Nate Boyle

Machine Learning To Detect Fraudulent Transactions - Report

The Problem: Can a machine learning model successfully identify fraudulent transactions? And if so, what features/fields/columns provide the most insight as to whether or not fraud has occurred? I will analyze a dataset to see which features produce the strongest fraud signals and then use those selected, or engineered, features to train and test a model to identify fraud.

The Data: IEEE-CIS Fraud Detection
https://www.kaggle.com/competitions/ieee-fraud-detection/overview

This dataset is provided by Vesta Corporation, a company that provides e-commerce payment solutions. It comes from the IEEE-CIS Fraud Detection competition on Kaggle that took place in Fall of 2019.

There are two pairs of training and testing datasets with an example submission dataset that I did not use for this assignment.

The two training datasets are:
- train_identity.csv (144233, 41)
- train_transaction.csv (590540, 394)

The two testing datasets are:
- test_identity.csv
- test_transaction.csv

Both identity datasets contain anonymized information about the card holder and what device they performed the transaction on. Most columns are in the form of id-## (01-38) with the two named columns being DeviceType and DeviceInfo.

Both transaction datasets contain anonymized information about the transaction itself, such as location, information about the card used, type of product bought, transaction amount, date and time of transaction, email domains of purchaser and recipient, and over 300 alphanumeric columns (C1-14, D1-15, M1-9, and V1-339).

All datasets contain a TransactionID primary key column.

Only the train_transaction dataset contains an isFraud label, the train_identity must joined with it using the TransactionID column.

I chose this dataset because I find using data science or machine learning to identify fraud fascinating. And given my background in finance, tech, and compliance it was a natural fit.

The Cleaning & Analysis: I combined these sections because my analysis informed the cleaning, as I cleaned with the intent to identify columns with a large **isFraud** (or is-*not*-Fraud) signal. The goal of my analysis was to determine which columns were good to keep or transform for a final dataset that a model would be trained on.
Note: roughly 3.5% of transactions are fraudulent in this set, so anything near or over double that I considered a strong fraud signal.

C&A for the transaction dataset:

Trimming the fat:

First I dropped any columns with a super majority of missing values, such as those below:

```
Missing % by column (top 20 highest):
dist2   94
D7      93
D13     90
D14     89
D12     89
D6      88
D9      87
D8      87
V153    86
V149    86
V141    86
V146    86
V154    86
V162    86
V142    86
V158    86
V161    86
V157    86
V138    86
V139    86
```

In fact, with the exception of the **M1-9** columns I dropped all of the alphanumeric columns in the transaction dataset. While they may have some value, for this assignment spending that much time deciphering which of the over 300 columns have model training value

Distance - dist1 and dist 2:

Although I kept a transformed version of **dist2** as its presence had a large fraud signal:

```
Fraud rate overall: 3.5000000000000004 %

Fraud rate when dist2 present:
9.92 %
Fraud rate when dist2 missing:
3.06 %
```

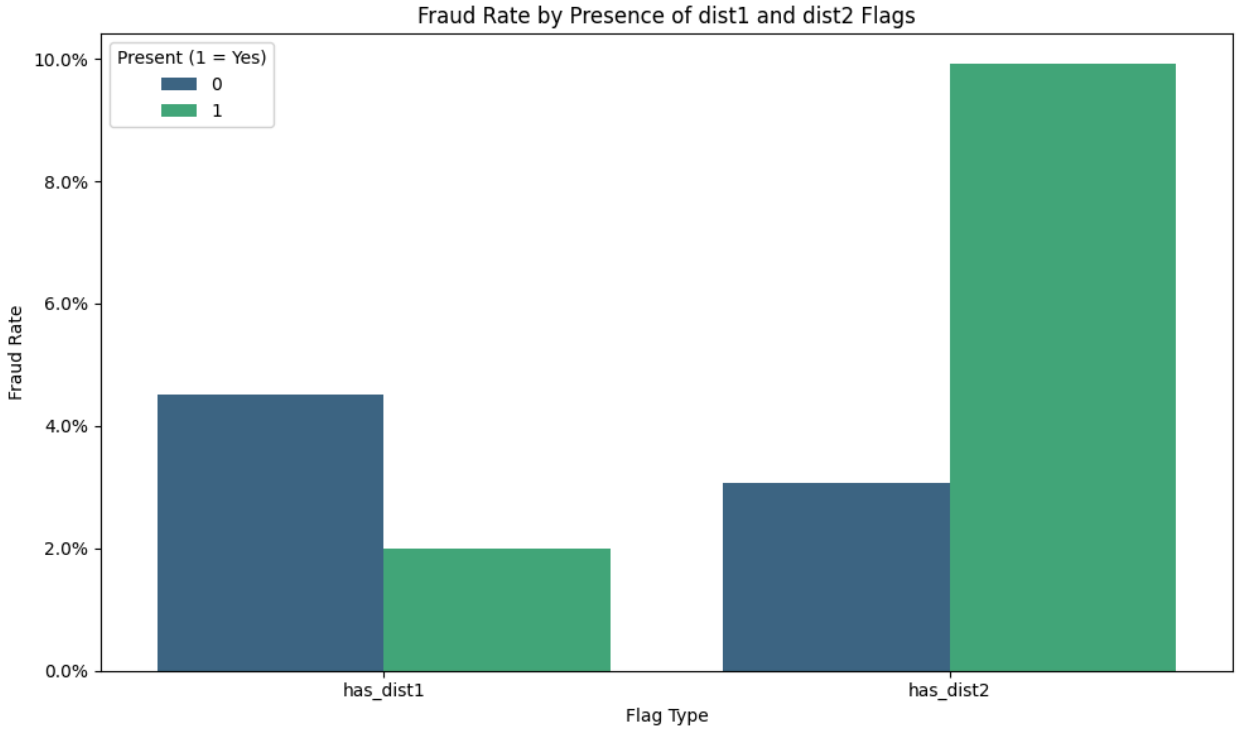
As you can see above, **dist2** being present meant fraud was more than 3 times more likely than when it was absent. So I created a new binary column **has_dist2**:

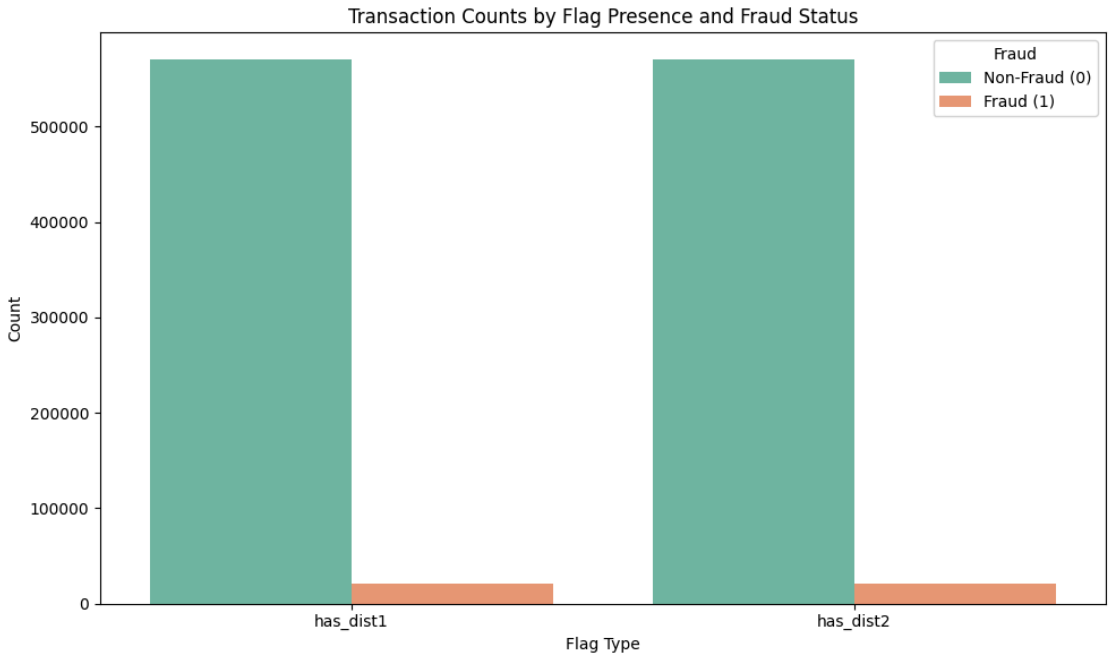
```
Fraud rate by has_dist2:
has_dist2
0    3
1   10
Name: isFraud, dtype: float64

Counts:
has_dist2
0    552913
1     37627
Name: count, dtype: int64
```

There was a similar situation with **dist1**:

```
Fraud rate when dist1 present vs missing:
dist1
False   5
True    2
```

The result was that two new dist binary columns were created and the original raw dist columns were dropped. See the images below to visualize the fraud rate discrepancies:





This was a pattern I would see in many more columns and I believe it has to do with the way Vesta's software works. Users committing fraud may be more or less likely to fill out certain fields, and based on the users' behavior they may be prompted or triggered to fill out other certain fields. Add in the fact that some fields are automatically populated based on user behavior and it starts to paint a larger picture that whether a value is null or not in and of itself is capable of creating a strong fraud signal.

M# Columns:

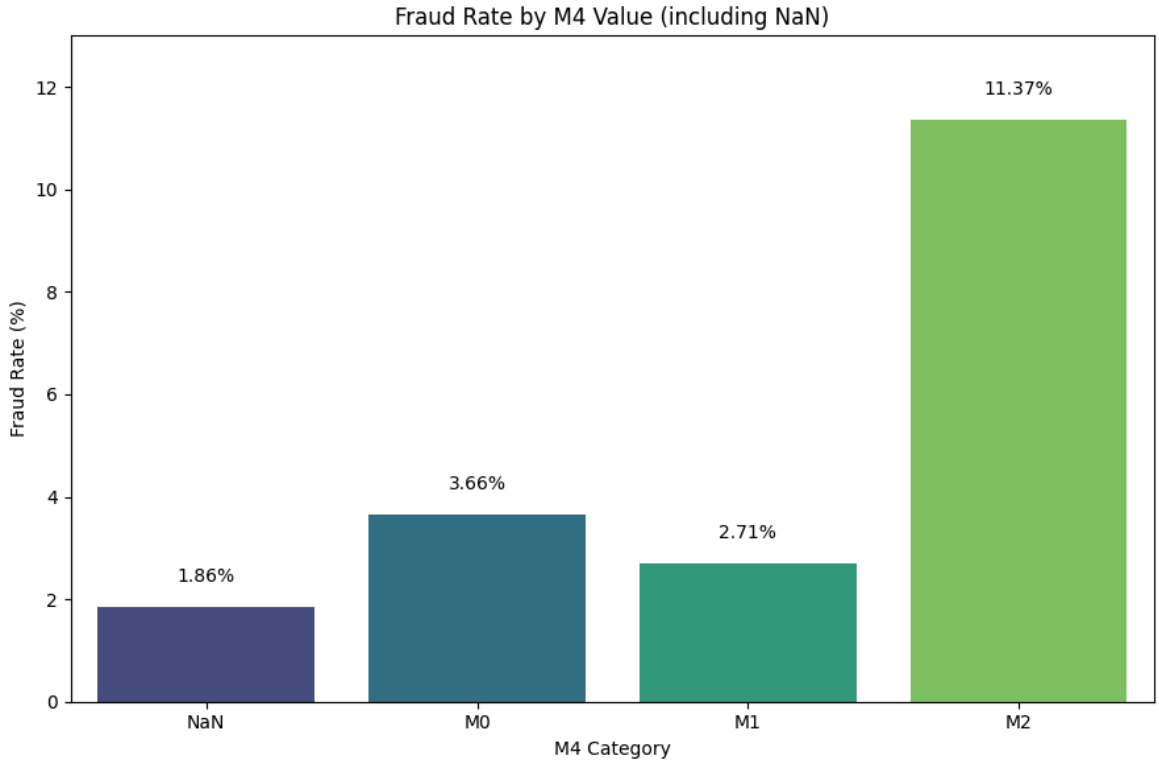
See below for the kinds of values found in columns **M1-9**:

M1	M2	M3	M4	M5	M6	M7	M8	M9
T	T	T	M0	T	F	F	F	T
T	F	F	M0	F	T	F	F	F
T	F	F	NaN	NaN	T	NaN	NaN	NaN
T	T	T	M0	F	T	NaN	NaN	NaN
T	F	F	NaN	NaN	T	F	F	F

Most were T/F/NaN, except for **M4**:

M4	count	mean	fraud_%
M2	59865	0	11
M0	196405	0	4
M1	52826	0	3
NaN	281444	0	2

As we can see **M4** not only had different values from the rest but there was clearly a strong signal for fraud based on one of those values.



There was also an interesting trend regarding fraud rates with the other **M#** columns:

Fraud summary for M1:

	count	mean	fraud_pct
M1			
F	25	0	0
T	319415	0	2
NaN	271100	0	5

Fraud summary for M2:

	count	mean	fraud_pct
M2			
F	33972	0	3
T	285468	0	2
NaN	271100	0	5

Fraud summary for M3:

	count	mean	fraud_pct
M3			
F	67709	0	3
T	251731	0	2
NaN	271100	0	5

Fraud summary for M5:

	count	mean	fraud_pct
M5			
F	132491	0	3
T	107567	0	4
NaN	350482	0	4

Fraud summary for M6:

	count	mean	fraud_pct
M6			
F	227856	0	2
T	193324	0	2
NaN	169360	0	7

Fraud summary for M7:

	count	mean	fraud_pct
M7			
F	211374	0	2
T	32901	0	2
NaN	346265	0	5

Fraud summary for M8:

	count	mean	fraud_pct
M8			
F	155251	0	2
T	89037	0	2
NaN	346252	0	5

Fraud summary for M9:

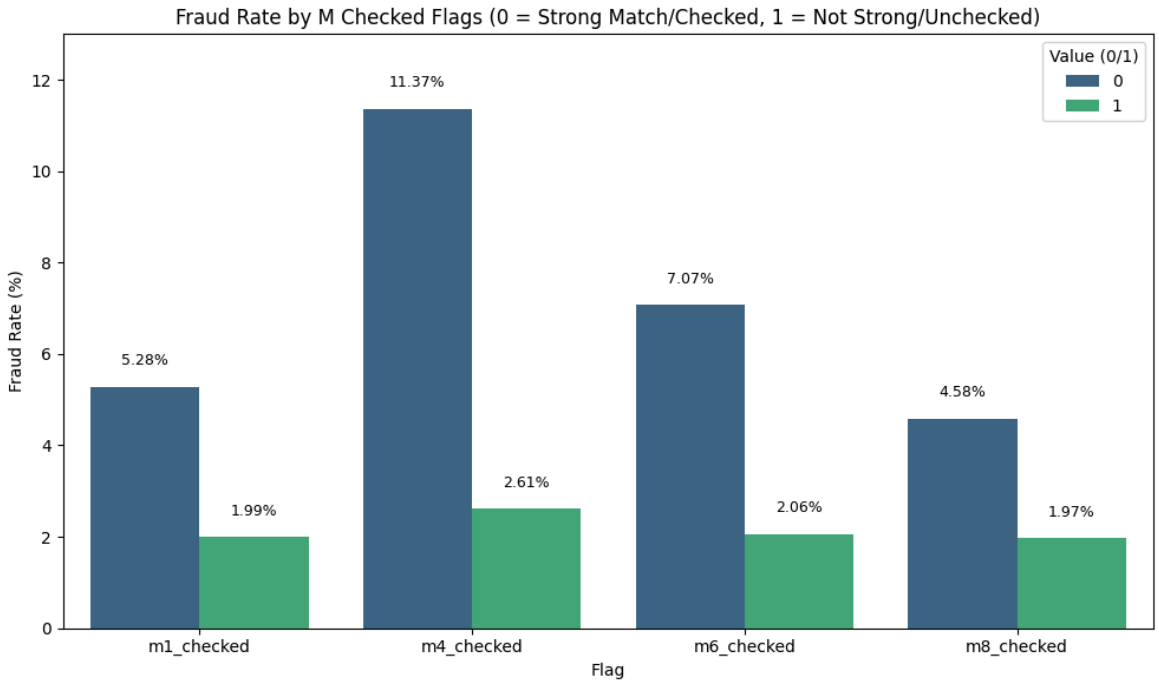
	count	mean	fraud_pct
M9			
F	38632	0	3
T	205656	0	2
NaN	346252	0	5

As you can see there is nearly a higher fraud signal for all of them when missing a value so I decided to create binary values for these 8 columns as well as **M4** and the results were surprising:

```
Fraud rate by M1_checked:
              count  mean  fraud_pct
M1_checked
0            271100     0          5
1            319440     0          2

Fraud rate by M2_checked:
              count  mean  fraud_pct
M2_checked
0            271100     0          5
1            319440     0          2

Fraud rate by M3_checked:
              count  mean  fraud_pct
M3_checked
0            271100     0          5
1            319440     0          2

Fraud rate by M5_checked:
              count  mean  fraud_pct
M5_checked
0            350482     0          4
1            240058     0          3
```

```
Fraud rate by M6_checked:
              count  mean  fraud_pct
M6_checked
0            169360     0          7
1            421180     0          2

Fraud rate by M7_checked:
              count  mean  fraud_pct
M7_checked
0            346265     0          5
1            244275     0          2

Fraud rate by M8_checked:
              count  mean  fraud_pct
M8_checked
0            346252     0          5
1            244288     0          2

Fraud rate by M9_checked:
              count  mean  fraud_pct
M9_checked
0            346252     0          5
1            244288     0          2

Fraud rate by M4_checked:
              count  mean  fraud_pct
M4_checked
0             59865     0         11
1            530675     0          3
```

Columns **M1-M3_checked** had identical NaN counts and fraud percentages, column **M5_checked** had very little fraud signal, column **M6_checked** had the second highest fraud signal after **M4_checked,** and **M7-M9_checked** basically had identical NaN counts and fraud percentages. So **M2,M3,M7,** and **M9_checked** were dropped for redundancy and **M5** was dropped for having a low signal. Please note that for **M4_checked**, 0 meant a value of 'M2' and NaN, 'M0', 'M1' were all grouped into 1. Also note that all original **M#** columns were dropped as well. See below for a fraud rate visualization of the kept columns.



Addresses - addr1 and addr2:

Given the pattern that was beginning to emerge I immediately did a binary fraud check for **addr1** and **addr2** and the results did not disappoint:

```
Fraud rate by addr1_present:
                count  mean  fraud_pct
addr1_present
0               65706     0         12
1              524834     0          2

Fraud rate by addr2_present:
                count  mean  fraud_pct
addr2_present
0               65706     0         12
1              524834     0          2
```

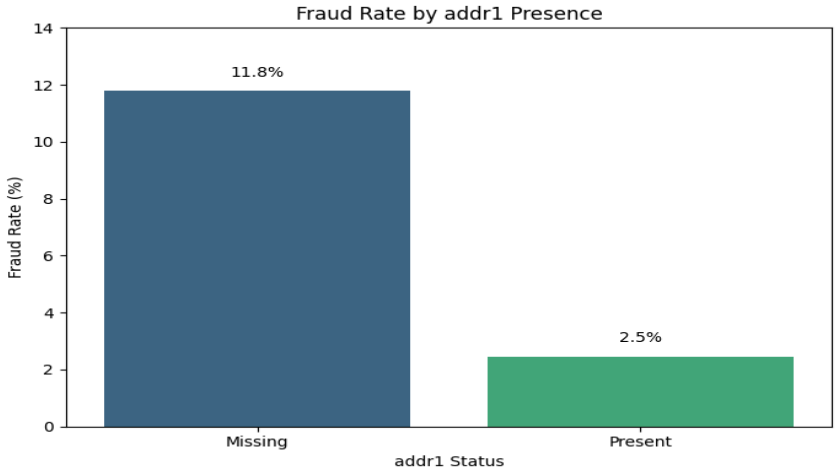
Not only was there a strong fraud signal for NaN values, but like some of the **M#** columns, **addr1** and **addr2** had identical NaN counts and fraud percentages, so I dropped **addr2** (and **addr2_present**) for redundancy. But before dropping the raw **addr1** I want to examine it a bit further. See summary statistics below:

```
Describe for addr1:
count   524,834
mean        291
std         102
min         100
25%         204
50%         299
75%         330
max         540
Name: addr1, dtype: float64
```

I think decided to bin these values loosely off of the quartiles and see if there was any meaningful fraud signal difference between them:

```
            count  mean  fraud_pct
addr1_bin
100-199    101112     0          2
200-299    194975     0          2
300-399    139644     0          3
400-540     89103     0          3
```

Annnnnd there was pretty much nothing. So I ended up dropping **addr1** as well and just keeping **addr1_present**. See below for the resulting visualization:



Email domains - P_emaildomain and R_emaildomain:
Like the past few column sets I started off examining the relationship between NaN values and fraud scores:

Fraud rate by P_emaildomain present (1) vs missing (0):

P_emaildomain	count	mean	fraud_pct
False	94456	0	3
True	496084	0	4

Fraud rate by R_emaildomain present (1) vs missing (0):

R_emaildomain	count	mean	fraud_pct
False	453249	0	2
True	137291	0	8

R_emaildomain seemed like it was worthy of converting into a binary column like many others that came before it in my analysis, I believe this is because the value represents the recipients email but only certain kinds of transactions require that, perhaps ones more likely to be fraud. I wanted to examine **P_emaildomain** further before giving up on it. First I examined the top ten, then the top 8 with the remainder in an 'Other' category and NaN as a 'Missing' category.

Missing (NaN): 15.99 %

Top 10 P_emaildomain by percentage:

P_emaildomain	
gmail.com	46.03 %
yahoo.com	20.35 %
hotmail.com	9.12 %
anonymous.com	7.46 %
aol.com	5.7 %
comcast.net	1.59 %
icloud.com	1.26 %
outlook.com	1.03 %
msn.com	0.82 %
att.net	0.81 %

P_email_grouped	
gmail.com	38.67 %
yahoo.com	17.09 %
Missing	15.99 %
hotmail.com	7.66 %
Other	6.27 %
anonymous.com	6.27 %
aol.com	4.79 %
comcast.net	1.34 %
icloud.com	1.06 %
outlook.com	0.86 %

I then examined the fraud rates:

Fraud rate by P_email_grouped (sorted by highest fraud %):

P_email_grouped	count	fraud_pct
outlook.com	5096	9
hotmail.com	45250	5
gmail.com	228355	4
icloud.com	6267	3
comcast.net	7888	3
Missing	94456	3
anonymous.com	36998	2
yahoo.com	100934	2
Other	37007	2
aol.com	28289	2

Surprisingly, outlook.com had the highest fraud percentage, maybe this means that corporate or Microsoft accounts are more likely to get hacked. I then further grouped the existing groups based on fraud percentage:

Fraud rate by P_email_grouped_v2 (sorted by highest fraud %):

P_email_grouped_v2	count	fraud_pct
outlook.com	5096	9
hotmail.com	45250	5
gmail.com	228355	4
Missing	94456	3
Other	217383	2

After landing on these results I dropped **P_emaildomain** and **P_email_grouped** kept **P_email_grouped_v2**.

Card columns - 'card1', 'card2', 'card3', 'card4', 'card5', 'card6':
See below for example values of these six columns:

card1	card2	card3	card4	card5	card6
13926	NaN	150	discover	142	credit
2755	404	150	mastercard	102	credit
4663	490	150	visa	166	debit
18132	567	150	mastercard	117	debit
4497	514	150	mastercard	102	credit

card4 and **card6** are clearly the vendor and type of card, the other columns are numbers (not card numbers) identifying the card in other ways, like country code.
Here is a breakdown of the unique values:

```
card1: 13553 unique values | 0 missing
card2: 500 unique values | 8933 missing
card3: 114 unique values | 1565 missing
card4: 4 unique values | 1577 missing
card5: 119 unique values | 4259 missing
card6: 4 unique values | 1571 missing
```

card4 and **card6** seem ripe to keep as categories for the eventual training of the model but I wanted to check fraud rates before committing to them:

Fraud rate by card4:

card4	count	mean	fraud_pct
american express	8328	0	3
discover	6651	0	8
mastercard	189217	0	3
visa	384767	0	3
NaN	1577	0	3

Fraud rate by card6:

card6	count	mean	fraud_pct
charge card	15	0	0
credit	148986	0	7
debit	439938	0	2
debit or credit	30	0	0
NaN	1571	0	2

And they both have strong signals for fraud, so **card4** and **card6** are kept as is.
For the remain card columns I created summary statistics to see how they could potentially be binned and look for any fraud patterns:

```
Describe for card1:
count   590,540
mean      9,899
std       4,901
min       1,000
25%       6,019
50%       9,678
75%      14,184
max      18,396
Name: card1, dtype: float64

Describe for card2:
count   581,607
mean        363
std         158
min         100
25%         214
50%         361
75%         512
max         600
Name: card2, dtype: float64
```

```
Describe for card3:
count   588,975
mean        153
std          11
min         100
25%         150
50%         150
75%         150
max         231
Name: card3, dtype: float64

Describe for card5:
count   586,281
mean        199
std          41
min         100
25%         166
50%         226
75%         226
max         237
Name: card5, dtype: float64
```

Then produced fraud percentage per bin statistics:

```
Fraud rate by card1 bin (non-null only):
              count  mean  fraud_pct
card1_bin
1000-5999    144089     0          4
6000-9999    159127     0          4
10000-13999  134729     0          3
14000-18396  152595     0          3
```

```
Fraud rate by card2 bin (non-null only):
             count  mean  fraud_pct
card2_bin
100-199     135148     0          4
200-299      66038     0          3
300-399     122055     0          3
400-499     100319     0          3
500-599     157853     0          4
```

```
Fraud rate by card3 bin (non-null only):
            count  mean  fraud_pct
card3_bin
100-149      8950     0          4
150        521287     0          2
151-159       119     0          3
160-199     57379     0         13
200-231      1240     0          3
```

```
Fraud rate by card5 bin (non-null only):
            count  mean  fraud_pct
card5_bin
100-165    108082     0          6
166-199     78291     0          2
200-225    100269     0          4
226-237    299639     0          3
```

card3_bin produced one of the strongest fraud signals we've seen yet and **card5_bin** produced a relatively strong fraud signal, so their bins were kept. All remaining card columns, raw or created, were dropped except **card4** and **card6.**

Product categories - ProductCD:

Obviously built for being a category column, I just did a quick fraud rate check to see if the column was worth keeping:

```
Fraud rate by ProductCD (sorted by highest fraud %):
            count  fraud_pct
ProductCD
C           68519         12
S           11628          6
H           33024          5
R           37699          4
W          439670          2
```

And it was.

Transaction date and time - TransactionDT:

Here are some example values:

590530	15810926
590531	15810935
590532	15811007
590533	15811029
590534	15811030
590535	15811047
590536	15811049
590537	15811079
590538	15811088
590539	15811131

These appear to be seconds, and after doing some research into the dataset it was confirmed that these are seconds since the beginning of the timespan of the dataset. Below are some summary statistics:

	TransactionDT
count	590,540
mean	7,372,311
std	4,617,224
min	86,400
25%	3,027,058
50%	7,306,528
75%	11,246,620
max	15,811,131

The 86,400 min value (24 hours) means that the first, or earliest, record in the dataset was one day after they began collecting data, and the max 15,811,131 says the last record is a little over 6 months later.

I decided to bin these values into days of the week and hours of the day to see the fraud rates:

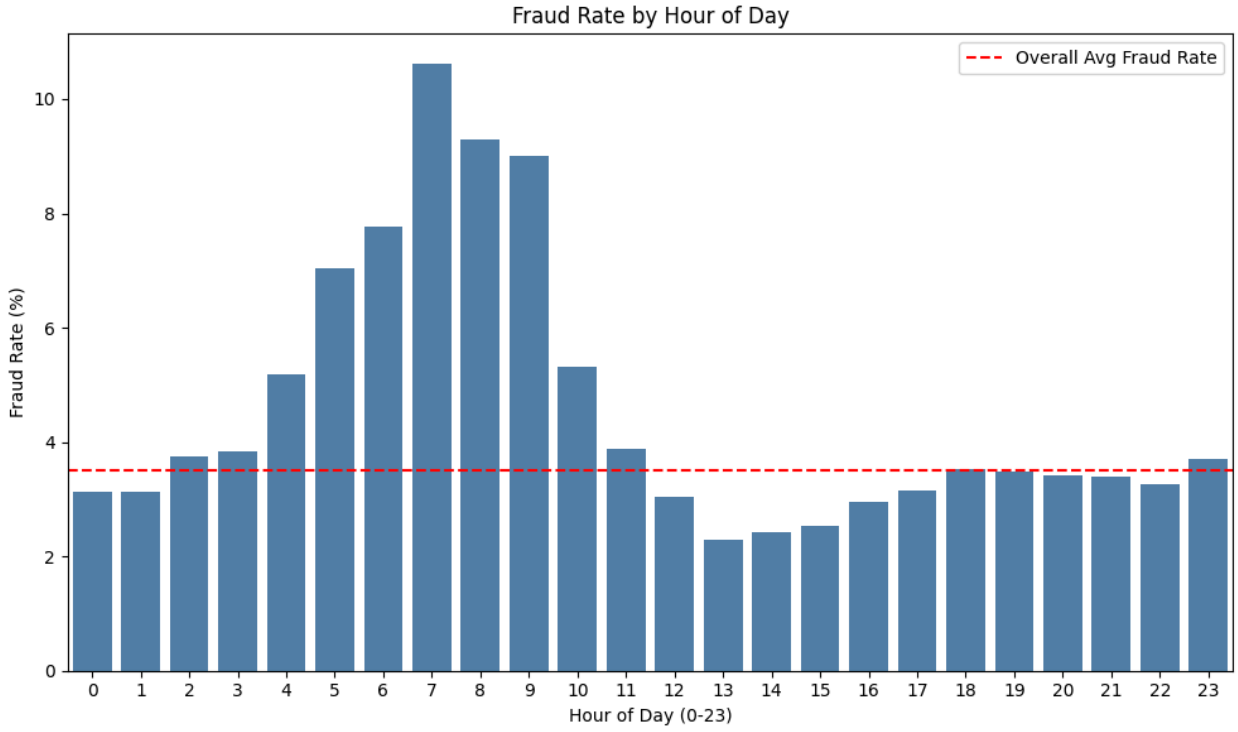
Fraud rate by hour of day (0–23):

hour	count	fraud_pct
0	37795	3
1	32797	3
2	26732	4
3	20802	4
4	14839	5
5	9701	7
6	6007	8
7	3704	11
8	2591	9
9	2479	9
10	3627	5
11	6827	4
12	12451	3
13	20315	2
14	28328	2
15	33859	3
16	38698	3
17	40723	3
18	41639	4
19	42115	3
20	41782	3
21	41641	3
22	41139	3
23	39949	4

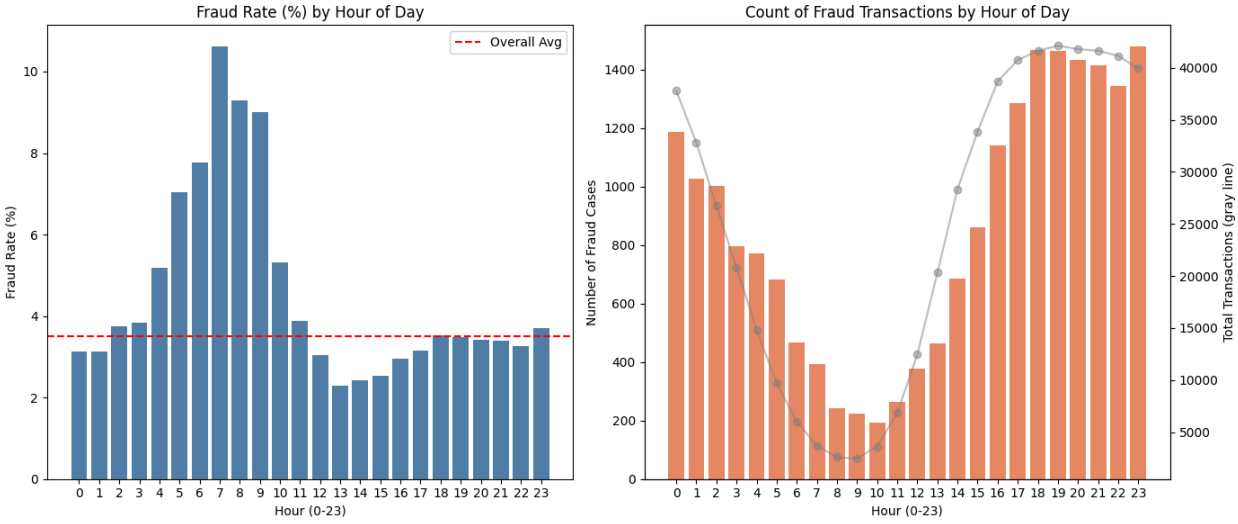
Fraud rate by day of week (0–6):

day_of_week	count	fraud_pct
0	86377	4
1	98502	4
2	79834	4
3	70223	4
4	85433	3
5	84815	3
6	85356	3

Not much variation for **day_of_the_week**, but clearly some strong fraud signals for the early hours of the morning. See the visualization below:



What is interesting though is that when you plot actual fraud counts the graph gets inverted:



This means that people in general are making more transactions at other times of the day, or less during the early morning hours, and it is diluting, or highlighting, the fraudulent activity. Either way, keeping **hour** as a categorical variable has merit.

Transaction amount - TransactionAMT:

The last column in the transaction dataset. Below are some summary statistics:

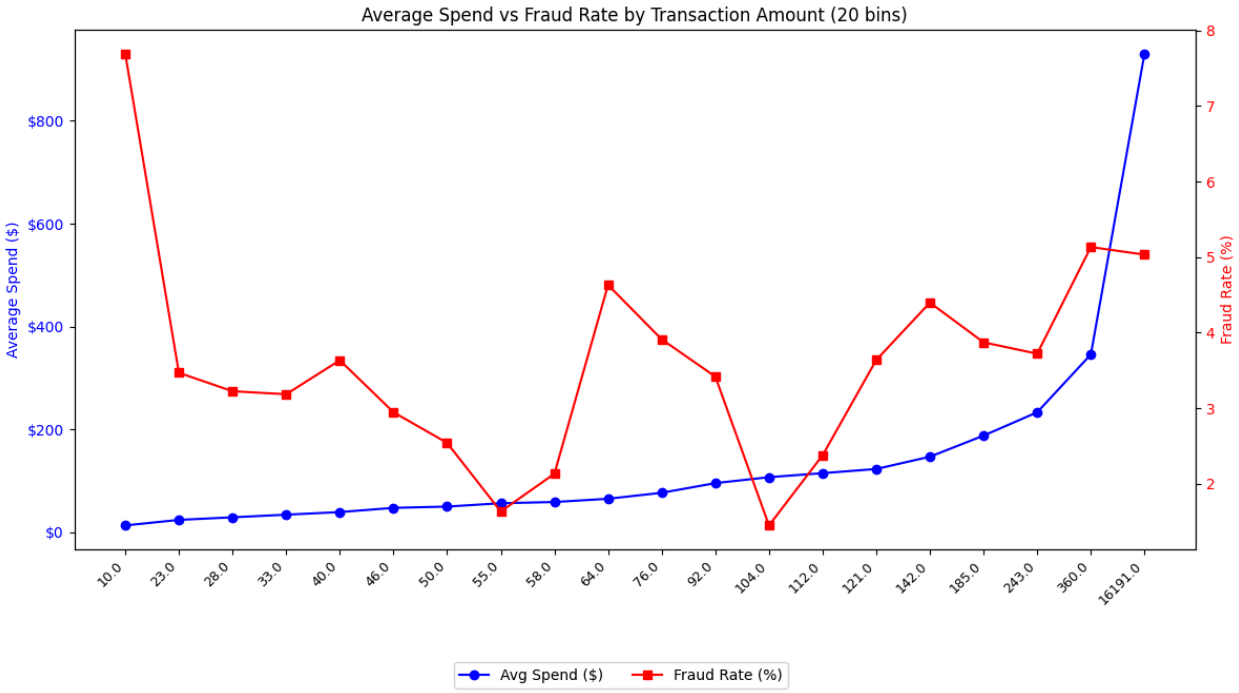
	TransactionAmt
count	590,540
mean	135
std	239
min	0
25%	43
50%	69
75%	125
max	31,937

I created bins just to see if there were any meaningful differences in terms of fraud:

Fraud rate by TransactionAmt bin:

amt_bin	count	mean	fraud_pct
0-50	187515	0	4
50-100	160742	0	3
100-200	141813	0	3
200-500	76146	0	4
500-1000	16744	0	6
1000-5000	7562	0	2
5000+	18	0	6

Nothing really stood out.



I expected fraud rates to climb as amount did, but there is a huge spike at transactions below $20. After doing some research these are typically ‘test’ transactions fraudsters do to see if a card works. Either way, **TransactionAMT** was kept as is with no binning.

C&A for the identity dataset:

Trimming the fat:

Like with the transaction dataset I dropped many columns at first, mostly the ones with a sparse amount of non-null values:

```
Data columns (total 41 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   TransactionID  118666 non-null  int64
 1   id_01          118666 non-null  float64
 2   id_02          118666 non-null  float64
 3   id_03          58643 non-null   float64
 4   id_04          58643 non-null   float64
 5   id_05          115313 non-null  float64
 6   id_06          115313 non-null  float64
 7   id_07          4527 non-null    float64
 8   id_08          4527 non-null    float64
 9   id_09          66260 non-null   float64
 10  id_10          66260 non-null   float64
 11  id_11          118666 non-null  float64
 12  id_12          118666 non-null  object
 13  id_13          103295 non-null  float64
 14  id_14          77285 non-null   float64
 15  id_15          118666 non-null  object
 16  id_16          116218 non-null  object
 17  id_17          117485 non-null  float64
 18  id_18          38973 non-null   float64
 19  id_19          117442 non-null  float64
 20  id_20          117394 non-null  float64
 21  id_21          4534 non-null    float64
 22  id_22          4538 non-null    float64
 23  id_23          4538 non-null    object
 24  id_24          4170 non-null    float64
 25  id_25          4511 non-null    float64
 26  id_26          4532 non-null    float64
 27  id_27          4538 non-null    object
 28  id_28          118666 non-null  object
 29  id_29          118666 non-null  object
 30  id_30          75539 non-null   object
 31  id_31          118367 non-null  object
 32  id_32          75541 non-null   float64
 33  id_33          71327 non-null   object
 34  id_34          75134 non-null   object
 35  id_35          118666 non-null  object
 36  id_36          118666 non-null  object
 37  id_37          118666 non-null  object
 38  id_38          118666 non-null  object
 39  DeviceType     118621 non-null  object
 40  DeviceInfo     118666 non-null  object
```

These were the columns left after the initial trimming:

TransactionID	id_01	id_02	id_11	id_12	id_15	id_28	id_29	id_35	id_36	id_37	id_38	DeviceType	DeviceInfo
3577495	-5	56,061	100	Found	Found	Found	Found	T	F	T	T	desktop	Windows
3577499	-5	89,753	100	NotFound	Found	Found	Found	F	F	T	F	desktop	Windows
3577501	-5	59,763	100	NotFound	Found	Found	Found	F	F	T	F	desktop	Windows
3577506	0	82,599	100	NotFound	Found	Found	Found	T	T	T	T	desktop	Trident/7.0
3577509	-5	253,305	100	NotFound	Found	Found	Found	F	F	T	F	mobile	SM-G930F Build/NRD90M
3577521	-15	145,955	100	NotFound	Found	Found	Found	F	F	T	F	mobile	F3111 Build/33.3.A.1.97
3577526	-5	172,059	100	NotFound	New	New	NotFound	T	F	T	F	mobile	A574BL Build/NMF26F
3577529	-20	632,381	100	NotFound	New	New	NotFound	F	F	T	F	mobile	Moto E (4) Plus Build/NMA26.42-152
3577531	-5	55,528	100	NotFound	Found	Found	Found	T	F	T	F	desktop	MacOS
3577534	-45	339,406	100	NotFound	New	New	NotFound	F	F	T	F	mobile	RNE-L03 Build/HUAWEIRNE-L03

These are how many unique values exist for the remaining columns:

```
TransactionID: 118621 unique values
id_01: 72 unique values
id_02: 99150 unique values
id_11: 362 unique values
id_12: 2 unique values
id_15: 3 unique values
id_28: 2 unique values
id_29: 2 unique values
id_35: 2 unique values
id_36: 2 unique values
id_37: 2 unique values
id_38: 2 unique values
DeviceType: 2 unique values
DeviceInfo: 1786 unique values
```

I believed there would be a way to group or cut down on the unique values for **DeviceInfo** to make it a better category column. Here are the top ten:

```
Top 10 DeviceInfo values:
DeviceInfo
Windows                 47720
iOS Device              19782
MacOS                   12573
Trident/7.0              7434
rv:11.0                  1864
rv:57.0                   962
SM-J700M Build/MMB29K     549
SM-G610M Build/MMB29K     461
SM-G531H Build/LMY48B     410
rv:59.0                   362
```

```
With percentages:
DeviceInfo
Windows                 40.23%
iOS Device              16.68%
MacOS                    10.6%
Trident/7.0              6.27%
rv:11.0                  1.57%
rv:57.0                  0.81%
SM-J700M Build/MMB29K    0.46%
SM-G610M Build/MMB29K    0.39%
SM-G531H Build/LMY48B    0.35%
rv:59.0                  0.31%
```

Based off of the patterns I was seeing (Windows, MacOS, iOS, SM - Samsung, rv - Firefox) I settled with this grouping:

```
New grouped distribution:
DeviceInfo_grouped_adv
Windows / Trident    55207
iOS                  19782
Other                17462
MacOS                12574
Samsung SM-           9248
Firefox rv: (old)     4348
```

```
With percentages:
DeviceInfo_grouped_adv
Windows / Trident    46.54 %
iOS                  16.68 %
Other                14.72 %
MacOS                 10.6 %
Samsung SM-            7.8 %
Firefox rv: (old)     3.67 %
```

And you can see this was good for identifying fraud signals:

```
Fraud rate by DeviceInfo_grouped_adv:
                        count  mean  fraud_pct
DeviceInfo_grouped_adv
Firefox rv: (old)        4348     0          8
MacOS                   12574     0          2
Other                   17462     0         14
Samsung SM-              9248     0         13
Windows / Trident       55207     0          6
iOS                     19782     0          6
NaN                    471919     0          3
```

So I kept **DeviceInfo_grouped_adv** and dropped **DeviceInfo**.

There were a similar results with the raw **DeviceType**:

Fraud rate by DeviceType in merged dense df:

DeviceType	count	mean	fraud_pct
desktop	73450	0	6
mobile	45171	0	10
NaN	471919	0	3

Here are the fraud percentages for the remaining **id_##**:

Fraud rate by id_01:

id_01	count	mean	fraud_pct
-72	1	1	100
-18	20	1	65
-88	2	0	50
-19	7	0	43
-11	12	0	42
...	...	...	...
-14	10	0	0
-13	3	0	0
-8	2	0	0
-7	5	0	0
-6	5	0	0

Fraud rate by id_02:

id_02	count	mean	fraud_pct
992,213	1	1	100
980,455	1	1	100
979,775	1	1	100
978,604	1	1	100
977,917	1	1	100
...	...	...	...
98,327	1	0	0
98,328	1	0	0
98,331	1	0	0
98,333	1	0	0
98,382	1	0	0

Fraud rate by id_11:

id_11	count	mean	fraud_pct
93	1	1	100
99	2	1	100
96	1	1	100
98	1	1	100
92	1	1	100
...	...	...	...
94	7	0	0
94	16	0	0
94	1	0	0
93	13	0	0
94	3	0	0

Fraud rate by id_12:

id_12	count	mean	fraud_pct
NotFound	99610	0	8
Found	19011	0	6
NaN	471919	0	3

Fraud rate by id_15:

id_15	count	mean	fraud_pct
Found	59925	0	10
Unknown	2410	0	8
New	56286	0	5
NaN	471919	0	3

Fraud rate by id_28:

id_28	count	mean	fraud_pct
Found	67420	0	9
New	51201	0	4
NaN	471919	0	3

Fraud rate by id_29:

id_29	count	mean	fraud_pct
Found	66220	0	10
NotFound	52401	0	4
NaN	471919	0	3

Fraud rate by id_35:

id_35	count	mean	fraud_pct
F	43075	0	12
T	75546	0	4
NaN	471919	0	3

Fraud rate by id_36:

id_36	count	mean	fraud_pct
F	112461	0	8
T	6160	0	3
NaN	471919	0	3

Fraud rate by id_37:

id_37	count	mean	fraud_pct
T	97476	0	8
F	21145	0	5
NaN	471919	0	3

Fraud rate by id_38:

id_38	count	mean	fraud_pct
F	59181	0	9
T	59440	0	6
NaN	471919	0	3

As you can see all of the T/F/NaN columns have reasonably strong fraud signals, and **id_01**, **id_02**, and **id_11** do at their upper bounds. Here are summary statistics for **id_01**, **id_02**, and **id_11:**

```
Describe for id_01:
count   118,621
mean         -9
std          14
min        -100
25%          -5
50%          -5
75%          -5
max           0
```

```
Describe for id_02:
count   118,621
mean    164,793
std     154,126
min           1
25%      63,941
50%     117,281
75%     212,559
max     999,595
```

```
Describe for id_11:
count   118,621
mean        100
std           1
min          90
25%         100
50%         100
75%         100
max         100
```

I decided to bin **id_01** and **id_11:**

```
Fraud rate by id_01 bin:
             count  mean  fraud_pct
id_01_bin
-100 to -4  101051     0          8
-5 to 0      17570     0          5
```

```
Fraud rate by id_11 bin (90-100 by 2):
            count  mean  fraud_pct
id_11_bin
90-90         229     0          4
91-92         410     0          3
93-94        1127     0          3
95-96        3497     0          9
97-100     113358     0          7
```

The fraud signals were strong enough to keep. The raw **id_01** and **id_11** were dropped.
For id_02, binning based on quartiles did yield some interesting fraud signals:

```
Fraud rate by id_02 quartiles:
               count  mean  fraud_pct
id_02_quartile
Q1             29656     0          5
Q2             29655     0          6
Q3             29655     0          8
Q4             29655     0         10
```

But given the continuous nature of the field I decided to keep as is.

C&A - Training and testing:

Model selection:

Now that I had my columns cleaned and selected I needed to choose a model.

The model I settled on was LightGBM (Light Gradient Boosting Machine), primarily because it handles categorical data and missing values extremely well, and given the amount of category data I had and the fact that roughly 80% of the values from the identity dataset columns were missing, this was important.

My dataset contained multiple data types, a mix of integers, floats, and categories, and LightGBM is particularly good at handling a mixture of data types, so not one-hot encoding or inputting missing values was required.

LightGBM also handles imbalanced data quite well, which is beneficial for the ~3.5% overall fraud rate. This handling of imbalance probably explains why LightGBM is a popular model for fraud datasets overall.

A big part of my analysis was identifying which features would assist in predicting fraud, LightGBM is able to provide importance ranking for features post training and testing which we will see in the results section:

Results:

The model worked on a five fold approach, meaning the training set was split into 5 pieces at a time, where one of the 5 pieces was hidden for testing while the other 4 pieces were used to train the model. After the 4 pieces were trained on the 5th piece was used to validate the training, this is called cross-validation. Useful for situations like mine where the competition is over and you cannot validate the testing dataset.

The model does this 4 for 1 training several rounds (600) within a fold until the results do not improve. Below are the results for the 5 folds:

```
Fold 1/5
Training until validation scores don't improve for 600 rounds
[200]   train's auc: 0.85844    valid's auc: 0.836628
[400]   train's auc: 0.874671   valid's auc: 0.842792
[600]   train's auc: 0.885526   valid's auc: 0.845852
[800]   train's auc: 0.894728   valid's auc: 0.847797
[1000]  train's auc: 0.901986   valid's auc: 0.848994
[1200]  train's auc: 0.907348   valid's auc: 0.849477
[1400]  train's auc: 0.912504   valid's auc: 0.849703
[1600]  train's auc: 0.917276   valid's auc: 0.849998
[1800]  train's auc: 0.921764   valid's auc: 0.849999
[2000]  train's auc: 0.925343   valid's auc: 0.84988
[2200]  train's auc: 0.928923   valid's auc: 0.849663
Early stopping, best iteration is:
[1725]  train's auc: 0.920149   valid's auc: 0.850201
Fold AUC: 0.85020

Fold 2/5
Training until validation scores don't improve for 600 rounds
[200]   train's auc: 0.858587   valid's auc: 0.837252
[400]   train's auc: 0.874829   valid's auc: 0.844585
[600]   train's auc: 0.885576   valid's auc: 0.848388
[800]   train's auc: 0.893592   valid's auc: 0.850835
[1000]  train's auc: 0.901231   valid's auc: 0.852314
[1200]  train's auc: 0.906603   valid's auc: 0.852832
[1400]  train's auc: 0.91184    valid's auc: 0.853372
[1600]  train's auc: 0.916879   valid's auc: 0.853852
[1800]  train's auc: 0.921005   valid's auc: 0.854011
[2000]  train's auc: 0.925067   valid's auc: 0.854169
[2200]  train's auc: 0.928574   valid's auc: 0.854266
[2400]  train's auc: 0.931726   valid's auc: 0.854162
[2600]  train's auc: 0.935121   valid's auc: 0.854237
Early stopping, best iteration is:
[2162]  train's auc: 0.927918   valid's auc: 0.854336
Fold AUC: 0.85434

Fold 3/5
Training until validation scores don't improve for 600 rounds
[200]   train's auc: 0.859472   valid's auc: 0.831323
[400]   train's auc: 0.87564    valid's auc: 0.839371
[600]   train's auc: 0.886168   valid's auc: 0.843727
[800]   train's auc: 0.894732   valid's auc: 0.846422
[1000]  train's auc: 0.901577   valid's auc: 0.848266
[1200]  train's auc: 0.907603   valid's auc: 0.849169
[1400]  train's auc: 0.913031   valid's auc: 0.84988
[1600]  train's auc: 0.917621   valid's auc: 0.850565
[1800]  train's auc: 0.921889   valid's auc: 0.850641
[2000]  train's auc: 0.925628   valid's auc: 0.850674
[2200]  train's auc: 0.928937   valid's auc: 0.850597
[2400]  train's auc: 0.931953   valid's auc: 0.850498
Early stopping, best iteration is:
[1963]  train's auc: 0.925051   valid's auc: 0.850765
Fold AUC: 0.85076
```

```
Fold 4/5
Training until validation scores don't improve for 600 rounds
[200]   train's auc: 0.858578   valid's auc: 0.832524
[400]   train's auc: 0.875367   valid's auc: 0.83975
[600]   train's auc: 0.885611   valid's auc: 0.843012
[800]   train's auc: 0.894062   valid's auc: 0.845174
[1000]  train's auc: 0.901105   valid's auc: 0.846282
[1200]  train's auc: 0.907019   valid's auc: 0.846995
[1400]  train's auc: 0.912478   valid's auc: 0.847162
[1600]  train's auc: 0.91768    valid's auc: 0.847625
[1800]  train's auc: 0.921924   valid's auc: 0.847495
[2000]  train's auc: 0.925526   valid's auc: 0.847491
[2200]  train's auc: 0.928897   valid's auc: 0.847479
Early stopping, best iteration is:
[1742]  train's auc: 0.920764   valid's auc: 0.847748
Fold AUC: 0.84775

Fold 5/5
Training until validation scores don't improve for 600 rounds
[200]   train's auc: 0.858693   valid's auc: 0.833703
[400]   train's auc: 0.875102   valid's auc: 0.840845
[600]   train's auc: 0.885497   valid's auc: 0.844447
[800]   train's auc: 0.89429    valid's auc: 0.846698
[1000]  train's auc: 0.901159   valid's auc: 0.848196
[1200]  train's auc: 0.90689    valid's auc: 0.848991
[1400]  train's auc: 0.912351   valid's auc: 0.849489
[1600]  train's auc: 0.917196   valid's auc: 0.849777
[1800]  train's auc: 0.921248   valid's auc: 0.849663
[2000]  train's auc: 0.925421   valid's auc: 0.849473
[2200]  train's auc: 0.928915   valid's auc: 0.849175
Early stopping, best iteration is:
[1709]  train's auc: 0.919578   valid's auc: 0.8498
Fold AUC: 0.84980

Full OOF CV AUC: 0.85056
```

The Full OOF CV AUC (Out-of-Fold Cross-Validated Area Under the ROC Curve) score was 0.85056, meaning that 85% percent of the time when a pair of fraud and not-fraud records were evaluated the model correctly assigned a higher fraud probability to the actual fraudulent record.

I am pretty happy with these results, but this was only part of solving ‘The Problem’, the other part was identifying, or at this point, validating, what features were most useful. The validation of certain features can be determined using importance scores, which are numbers calculated by how much the feature reduces the loss function. This loss function reduction is called a ‘gain’ and an importance score for a feature is the sum of all of the gains the feature had on the loss function. Below are importance scores for the 15 most important features in terms of predicting fraud:

feature	importance
transactionamt	3,140,033
hour	1,352,550
r_email_present	1,175,843
id_02	1,007,104
productcd	947,495
p_email_grouped_v2	760,064
card6	562,390
card5_bin	525,011
card3_bin	466,954
deviceinfo_grouped_adv	369,765
card4	285,374
devicetype	149,310
id_35	137,189
id_29	128,370
m4_checked	126,078

Many of these are not surprising, **transactionamt**, **productcd**, card vendor and type (**card4** and **card6**), but many of the features I engineered had very high important scores, such as **hour**, **r_email_present**, **p_email_grouped_v2**, or **deviceinfo_grouped_adv**.

Conclusion:

The cleaning and analysis approach was successfully able to train a model to assign a correct higher risk probability, and not only select, but engineer features that proved useful in assigning fraud risk probability.

I was particularly happy with ‘hour’ having such a high importance score, a feature derived from the raw transaction date time data. It, along with pretty much all of the high importance scoring features, validated my analysis technique of looking for strong fraud signals when selecting, cleaning, or engineering features.

And for the categorical features, yes I was happy with the engineered ones that had high importance scores, but it proved the revelation I came upon in my analysis, that NaN or missing values themselves can be great sources of data and insight. It kind of reminds me of the old adage “the absence of evidence is not the evidence of absence”. This largely had to do with the way Vesta’s software works, and how fields are filled in or required or not required, or triggered or not triggered, to be populated. Either way, this revelation regarding null values is truly a bullet point in the timeline of my data science journey and lesson I will take with me going forward.