Table of contents

State Farm

Purpose

The purpose of this project is to demonstrate data science techniques on datasets provided by State Farm insurance company. The first step is to load and clean the data, as well as conduct exploratory data analysis to understand the data. Following EDA, a few classification models will be built and compared. A logistic regression and another model will be chosen as the final models. We will then compare and contrast the different models based on respective strengths and weaknesses. Finally, predictions will be made on the test data, in the form of class probabilities for belonging to the positive class.

Intro

In [ ]:
```
# import libraries
import pandas as pd
import plotly_express as px
import plotly.graph_objects as go
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn import svm
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import lightgbm as lgb
from imblearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OrdinalEncoder, LabelEncoder
from sklearn.neural_network import MLPClassifier
from tensorflow import keras
from tensorflow.keras.optimizers import Adam
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import SimpleImputer, KNNImputer, IterativeImputer
from sklearn.metrics import accuracy_score, auc, roc_auc_score, roc_curve, f1_score, classification_report, confusion_m
from imblearn.over_sampling import SMOTE
# show graphs in html
import plotly.io as pio
pio.renderers.default = "plotly_mimetype+notebook"
```

In [ ]:

```
# read dataset
train = pd.read_csv('datasets/exercise_40_train.csv')
test = pd.read_csv('datasets/exercise_40_test.csv')
```

In [ ]:

```
# set max column length to 110
pd.set_option('display.max_columns', 110)
```

Train Dataset

In [ ]:

```
# look at dataset
train.head()
```

Out[ ]:

	y	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13
0	0	0.165254	18.060003	Wed	1.077380	-1.339233	-1.584341	0.0062%	0.220784	1.816481	1.171788	109.626841	4.644568	4.814885
1	1	2.441471	18.416307	Friday	1.482586	0.920817	-0.759931	0.0064%	1.192441	3.513950	1.419900	84.079367	1.459868	1.443983
2	1	4.427278	19.188092	Thursday	0.145652	0.366093	0.709962	-8e-04%	0.952323	0.782974	-1.247022	95.375221	1.098525	1.216059
3	0	3.925235	19.901257	Tuesday	1.763602	-0.251926	-0.827461	-0.0057%	-0.520756	1.825586	2.223038	96.420382	-1.390239	3.962961
4	0	2.868802	22.202473	Sunday	3.405119	0.083162	1.381504	0.0109%	-0.732739	2.151990	-0.275406	90.769952	7.230125	3.877312

At first glance, we see various problems with the dataset, and we collect some ideas of how to deal with those problems: label encode x3, remove % in x7, fill missing values, remove dollar sign in x19, binarize x24, binarize x31, label encode x33, label encode x39, label encode x60, label encode x64, label encode x65, label encode x77, binarize x93, binarize x99. The most efficient method would be to use a pipeline to label encode and impute missing values.

In [ ]:
```
# summary info on columns
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 40000 entries, 0 to 39999
Columns: 101 entries, y to x100
dtypes: float64(86), int64(3), object(12)
memory usage: 30.8+ MB
```

In [ ]:
```
# looking at shape of data
train.shape
```

Out[ ]:
```
(40000, 101)
```

In [ ]:
```
# looking at column names
train.columns
```

Out[ ]:
```
Index(['y', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9',
       ...
       'x91', 'x92', 'x93', 'x94', 'x95', 'x96', 'x97', 'x98', 'x99', 'x100'],
      dtype='object', length=101)
```

In [ ]:
```
# remove special characters
train.x7 = train.x7.str.replace('%', '').astype(float)
train.x19 = train.x19.str.replace('$', '').astype(float)
```

```
C:\Users\XIX\AppData\Local\Temp\ipykernel_32516\2428904411.py:3: FutureWarning:

The default value of regex will change from True to False in a future version. In addition, single character regular ex
pressions will *not* be treated as literal strings when regex=True.
```

In [ ]:
```
# Check proper implementation
train[['x7', 'x19']].head()
```

Out[ ]:

	x7	x19
0	0.0062	-908.650758
1	0.0064	-1864.962288
2	-0.0008	-543.187403
3	-0.0057	-182.626381
4	0.0109	967.007091

We needed to remove the special characters from the dataset, and then convert those columns into float. By default, x19 was rounded to 6 decimal places. This should have a minimal effect on the model performance.

In [ ]:
```
# looking at categories
train.select_dtypes(['object'])
```

Out[ ]:

	x3	x24	x31	x33	x39	x60	x65	x77	x93	x99
0	Wed	female	no	Colorado	5-10 miles	August	farmers	mercedes	no	yes
1	Friday	male	no	Tennessee	5-10 miles	April	allstate	mercedes	no	yes
2	Thursday	male	no	Texas	5-10 miles	September	geico	subaru	no	yes
3	Tuesday	male	no	Minnesota	5-10 miles	September	geico	nissan	no	yes
4	Sunday	male	yes	New York	5-10 miles	January	geico	toyota	yes	yes
...	...	...	...	...	...	...	...	...	...	...
39995	Sun	female	no	NaN	5-10 miles	July	farmers	NaN	no	yes
39996	Thursday	male	yes	Illinois	5-10 miles	July	progressive	ford	no	yes
39997	Monday	male	yes	NaN	5-10 miles	August	geico	ford	no	yes
39998	Tuesday	male	no	Ohio	5-10 miles	December	farmers	NaN	no	yes
39999	Thursday	NaN	no	Florida	5-10 miles	January	progressive	toyota	no	NaN

40000 rows × 10 columns

We need to take a better look at the object columns with EDA.

```
In [ ]: # rows with missing values
        train.isna().any(axis=1).sum()
```

```
Out[ ]: 39999
```

We see that most rows have at least one missing value

```
In [ ]: # checking for rows where all values are missing
        train.isna().all(axis=0).sum()
```

```
Out[ ]: 0
```

Dataset does not contain any rows where all values are missing.

```
In [ ]: # looking for duplicates
        train.duplicated().sum()
```

```
Out[ ]: 0
```

Test Dataset

```
In [ ]: # look at test set
        test.head()
```

Out[ ]:

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13
0	4.747627	20.509439	Wednesday	2.299105	-1.815777	-0.752166	0.0098%	-3.240309	0.587948	-0.260721	101.113628	-0.812035	3.251085
1	1.148654	19.301465	Fri	1.862200	-0.773707	-1.461276	0.0076%	0.443209	0.522113	-1.090886	104.791999	8.805876	1.651993
2	4.986860	18.769675	Saturday	1.040845	-1.548690	2.632948	-5e-04%	-1.167885	5.739275	0.222975	102.109546	7.831517	3.055358
3	3.709183	18.374375	Tuesday	-0.169882	-2.396549	-0.784673	-0.016%	-2.662226	1.548050	0.210141	82.653354	0.436885	1.578106
4	3.801616	20.205541	Monday	2.092652	-0.732784	-0.703101	0.0186%	0.056422	2.878167	-0.457618	75.036421	8.034303	1.631426

In [ ]:
```
# shape of dataset
test.shape
```

Out[ ]:
```
(10000, 100)
```

In [ ]:
```
# look at info on columns
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 100 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   x1      10000 non-null  float64
 1   x2      10000 non-null  float64
 2   x3      10000 non-null  object 
 3   x4      10000 non-null  float64
 4   x5      9398 non-null   float64
 5   x6      10000 non-null  float64
 6   x7      10000 non-null  object 
 7   x8      10000 non-null  float64
 8   x9      10000 non-null  float64
 9   x10     10000 non-null  float64
 10  x11     8671 non-null   float64
 11  x12     10000 non-null  float64
 12  x13     10000 non-null  float64
 13  x14     7572 non-null   float64
 14  x15     10000 non-null  float64
 15  x16     7247 non-null   float64
 16  x17     10000 non-null  float64
 17  x18     10000 non-null  float64
 18  x19     10000 non-null  object 
 19  x20     10000 non-null  float64
 20  x21     10000 non-null  float64
 21  x22     9387 non-null   float64
 22  x23     10000 non-null  float64
 23  x24     9031 non-null   object 
 24  x25     10000 non-null  float64
 25  x26     9383 non-null   float64
 26  x27     10000 non-null  float64
 27  x28     10000 non-null  float64
 28  x29     10000 non-null  float64
 29  x30     1915 non-null   float64
 30  x31     10000 non-null  object 
 31  x32     10000 non-null  float64
 32  x33     8230 non-null   object 
 33  x34     10000 non-null  float64
 34  x35     10000 non-null  float64
 35  x36     10000 non-null  float64
 36  x37     10000 non-null  float64
 37  x38     9435 non-null   float64
 38  x39     10000 non-null  object 
 39  x40     10000 non-null  float64
 40  x41     7596 non-null   float64
 41  x42     7582 non-null   float64
 42  x43     10000 non-null  float64
 43  x44     1434 non-null   float64
 44  x45     7937 non-null   float64
 45  x46     10000 non-null  float64
 46  x47     10000 non-null  float64
 47  x48     10000 non-null  float64
 48  x49     6746 non-null   float64
 49  x50     10000 non-null  float64
 50  x51     10000 non-null  float64
 51  x52     5920 non-null   float64
 52  x53     10000 non-null  float64
 53  x54     6794 non-null   float64
 54  x55     5576 non-null   float64
 55  x56     10000 non-null  float64
 56  x57     1923 non-null   float64
 57  x58     10000 non-null  float64
 58  x59     10000 non-null  int64
 59  x60     10000 non-null  object
 60  x61     8234 non-null   float64
 61  x62     10000 non-null  float64
 62  x63     9413 non-null   float64
 63  x64     8738 non-null   float64
 64  x65     10000 non-null  object
 65  x66     10000 non-null  float64
 66  x67     9380 non-null   float64
 67  x68     9400 non-null   float64
 68  x69     10000 non-null  float64
 69  x70     10000 non-null  float64
 70  x71     10000 non-null  float64
 71  x72     10000 non-null  float64
 72  x73     10000 non-null  float64
 73  x74     6837 non-null   float64
 74  x75     8734 non-null   float64
 75  x76     8644 non-null   float64
 76  x77     7682 non-null   object
 77  x78     7134 non-null   float64
 78  x79     9390 non-null   float64
 79  x80     8685 non-null   float64
 80  x81     10000 non-null  float64
 81  x82     10000 non-null  float64
 82  x83     9428 non-null   float64
 83  x84     10000 non-null  float64
 84  x85     7581 non-null   float64
 85  x86     9398 non-null   float64
 86  x87     10000 non-null  float64
 87  x88     9409 non-null   float64
 88  x89     7325 non-null   float64
 89  x90     10000 non-null  float64
 90  x91     8690 non-null   float64
 91  x92     9374 non-null   float64
 92  x93     10000 non-null  object
 93  x94     9385 non-null   float64
 94  x95     6828 non-null   float64
 95  x96     8372 non-null   float64
 96  x97     10000 non-null  float64
 97  x98     10000 non-null  int64
 98  x99     6700 non-null   object
 99  x100    10000 non-null  float64
dtypes: float64(86), int64(2), object(12)
memory usage: 7.6+ MB
```

In [ ]:
```
# looking at missing values
test.isna().sum()
```

Out[ ]:
```
x1         0
x2         0
x3         0
x4         0
x5       602
        ...
x96     1628
x97        0
x98        0
x99     3300
x100       0
Length: 100, dtype: int64
```

In [ ]:
```
# remove special characters
test.x7 = test.x7.str.replace('%', '').astype(float)
test.x19 = test.x19.str.replace('$', '').astype(float)
```

```
C:\Users\XIX\AppData\Local\Temp\ipykernel_32516\4155689457.py:3: FutureWarning:

The default value of regex will change from True to False in a future version. In addition, single character regular ex
pressions will *not* be treated as literal strings when regex=True.
```

In [ ]:
```
# Check proper implementation
test[['x7', 'x19']].head()
```

Out[ ]:

	x7	x19
0	0.0098	120.216190
1	0.0076	-267.562586
2	-0.0005	-311.292903
3	-0.0160	2229.149400
4	0.0186	-469.049530

Introductory Conclusions

We cleaned the data from the obvious issues, such as special characters and changing dtypes. We see many missing values as well as categorical columns in the dataset. We applied the same cleaning methods to both the training and test sets.

EDA

Train Dataset

In [ ]:
```
# values of column
train.x3.value_counts(dropna=False)
```

Out[ ]:
```
Wednesday    4930
Monday       4144
Friday       3975
Tuesday      3915
Sunday       3610
Saturday     3596
Tue          2948
Thursday     2791
Mon          2200
Wed          2043
Sat          1787
Thur         1643
Fri          1620
Sun           798
Name: x3, dtype: int64
```

In [ ]:
```python
# being consistent with labeling, short notation
train.x3 = train.x3.str.replace('Sunday', 'Sun')
train.x3 = train.x3.str.replace('Monday', 'Mon')
train.x3 = train.x3.str.replace('Tuesday', 'Tue')
train.x3 = train.x3.str.replace('Wednesday', 'Wed')
train.x3 = train.x3.str.replace('Thursday', 'Thur')
train.x3 = train.x3.str.replace('Friday', 'Fri')
train.x3 = train.x3.str.replace('Saturday', 'Sat')
```

We combined the corresponding days to the shorthand notation.

In [ ]:
```python
# values of column
train.x24.value_counts(dropna=False)
```

Out[ ]:
```
female    18158
male      17986
NaN        3856
Name: x24, dtype: int64
```

In [ ]:
```python
# check values
train.x33.value_counts(dropna=False)
```

Out[ ]:
```
NaN               7171
California        3393
Texas             2252
Florida           1802
New York          1714
Illinois          1240
Pennsylvania      1233
Ohio              1114
Michigan           982
Georgia            918
North Carolina     910
New Jersey         870
Virginia           791
Washington         750
Tennessee          690
Indiana            674
Arizona            665
Massachusetts      638
Wisconsin          635
Missouri           634
Minnesota          611
Maryland           581
Alabama            554
Colorado           530
Louisiana          501
South Carolina     491
Kentucky           478
Oregon             452
Connecticut        422
Oklahoma           397
Kansas             378
Nevada             373
Utah               370
Mississippi        361
Iowa               353
Arkansas           346
New Mexico         333
Nebraska           323
West Virginia      305
Hawaii             282
Idaho              277
Maine              247
Rhode Island       246
New Hampshire      231
Montana            195
```

```
Vermont            195
Wyoming            189
DC                 186
South Dakota       183
North Dakota       181
Delaware           177
Alaska             176
Name: x33, dtype: int64
```

There are 52 values for what is a states column. Total should be 50 + 1 with D.C. Therefore, the missing value is not a missing state and is unlikely to be a territory from the list. The values will be imputed in the pipeline.

In [ ]:
```
# Change values to 1
train.x39 = train.x39.str.replace('5-10 miles', '1').astype(int)
```

All rows of this column are the same, so we will change the value to 1.

In [ ]:
```
# checking values
train.x60.value_counts(dropna=False)
```

Out[ ]:
```
December     8136
January      7922
July         7912
August       7907
June         1272
September    1245
February     1213
November     1043
April         951
March         807
May           799
October       793
Name: x60, dtype: int64
```

This column represents months. No duplicate naming is seen here, and all 12 months are present.

In [ ]:
```
# checking values
train.x65.value_counts(dropna=False)
```

Out[ ]:
```
progressive    10877
allstate       10859
esurance        7144
farmers         5600
geico           5520
Name: x65, dtype: int64
```

This column represents the different insurance companies.

In [ ]:
```
# checking values
train.x77.value_counts(dropna=False)
```

Out[ ]:
```
NaN          9257
ford         9005
subaru       5047
chevrolet    5011
mercedes     4494
toyota       3555
nissan       2575
buick        1056
Name: x77, dtype: int64
```

This column represents different vehicle manufacturers. As it is unlikely that the missing values are all one manufacturer missing from the list, these values will have to be imputed.

In [ ]:
```
# checking values
train.x93.value_counts(dropna=False)
```

Out[ ]:
```
no     35506
yes     4494
Name: x93, dtype: int64
```

In [ ]:
```
# values of column
train.x99.value_counts(dropna=False)
```

Out[ ]:
```
yes    27164
NaN    12836
Name: x99, dtype: int64
```

Missing values in this column are more likely to be no, rather than missing yes values. Therefore, we will fill in missing vales with no.

In [ ]:
```
# fill missing values with no
train.x99.fillna('no', inplace=True)
```

```
In [ ]: # check proper implementation
        train.x99.value_counts(dropna=False)
```

```
Out[ ]: yes    27164
        no     12836
        Name: x99, dtype: int64
```

Filled missing values with no.

```
In [ ]: # summary statistics on data
        train.describe()
```

Out[ ]:

	y	x1	x2	x4	x5	x6	x7	x8	x9	
count	40000.000000	40000.000000	40000.000000	40000.000000	37572.000000	40000.000000	40000.000000	40000.000000	40000.000000	40000.(
mean	0.145075	2.999958	20.004865	0.002950	0.005396	0.007234	0.000033	0.004371	2.722334	0..
std	0.352181	1.994490	1.604291	1.462185	1.297952	1.358551	0.009965	1.447223	1.966828	1.(
min	0.000000	-3.648431	13.714945	-5.137161	-5.616412	-6.113153	-0.043800	-6.376810	-3.143438	-3.
25%	0.000000	1.592714	18.921388	-1.026798	-0.872354	-0.909831	-0.006700	-0.971167	1.340450	-0..
50%	0.000000	2.875892	20.005944	0.002263	0.008822	0.007335	0.000100	0.002226	2.498876	0..
75%	0.000000	4.270295	21.083465	1.043354	0.892467	0.926222	0.006800	0.985023	3.827712	1.
max	1.000000	13.837591	27.086468	5.150153	5.698128	5.639372	0.037900	5.869889	18.006669	4.1

```
In [ ]: # show correlation
        fig = px.imshow(train.corr(), aspect='auto', title='Train Correlations')
        fig.show()
```

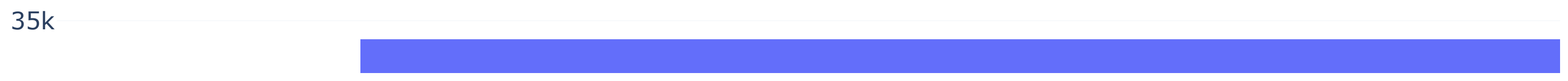
This figure shows the correlations between the features and the target variable. Overall, we see no correlations of note.

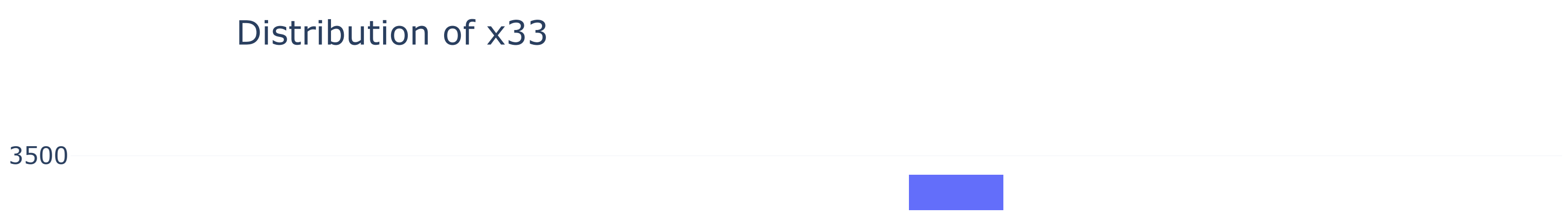
```
In [ ]: # distribution of object columns
        for col in train.select_dtypes('object'):
            fig = px.histogram(train[col], title='Distribution of '+str(col), template='plotly_white')
            fig.show()
```

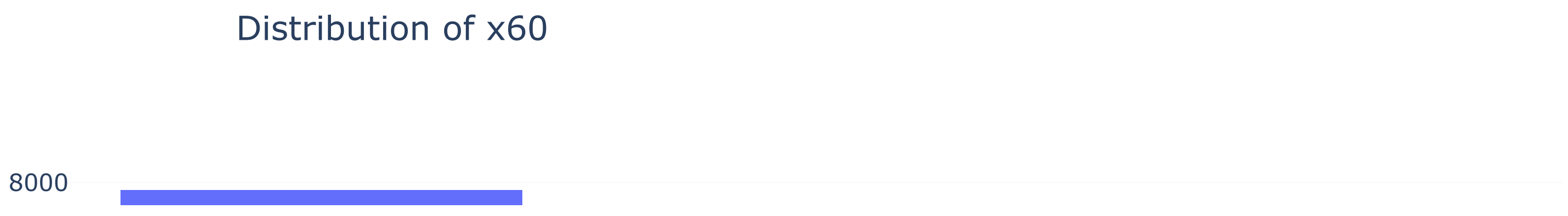
Distribution of x3
7000

Distribution of x24
18k

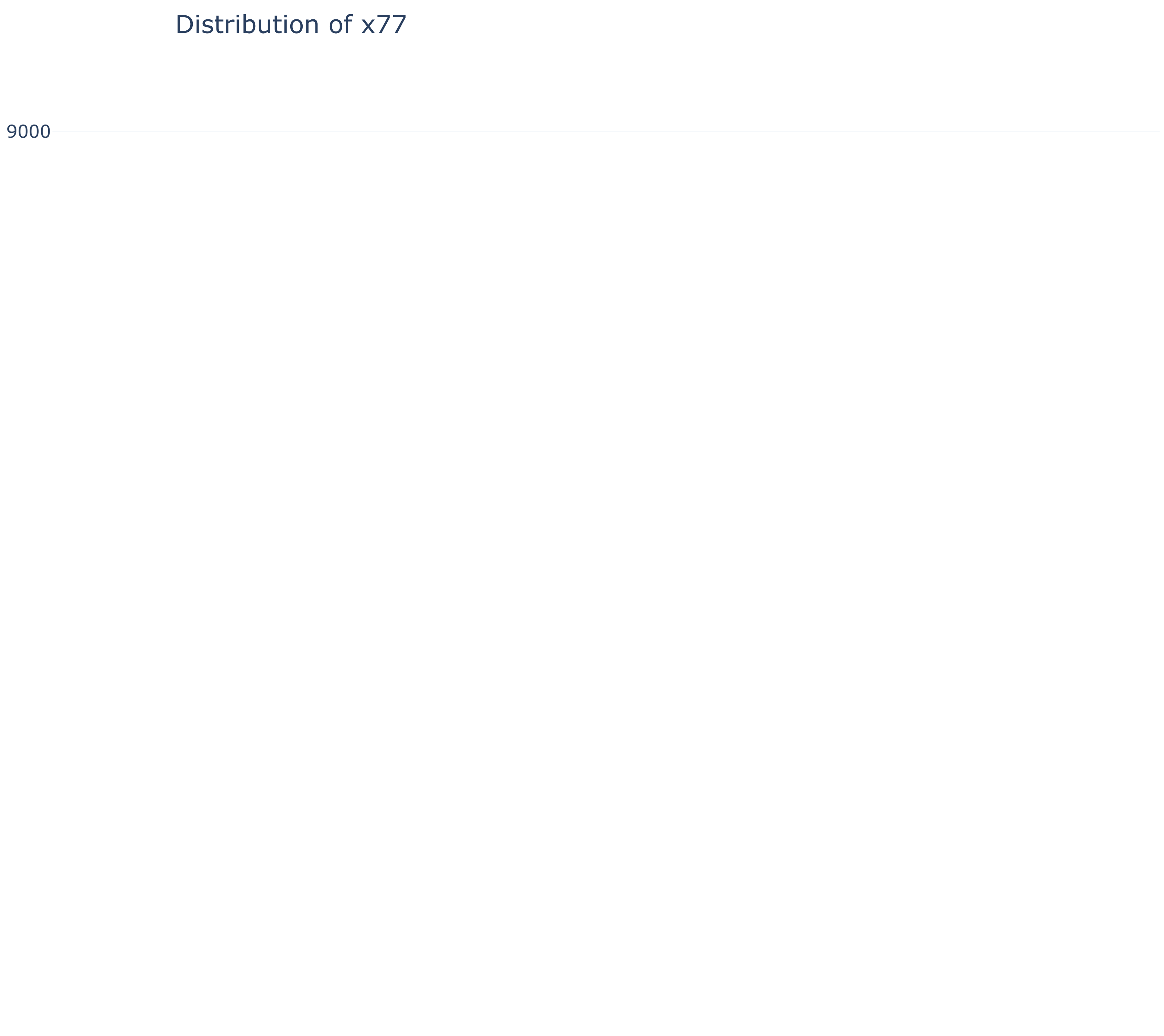
Distribution of x31



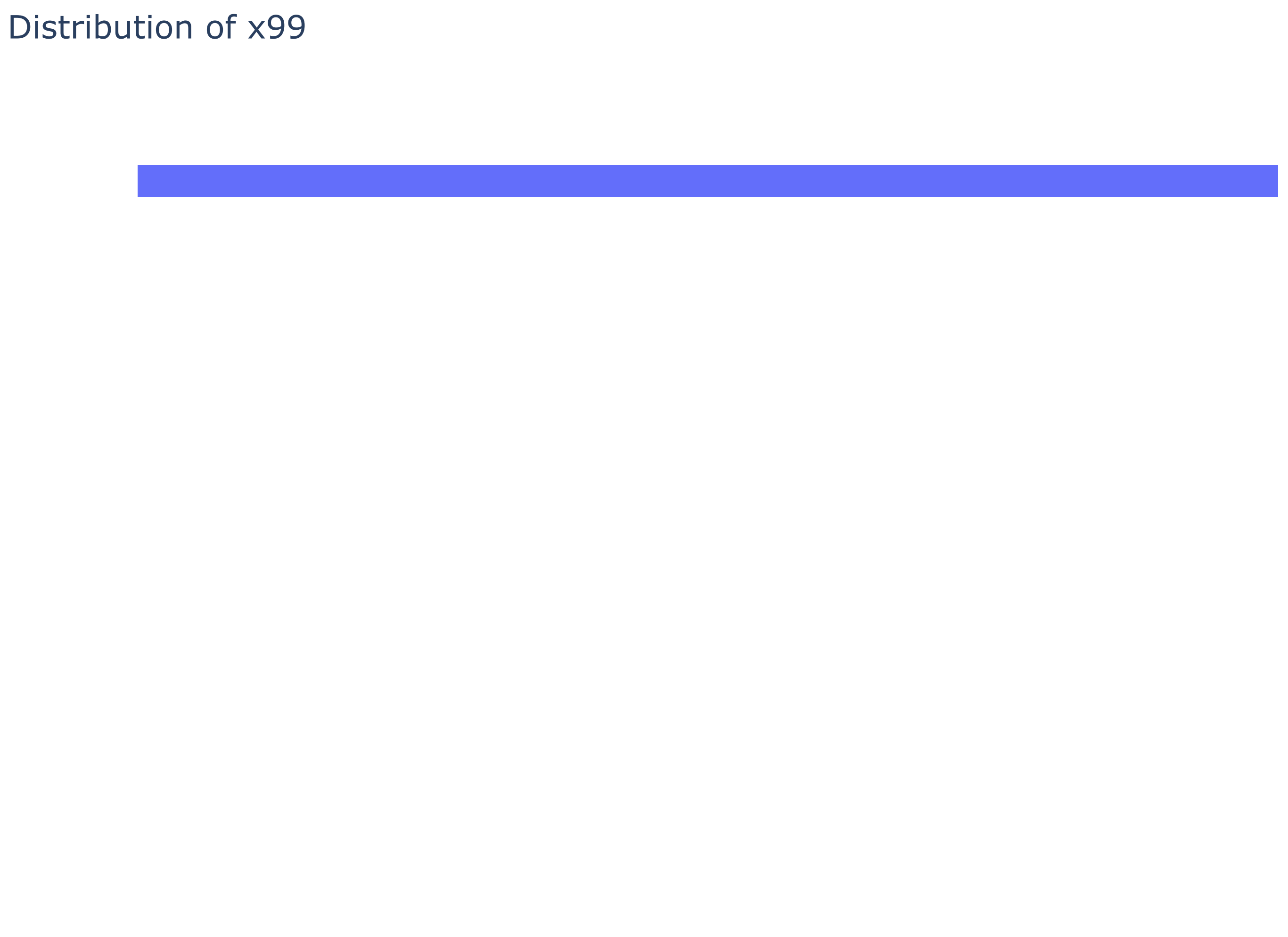

Distribution of x33
3500


Distribution of x60
8000

Distribution of x65


Distribution of x77
9000

Distribution of x93
35k



The most common days are Wednesday Tuesday and Monday. The distribution of gender is balanced. Column x31 is distributed towards no, while the most common states are California and Texas. The months are distributed towards the winter and summer months. The most popular insurance companies are Progressive and Allstate, while the least common is Geico. The most common car manufacturer is Ford, while the least common is Buick. Column x93 is distributed towards no, while x99 is distributed towards yes. The distribution of these columns are likely to change after imputation.

Test Dataset

In [ ]:
```
# values of column
test.x3.value_counts(dropna=False)
```

Out[ ]:
```
Wednesday    1224
Friday       1089
Tuesday      1010
Monday       1005
Sunday        953
Saturday      846
Thursday      702
Tue           688
Wed           524
Mon           522
Thur          426
Sat           425
Fri           382
Sun           204
Name: x3, dtype: int64
```

In [ ]:
```
# being consistent with labeling, short notation
test.x3 = test.x3.str.replace('Sunday', 'Sun')
test.x3 = test.x3.str.replace('Monday', 'Mon')
test.x3 = test.x3.str.replace('Tuesday', 'Tue')
test.x3 = test.x3.str.replace('Wednesday', 'Wed')
test.x3 = test.x3.str.replace('Thursday', 'Thur')
test.x3 = test.x3.str.replace('Friday', 'Fri')
test.x3 = test.x3.str.replace('Saturday', 'Sat')
```

We combined the corresponding days to the shorthand notation.

In [ ]:
```
# values of column
test.x24.value_counts(dropna=False)
```

Out[ ]:
```
female    4532
male      4499
NaN        969
Name: x24, dtype: int64
```

Missing values need to be imputed.

In [ ]:
```
# check values
test.x33.value_counts(dropna=False)
```

```
Out[ ]: NaN               1770
        California         841
        Texas              593
        Florida            475
        New York           462
        Pennsylvania       321
        Illinois           306
        Ohio               278
        Michigan           245
        North Carolina     238
        Georgia            236
        New Jersey         204
        Washington         189
        Virginia           188
        Massachusetts      178
        Indiana            162
        Colorado           160
        Tennessee          157
        Oklahoma           153
        Missouri           153
        Alabama            149
        Minnesota          148
        Wisconsin          145
        Maryland           139
        South Carolina     132
        Arizona            124
        Louisiana          119
        Kentucky           114
        Arkansas           113
        Utah               109
        Oregon             102
        Connecticut        100
        Iowa                89
        Nevada              88
        Kansas              87
        Mississippi         85
        Nebraska            77
        New Hampshire       73
        Idaho               67
        West Virginia       65
        New Mexico          62
        Rhode Island        57
        Maine               54
        South Dakota        50
        North Dakota        48
```

```
Hawaii              46
Alaska              45
DC                  44
Vermont             41
Wyoming             41
Montana             40
Delaware            38
Name: x33, dtype: int64
```

Again, there are 52 values for a missing value with the most counts.

In [ ]:
```
# Change values to 1
test.x39 = test.x39.str.replace('5-10 miles', '1').astype(int)
```

All rows of this column are the same, so we will change the value to 1.

In [ ]:
```
# checking values
test.x60.value_counts(dropna=False)
```

Out[ ]:
```
August       2055
July         2050
December     2028
January      1935
September     295
June          279
February      277
April         240
November      238
May           211
March         210
October       182
Name: x60, dtype: int64
```

No duplicate naming is seen here, and all 12 months are present.

In [ ]:
```
# checking values
test.x65.value_counts(dropna=False)
```

Out[ ]:
```
progressive    2703
allstate       2686
esurance       1828
farmers        1451
geico          1332
Name: x65, dtype: int64
```

This column represents the different insurance companies.

In [ ]:
```
# checking values
test.x77.value_counts(dropna=False)
```

Out[ ]:
```
ford         2325
NaN          2318
chevrolet    1265
subaru       1209
mercedes     1081
toyota        903
nissan        617
buick         282
Name: x77, dtype: int64
```

This column represents different vehicle manufacturers.

In [ ]:
```
# checking values
test.x93.value_counts(dropna=False)
```

Out[ ]:
```
no     8848
yes    1152
Name: x93, dtype: int64
```

In [ ]:
```
# values of column
test.x99.value_counts(dropna=False)
```

Out[ ]:
```
yes    6700
NaN    3300
Name: x99, dtype: int64
```

Missing values in this column are more likely to be no, rather than missing yes values. Therefore, we will fill in missing vales with no, just as we did with the training set.

In [ ]:
```
# fill missing values with no
test.x99.fillna('no', inplace=True)
```

In [ ]:
```
# check proper implementation
test.x99.value_counts(dropna=False)
```

Out[ ]:
```
yes    6700
no     3300
Name: x99, dtype: int64
```

Filled missing values with no.

In [ ]:
```
# summary statistics on data
test.describe()
```

Out[ ]:

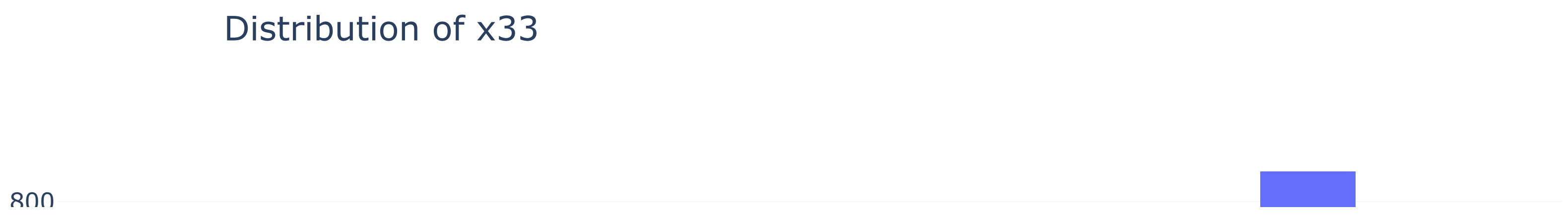
	x1	x2	x4	x5	x6	x7	x8	x9	x10	
count	10000.000000	10000.000000	10000.000000	9398.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	8671.00
mean	2.944648	20.003002	0.004528	0.001215	0.001926	0.000008	-0.003416	2.710221	0.506369	99.91
std	2.018091	1.600368	1.449873	1.290027	1.363301	0.009927	1.442214	1.985433	1.028552	13.25
min	-2.639067	13.790389	-4.768309	-4.662646	-5.720785	-0.036100	-5.627568	-3.160208	-3.452189	51.48
25%	1.522883	18.926348	-1.025638	-0.878598	-0.931918	-0.006800	-0.978422	1.328622	-0.196678	90.98
50%	2.817275	20.013331	-0.007336	-0.009562	0.001364	0.000100	0.000347	2.467988	0.509366	99.91
75%	4.223699	21.083448	1.041062	0.882272	0.925603	0.006700	0.980095	3.797335	1.200406	108.72
max	11.737364	25.808760	4.653302	4.709272	5.096100	0.048300	5.326779	17.165790	4.666843	148.31

In [ ]:
```
# distribution of object columns
for col in test.select_dtypes('object'):
    fig = px.histogram(test[col], title='Distribution of '+ str(col), template='plotly_white')
    fig.show()
```

Distribution of x3
1800

Distribution of x24
4500

Distribution of x31


Distribution of x33
800

Distribution of x60
2000

Distribution of x65

Distribution of x77

Distribution of x93
9000

We see similar distributions in these columns to the respective columns in the training set.

EDA Conclusions

We observe some patterns in the dataset. We see certain weekdays and certain months are more prevalent in the datasets. Comparing the train and test datasets, we see many columns have similar distributions.

Preprocessing

In [ ]:
```
# separate features and target
X = train.drop(columns='y')
y = train.y
```

In [ ]:
```
# values of the target
y.value_counts()
```

Out[ ]:
```
0    34197
1     5803
Name: y, dtype: int64
```

Target values are very imbalanced, therefore, we wil train models to optimize AUC or F1 scores. The appropriate metric depends on the specific problem and the business needs.

If the business problem involves minimizing false positives and false negatives equally, then optimizing on AUC may be appropriate, as AUC measures the ability of a model to distinguish between positive and negative classes.

However, if the business problem is such that minimizing false positives is more important than minimizing false negatives, or vice versa, then optimizing on F1 score may be more appropriate. F1 score is the harmonic mean of precision and recall and is a good metric to use when there is an uneven class distribution.

In [ ]:
```
# ordinal encoding days and months in order
weekday_names = ['Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat', 'Sun']
month_names = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'Novem

encoder_day = OrdinalEncoder(categories=[weekday_names])
encoder_month = OrdinalEncoder(categories=[month_names])

days = pd.DataFrame(encoder_day.fit_transform(X.x3.to_numpy().reshape(-1,1)), columns=['day'])
months = pd.DataFrame(encoder_month.fit_transform(X.x60.to_numpy().reshape(-1,1)), columns=['month'])
```

In [ ]:
```
# replace columns with ordinal columns
X['x3'] = days
X['x60'] = months
```

In [ ]:
```
# check for proper implementation
X.head()
```

Out[ ]:

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14
0	0.165254	18.060003	2.0	1.077380	-1.339233	-1.584341	0.0062	0.220784	1.816481	1.171788	109.626841	4.644568	4.814885	1.541740
1	2.441471	18.416307	4.0	1.482586	0.920817	-0.759931	0.0064	1.192441	3.513950	1.419900	84.079367	1.459868	1.443983	NaN
2	4.427278	19.188092	3.0	0.145652	0.366093	0.709962	-0.0008	0.952323	0.782974	-1.247022	95.375221	1.098525	1.216059	0.450624
3	3.925235	19.901257	1.0	1.763602	-0.251926	-0.827461	-0.0057	-0.520756	1.825586	2.223038	96.420382	-1.390239	3.962961	NaN
4	2.868802	22.202473	6.0	3.405119	0.083162	1.381504	0.0109	-0.732739	2.151990	-0.275406	90.769952	7.230125	3.877312	0.392002

Encoding all columns with ordinal encoding did not retain the order of days and months. Since there appears to be a trend in the data with respect to days and months, we want to retain the proper order of these labels. So we will encode these columns first, and then encode the other categorical columns later.

In [ ]:
```
# preprocessing steps
preprocessor = Pipeline([('ordinal_encoder', OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1)), ('i

# Preprocess the test data
X_processed = preprocessor.fit_transform(X)
```

In [ ]:
```
# implement SMOTE for class balance
oversampler = SMOTE(random_state=19)
X_final, y_final = oversampler.fit_resample(X_processed, y)
```

In [ ]:
```
# shape of the final dataframe
X_final.shape
```

Out[ ]:
```
(68394, 100)
```

In [ ]:
```
# targets are now balanced
y_final.value_counts()
```

Out[ ]:
```
0    34197
1    34197
Name: y, dtype: int64
```

In [ ]:
```
# train and valid split
X_train, X_valid, y_train, y_valid = train_test_split(
    X_final, y_final, test_size=0.25, random_state=19)
```

Preprocessing Conclusions

We preprocessed the data to convert the categorical columns into numerically labeled columns. Although some of our models selected can handle categorical values, we prefer to train the models on continuous values. We imputed the missing vales with simple imputer, scaled the data, and then implemented SMOTE to address class imbalance. Finally, we split the data into train and validation sets for hyperparameter tuning.

Modeling

Tuning with Grid Search CV

In [ ]:
```
# Gridsearch CV for hyperparameter tuning

# Create a LightGBM dataset
lgb_train = lgb.Dataset(X_train, y_train)
lgb_valid = lgb.Dataset(X_valid, y_valid, reference=lgb_train)

# Define the parameter grid for the LightGBM model
param_grid = {
    'boosting_type': ['gbdt'],
    'num_leaves': [10, 15, 20],
    'max_depth': [3, 4, 5],
    'learning_rate': [0.1, 0.2],
    'n_estimators': [100, 200, 300],
    'random_state': [19]
}

# Define the parameters for the LightGBM model
params = {
    'objective': 'binary',
    'metric': 'auc',
}

# Create the GridSearchCV object
grid_search = GridSearchCV(LGBMClassifier(**params), param_grid, cv=2, scoring='roc_auc',verbose=3, n_jobs=-1)

# Fit the GridSearchCV object to the data
grid_search.fit(X_train, y_train)
```

```
# Print the best parameters and the best score
print("Best parameters: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)
```

```
Fitting 2 folds for each of 54 candidates, totalling 108 fits
Best parameters:  {'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200, 'num_leaves': 2
0, 'random_state': 19}
Best score:  0.963786706705483
```

In [ ]:
```
# XG boost hyperparameter tuning
param_grid = {
    'booster':['gbtree'],
    'max_depth': [3, 4],
    'learning_rate': [0.1],
    #'n_estimators': [100, 200, 300],
    'eval_metric':['auc']
}

# Create the XGBoost model
xgb = XGBClassifier(random_state=19)

# Create the GridSearchCV object
grid_search = GridSearchCV(xgb, param_grid, cv=2, scoring='roc_auc', verbose=3, n_jobs=-1)


# Fit the GridSearchCV object to the data
grid_search.fit(X_train, y_train)

# Print the best parameters and the best score
print("Best parameters: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)
```

```
Fitting 2 folds for each of 2 candidates, totalling 4 fits
Best parameters:  {'booster': 'gbtree', 'eval_metric': 'auc', 'learning_rate': 0.1, 'max_depth': 4}
Best score:  0.9624760773067411
```

We used Grid Search CV to tune hyperparameters of each model we selected, and we will use the best parameters in the pipeline.

Tuning Neural Network

In [ ]:
```
# tuning neural network
optimizer = Adam(learning_rate=0.001)

model = keras.models.Sequential()
```

```
model.add(
    keras.layers.Dense(
        units=100, input_dim=X_train.shape[1], activation='relu'
    ))
model.add(keras.layers.Dense(
        units=75, activation='relu'
    ))
model.add(keras.layers.Dense(
        units=50, activation='relu'
    ))
model.add(keras.layers.Dense(
        units=25, activation='relu'
    ))
model.add(keras.layers.Dense(
        units=5, activation='relu'
    ))
model.add(keras.layers.Dense(
        units=1, activation='sigmoid'
    ))
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['AUC'])
model.fit(X_train, y_train, epochs=10, verbose=2,
          validation_data=(X_valid, y_valid))
```

```
Epoch 1/10
1603/1603 - 18s - loss: 0.4934 - auc: 0.8402 - val_loss: 0.4097 - val_auc: 0.8971 - 18s/epoch - 11ms/step
Epoch 2/10
1603/1603 - 9s - loss: 0.3667 - auc: 0.9168 - val_loss: 0.3585 - val_auc: 0.9216 - 9s/epoch - 6ms/step
Epoch 3/10
1603/1603 - 9s - loss: 0.3070 - auc: 0.9423 - val_loss: 0.3318 - val_auc: 0.9334 - 9s/epoch - 6ms/step
Epoch 4/10
1603/1603 - 11s - loss: 0.2631 - auc: 0.9577 - val_loss: 0.3301 - val_auc: 0.9362 - 11s/epoch - 7ms/step
Epoch 5/10
1603/1603 - 10s - loss: 0.2297 - auc: 0.9676 - val_loss: 0.3399 - val_auc: 0.9381 - 10s/epoch - 6ms/step
Epoch 6/10
1603/1603 - 10s - loss: 0.2064 - auc: 0.9738 - val_loss: 0.3424 - val_auc: 0.9422 - 10s/epoch - 6ms/step
Epoch 7/10
1603/1603 - 10s - loss: 0.1821 - auc: 0.9795 - val_loss: 0.3408 - val_auc: 0.9417 - 10s/epoch - 6ms/step
Epoch 8/10
1603/1603 - 10s - loss: 0.1648 - auc: 0.9831 - val_loss: 0.3326 - val_auc: 0.9454 - 10s/epoch - 6ms/step
Epoch 9/10
1603/1603 - 10s - loss: 0.1497 - auc: 0.9859 - val_loss: 0.3425 - val_auc: 0.9464 - 10s/epoch - 6ms/step
Epoch 10/10
1603/1603 - 10s - loss: 0.1390 - auc: 0.9877 - val_loss: 0.3196 - val_auc: 0.9487 - 10s/epoch - 6ms/step
```

```
Out[ ]: <keras.callbacks.History at 0x1a876c620a0>
```

A more complicated neural network with more layers and epochs can lead to overfitting. We trained models with 0.99 AUC with the training set, but with 0.95 AUC with the validation set.

Model Pipeline

In [ ]:
```
# Classifier pipeline
pipe_lr = Pipeline([('lr_classifier', LogisticRegression(random_state=19, max_iter=2000))])
pipe_dt = Pipeline([('dt_classifier', DecisionTreeClassifier(random_state=19, max_depth=3))])
pipe_rf = Pipeline([('rf_classifier', RandomForestClassifier(random_state=19, n_estimators=40))])
pipe_sv = Pipeline([('svm_classifier', svm.LinearSVC(random_state=19, max_iter=2000))])
pipe_xg = Pipeline([('xg_classifier', XGBClassifier(random_state=19, n_estimators=200, learning_rate=0.1, eval_metric='
pipe_lb = Pipeline([('lb_classifier', LGBMClassifier(boosting_type='gbdt', random_state=19, objective='binary', metric=
pipe_ml = Pipeline([('ml_classifier', MLPClassifier(max_iter=200, random_state=19, early_stopping=True, n_iter_no_chang

pipelines = [pipe_lr, pipe_dt, pipe_rf, pipe_sv, pipe_xg, pipe_lb, pipe_ml]

best_auc = 0
best_classifier = 0
best_pipeline = ""

pipe_dict = {0: 'Logistic Regression', 1: 'Decision Tree', 2: 'Random Forest', 3: 'SVM', 4: 'XG Boost', 5: 'Light GBM',

# Use cross-validation to evaluate the models
for i, model in enumerate(pipelines):
    model.fit(X_train, y_train)
    scores = cross_val_score(model, X_final, y_final, cv=5, scoring='roc_auc')
    print('{} Cross-Validation AUC: {:.2f}'.format(pipe_dict[i], scores.mean()))
    if scores.mean() > best_auc:
        best_auc = scores.mean()
        best_pipeline = model
        best_classifier = i

# Print the best classifier
print('\nClassifier with the best AUC-ROC score: {}'.format(pipe_dict[best_classifier]))
```

```
Logistic Regression Cross-Validation AUC: 0.77
Decision Tree Cross-Validation AUC: 0.73
Random Forest Cross-Validation AUC: 0.98
```

```
c:\Users\XIX\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.

c:\Users\XIX\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.

c:\Users\XIX\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.

c:\Users\XIX\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.

c:\Users\XIX\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.

c:\Users\XIX\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.
```

```
SVM Cross-Validation AUC: 0.77
XG Boost Cross-Validation AUC: 0.96
Light GBM Cross-Validation AUC: 0.96
Neural Network Cross-Validation AUC: 0.94

Classifier with the best AUC-ROC score: Random Forest
```

We tried to implement two other imputers, KNN and iterative imputer. However, they were too computationally intensive for this system. KNN and iterative imputer use machine learning to impute the missing values, and increased accuracy of the imputed values comes at a cost in terms of model training time. Consequently, we will use simple imputation. The models were trained on the training set, and cross validation was used to determine average AUC scores.

In [ ]:
```python
# dummy model
pipe_dm = Pipeline([('dm_classifier', DummyClassifier(random_state=19, strategy='most_frequent'))])
pipe_dm.fit(X_processed, y)

scores = cross_val_score(pipe_dm, X_processed, y, cv=5, scoring='roc_auc')
final_score = sum(scores) / len(scores)
print('Average model AUC ROC score:', final_score)
```

```
Average model AUC ROC score: 0.5
```

In [ ]:
```
# accuracy function of dummy model on imbalanced data
accuracy_score(y, pipe_dm.predict(X))
```

Out[ ]:
```
0.854925
```

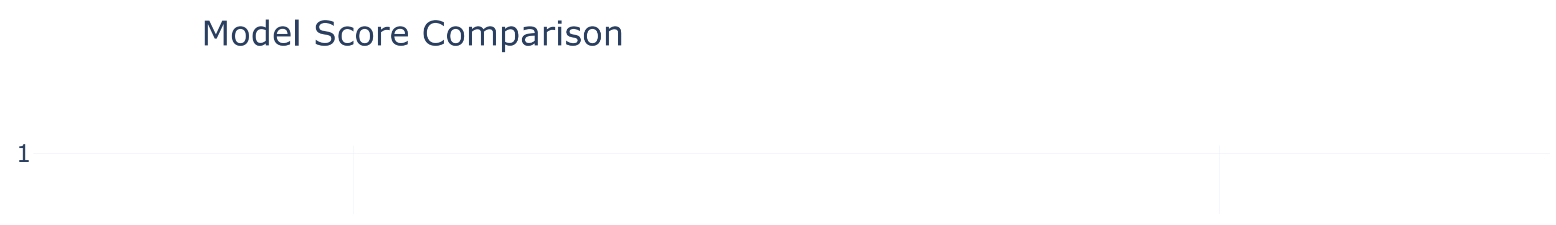
In [ ]:
```
# accuracy function of balanced data
accuracy_score(y_final, pipe_dm.predict(X_final))
```

Out[ ]:
```
0.5
```

A dummy model was trained to illustrate two things: the effect of class imbalance, and the difference between AUC and accuracy. This dummy is a baseline model that disregards the features, and always predicts the majority class, 0. As we can see, the accuracy of the model is 0.85, while the AUC score is also 0.5, when we use imbalanced data. However, accuracy is not a useful metric with imbalanced targets, because it does not properly illustrate the model's performance on the minority class with false negatives.Once we balance the classes, the accuracy of the dummy model drops down to 0.5.

Compare Model Scores

In [ ]:
```
# series of model scores
data = {'Logistic Regression': 0.7728, 'Decision Tree': 0.7378 , 'Random Forest': 0.9754, 'SVM': 0.7729, 'XG Boost': 0.
comp = pd.Series(data, name='AUC Score')

# model scores
fig = px.scatter(comp, color=comp.index, size=comp, title='Model Score Comparison', symbol=comp, labels={'index': 'Mode
fig.show()
```

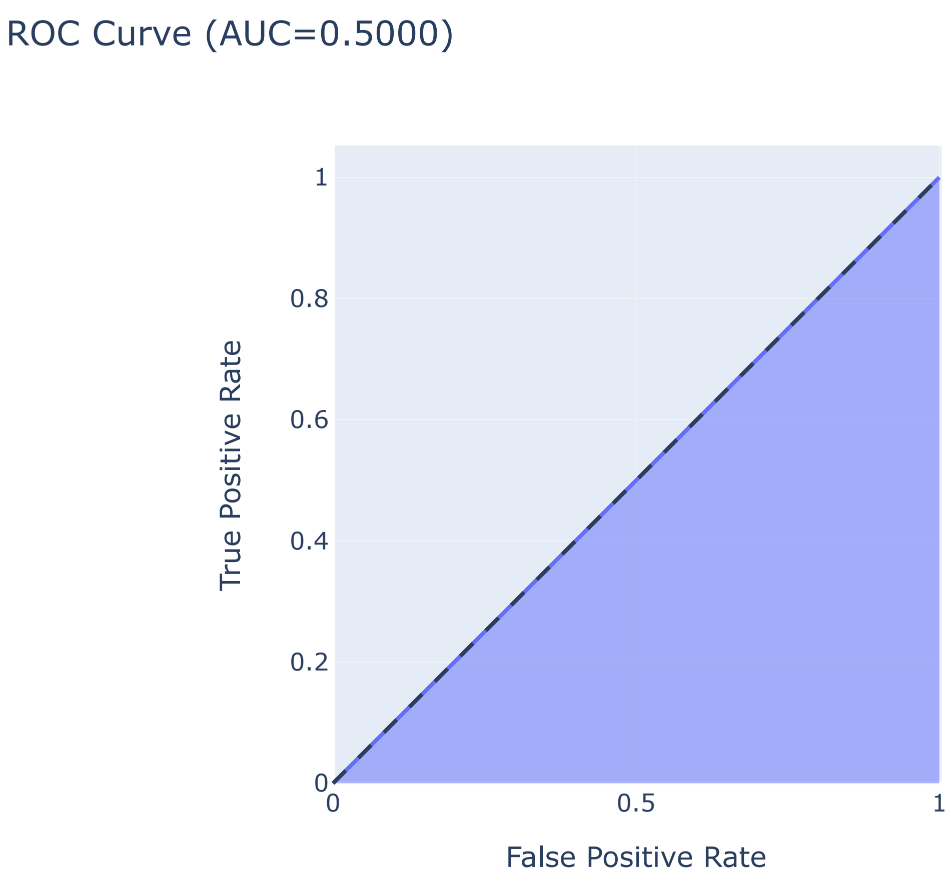


Dummy Model AUC

```
In [ ]: # dummy model
probabilities_valid = pipe_dm.predict_proba(X_valid)
probabilities_one_valid = probabilities_valid[:, 1]

auc_roc = roc_auc_score(y_valid, probabilities_one_valid)

print(auc_roc)

# ROC AUC curve of results
```

```
fpr, tpr, thresholds = roc_curve(y_valid, probabilities_one_valid)

fig = px.area(
    x=fpr, y=tpr,
    title=f'ROC Curve (AUC={auc(fpr, tpr):.4f})',
    labels=dict(x='False Positive Rate', y='True Positive Rate'),
    width=700, height=500
)
fig.add_shape(
    type='line', line=dict(dash='dash'),
    x0=0, x1=1, y0=0, y1=1
)

fig.update_yaxes(scaleanchor="x", scaleratio=1)
fig.update_xaxes(constrain='domain')
fig.show()
```

```
0.5
```

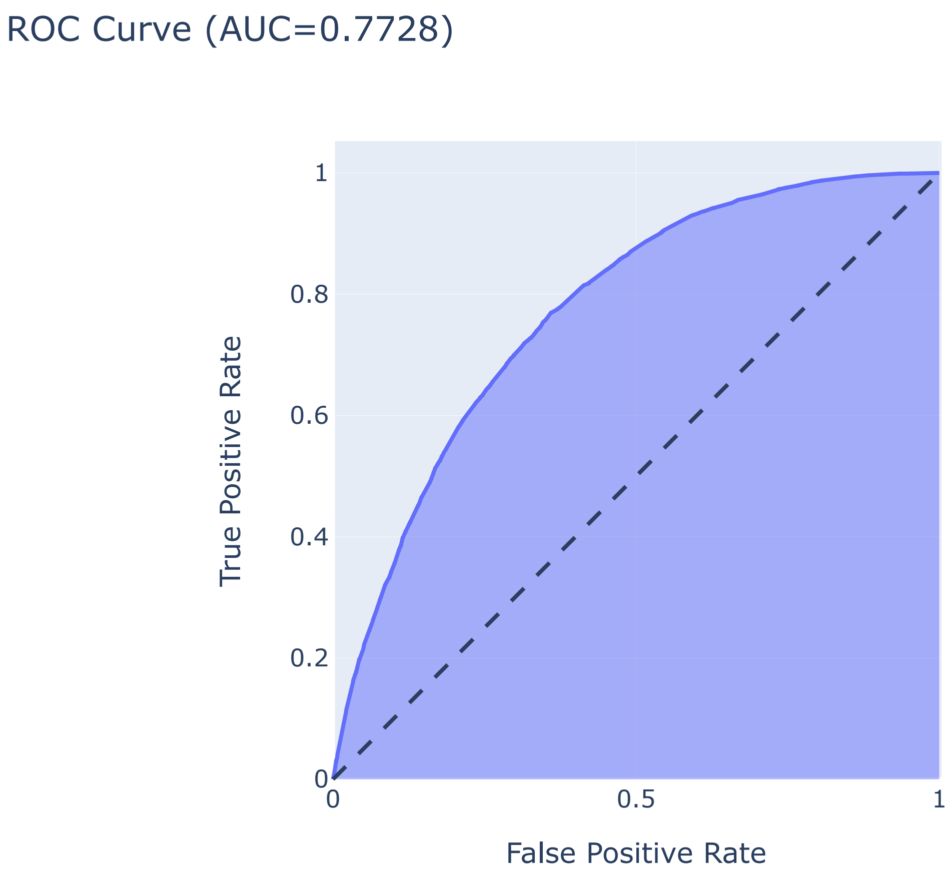


Logistic Regression AUC

In [ ]:
```
probabilities_valid = pipe_lr.predict_proba(X_valid)
probabilities_one_valid = probabilities_valid[:, 1]

auc_roc = roc_auc_score(y_valid, probabilities_one_valid)

print(auc_roc)

# ROC AUC curve of results
fpr, tpr, thresholds = roc_curve(y_valid, probabilities_one_valid)
```

```
fig = px.area(
    x=fpr, y=tpr,
    title=f'ROC Curve (AUC={auc(fpr, tpr):.4f})',
    labels=dict(x='False Positive Rate', y='True Positive Rate'),
    width=700, height=500
)
fig.add_shape(
    type='line', line=dict(dash='dash'),
    x0=0, x1=1, y0=0, y1=1
)

fig.update_yaxes(scaleanchor="x", scaleratio=1)
fig.update_xaxes(constrain='domain')
fig.show()
```

```
0.7728281007460819
```

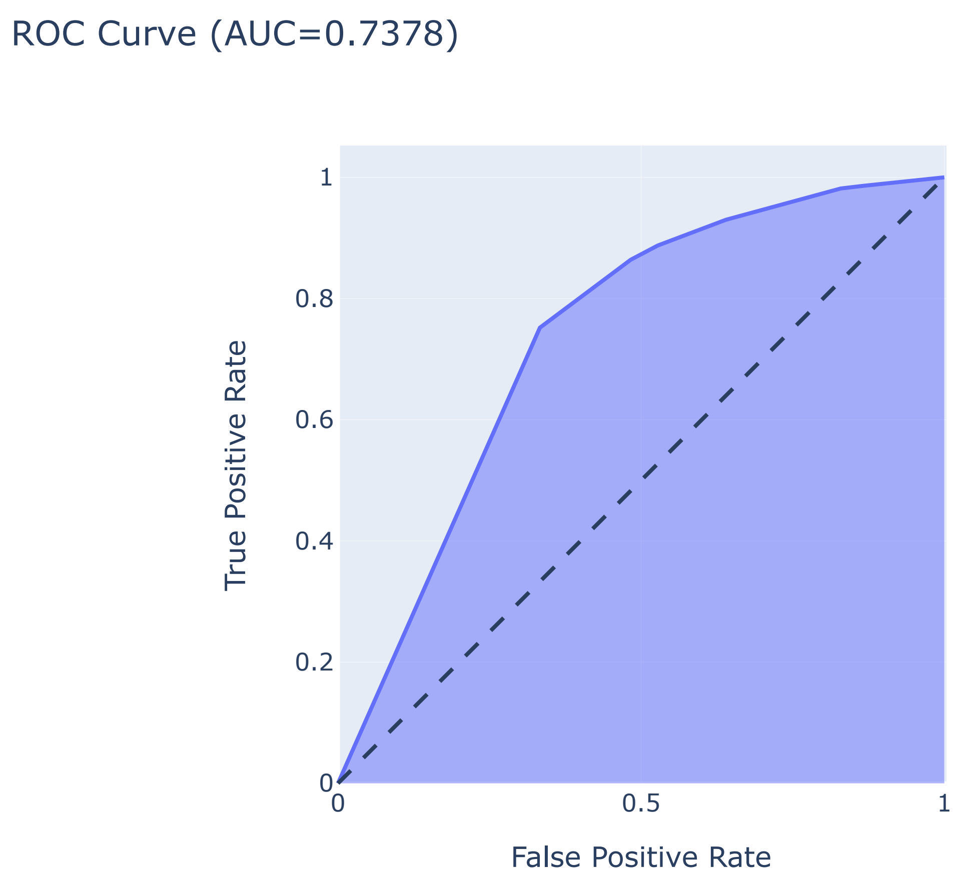


Decision Tree AUC

In [ ]:
```
probabilities_valid = pipe_dt.predict_proba(X_valid)
probabilities_one_valid = probabilities_valid[:, 1]

auc_roc = roc_auc_score(y_valid, probabilities_one_valid)

print(auc_roc)

# ROC AUC curve of results
fpr, tpr, thresholds = roc_curve(y_valid, probabilities_one_valid)
```

```
fig = px.area(
    x=fpr, y=tpr,
    title=f'ROC Curve (AUC={auc(fpr, tpr):.4f})',
    labels=dict(x='False Positive Rate', y='True Positive Rate'),
    width=700, height=500
)
fig.add_shape(
    type='line', line=dict(dash='dash'),
    x0=0, x1=1, y0=0, y1=1
)

fig.update_yaxes(scaleanchor="x", scaleratio=1)
fig.update_xaxes(constrain='domain')
fig.show()
```

```
0.7377709495708669
```

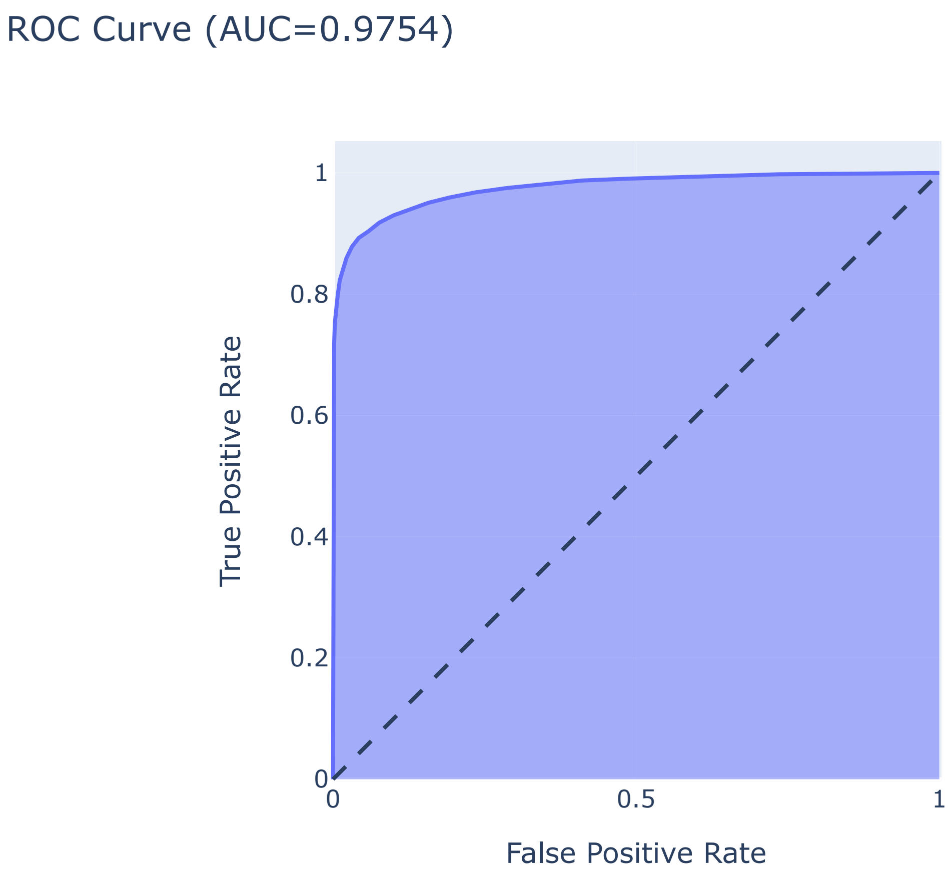


Random Forest AUC

In [ ]:
```
probabilities_valid = pipe_rf.predict_proba(X_valid)
probabilities_one_valid = probabilities_valid[:, 1]

auc_roc = roc_auc_score(y_valid, probabilities_one_valid)

print(auc_roc)

# ROC AUC curve of results
fpr, tpr, thresholds = roc_curve(y_valid, probabilities_one_valid)
```

```
fig = px.area(
    x=fpr, y=tpr,
    title=f'ROC Curve (AUC={auc(fpr, tpr):.4f})',
    labels=dict(x='False Positive Rate', y='True Positive Rate'),
    width=700, height=500
)
fig.add_shape(
    type='line', line=dict(dash='dash'),
    x0=0, x1=1, y0=0, y1=1
)

fig.update_yaxes(scaleanchor="x", scaleratio=1)
fig.update_xaxes(constrain='domain')
fig.show()
```

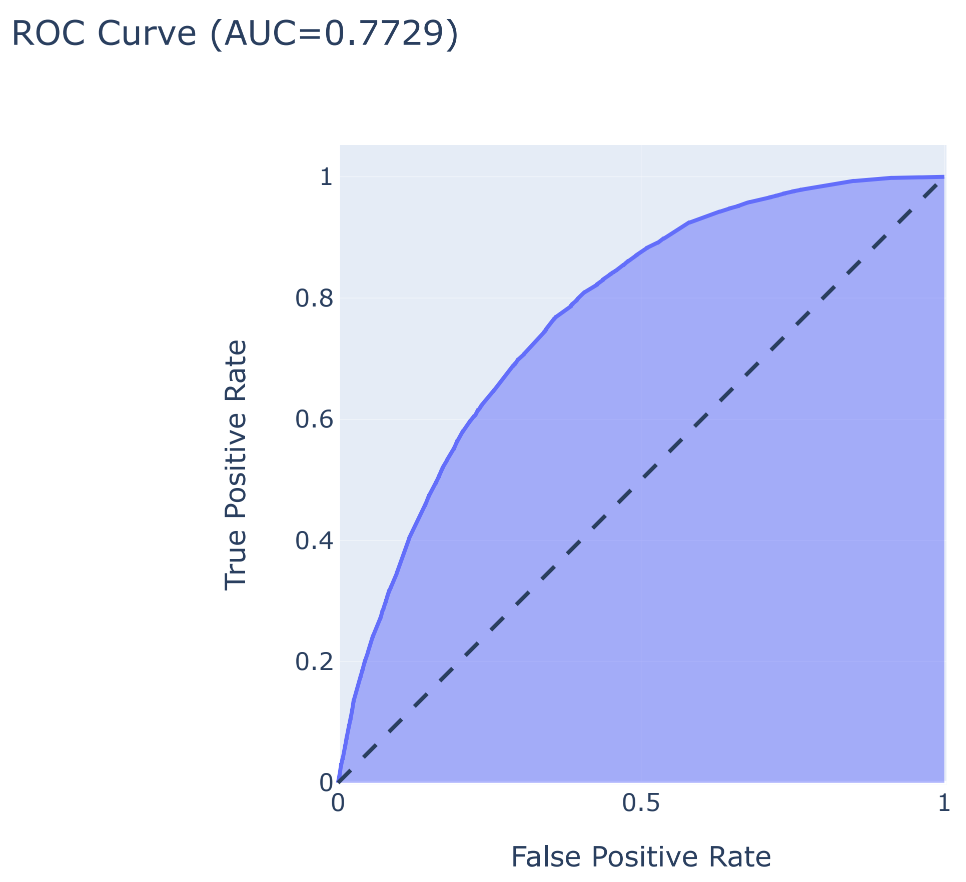
```
0.9754461250233604
```



SVM AUC

In [ ]:
```
probabilities_valid = pipe_sv.decision_function(X_valid)
auc_roc = roc_auc_score(y_valid, probabilities_valid)

print(auc_roc)

# ROC AUC curve of results
fpr, tpr, thresholds = roc_curve(y_valid, probabilities_valid)

fig = px.area(
    x=fpr, y=tpr,
    title=f'ROC Curve (AUC={auc(fpr, tpr):.4f})',
    labels=dict(x='False Positive Rate', y='True Positive Rate'),
    width=700, height=500
)
fig.add_shape(
    type='line', line=dict(dash='dash'),
    x0=0, x1=1, y0=0, y1=1
)

fig.update_yaxes(scaleanchor="x", scaleratio=1)
fig.update_xaxes(constrain='domain')
fig.show()
```

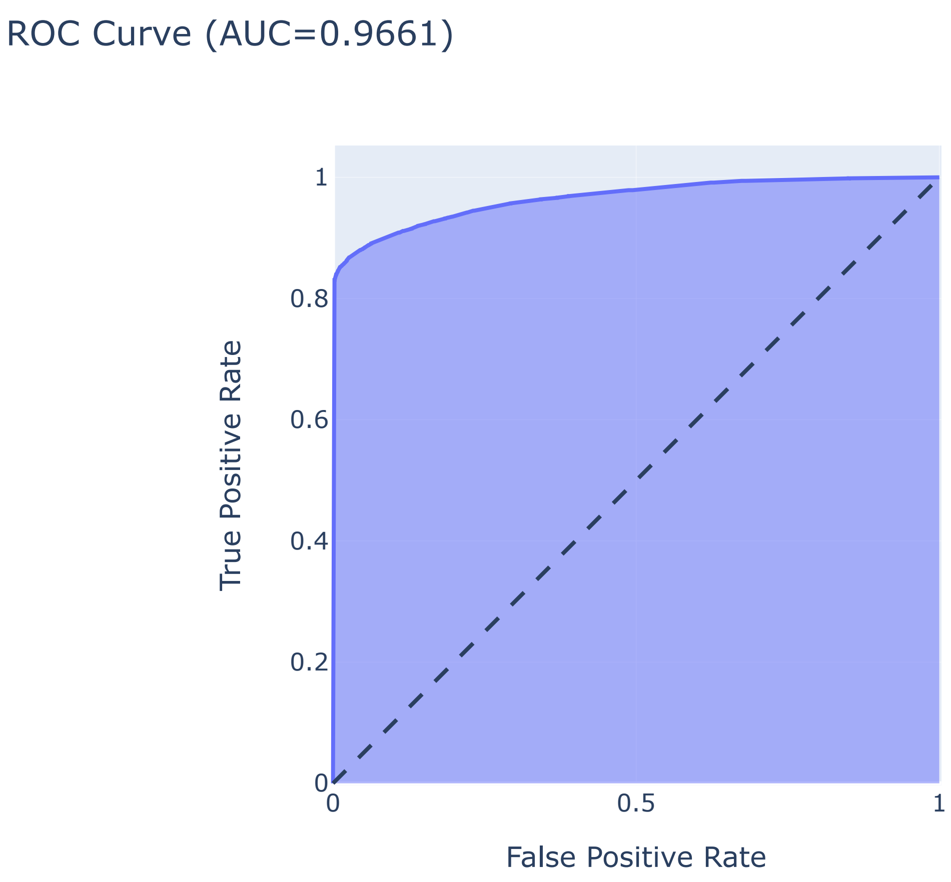
```
0.772904564076928
```



XG Boost AUC

```
In [ ]: probabilities_valid = pipe_xg.predict_proba(X_valid)
        probabilities_one_valid = probabilities_valid[:, 1]

        auc_roc = roc_auc_score(y_valid, probabilities_one_valid)

        print(auc_roc)

        # ROC AUC curve of results
        fpr, tpr, thresholds = roc_curve(y_valid, probabilities_one_valid)

        fig = px.area(
            x=fpr, y=tpr,
            title=f'ROC Curve (AUC={auc(fpr, tpr):.4f})',
            labels=dict(x='False Positive Rate', y='True Positive Rate'),
            width=700, height=500
        )
        fig.add_shape(
            type='line', line=dict(dash='dash'),
            x0=0, x1=1, y0=0, y1=1
        )

        fig.update_yaxes(scaleanchor="x", scaleratio=1)
        fig.update_xaxes(constrain='domain')
        fig.show()
```

```
0.9660532498899361
```

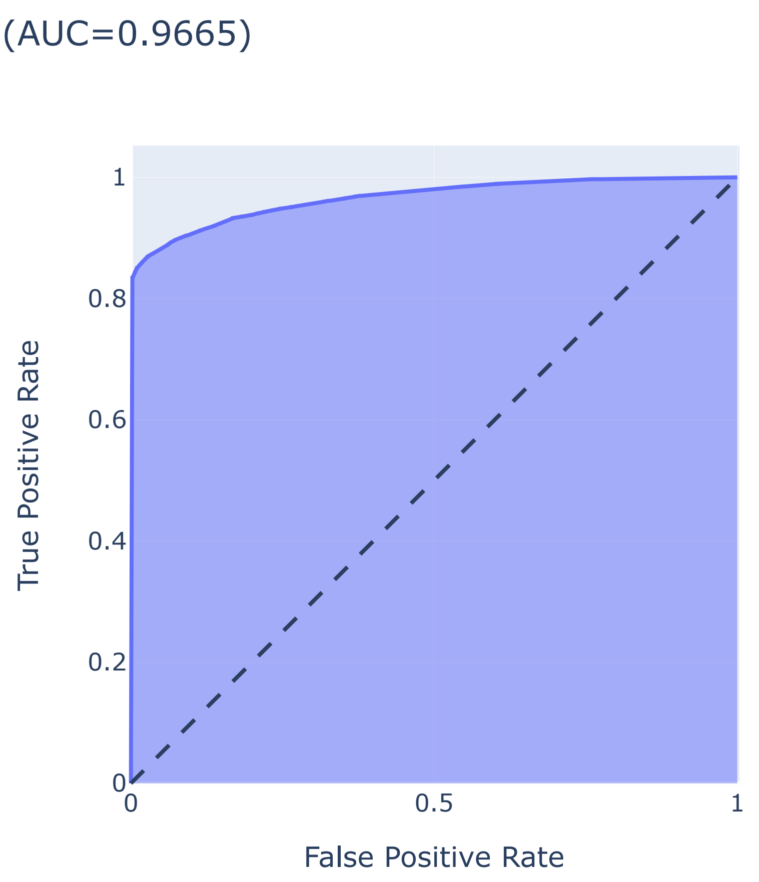


Light GBM AUC

```
In [ ]: probabilities_valid = pipe_lb.predict_proba(X_valid)
        probabilities_one_valid = probabilities_valid[:, 1]

        auc_roc = roc_auc_score(y_valid, probabilities_one_valid)

        print(auc_roc)

        # ROC AUC curve of results
        fpr, tpr, thresholds = roc_curve(y_valid, probabilities_one_valid)
```

```
fig = px.area(
    x=fpr, y=tpr,
    title=f'ROC Curve (AUC={auc(fpr, tpr):.4f})',
    labels=dict(x='False Positive Rate', y='True Positive Rate'),
    width=700, height=500
)
fig.add_shape(
    type='line', line=dict(dash='dash'),
    x0=0, x1=1, y0=0, y1=1
)

fig.update_yaxes(scaleanchor="x", scaleratio=1)
fig.update_xaxes(constrain='domain')
fig.show()
```

```
0.9665248900523244
```



Modelling Conclusions

AUC ROC is a metric that compares the True positive rate with the False Positive Rate. The dashed line through the curve represents 0.50, the score of a random model. AUC scores closer to 0 are poor performing, while a perfect AUC score is 1. We see most models performed well, and some performed excellent, when compared to a random model.

Logistic regression is a model that is simple, fast, and easily interpretable. Logistic regression works well with linearly separable data, and it can handle large datasets with low computational cost. A weakness of this model include its assumption that the input features are linearly separable, which may lead to poor performance, high bias, and underfitting when the data is too complex. Decision trees are also easily interpretable, and they can handle categorical data. It can handle categorical data by implementing one-hot encoding.

Decision trees can also capture non-linear relationships. Weaknesses include its inclination to overfit the training data, and not generalize new data. Random forest can also handle categorical and continuous data, and it reduces overfitting by using multiple trees. Random forest is less interpretable than the previous methods, and requires hyperparameter tuning to reduce overfitting. Linear SVC is good for binary classification tasks, and can handle high-dimensional data. SVC models do not work well with imbalanced classes, can be sensitive to outliers, and are slow to train on large datasets. XG boost models can handle both categorical and continuous data, and reduce overfitting by using multiple trees. XG boost models may require significant tuning, which is a downside for those who are not familiar with this algorithm. Light GBM is similar to XG boost, but can handle larger datasets faster and with less memory. However, this model requires hyperparameter tuning to reduce overfitting. MLP models and other neural networks can handle complex relationships between features and targets. Neural networks can be computationally extensive, require hyperparameter tuning, and can suffer from overfitting.

Overall, the best model to use depends on the problem at hand, the size and complexity of the data, and the level of interpretability.

Feature Importance

Logistic Regression

In [ ]:
```
# Logistic regression pipeline feature importance
pipe_lr.fit(X_train, y_train)

logreg_classifier = pipe_lr.named_steps['lr_classifier']
logreg_importances = logreg_classifier.coef_
logreg_indices = np.argsort(logreg_importances)[::-1]
```

In [ ]:
```
# making dataframe of important coefficients
lr_importance = pd.DataFrame(logreg_importances, columns=X.columns)
lr_importance = lr_importance.T
lr_top_10_df = lr_importance.nlargest(10, columns=0)
```

In [ ]:
```
fig = px.pie(lr_top_10_df, names=lr_top_10_df.index, values=0, title='Top 10 Linear Regression Coefficients')
fig.show()
```

Top 10 Linear Regression Coefficients

Decision Tree

In [ ]:
```
# decision tree pipeline feature importance
pipe_dt.fit(X_train, y_train)

dt_classifier = pipe_dt.named_steps['dt_classifier']
dt_importances = dt_classifier.feature_importances_
dt_indices = np.argsort(dt_importances)[::-1]

top_10_features = []
for f in range(10):
    feature_index = dt_indices[f]
    feature_name = train.columns[feature_index]
    top_10_features.append((feature_name, dt_importances[feature_index]))

dt_top_10_df = pd.DataFrame(top_10_features, columns=['Feature', 'Importance'])
```

In [ ]:
```
fig = px.pie(dt_top_10_df.head(2), title='Top Features of Decision Tree', names='Feature', values='Importance')
fig.show()
```

Top Features of Decision Tree

Random Forest

In [ ]:
```
# random forest pipeline feature importance
pipe_rf.fit(X_train, y_train)

rf_classifier = pipe_rf.named_steps['rf_classifier']
rf_importances = rf_classifier.feature_importances_
rf_indices = np.argsort(rf_importances)[::-1]

top_10_features = []
for f in range(10):
    feature_index = rf_indices[f]
    feature_name = train.columns[feature_index]
    top_10_features.append((feature_name, rf_importances[feature_index]))

rf_top_10_df = pd.DataFrame(top_10_features, columns=['Feature', 'Importance'])
```

In [ ]:
```
fig = px.pie(rf_top_10_df, title='Top 10 Features of Random Forest', names='Feature', values='Importance')
fig.show()
```

Top 10 Features of Random Forest

Support Vector

```
In [ ]: # Support vector pipeline feature importance
pipe_sv.fit(X_train, y_train)

svm_classifier = pipe_sv.named_steps['svm_classifier']
svm_importances = svm_classifier.coef_
svm_indices = np.argsort(svm_importances)[::-1]

# making dataframe of important coefficients
sv_importance = pd.DataFrame(svm_importances, columns=X.columns)
sv_importance = sv_importance.T
sv_top_10_df = sv_importance.nlargest(10, columns=0)
```

```
c:\Users\XIX\anaconda3\lib\site-packages\sklearn\svm\_base.py:1206: ConvergenceWarning:

Liblinear failed to converge, increase the number of iterations.
```

```
In [ ]: fig = px.pie(sv_top_10_df, names=sv_top_10_df.index, values=0, title='Top 10 Support Vector Coefficients')
        fig.show()
```

Top 10 Support Vector Coefficients

XG Boost

```
In [ ]: # xg boost pipeline feature importance
pipe_xg.fit(X_train, y_train)

xg_classifier = pipe_xg.named_steps['xg_classifier']
xg_importances = xg_classifier.feature_importances_
xg_indices = np.argsort(xg_importances)[::-1]

top_10_features = []
for f in range(10):
    feature_index = xg_indices[f]
    feature_name = train.columns[feature_index]
    top_10_features.append((feature_name, xg_importances[feature_index]))

xg_top_10_df = pd.DataFrame(top_10_features, columns=['Feature', 'Importance'])
```

```
In [ ]: fig = px.pie(xg_top_10_df, title='Top 10 Features of XG Boost', names='Feature', values='Importance')
fig.show()
```

Top 10 Features of XG Boost

Light GBM

In [ ]:
```
# light boost pipeline feature importance
pipe_lb.fit(X_train, y_train)

lb_classifier = pipe_lb.named_steps['lb_classifier']
lb_importances = lb_classifier.feature_importances_
lb_indices = np.argsort(lb_importances)[::-1]

top_10_features = []
for f in range(10):
    feature_index = lb_indices[f]
    feature_name = train.columns[feature_index]
    top_10_features.append((feature_name, lb_importances[feature_index]))

lb_top_10_df = pd.DataFrame(top_10_features, columns=['Feature', 'Importance'])
```

In [ ]:

```
fig = px.pie(lb_top_10_df, title='Top 10 Features of XG Boost', names='Feature', values='Importance')
fig.show()
```

Top 10 Features of XG Boost

If scoring metrics can not be used to chose a model, feature importance can help pick a model based on explainability. Explainability is how to take a machine learning model and express the behavior in human terms. With complex models, you can not fully understand how the model parameters impact predictions. With feature importance, we can pick a model based on how it makes predictions, and which features are most important to each model. Even without feature importance, a model can still be selected based on its interpretability, as simpler models are easier to explain to stakeholders.

Another factor in choosing a model is the resource requirement of the machine learning algorithms. More complex models require more memory or computing power to train or make predictions. With limited resources, model selection is limited to simpler models.

Furthermore, we can use visualizations to show how predictions of two models differ from actual values. A confusion matrix can show true positive and true negative values, and a visualization of the confusion matrix can illustrate the results of the classification model's predictions.

Simulate Scoring with Test Set

In [ ]:
```
# confusion matrix map
fig = go.Figure(data=go.Heatmap(z=[[1205, 185], [8557, 53]], text=[['False Negatives', 'True Positives'], ['True Negati
                texttemplate="%{text}", textfont={"size":20}))

fig.show()
```

```
In [ ]: # validation predictions of logistic regression
        valid_pred_lr = pipe_lr.predict(X_valid)
```

```
In [ ]: # confusion matrix of validation set of logistic regression
        fig = px.imshow(confusion_matrix(y_valid, valid_pred_lr), text_auto=True, labels=dict(y="Actual", x="Predicted"),
                        x=['Negative', 'Positive'],
                        y=['Negative', 'Positive'], title='Confusion Matrix of Logistic Regression')
        fig.show()
```

Confusion Matrix of Logistic Regression

The true negative value is 5841, while the true positive value is 6154. Overall, the model performed moderately at predicting the negative and positive class. The model had nearly half as many incorrect positive, and less than half as many negative class predictions, as the respective correct predictions.

In [ ]:
```
# validation predictions of xg boost
valid_pred_rf = pipe_rf.predict(X_valid)
```

In [ ]:
```
# confusion matrix of validation set of xg boost
fig = px.imshow(confusion_matrix(y_valid, valid_pred_rf), text_auto=True, labels=dict(y="Actual", x="Predicted"),
                x=['Negative', 'Positive'],
                y=['Negative', 'Positive'], title='Confusion Matrix of Random Forest')
fig.show()
```

Confusion Matrix of Random Forest

The confusion matrix illustrates the true negative value of 8195 and a true positive vale of 7625, which are predicted values that match actual values. Overall, the model was excellent at predicting the negative class, and fairly good at predicting the positive class. This is further supported by the false negative value of 915, which are the instances where the model incorrectly predicted a negative class. Our model performed best when we using SMOTE to balance our datasets. SMOTE works by using the K nearest neighbors algorithm to create synthetic examples of the minority class, thereby balancing the data.

The confusion matrix on the validation set is used to illustrate how we expect the model will perform on the test set.

In [ ]:
```
# validation f1 score of logistic regression
f1_score(y_valid, valid_pred_lr)
```

Out[ ]:
```
0.7068688260969446
```

In [ ]:
```
# classification report
print(classification_report(y_valid, valid_pred_lr))
```

```
              precision    recall  f1-score   support

           0       0.71      0.68      0.70      8559
           1       0.69      0.72      0.71      8540

    accuracy                           0.70     17099
   macro avg       0.70      0.70      0.70     17099
weighted avg       0.70      0.70      0.70     17099
```

The classification report breaks down the precision and recall of the model with respect to each class. Precision tells us how well the model identifies relevant instances, while recall tells us how well the model captures all relevant instances. A model high precision and recall is a strong model. With the Logistic regression model, we see moderate precision and recall with the negative class. The positive class has similar precision and recall. Consequently, the f1 scores of the negative and positive classes are both moderate.

In [ ]:
```
# validation f1 score of random forest
f1_score(y_valid, valid_pred_rf)
```

Out[ ]:
```
0.9226208482061831
```

In [ ]:
```
# classification report
print(classification_report(y_valid, valid_pred_rf))
```

```
              precision    recall  f1-score   support

           0       0.90      0.96      0.93      8559
           1       0.95      0.89      0.92      8540

    accuracy                           0.93     17099
   macro avg       0.93      0.93      0.93     17099
weighted avg       0.93      0.93      0.93     17099
```

In our case with random forest, we see high precision and recall in the negative class. The positive class has high precision, and slightly lower recall. As F1 score is the harmonic mean of precision and recall, both classes have a high F1 score.

Test Set scoring Predictions

Based on the confusion matrices and classification reports, we expect the random forest model to perform better. The random forest model had more true positive and true negative values than the logistic regression model, when comparing performance on the validation set.

Final Model Predictions

In [ ]:
```
# final Linear regression
final_lr = pipe_lr.fit(X_final, y_final)
```

In [ ]:
```
# Final xg boost model
final_rf = pipe_rf.fit(X_final, y_final)
```

Now that we have selected our final models, we use the full training set to fit the models.

In [ ]:
```
# ordinal encoding days and months in order

days_test = pd.DataFrame(encoder_day.fit_transform(test.x3.to_numpy().reshape(-1,1)), columns=['day'])
months_test = pd.DataFrame(encoder_month.fit_transform(test.x60.to_numpy().reshape(-1,1)), columns=['month'])
```

In [ ]:
```
# replace columns with ordinal columns
test['x3'] = days_test
test['x60'] = months_test
```

In [ ]:
```
test.head()
```

Out[ ]:

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x1
0	4.747627	20.509439	2.0	2.299105	-1.815777	-0.752166	0.0098	-3.240309	0.587948	-0.260721	101.113628	-0.812035	3.251085	-0.00443
1	1.148654	19.301465	4.0	1.862200	-0.773707	-1.461276	0.0076	0.443209	0.522113	-1.090886	104.791999	8.805876	1.651993	Nal
2	4.986860	18.769675	5.0	1.040845	-1.548690	2.632948	-0.0005	-1.167885	5.739275	0.222975	102.109546	7.831517	3.055358	2.03643
3	3.709183	18.374375	1.0	-0.169882	-2.396549	-0.784673	-0.0160	-2.662226	1.548050	0.210141	82.653354	0.436885	1.578106	Nal
4	3.801616	20.205541	0.0	2.092652	-0.732784	-0.703101	0.0186	0.056422	2.878167	-0.457618	75.036421	8.034303	1.631426	0.64373

In [ ]:
```python
# Preprocess the test data
X_test_transformed = preprocessor.transform(test)
```

In [ ]:
```python
# shape of test set
X_test_transformed.shape
```

Out[ ]:
```
(10000, 100)
```

We follow the same preprocessing steps as the training set, to transform the test set for the model.

In [ ]:
```python
# test set predictions
valid_pred_lr = final_lr.predict_proba(X_test_transformed)
valid_pred_rf = final_rf.predict_proba(X_test_transformed)
```

We run predictions on the transformed test datasets, and extract the probabilities of each class. The model will assign a class based on the highest predicted probability. The default threshold is 0.5. If class 0 predicted probability is higher than the 0.5 threshold, the model will predict a class of 0. Conversely, if the predicted probability of the positive class is greater than the threshold, the model will predict class 1. Probabilities allow us to determine how confident the model is in each class prediction, as probabilities closer to 1 are more certain than those closer to 0.5.

In [ ]:
```python
# probabilities of positive class
lr_list = valid_pred_lr[:,1].tolist()
```

We extract the predicted probabilities of the positive class. In other words, these values represent the predicted probability that the target is class 1.

In [ ]:
```
# Create a DataFrame from the list
lr_df = pd.DataFrame(lr_list)

# Save the DataFrame to a CSV file
#lr_df.to_csv('predictions/glmresults.csv', index=False, header=False)
```

We save the predictions to a csv file, where each value is the predicted probability of the positive class.

In [ ]:
```
# probabilities of positive class
rf_list = valid_pred_rf[:,1].tolist()
```

We extract the probabilities of the positive class.

In [ ]:
```
# Create a DataFrame from the list
rf_df = pd.DataFrame(rf_list)

# Save the DataFrame to a CSV file
#rf_df.to_csv('predictions/nonglmresults.csv', index=False, header=False)
```

We save the values to another csv file.

In [ ]:
```
# logistic regression class predictions
lr_class = pd.DataFrame(final_lr.predict(X_test_transformed))
```

In [ ]:
```
# class prediction counts
lr_class.value_counts()
```

Out[ ]:
```
1    9700
0     300
dtype: int64
```

The logistic regression model made 6347 negative predictions, and 3653 positive predictions.

In [ ]:
```
# xg boost class predictions
rf_class = pd.DataFrame(final_rf.predict(X_test_transformed))
```

In [ ]:
```
# class prediction counts
rf_class.value_counts()
```

Out[ ]:
```
0    7726
1    2274
dtype: int64
```

The random forest model made 896 more negative predictions than the logistic regression model.

Executive Summary

One of the main issues with the dataset was the amount of missing values. Deleting missing values leads to a loss of valuable information, and model performance would suffer, unless the proportion of missing data is minimal. Imputing these missing values could recover some of the missing information, which can result in a better model. However, the reason why the data is missing, as well as the imputation method implemented, can have a significant impact on the model performance.

The datasets were cleaned and preprocessed with ordinal encoding, standard scaling, simple imputing, and SMOTE. Ordinal encoding allowed us to convert categorical features into numerical labels, to then train our models. Standard scaling was implemented to improve the performance of the models, as features with much higher scales will be given greater weights, merely because they have larger values. By scaling features to the same level, we ensure the model interprets the weights of each feature equally. Simple imputer was used to fill in the missing values, while SMOTE was used to balance the minority class.

Several models were trained, and two were selected: logistic regression and random forest. Logistic regression serves as a baseline model for the more complex random forest. The logistic model is simple and easy to to interpret, however, it assumes a linear relationship between the features and the target. Random forest is a more powerful model that can handle a large number of inputs, without suffering from overfitting, and the model is also less prone to overfitting with outliers and noisy data. However, random forest can still overfit noisy data, when datasets contain a large number of irrelevant features. Furthermore, random forest models are difficult to interpret, as they are comprised of several decision trees.

When it comes to selecting between the two models, our main determinant was model performance. If our decision was not based on performance, but on interpretability, we would choose logistic regression. However, Based on model performance on the validation set, I expect random forest to perform better on the test set. In addition, the models in the pipeline that did not assume linearity performed better.

AUC, or area under the curve, calculates the true positive rate against the false positive rate, where 1 represents a perfect model, and 0 is the worst model. As the AUC of a model is more often lower on the test set than on the validation set, we assume the random forest model will perform significantly better than logistic regression on the test set, as it has a high AUC on the validation set. In addition, the random forest made more correct predictions in both the positive and negative classes, as evident by the confusion matrix of the validation set. We estimate the AUC score of the logistic regression model to be between 0.60-0.80, while the AUC of the random forest model may be between 0.9-1.0. AUC was the appropriate metric to evaluate our models, as accuracy is not suitable for data with imbalanced classes. This was further illustrated by the dummy model. We found using SMOTE significantly increased our model performance among non-linear models, as SMOTE balanced the minority class in the data.

If we could not use a scoring metric to compare the two models, we can compare the predictions of the two models on the test set. We can compare the true positive and true negative values of both models, as the model with more correct predictions will perform better. We can also compare the false positive and false negative values of both models.

Overall, the appropriate model and scoring to implement depends on the the data and the business needs. Many factors can determine the appropriate machine learning algorithm to use, from limited resources and large datasets, to categorical values and model interpretability.