there are two columns: cardCVV and enteredCVV, which are numbers and I also did not want my models to assume they were continuous. That would mean, I need to treat them as categorical, but that would create a lot of dummy variables. I felt that what was important in these two columns, is whether the entered CVV for the transaction is the card CVV. So, I made a new variable called cardCVV_match, that assigns it a 1 if the entered CVV matches the card CVV and a 0 if it was not a match, and I added this new column into the transaction_df.

In [8]:

```
#Dealing with Credit Card information

cardCVV_match = []

for i in range(len(transaction_df.index)):
    if transaction_df['cardCVV'][i] == transaction_df['enteredCVV'][i]:
        cardCVV_match.append(1)
    else:
        cardCVV_match.append(0)

cardCVVmatch_df = pd.DataFrame(cardCVV_match)
```

```
cardCVVmatch_df.columns = ['cardCVV_match']
transaction_df = transaction_df.join(cardCVVmatch_df)
```

Now, my data set is left with a lot of categorical features. Namely: merchantName, acqCountry, merchantCountryCode, posEntryMode, posConditionCode, merchantCategoryCode, transactionType, transactionDayCat, transactionMonthCat, and cardCVV_match. Since, One Hot Encoding has been suggested, I will perform this encoding to my categorical features. For the binary features: isFraud, cardPresent, and expirationDateKeyInMatch, I decided that a one hot encoding for them will be a little bit redundant, so I just assigned them a 1 if it is True and a 0 if it is False. Lastly, I added all these new encoded columns into transaction_df.

In [9]:

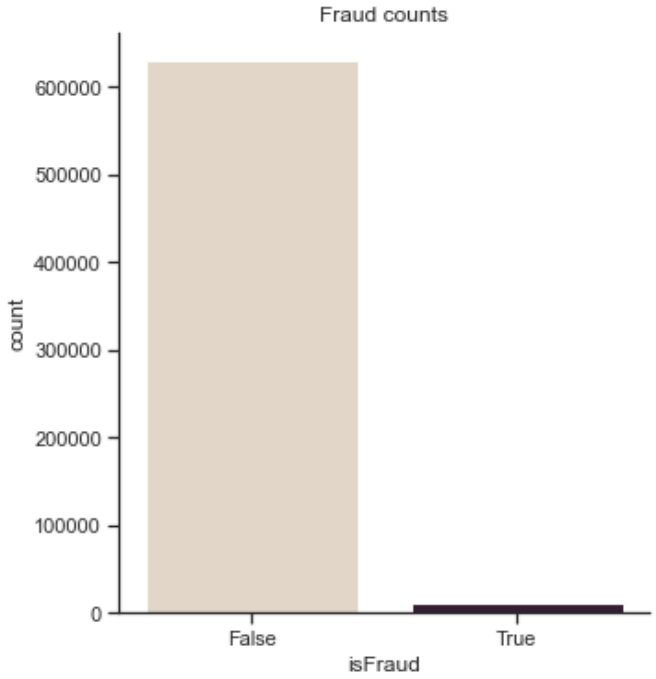
```
#Encoding categorical data

#One-hot encoding the categorical columns
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['acqCountry'], prefix
= 'acqCountry'))
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['merchantCountryCode']
, prefix = 'merchantCountryCode'))
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['posEntryMode'], prefi
x = 'posEntryMode'))
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['posConditionCode'], p
refix = 'posConditionCode'))
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['merchantCategoryCode'
], prefix = 'merchantCategoryCode'))
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['transactionType'], p
refix = 'transactionType'))
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['transactionDayCat'],
prefix = 'transactionDayCat'))
transaction_df = transaction_df.join(pd.get_dummies(transaction_df['transactionMonthCat']
, prefix = 'transactionMonthCat'))

#Encoding binary columns
isFraud_binary = []
cardPresent_binary = []
expirationDateKeyInMatch_binary = []

#isFraud
for i in range(len(transaction_df.index)):
    if transaction_df['isFraud'][i] == True:
        isFraud_binary.append(1)
    else:
        isFraud_binary.append(0)

#cardPresent
for j in range(len(transaction_df.index)):
    if transaction_df['cardPresent'][j] == True:
        cardPresent_binary.append(1)
    else:
        cardPresent_binary.append(0)

#expirationDateKeyInMatch
for k in range(len(transaction_df.index)):
    if transaction_df['expirationDateKeyInMatch'][k] == True:
        expirationDateKeyInMatch_binary.append(1)
    else:
        expirationDateKeyInMatch_binary.append(0)

#Turning the arrays into dataframes and adding them to the end of transaction_df
isFraud_binary_df = pd.DataFrame(isFraud_binary)
isFraud_binary_df.columns = ['isFraud_binary']
cardPresent_binary_df = pd.DataFrame(cardPresent_binary)
cardPresent_binary_df.columns = ['cardPresent_binary']
expirationDateKeyInMatch_binary_df = pd.DataFrame(expirationDateKeyInMatch_binary)
expirationDateKeyInMatch_binary_df.columns = ['expirationDateKeyInMatch_binary']
transaction_df = transaction_df.join(cardPresent_binary_df)
transaction_df = transaction_df.join(expirationDateKeyInMatch_binary_df)
transaction_df = transaction_df.join(isFraud_binary_df)
```

PART 2: Data Visualization

For this first plot below, I decided I wanted to see the number of Non-fraudulent observations vs the fraudulent observations. Which appears very obvious, we have a huge data imbalance.

In [10]:

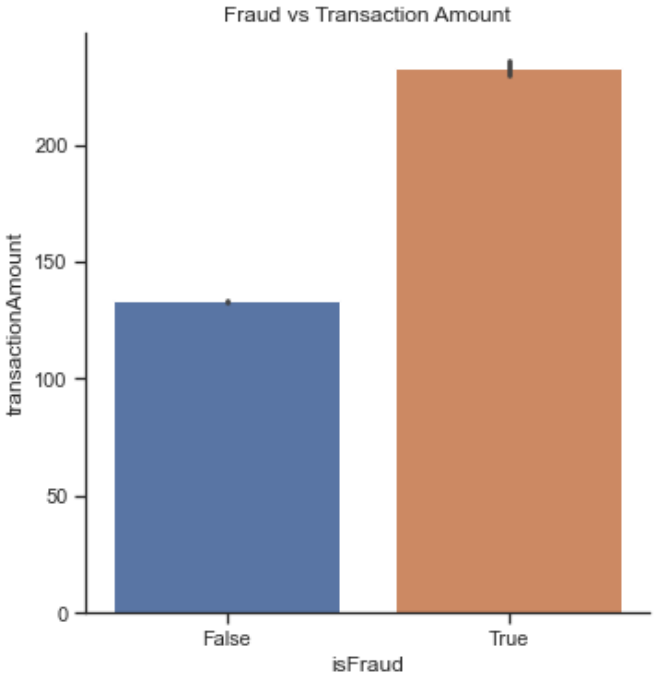
```
#Visualizing isFraud
sns.catplot(x="isFraud", kind="count",palette="ch:.25", data=transaction_df).set(title='
Fraud counts')
plt.show()
```



In the second plot below, I wanted to see the transaction amount when a transaction is a fraud and when it is not a fraud. It appears that when a transaction is a fraud, the transaction amount is a lot higher.

In [11]:

```
#Visualizing the transaction amount when it is fraud and non-fraud
sns.catplot(x="isFraud", y="transactionAmount", kind="bar", data=transaction_df).set(titl
e='Fraud vs Transaction Amount')
plt.show()
```

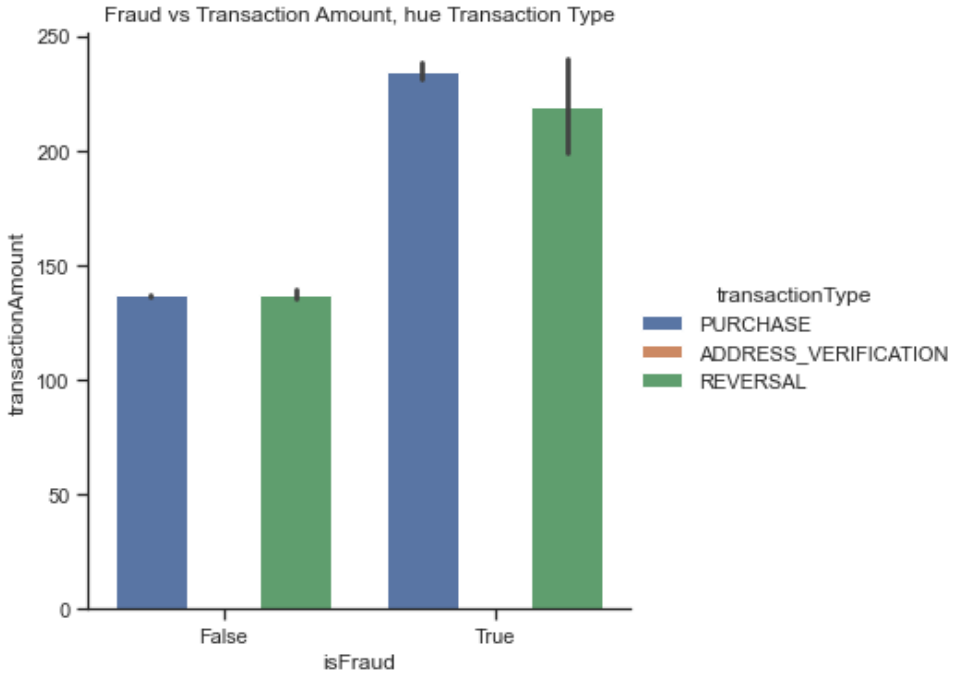


In the third plot below, I wanted to know when an activity is fraudulent, what type of transaction it most likely is

and what is the transaction amount. The plot shows us that very few transactions are for address verification, and when we have a fraudulent transaction there is a pretty good chance that the transaction amount is high and the transaction type is purchase.

In [12]:

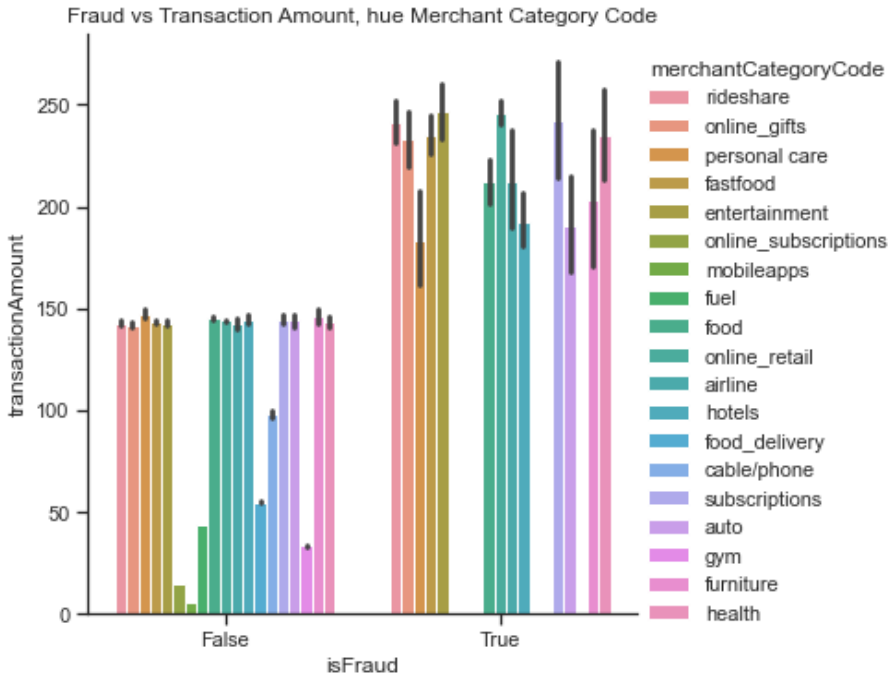
```
#Visualizing fraud vs transaction amount with transaction type hue
sns.catplot(x="isFraud", y="transactionAmount", hue="transactionType", kind="bar", data=
transaction_df).set(title='Fraud vs Transaction Amount, hue Transaction Type')
plt.show()
```



Similar to the third plot, in this fourth plot I was curious which type of merchant category usually has the highest amount of transaction when there is a fraudulent activity happening. Looking at the plot: entertainment, subscription, and online retail has the highest transaction amount for when it is a fraud. Also, perhaps phone/cable frauds are not very common?

In [13]:

```
#Visualizing fraud vs transaction amount with merchant category code hue
sns.catplot(x="isFraud", y="transactionAmount", hue="merchantCategoryCode", kind="bar",
data=transaction_df).set(title='Fraud vs Transaction Amount, hue Merchant Category Code')
plt.show()
```



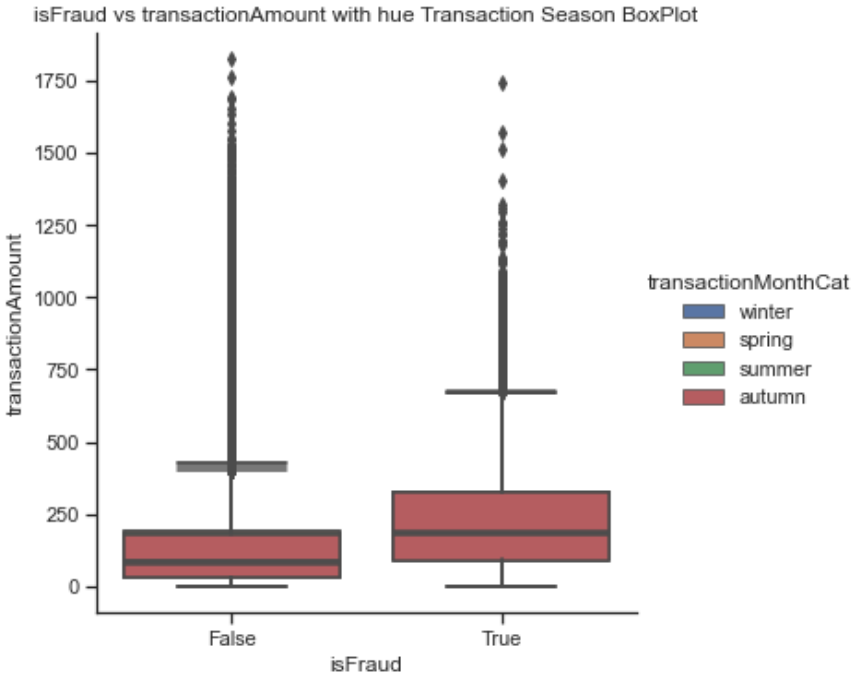
For this next plot, I wanted to see if the season has anything to do with whether a transaction is a fraud or not and how much money is involved. It seems that most transaction amount are in the 100 - 200 range and happens mostly in autumn. It is also very skewed.

In [18]:

```
sns.catplot(x="isFraud", y="transactionAmount", hue="transactionMonthCat",
            kind="box", dodge=False, data=transaction_df).set(title='isFraud vs transact
ionAmount with hue Transaction Season BoxPlot')
```

Out[18]:

```
<seaborn.axisgrid.FacetGrid at 0x16079912eb0>
```



For my next four plots, I wanted to see some of the variables when they are from fraudulent observations and comparing them to the non-fraudulent observations. Therefore, I will split the data into two groups: fraud_true and fraud_false.

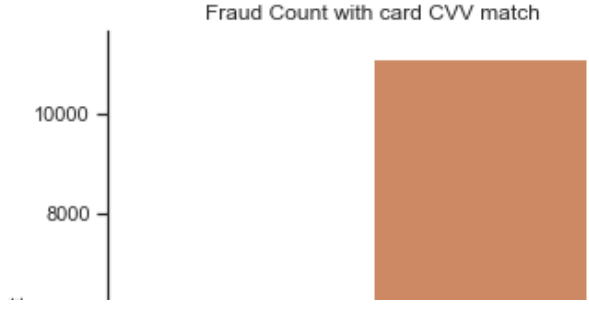
In [15]:

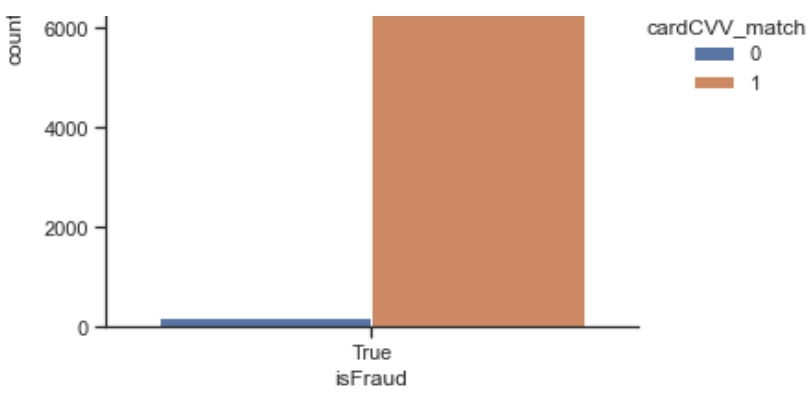
```
#Splitting the data into Fraud and Non-Fraud for plotting purposes

fraud_true = transaction_df[transaction_df['isFraud'] == True]
fraud_false = transaction_df[transaction_df['isFraud'] == False]
```

The two plots below, shows the number of fraudulent and non-fraudulent activities and whether their entered CVV matched their card CVV. We can kind of see that the blue bar for fraudulent plot is a bit bigger relative to its own ornage bar compared to the non-fraudulent one.

In [16]:

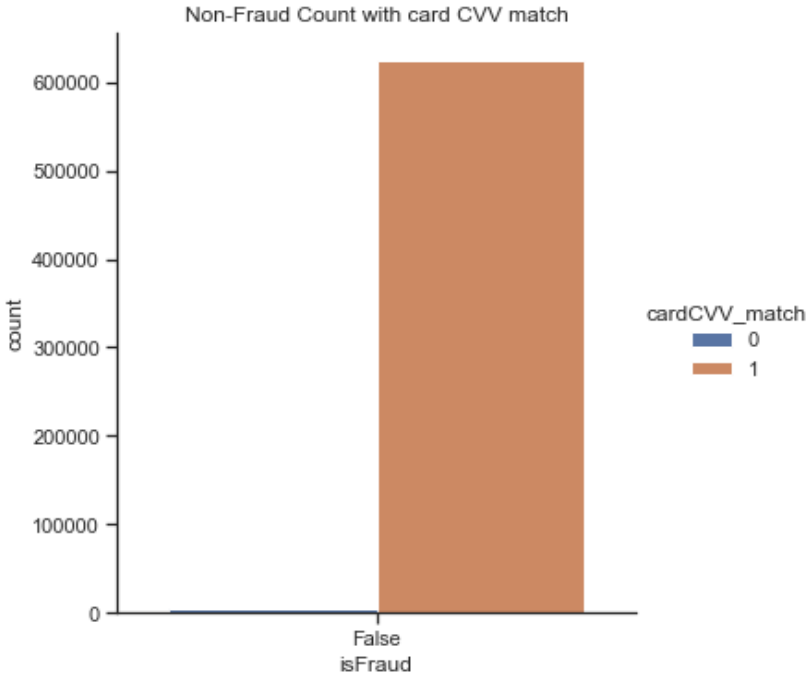
```
#Fraudulent observations and card CVV matches. 1 is a match and 0 is not a match.
sns.catplot(x='isFraud', hue='cardCVV_match', kind="count", data = fraud_true).set(title
='Fraud Count with card CVV match')
plt.show()
```





In [17]:

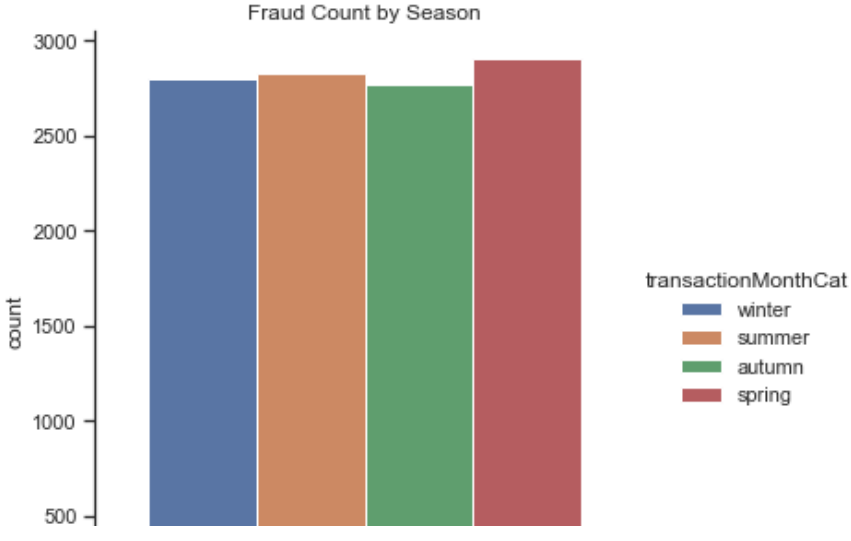
```
#Non-fraudulent observations and card CVV matches. 1 is a match and 0 is not a match.
sns.catplot(x='isFraud', hue='cardCVV_match', kind="count", data = fraud_false).set(titl
e='Non-Fraud Count with card CVV match')
plt.show()
```



Comparing the Non-fraud and the Fraud count by season plots below, it seems like most transactions happen during autumn, but in spring there is slightly a bit more fraudulent activities happening than the rest of the year.

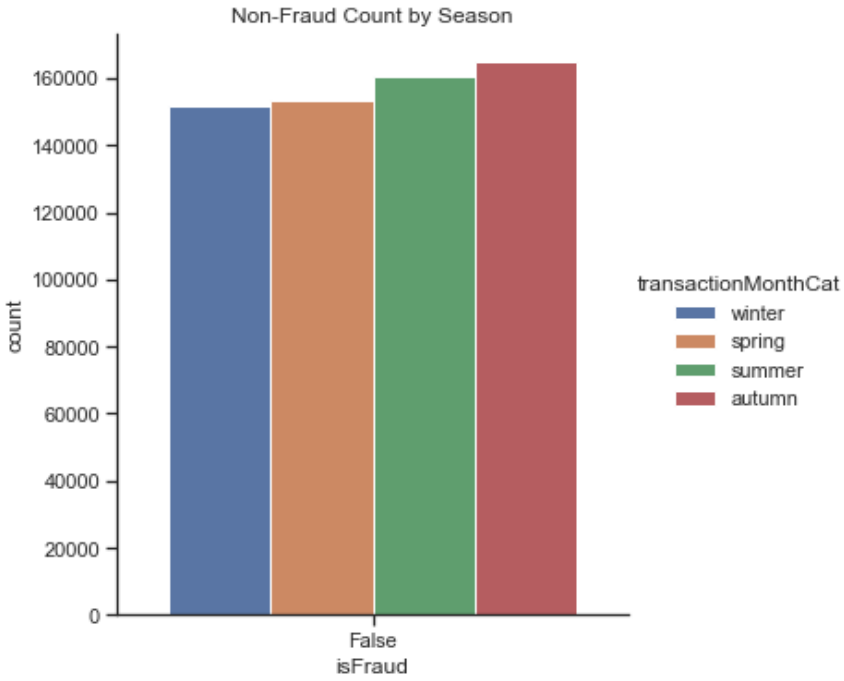
In [19]:

```
#Showing the counts of fraudulent transactions by season
sns.catplot(x='isFraud', hue='transactionMonthCat', kind="count", data = fraud_true).set
(title='Fraud Count by Season')
plt.show()
```



In [20]:

```
#Showing the counts of non-fraudulent transactions by season
sns.catplot(x='isFraud', hue='transactionMonthCat', kind="count", data = fraud_false).se
t(title='Non-Fraud Count by Season')
plt.show()
```



I have tried plotting other types of graphs appropriate for categorical data, such as a strip plot and rel plot. But, due to the large amount of observations, these graphs were not insightful. So, I stuck mostly with bar plots.

Now that I have plotted some of our data, I will remove the old columns that I have already encoded, and the columns that I have previously mentioned dropping.

In [21]:

```
#Dropping the original categorical columns and previously mentioned columns
transaction_df = transaction_df.drop('acqCountry', 1)
transaction_df = transaction_df.drop('merchantCountryCode', 1)
transaction_df = transaction_df.drop('merchantCategoryCode', 1)
transaction_df = transaction_df.drop('transactionType', 1)
transaction_df = transaction_df.drop('merchantName', 1)
transaction_df = transaction_df.drop('currentExpDate', 1)
transaction_df = transaction_df.drop('accountOpenDate', 1)
transaction_df = transaction_df.drop('dateOfLastAddressChange', 1)
transaction_df = transaction_df.drop('posEntryMode', 1)
transaction_df = transaction_df.drop('posConditionCode', 1)
transaction_df = transaction_df.drop('accountNumber', 1)
transaction_df = transaction_df.drop('customerId', 1)
transaction_df = transaction_df.drop('transactionDateTime', 1)
transaction_df = transaction_df.drop('isFraud', 1)
transaction_df = transaction_df.drop('cardPresent', 1)
transaction_df = transaction_df.drop('expirationDateKeyInMatch', 1)
transaction_df = transaction_df.drop('cardCVV', 1)
transaction_df = transaction_df.drop('enteredCVV', 1)
transaction_df = transaction_df.drop('cardLast4Digits', 1)
transaction_df = transaction_df.drop('transactionDayCat', 1)
transaction_df = transaction_df.drop('transactionMonthCat', 1)
```

PART 3: Modeling and Cross Validation

I will first split the original data set of 641914 observations into 20% testing set and 80% training set. I also know

I will first split the original data set of 641914 observations into 20% testing set and 80% training set. I also know that I will be doing some parameters tuning for my models, therefore I have set aside 20% of the training data set for validation.

In [22]:

```
#Splitting the data set into 80% train and 20% test

X = transaction_df.loc[:,list(transaction_df.columns[0:52])]
y = transaction_df.loc[:,list(transaction_df.columns[52:53])]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=82
90)
```

In [23]:

```
#Splitting the train set into another 80% train and 20% validate for parameter tuning

X_traintrain, X_validate, y_traintrain, y_validate = train_test_split(X_train, y_train,
test_size=0.2, random_state=4530)
```

The first model that I am going to use to train my data is Logistic Regression. Below, I am simply fitting it with its default parameters using my training set. Afterwards, I run a 5-fold cross validation using the validation data set that I have set aside and the cross validation F1 Score for this model is 0.

In [29]:

```
#Logistic Regression (original data, normal parameters)

#Fitting the Logistic regression
logReg1 = LogisticRegression(max_iter=1000).fit(X_traintrain.to_numpy(), y_traintrain.val
ues.ravel())

#Calculating the 5-fold cross validation score
scoreLog1 = cross_val_score(logReg1, X_validate.to_numpy(), y_validate.values.ravel(), cv
=5, scoring='f1')
print('Cross Validation F1 Score for Logistic Regression:', scoreLog1.mean())
```

```
Cross Validation F1 Score for Logistic Regression: 0.0
```

The F1 Score for the cross validation is extremely low. So, my next step was to adjust the weight parameters, in order to try to counter the fact that my dataset is currently imbalanced, which is what I think is causing the low F1 Score. To find a good weight ratio, I will be performing a grid search and choosing the best parameter. Just to have an idea which parameters I should be grid searching for, I was informed a ratio of 1 to (count of majority class/count of minority class). So my weight ratio should be somewhere around 1:(630612/11302), a ratio around 1:56. Just in case, I wanted to check some lower weight ratios as well, which is why I decided to run my grid search testing 1:15, 1:20, 1:30, 1:45, 1:50, 1:56, and 1:60.

In [25]:

```
#Grid Search for Logistic Regression (original data)

#Defining a list of weight ratios to check
balance = [{0:1,1:15},{0:1,1:20},{0:1,1:30},{0:1,1:45}, {0:1,1:50}, {0:1,1:56}, {0:1,1:
60}]
param_grid = dict(class_weight=balance)

# Defining evaluation procedure using repeated stratified k-fold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Defining the grid search
grid = GridSearchCV(estimator=logReg1, param_grid=param_grid, n_jobs=-1, cv=cv, scoring=
'roc_auc')

#Fitting the grid search using the training's train set
grid_result = grid.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Printing the result
means = grid_result.cv_results_['mean_test_score']
```

```
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
0.694057 (0.012861) with: {'class_weight': {0: 1, 1: 15}}
0.698359 (0.011370) with: {'class_weight': {0: 1, 1: 20}}
0.699419 (0.009941) with: {'class_weight': {0: 1, 1: 30}}
0.701346 (0.013764) with: {'class_weight': {0: 1, 1: 45}}
0.692508 (0.026906) with: {'class_weight': {0: 1, 1: 50}}
0.695037 (0.020701) with: {'class_weight': {0: 1, 1: 56}}
0.695840 (0.020516) with: {'class_weight': {0: 1, 1: 60}}
```

The grid search result came back with 1:45 being the best parameter on average, with 1:30 being the second best. Therefore, I will be running the Logistic Regression again with this new weight adjustments.

The cross validation F1 score for the model with 1:45 is 0.0627. And for the model with 1:30 weight adjustment, we get an F1 score of 0.0836. Which is an improvement from the previous default Logistic Regression model.

In [35]:

```
#Logistic Regression (original data, new parameter)

#Fitting the Logistic regression
logReg2 = LogisticRegression(max_iter=1000, class_weight={0:1,1:45}).fit(X_traintrain.to
_numpy(), y_traintrain.values.ravel())

#Calculating the 5-fold cross validation score
scoreLog2 = cross_val_score(logReg2, X_validate.to_numpy(), y_validate.values.ravel(), cv
=5, scoring='f1')
print('Cross Validation F1 Score for Logistic Regression:', scoreLog2.mean())
```

```
Cross Validation F1 Score for Logistic Regression: 0.06274432791956328
```

In [45]:

```
#Logistic Regression (original data, new parameter)

#Fitting the Logistic regression
logReg3 = LogisticRegression(max_iter=1000, class_weight={0:1,1:30}).fit(X_traintrain.to
_numpy(), y_traintrain.values.ravel())

#Calculating the 5-fold cross validation score
scoreLog3 = cross_val_score(logReg3, X_validate.to_numpy(), y_validate.values.ravel(), cv
=5, scoring='f1')
print('Cross Validation F1 Score for Logistic Regression:', scoreLog3.mean())
```

```
Cross Validation F1 Score for Logistic Regression: 0.08360273370561308
```

The next model that I wanted to use to train my data is Random Forest. Just as before, I will intially run this model using its default parameters. The 5-fold cross validation F1 score is 0.0043. Since it is a model that I ran with its default parameters, it appears to be doing a better job than Logistic Regression so far.

In [36]:

```
#Random Forest (original data, normal parameters)

#Fitting the Random Forest
rf1 = RandomForestClassifier()
rf1.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreRF1 = cross_val_score(rf1, X_validate.to_numpy(), y_validate.values.ravel(), cv=5, s
coring='f1')
print('Cross Validation F1 Score for Random Forest:', scoreRF1.mean())
```

```
Cross Validation F1 Score for Random Forest: 0.004324355912022732
```

I have decided to perform a grid search for the Random Forest using the same weights ratios that I used for

Logisitic Regression.

The grid search result came back that the ratio 1:15 yields the best average result, and 1:20 being the second. Which is what I will be adjusting for the next two Random Forest models.

In [28]:

```
#Grid Search for Random Forest (original data)

#Defining a list of weight ratios to check
balance = [{0:1,1:15},{0:1,1:20},{0:1,1:30},{0:1,1:45}, {0:1,1:50}, {0:1,1:56}, {0:1,1:
60}]
param_grid = dict(class_weight=balance)

# Defining evaluation procedure using repeated stratified k-fold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Defining the grid search
grid = GridSearchCV(estimator=rf1, param_grid=param_grid, n_jobs=-1, cv=cv, scoring='roc
_auc')

#Fitting the grid search using the training's train set
grid_result = grid.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Printing the result
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
0.658944 (0.008414) with: {'class_weight': {0: 1, 1: 15}}
0.658491 (0.010035) with: {'class_weight': {0: 1, 1: 20}}
0.654658 (0.009610) with: {'class_weight': {0: 1, 1: 30}}
0.656351 (0.007316) with: {'class_weight': {0: 1, 1: 45}}
0.655197 (0.009787) with: {'class_weight': {0: 1, 1: 50}}
0.652556 (0.009196) with: {'class_weight': {0: 1, 1: 56}}
0.652730 (0.009438) with: {'class_weight': {0: 1, 1: 60}}
```

In [37]:

```
#Random Forest (original data, new parameter)

#Fitting the Random Forest
rf2 = RandomForestClassifier(class_weight={0:1,1:15})
rf2.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreRF2 = cross_val_score(rf2, X_validate.to_numpy(), y_validate.values.ravel(), cv=5, s
coring='f1')
print('Cross Validation F1 Score for Random Forest:', scoreRF2.mean())
```

```
Cross Validation F1 Score for Random Forest: 0.0032432432432432435
```

In [46]:

```
#Random Forest (original data, new parameter)

#Fitting the Random Forest
rf3 = RandomForestClassifier(class_weight={0:1,1:20})
rf3.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreRF3 = cross_val_score(rf3, X_validate.to_numpy(), y_validate.values.ravel(), cv=5, s
coring='f1')
print('Cross Validation F1 Score for Random Forest:', scoreRF3.mean())
```

```
Cross Validation F1 Score for Random Forest: 0.001081081081081081
```

The newly 1:15 weight ratio adjusted Random Forest yields us a cross validation F1 score of 0.0032. Followed by our third Random Forest model with 1:20 weight, the resulted F1 score for cross validation is 0.0012. Next, I will

be looking at another model, AdaBoost.

In [38]:

```
#AdaBoost (original data, normal parameters)

#Fitting the AdaBoost
ada1 = AdaBoostClassifier()
ada1.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreADA1 = cross_val_score(ada1, X_validate.to_numpy(), y_validate.values.ravel(), cv=5,
scoring='f1')
print('Cross Validation F1 Score for AdaBoost:', scoreADA1.mean())
```

```
Cross Validation F1 Score for AdaBoost: 0.0
```

In [39]:

```
#AdaBoost (original data, new parameter)

#Fitting the AdaBoost
ada2 = AdaBoostClassifier(learning_rate = 2)
ada2.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreADA2 = cross_val_score(ada2, X_validate.to_numpy(), y_validate.values.ravel(), cv=5,
scoring='f1')
print('Cross Validation F1 Score for AdaBoost:', scoreADA2.mean())
```

```
Cross Validation F1 Score for AdaBoost: 0.02545214797500515
```

In [40]:

```
#AdaBoost (original data, new parameter)

#Fitting the AdaBoost
ada3 = AdaBoostClassifier(learning_rate = 0.01)
ada3.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreADA3 = cross_val_score(ada3, X_validate.to_numpy(), y_validate.values.ravel(), cv=5,
scoring='f1')
print('Cross Validation F1 Score for AdaBoost:', scoreADA3.mean())
```

```
Cross Validation F1 Score for AdaBoost: 0.0
```

For the AdaBoost, I am able to change the learning rate which is the contribution of each model to the weight. For my first try, I fit my training set with the default AdaBoost classifier. That gave me a cross validation score of 0. If I inrease my learning rate my model will train faster, which is good because our data set is very big. I tried increasing the learning rate from 1, to 2. That resulted in a cross validation F1 score of 0.0255. If I decrease the learning rate it will force the model to train slower, but it might give better performance. Therefore, I will run another AdaBoost model using a lower learning rate at 0.01. Decreasing the learning rate gave us a cross validation score of 0, which is the same as when learning rate was 1.

The next model that I wanted to look at is the XGBoost. As usual, I start off my model training with the default parameters. That gave us a cross validation F1 score of 0.

In [41]:

```
#XGBoost (original data, normal parameters)

#Fitting the XGBoost
xgb1 = XGBClassifier()
xgb1.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreXGB1 = cross_val_score(xgb1, X_validate.to_numpy(), y_validate.values.ravel(), cv=5,
scoring='f1')
```

```
print('Cross Validation F1 Score for XGBoost:', scoreXGB1.mean())
```

```
Cross Validation F1 Score for XGBoost: 0.0
```

Since XGBoost also allows us to adjust the weight directly, so I decided to run the grid search again using 10, 15, 20, 25, 30, 35, 40, and 45 to see if I can find a good parameter.

In [42]:

```
#Grid Search for XGBoost (original data)

#Defining a list of weight ratios to check
balance = [10, 15, 20, 25, 30, 35, 40, 45]
param_grid = dict(scale_pos_weight=balance)

# Defining evaluation procedure using repeated stratified k-fold
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#Defining the grid search
grid = GridSearchCV(estimator=xgb1, param_grid=param_grid, n_jobs=-1, cv=cv, scoring='roc_auc')

#Fitting the grid search using the training's train set
grid_result = grid.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Printing the result
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
0.746819 (0.008796) with: {'scale_pos_weight': 10}
0.745994 (0.008767) with: {'scale_pos_weight': 15}
0.745186 (0.008352) with: {'scale_pos_weight': 20}
0.744950 (0.008102) with: {'scale_pos_weight': 25}
0.742516 (0.008849) with: {'scale_pos_weight': 30}
0.743085 (0.008897) with: {'scale_pos_weight': 35}
0.741700 (0.009581) with: {'scale_pos_weight': 40}
0.740545 (0.008932) with: {'scale_pos_weight': 45}
```

It seems that for XGBoost a scale pos weight of 10 has the highest AUROC score, and when fitted with that adjusment, gives us a cross validation F1 score of 0.0508. I also wanted to try the second best result, which is 15 and that resulted in a cross validation F1 score of 0.0752.

In [43]:

```
#XGBoost (original data, new parameter)

#Fitting the XGBoost
xgb2 = XGBClassifier(scale_pos_weight=10)
xgb2.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreXGB2 = cross_val_score(xgb2, X_validate.to_numpy(), y_validate.values.ravel(), cv=5, scoring='f1')
print('Cross Validation F1 Score for XGBoost:', scoreXGB2.mean())
```

```
Cross Validation F1 Score for XGBoost: 0.05078879948646057
```

In [44]:

```
#XGBoost (original data, new parameter)

#Fitting the XGBoost
xgb3 = XGBClassifier(scale_pos_weight=15)
xgb3.fit(X_traintrain.to_numpy(), y_traintrain.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreXGB3 = cross_val_score(xgb3, X_validate.to_numpy(), y_validate.values.ravel(), cv=5,
```

```
scoring='f1')
print('Cross Validation F1 Score for XGBoost:', scoreXGB3.mean())
```

```
Cross Validation F1 Score for XGBoost: 0.07522334901687089
```

Another method that I tried to use to counter the problem of this imbalanced data set, is by resampling the training data. According to my research, if there are a lot of observations we can do undersampling. If there are very few observations, we can do oversampling. Due to our data set having 641914 observations, I opted to undersample my training data.

At first, I tried to undersample using Tomek Links, but that only reduced the observations by around 22000, which left the data set still very imbalanced. So, I tried to combine that with a One-Sided Selection undersampling, but it still did not remove as many observations as I would like. After not achieving a fraud and non-fraud observations ratio that I liked, I tried undersampling with the Edited Nerest Neighbors Rule. This removed more non-fraudulent observations, but it was still not enough. I tried Condensed Nearest Neighbor, but after running it for 2+ hours it was still not complete, so I had to give that option up. After trying Near-Miss version 3 - Majority class examples with minimum distance to each minority class example, I got a completely balanced set 50% fraudulent and 50% non-fraudulent observations. I will be using this undersampled data set to train my next models.

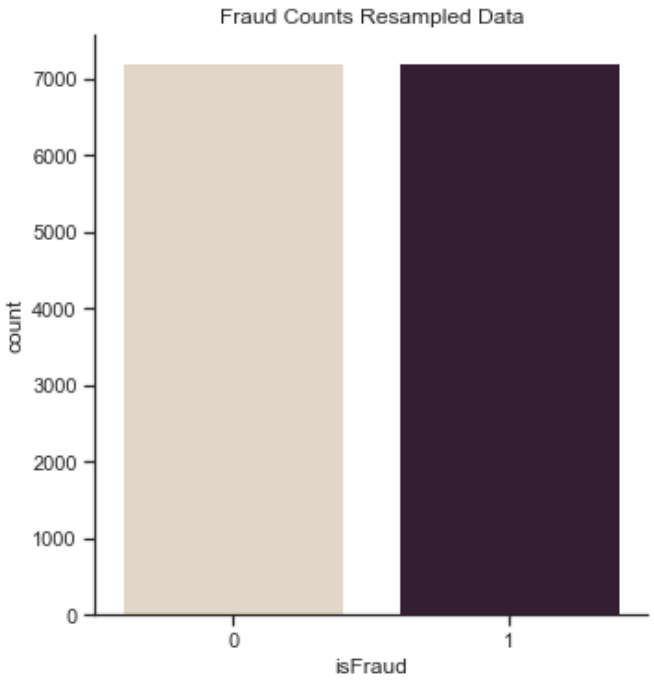
In [48]:

```
#Undersampling the training data

#Using NearMiss-3 technique to get an undersampled training set
undersample = NearMiss(version=3, n_neighbors_ver3=3)
X_undersample, y_undersample = undersample.fit_resample(X_traintrain, y_traintrain)
```

The plot shows that our undersampled data is balanced.

In [49]:

```
#Visualizing isFraud
sns.catplot(x="isFraud_binary", kind="count",palette="ch:.25", data=y_undersample).set(t
itle='Fraud Counts Resampled Data', xlabel='isFraud')
plt.show()
```



To start off with our new undersampled training data set, I started with the Logistic Regression, and testws it with cross validation on our validation set that is not undersampled. Which gives us a F1 Score of 0.

In [55]:

```
#Logistic Regression (resampled data, normal parameters)
```

```
#Fitting the Logistic regression
logReg4 = LogisticRegression(max_iter=1000).fit(X_undersample.to_numpy(), y_undersample.v
alues.ravel())

#Calculating the 5-fold cross validation score
scoreLog4 = cross_val_score(logReg4, X_validate.to_numpy(), y_validate.values.ravel(), cv
=5, scoring='f1')
print('Cross Validation F1 Score for Logistic Regression:', scoreLog4.mean())
```

```
Cross Validation F1 Score for Logistic Regression: 0.0
```

Next is training the undersampled data with the Random Forest, 5-fold cross validating this model on our validation set, yields a result of 0.0032 F1 Score.

In [56]:

```
#Random Forest (resampled data, normal parameter)

#Fitting the Random Forest
rf4 = RandomForestClassifier()
rf4.fit(X_undersample.to_numpy(), y_undersample.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreRF4 = cross_val_score(rf3, X_validate.to_numpy(), y_validate.values.ravel(), cv=5, s
coring='f1')
print('Cross Validation F1 Score for Random Forest:', scoreRF4.mean())
```

```
Cross Validation F1 Score for Random Forest: 0.0032403607383937826
```

Using AdaBoost, our model got a cross validation F1 score of 0.

In [57]:

```
#AdaBoost (resampled data, normal parameters)

#Fitting the AdaBoost
ada4 = AdaBoostClassifier()
ada4.fit(X_undersample.to_numpy(), y_undersample.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreADA4 = cross_val_score(ada4, X_validate.to_numpy(), y_validate.values.ravel(), cv=5,
scoring='f1')
print('Cross Validation F1 Score for AdaBoost:', scoreADA4.mean())
```

```
Cross Validation F1 Score for AdaBoost: 0.0
```

Lastly, trying with the XGBoost, this results in a cross validation F1 score of 0 again.

In [58]:

```
#XGBoost (original data, normal parameters)

#Fitting the XGBoost
xgb4 = XGBClassifier()
xgb4.fit(X_undersample.to_numpy(), y_undersample.values.ravel())

#Calcultaing the 5-fold cross validation score for this model
scoreXGB4 = cross_val_score(xgb4, X_validate.to_numpy(), y_validate.values.ravel(), cv=5,
scoring='f1')
print('Cross Validation F1 Score for XGBoost:', scoreXGB4.mean())
```

```
Cross Validation F1 Score for XGBoost: 0.0
```

I did not try other parameters for these algorithms using the undersampled data set, since they are already "balanced".

PART 4: Evaluation

To evaluate the models that we have trained, I will be calculating their F1 and AUROC scores, as well as plotting

To evaluate the models that we have trained, I will be calculating their F1 and AUROC scores, as well as plotting their Confusion Matrices.

In [59]:

```
#Predicting and evaluating the models using F1 Scores

F1Scores = []

#Log Regression Models
F1Scores.append(f1_score(y_test, logReg1.predict(X_test)))
F1Scores.append(f1_score(y_test, logReg2.predict(X_test)))
F1Scores.append(f1_score(y_test, logReg3.predict(X_test)))
F1Scores.append(f1_score(y_test, logReg4.predict(X_test)))

#Random Forest Models
F1Scores.append(f1_score(y_test, rf1.predict(X_test)))
F1Scores.append(f1_score(y_test, rf2.predict(X_test)))
F1Scores.append(f1_score(y_test, rf3.predict(X_test)))
F1Scores.append(f1_score(y_test, rf4.predict(X_test)))

#AdaBoost Models
F1Scores.append(f1_score(y_test, ada1.predict(X_test)))
F1Scores.append(f1_score(y_test, ada2.predict(X_test)))
F1Scores.append(f1_score(y_test, ada3.predict(X_test)))
F1Scores.append(f1_score(y_test, ada4.predict(X_test)))

#XGBoost Models
F1Scores.append(f1_score(y_test, xgb1.predict(X_test.values)))
F1Scores.append(f1_score(y_test, xgb2.predict(X_test.values)))
F1Scores.append(f1_score(y_test, xgb3.predict(X_test.values)))
F1Scores.append(f1_score(y_test, xgb4.predict(X_test.values)))

F1Score_table = pd.DataFrame({'Model': ['Logistic Regression M1','Logistic Regression M2'
,'Logistic Regression M3','Logistic Regression M4',
                                        'Random Forest M1','Random Forest M2','Random Forest
M3','Random Forest M4','AdaBoost M1','AdaBoost M2',
                                           'AdaBoost M3','AdaBoost M4','XGBoost M1','XGBoos
t M2','XGBoost M3','XGBoost M4'],
                              'F1 Score': [F1Scores[0],F1Scores[1],F1Scores[2],F1Scores[3],
F1Scores[4],F1Scores[5],
                                           F1Scores[6],F1Scores[7],F1Scores[8],F1Scores[9],F
1Scores[10],F1Scores[11],
                                           F1Scores[12], F1Scores[13], F1Scores[14], F1Sco
res[15]]})
display(F1Score_table)
```

	Model	F1 Score
0	Logistic Regression M1	0.000000
1	Logistic Regression M2	0.077183
2	Logistic Regression M3	0.082108
3	Logistic Regression M4	0.030621
4	Random Forest M1	0.011324
5	Random Forest M2	0.012227
6	Random Forest M3	0.012238
7	Random Forest M4	0.036291
8	AdaBoost M1	0.000000
9	AdaBoost M2	0.037524
10	AdaBoost M3	0.000000
11	AdaBoost M4	0.037388
12	XGBoost M1	0.000888
13	XGBoost M2	0.071815

	Model	F1 Score
14	XGBoost M3	0.100077
15	XGBoost M4	0.035991

Looking at the table above that has the Models and their corresponding F1 Scores tested on the original 20% Test data set, it seems that for Logistic Regression the best one is Model 3 with a weight adjustement of 1:30. The best Random Forest model is Model 4, a regular Random Forest model with no weight adjustment but trained on the undersampled data set. The best AdaBoost model is Model 2, when we changed the learning rate to 2. The best XGBoost model is Model 3, when we adjusted the scale_pos_weight to 15. Comparing all of these models, the one with the highest F1 Sore is XGBoost Model 3, with an F1 Score of 0.1

In [65]:

```
#Predicting and evaluating the models using AUROC

AUROC = []

#Log Regression Models
AUROC.append(roc_auc_score(y_test, logReg1.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, logReg2.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, logReg3.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, logReg4.predict_proba(X_test)[:, 1]))

#Random Forest Models
AUROC.append(roc_auc_score(y_test, rf1.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, rf2.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, rf3.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, rf4.predict_proba(X_test)[:, 1]))

#AdaBoost Models
AUROC.append(roc_auc_score(y_test, ada1.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, ada2.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, ada3.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, ada4.predict_proba(X_test)[:, 1]))

#XGBoost Models
AUROC.append(roc_auc_score(y_test, xgb1.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, xgb2.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, xgb3.predict_proba(X_test)[:, 1]))
AUROC.append(roc_auc_score(y_test, xgb4.predict_proba(X_test)[:, 1]))

AUROC_table = pd.DataFrame({'Model': ['Logistic Regression M1','Logistic Regression M2',
'Logistic Regression M3','Logistic Regression M4',
                                      'Random Forest M1','Random Forest M2','Random Forest
M3','Random Forest M4','AdaBoost M1','AdaBoost M2',
                                        'AdaBoost M3','AdaBoost M4','XGBoost M1','XGBoost
M2','XGBoost M3','XGBoost M4'],
                            'AUROC': [AUROC[0],AUROC[1],AUROC[2],AUROC[3],AUROC[4],AUROC
[5],AUROC[6],AUROC[7],AUROC[8],
                                      AUROC[9],AUROC[10],AUROC[11],AUROC[12],AUROC[13
], AUROC[14],AUROC[15]]})
display(AUROC_table)
```

	Model	AUROC
0	Logistic Regression M1	0.669176
1	Logistic Regression M2	0.704840
2	Logistic Regression M3	0.702971
3	Logistic Regression M4	0.399749
4	Random Forest M1	0.660917
5	Random Forest M2	0.660340
6	Random Forest M3	0.653925
7	Random Forest M4	0.527391
8	AdaBoost M1	0.744314
9	AdaBoost M2	0.429791

	Model	AUROC
10	AdaBoost M3	0.697901
11	AdaBoost M4	0.547128
12	XGBoost M1	0.753220
13	XGBoost M2	0.748484
14	XGBoost M3	0.751127
15	XGBoost M4	0.511020

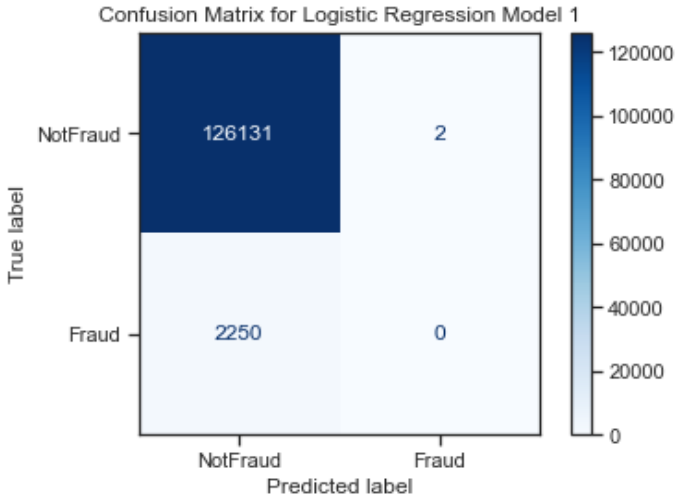
Next we will evaluate the models using AUROC. The best Logistic Regression model is Model 2, with a weight adjustment of 1:45. The best Random Forest model is Model 1, a basic random forest classifier with no adjustment to the weights or data. The best AdaBoost model is Model 1, which is the default AdaBoost classifier. Lastly, the best XGBoost model is Model 1, the default XGBoost model with no adjustments.

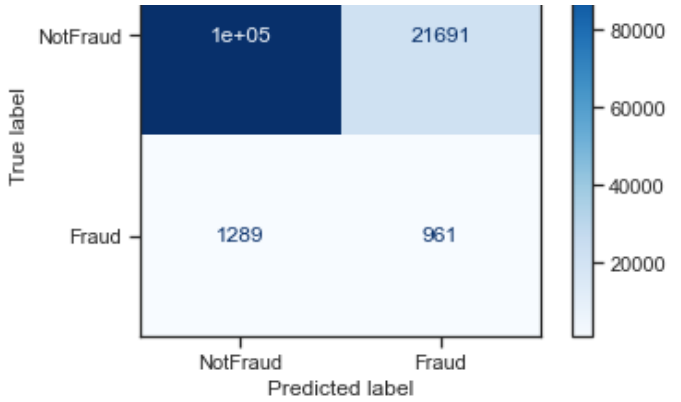
I have also decided to plot the confusion matrix for these models, to have more insight on how they are predicting fraudulent activities.

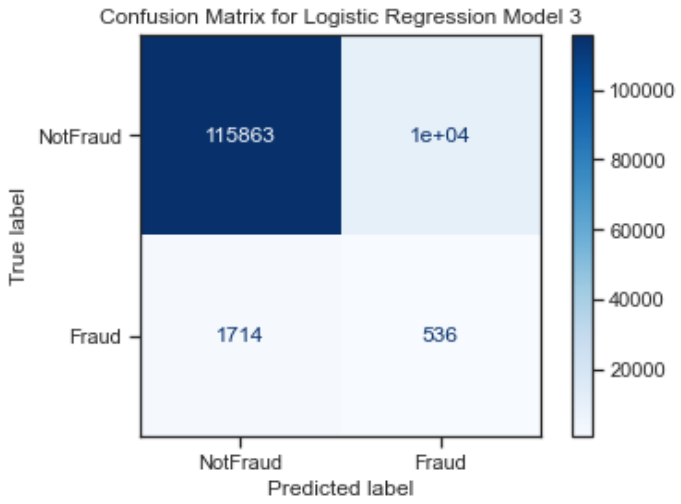
The Confusion Matrices for Logistic Regression shows that for our first model, we were not able to predict a single fraudulent activity. For the second model, we predicted some fraudulent activities but we also made more mistakes by predicting on observation as fraudulent when they were not. For the third model we caught less fraudulent activities but we also made less mistakes on predicting fraud when it was not. The last model, seems the worse on the confusion matrix. It seems as if we are predicting majority of the observations as fraudulent. Looking at them all, it seems like Model 3 looks the best.

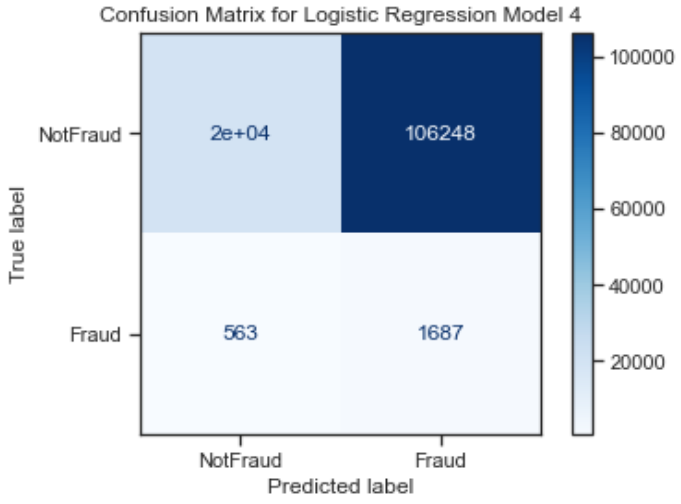
In [61]:

```
#Confusion Matrices for Logisitic Regression Models

plot_confusion_matrix(logReg1, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
Logistic Regression Model 1')
plot_confusion_matrix(logReg2, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
Logistic Regression Model 2')
plot_confusion_matrix(logReg3, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
Logistic Regression Model 3')
plot_confusion_matrix(logReg4, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
Logistic Regression Model 4')
plt.show()
```







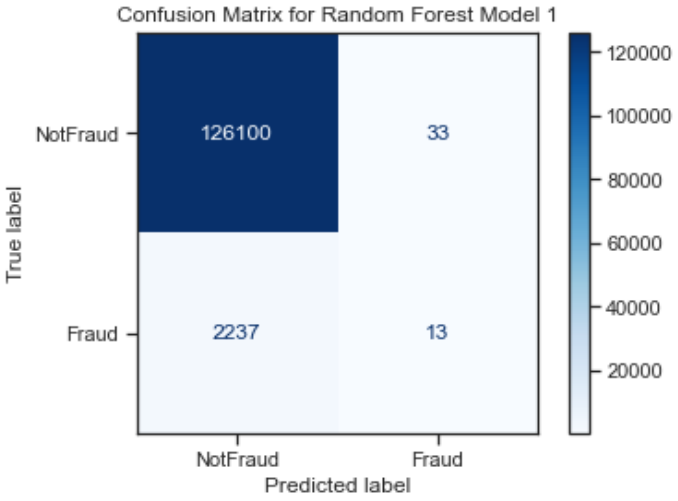


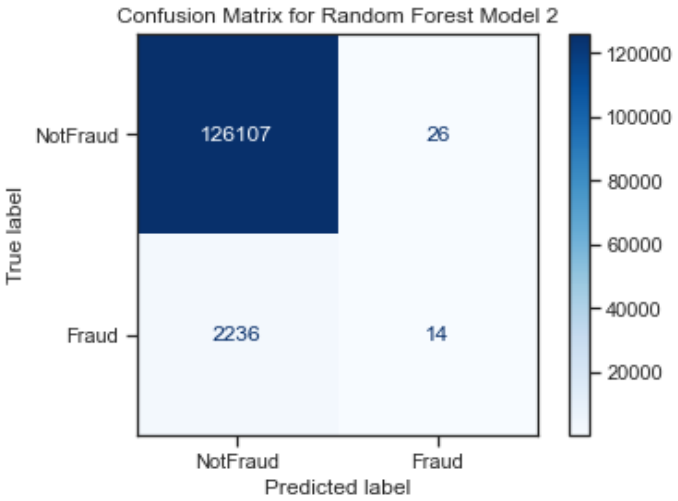
Plotting the Confusion Matrices for Random Forest, we can see that even at the first model there is already some prediction for fraud activities. In the second model we slightly increased the number of correct predictions for fraudulent activities and lowered the incorrect predictions by a little. Model 3 is very similar to Model 2, but little bit better. For our last Model 4, the model made a lot of mistakes by predicting observations to be fraudulent when they were not.

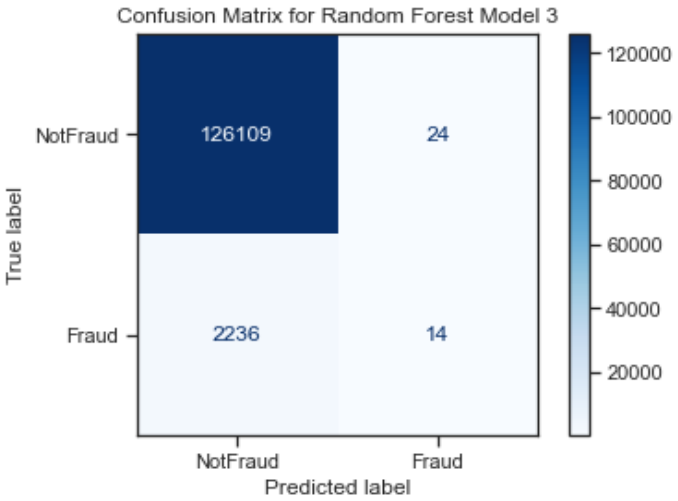
In [62]:

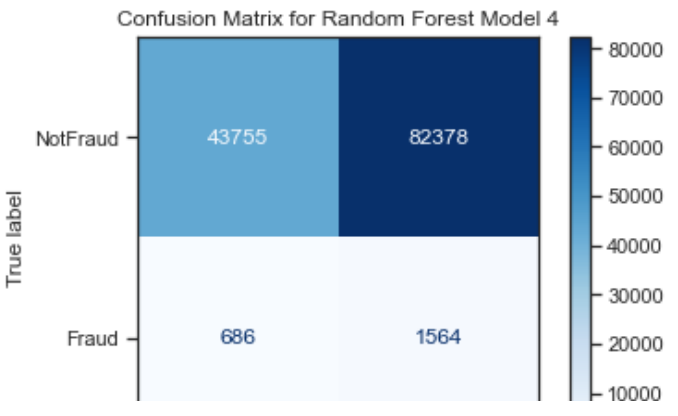
```
#Confusion Matrices for Random Forest Models

plot_confusion_matrix(rf1, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
Random Forest Model 1')
plot_confusion_matrix(rf2, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
Random Forest Model 2')
plot_confusion_matrix(rf3, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
```

```
Random Forest Model 3')
plot_confusion_matrix(rf4, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
Random Forest Model 4')
plt.show()
```





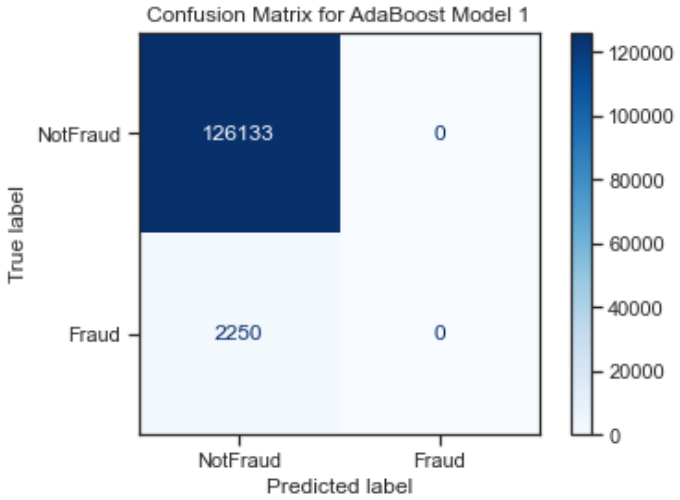


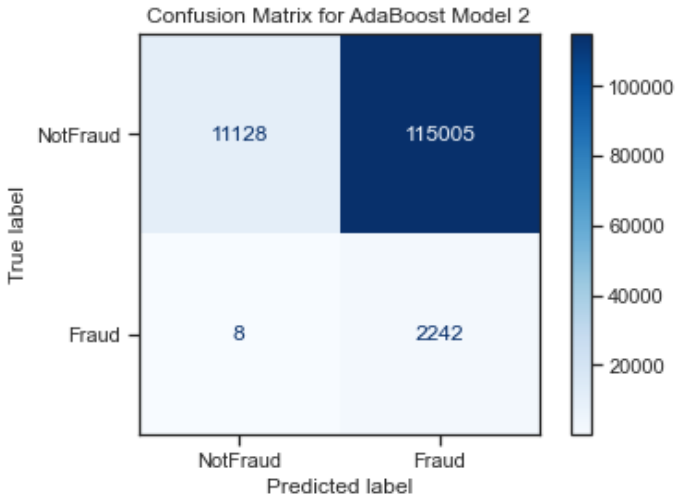


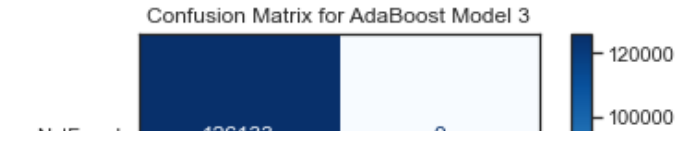
Next we will be looking at the AdaBoost Confusion Matrices. Model 1 starts off with no fraudulent prediction. Model 2 is overpredicting the number of fraudulent ativities. Model 3 is exactly the same as Model 1. Lastly, Model 4 also heavily predicts activities to be fraudulent when they were not. None of these AdaBoost models look too good.

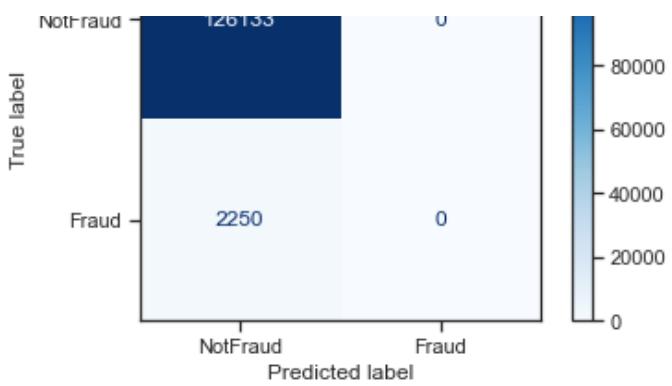
In [63]:

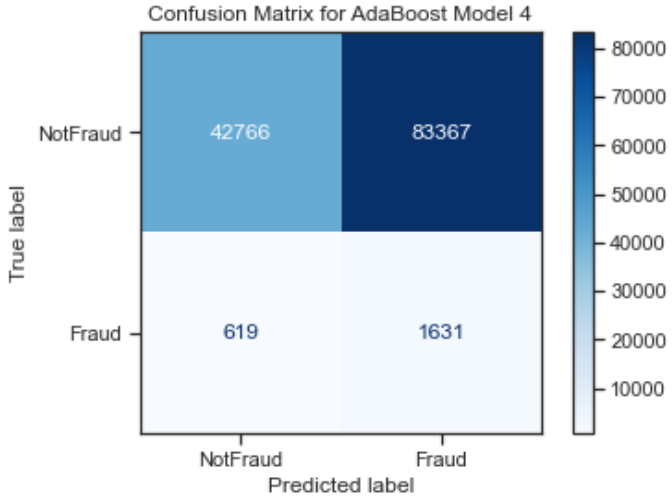
```
#Confusion Matrices for AdaBoost Models

plot_confusion_matrix(ada1, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
AdaBoost Model 1')
plot_confusion_matrix(ada2, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
AdaBoost Model 2')
plot_confusion_matrix(ada3, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
AdaBoost Model 3')
plot_confusion_matrix(ada4, X_test, y_test,
                                  display_labels=['NotFraud','Fraud'],
                                  cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
AdaBoost Model 4')
plt.show()
```







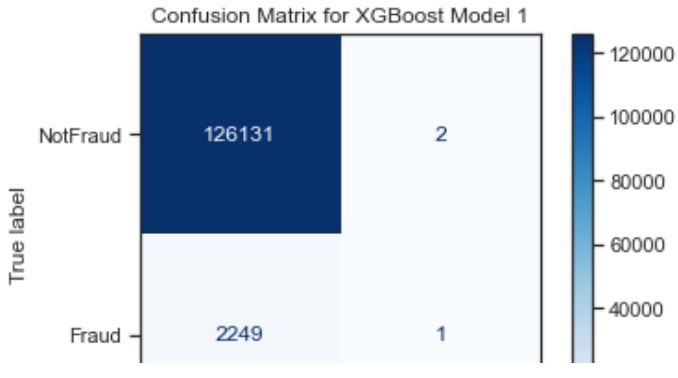


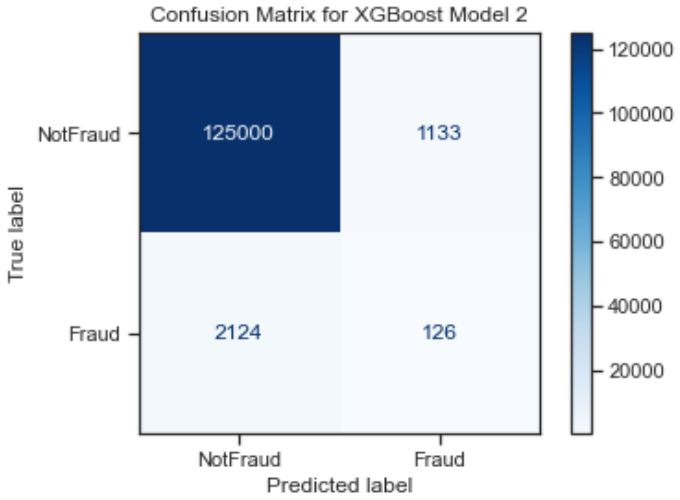


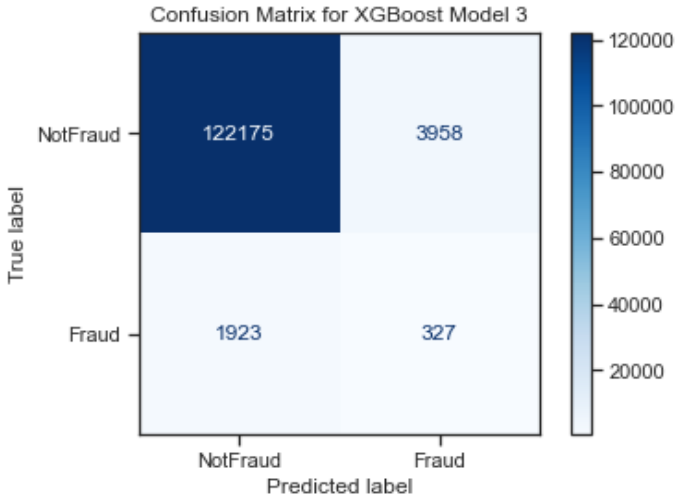
For our last algorithm we used XGBoost and the Confusion Matrices are plotted below. It shows that for the first model it was able to correctly identify one fraudulent transaction. In Model 2 we increased that to 126 but the number of incorrect predictions also got higher. Model 3 shows more improvement in fraud detection, but also higher false positive. Model 4 heavily predicts observations to be fraudulent when they are not. Looking at these confusion matrices, it seems like Model 3 is the best one.

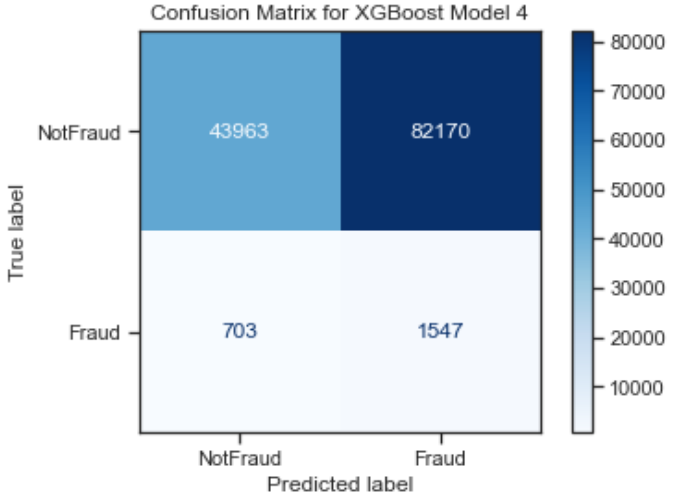
In [64]:

```
#Confusion Matrices for XGBoost Models

plot_confusion_matrix(xgb1, X_test.values, y_test,
                                display_labels=['NotFraud','Fraud'],
                                cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
XGBoost Model 1')
plot_confusion_matrix(xgb2, X_test.values, y_test,
                                display_labels=['NotFraud','Fraud'],
                                cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
XGBoost Model 2')
plot_confusion_matrix(xgb3, X_test.values, y_test,
                                display_labels=['NotFraud','Fraud'],
                                cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
XGBoost Model 3')
plot_confusion_matrix(xgb4, X_test.values, y_test,
                                display_labels=['NotFraud','Fraud'],
                                cmap=plt.cm.Blues).ax_.set_title('Confusion Matrix for
XGBoost Model 4')
plt.show()
```









In [66]:

```
#Adding the F1 Scores and AUROC together in a table

final_table = pd.DataFrame({'Model': ['Logistic Regression M1','Logistic Regression M2',
'Logistic Regression M3','Logistic Regression M4',
                                      'Random Forest M1','Random Forest M2','Random Forest
M3','Random Forest M4','AdaBoost M1','AdaBoost M2',
                                        'AdaBoost M3','AdaBoost M4','XGBoost M1','XGBoost
M2','XGBoost M3','XGBoost M4'],
                            'F1 Score': [F1Scores[0],F1Scores[1],F1Scores[2],F1Scores[3]
,F1Scores[4],F1Scores[5],
                                         F1Scores[6],F1Scores[7],F1Scores[8],F1Scores[9],F
```

```
1Scores[10],F1Scores[11],
                                        F1Scores[12], F1Scores[13], F1Scores[14], F1Sco
res[15]],
                            'AUROC': [AUROC[0],AUROC[1],AUROC[2],AUROC[3],AUROC[4],AUROC
[5],AUROC[6],AUROC[7],AUROC[8],
                                        AUROC[9],AUROC[10],AUROC[11],AUROC[12],AUROC[13
], AUROC[14],AUROC[15]]})

display(final_table)
```

	Model	F1 Score	AUROC
0	Logistic Regression M1	0.000000	0.669176
1	Logistic Regression M2	0.077183	0.704840
2	Logistic Regression M3	0.082108	0.702971
3	Logistic Regression M4	0.030621	0.399749
4	Random Forest M1	0.011324	0.660917
5	Random Forest M2	0.012227	0.660340
6	Random Forest M3	0.012238	0.653925
7	Random Forest M4	0.036291	0.527391
8	AdaBoost M1	0.000000	0.744314
9	AdaBoost M2	0.037524	0.429791
10	AdaBoost M3	0.000000	0.697901
11	AdaBoost M4	0.037388	0.547128
12	XGBoost M1	0.000888	0.753220
13	XGBoost M2	0.071815	0.748484
14	XGBoost M3	0.100077	0.751127
15	XGBoost M4	0.035991	0.511020

Conclusion

After cleaning and preparing the dataset, it was then split into a testing and training set. Which the training set was then further divided into another training set and a validation set for parameters tuning. Logistic Regression, Random Forest Classifier, AdaBoost, and XGBoost were all used to train the data set. Grid Search has been performed in order to find the best parameters, as well as Near Miss 3 undersampling the training data set to counter the problem of imbalance. After all the models were trained, we have a total of 4 machine learning algorithms and 16 different models across them. Fraudulent activities were predicted and evaluated using F1 Scores and AUROC. The results are in the table above, the best model is XGBoost Model 3, where we tuned the scale_pos_weight parameter to 15. This model gave us a F1 Score of 0.100077 and an AUROC score of 0.751127. As well as, looking at the Confusion Matrices it seems that this model performed the best out of all the other models. We can conclude that based on the F1 Score, AUROC score, and the number of correct fraudulent activity detection.

References

C. Albon, "Adaboost Classifier," Chris Albon, 20-Dec-2017. [Online]. Available: https://chrisalbon.com/machine_learning/trees_and_forests/adaboost_classifier/. [Accessed: 01-Dec-2020].

J. Brownlee, "How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras," Machine Learning Mastery, 09-Aug-2016. [Online]. Available: https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/. [Accessed: 22-Nov-2020].

J. Brownlee, "Undersampling Algorithms for Imbalanced Classification," Machine Learning Mastery, 20-Jan-2020. [Online]. Available: https://machinelearningmastery.com/undersampling-algorithms-for-imbalanced-classification/. [Accessed: 23-Nov-2020].

POS Entry Mode. [Online]. Available: https://www.mreports.com/documentation/ac/nonmerchant/80451.htm. [Accessed: 17-Nov-2020].

POS Entry Mode. [Online]. Available: https://www.mreports.com/documentation/ac/nonmerchant/80451.htm. [Accessed: 17-Nov-2020].