1 Vehicle Insurance Fraud Detection with Gradient Boosting

1.1 Business Problem

This project aims to build a model that can predict whether a vehicle insurance claim is fraud based on 30 different input features. Fraudulent claims are costly to both the company and the customer, as they can drive up premiums as the cost of doing business becomes higher. The ability to flag transactions as likely fraudulent allows for more targeted fraud detection. The challenge is to flag as many 'actual fraud' claims while not flagging too many non-fraudulent claims, since this increases the follow on work to investigate the claim.

1.2 Data Understanding

The data for this project came from Kaggle **[(Kaggle Dataset Link)](https://www.kaggle.com/shivamb/vehicle-claim-fraud-detection)** **(https://www.kaggle.com/shivamb/vehicle-claim-fraud-detection)** and originated with the company Oracle. It is somewhat old (from the 90s), but was one of the larger datasets available for insurance claim fraud detection. While there may be additional claim data available today, all of these categories are still applicable and should still be expected to provide reasonable predictions in present day.

This data has a significant class imbalance. Only 6% of the claims listed are labeled as fraudulent. This can be addressed using class weighting in the model or oversampling. Undersampling would be difficult for this data since there are so few fraudulent claims. The three oversampling methods that I tried are RandomOverSample, SMOTE, and ADASYN. Ultimately, class weighting performed the best in the Gradient Boosting models.

1.2.1 Features

There are 30 features in the data set, but ultimately only 19 of them are used in the model. These include things like whether the claimant or third party was at fault, day of week and month of claim, policy type, age of policy holder, and cost of vehicle. The data had many features that could be numerical but were grouped into ranges. Where possible I converted this back into numerical features. Categorical variables were one hot encoded for use with XGBoost whereas catboost handled the categorical variables when fitting the model.

1.3 Model

Two different gradient boosting models were trained to the data: XGBoost and CatBoost.

For XGBoost, class weighting, random over sampling, SMOTE, and ADASYN were all tested to deal with the class imbalance. Early stopping rounds were used to avoid overfitting the model.

For CatBoost, class weighting was used to deal with the class imbalance. The model was fit with a training pool and eval pool so that the iteration with the best validation score for the eval metric (I used logloss and tried AUC) was kept to avoid overfitting.

For both XGBoost and CatBoost I used GridSearchCV to tune hyperparamaters. I tested this using AUC, logloss, Recall, and f1 score as the eval metrics. Ultimately, I used AUC to find the best model.

After tuning hyperparameters for the models with all 30 features, I explored feature importances for the best model and tried the training it on data with less features, selected for their prediction value change. I found that the model with the 19 most important features by prediction value change performed slightly better than models with all 30 features after hyperparameter tuning.

1.3.1 Criteria for Grading the Model

Because we are trying to balance identifying as much fraud as possible while not flagging too many non-fraudulent claims, we will rely heavily on the balance between the False Negative Rate (FNR) and the False Positive Rate (FPR) to determine if the model is performing well. It is more important to identify fraud than have a higher accuracy, but this can't result in flagging everything.

Furthermore, if we relied on accuracy alone, just guessing 'not fraud' every time in such an imbalanced data set would result in a 94% accuracy score.

1.4 Data Exploration

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, GridSearchCV, Stra
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, recall_score, precision_sco
from sklearn.pipeline import Pipeline
from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN
from imblearn import pipeline
from sklearn.metrics import roc_auc_score, roc_curve, auc, confusion_ma
from sklearn.metrics import plot_confusion_matrix, classification_repor
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
import random
import xgboost as xgb
import seaborn as sns
import catboost
from catboost.utils import get_roc_curve, get_fpr_curve, get_fnr_curve,
import shap
%matplotlib inline
```

In [3]:

```
df=pd.read_csv('fraud_oracle.csv')
df.head()
```

Out[3]:

	Month	WeekOfMonth	DayOfWeek	Make	AccidentArea	DayOfWeekClaimed	MonthClaimed	We
0	Dec	5	Wednesday	Honda	Urban	Tuesday	Jan	
1	Jan	3	Wednesday	Honda	Urban	Monday	Jan	
2	Oct	5	Friday	Honda	Urban	Thursday	Nov	
3	Jun	2	Saturday	Toyota	Rural	Friday	Jul	
4	Jan	5	Monday	Honda	Urban	Tuesday	Feb	

5 rows × 33 columns

In [4]:
```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15420 entries, 0 to 15419
Data columns (total 33 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Month                 15420 non-null  object
 1   WeekOfMonth           15420 non-null  int64
 2   DayOfWeek             15420 non-null  object
 3   Make                  15420 non-null  object
 4   AccidentArea          15420 non-null  object
 5   DayOfWeekClaimed      15420 non-null  object
 6   MonthClaimed          15420 non-null  object
 7   WeekOfMonthClaimed    15420 non-null  int64
 8   Sex                   15420 non-null  object
 9   MaritalStatus         15420 non-null  object
 10  Age                   15420 non-null  int64
 11  Fault                 15420 non-null  object
 12  PolicyType            15420 non-null  object
 13  VehicleCategory       15420 non-null  object
 14  VehiclePrice          15420 non-null  object
 15  FraudFound_P          15420 non-null  int64
 16  PolicyNumber          15420 non-null  int64
 17  RepNumber             15420 non-null  int64
 18  Deductible            15420 non-null  int64
 19  DriverRating          15420 non-null  int64
 20  Days_Policy_Accident  15420 non-null  object
 21  Days_Policy_Claim     15420 non-null  object
 22  PastNumberOfClaims    15420 non-null  object
 23  AgeOfVehicle          15420 non-null  object
 24  AgeOfPolicyHolder     15420 non-null  object
 25  PoliceReportFiled     15420 non-null  object
 26  WitnessPresent        15420 non-null  object
 27  AgentType             15420 non-null  object
 28  NumberOfSuppliments   15420 non-null  object
 29  AddressChange_Claim   15420 non-null  object
 30  NumberOfCars          15420 non-null  object
 31  Year                  15420 non-null  int64
 32  BasePolicy            15420 non-null  object
dtypes: int64(9), object(24)
memory usage: 3.9+ MB
```

In [5]:
```
1 df['FraudFound_P'].value_counts(normalize=True)
```

Out[5]:
```
0    0.940143
1    0.059857
Name: FraudFound_P, dtype: float64
```

In [6]:
```
cols=df.columns

for col in cols:
    print(df[col].value_counts())
```

```
Jan    1411
May    1367
Mar    1360
Jun    1321
Oct    1305
Dec    1285
Apr    1280
Feb    1266
Jul    1257
Sep    1240
Nov    1201
Aug    1127
Name: Month, dtype: int64
3    3640
2    3558
4    3398
1    3187
5    1637
Name: WeekOfMonth, dtype: int64
```

In [7]:
```
df[df['DayOfWeekClaimed']=='0']
```

Out[7]:

	Month	WeekOfMonth	DayOfWeek	Make	AccidentArea	DayOfWeekClaimed	MonthClaimed
1516	Jul	2	Monday	Honda	Rural	0	0

1 rows × 33 columns

There is only one row with missing data. Drop this observation.

In [8]:
```
df=df[df['DayOfWeekClaimed']!='0']
```

Year, Policy number and Rep Number can be dropped from features since they are not predictors of Fraud. If a certain policy holder or rep is involved in vehicle fraud then it is possible that this could be used to identify fraud, but it will not help with general classification of claims.

In [9]:
```
df.drop(columns=['PolicyNumber','RepNumber','Year'],inplace=True)
```

In [10]:
```
#Look at rows where Age is 0
df[df['Age']==0]['AgeOfPolicyHolder'].value_counts()
```

Out[10]:
```
16 to 17    319
Name: AgeOfPolicyHolder, dtype: int64
```

All rows where Age is 0 show the age of policy holder '16 to 17'. Lets look if age of policy holder tends to correspond to Age.

In [11]:
```
df.groupby('AgeOfPolicyHolder')['Age'].mean()
```

Out[11]:
```
AgeOfPolicyHolder
16 to 17      0.000000
18 to 20     16.400000
21 to 25     18.814815
26 to 30     22.941272
31 to 35     30.548006
36 to 40     40.483304
41 to 50     50.423267
51 to 65     60.441092
over 65      72.783465
Name: Age, dtype: float64
```

The mean values don't perfectly correspond, which can be expected since the person involved in a claim is not always the policy holder. But, they do correspond to roughly the age range, so we will assume that the 16 to 17 category corresponds to an age of 16 so that we don't lose those rows of data.

In [12]:
```
df['Age']=df['Age'].apply(lambda x: 16 if x==0 else x)
```

In [13]:
```
df.groupby('AgeOfPolicyHolder')['Age'].mean()
```

Out[13]:
```
AgeOfPolicyHolder
16 to 17     16.000000
18 to 20     16.400000
21 to 25     18.814815
26 to 30     22.941272
31 to 35     30.548006
36 to 40     40.483304
41 to 50     50.423267
51 to 65     60.441092
over 65      72.783465
Name: Age, dtype: float64
```

1.4.1 Convert Feature Data Types

A number of the features are in object format when they could be numeric. This includes month, week, vehicle age, number of supplements. Some have difficult formats to deal with where one of the categories is "greater than" a certain value. We'll create identifier features for these while setting the values in the numeric column to -1.

In [14]:
```
# Correct Month and MonthClaimed columns
month_map={'Jan':1,'Feb':2,'Mar':3,'Apr':4,'May':5,'Jun':6,'Jul':7,
           'Aug':8,'Sep':9,'Oct':10,'Nov':11,'Dec':12}

df['Month']=df['Month'].map(month_map)
df['MonthClaimed']=df['MonthClaimed'].map(month_map)
```

In [15]:
```
#Correct DayOfWeek and DayOfWeekClaimed
day_map={'Sunday':1,'Monday':2,'Tuesday':3,'Wednesday':4,'Thursday':5,
        'Friday':6,'Saturday':7}

df['DayOfWeek']=df['DayOfWeek'].map(day_map)
df['DayOfWeekClaimed']=df['DayOfWeekClaimed'].map(day_map)
```

In [16]:
```
#replace vehicle price ranges with random prices in that range

def fix_vehicle_price(x):
    if x=='less than 20000':
        return random.randint(5000,19999)
    elif x=='more than 69000':
        return random.randint(70000,89000)
    else:
        lower=int(x[:5])
        upper=int(x[-5:])
        return random.randint(lower,upper)

df['VehiclePrice']=df['VehiclePrice'].apply(fix_vehicle_price)
```

In [17]:
```
#Examine Days_Policy_Accident and Days_Policy_Claim
print(df['Days_Policy_Accident'].value_counts())
print(df['Days_Policy_Claim'].value_counts())
```

```
more than 30    15246
none               55
8 to 15            55
15 to 30           49
1 to 7             14
Name: Days_Policy_Accident, dtype: int64
more than 30    15342
15 to 30           56
8 to 15            21
Name: Days_Policy_Claim, dtype: int64
```

Because only ~1% of all claims in the data set had accidents/claims less than 30 after buying policy, we will create new columns denoting 'accident less than 30 day policy' and 'claim less than 30 day policy' and delete the original columns

In [18]:
```
df['Less30DaysPolicyAccident']=df['Days_Policy_Accident'].apply(
    lambda x: 0 if x=='more than 30' else 1
)
df['Less30DaysPolicyClaim']=df['Days_Policy_Claim'].apply(
    lambda x: 0 if x=='more than 30' else 1
)
df.drop(columns=['Days_Policy_Accident','Days_Policy_Claim'],inplace=Tr
```

In [19]:
```
#Examine AgeOfVehicle
df['AgeOfVehicle'].value_counts()
```

Out[19]:
```
7 years        5807
more than 7    3981
6 years        3448
5 years        1357
new             372
4 years         229
3 years         152
2 years          73
Name: AgeOfVehicle, dtype: int64
```

Convert these values to numbers. New will be 1. More than 7, set to 10 and have identifying column.

In [20]:
```
vehicle_age_map={'new': 1,
                 '2 years': 2,
                 '3 years': 3,
                 '4 years': 4,
                 '5 years': 5,
                 '6 years': 6,
                 '7 years': 7,
                 'more than 7': 10}  #set to -1
df['AgeOfVehicle']=df['AgeOfVehicle'].map(vehicle_age_map)
df['VehicleAgeOver7']=df['AgeOfVehicle'].apply(
    lambda x: 1 if x==10 else 0
)
df['VehicleAgeOver7'].value_counts()
```

Out[20]:
```
0    11438
1     3981
Name: VehicleAgeOver7, dtype: int64
```

In [21]:
```
#Examine AgeOfPolicyHolder
df['AgeOfPolicyHolder'].value_counts()
```

Out[21]:
```
31 to 35    5593
36 to 40    4043
41 to 50    2828
51 to 65    1392
26 to 30     613
over 65      508
16 to 17     319
21 to 25     108
18 to 20      15
Name: AgeOfPolicyHolder, dtype: int64
```

In [22]:
```
random.seed(42)
```

In [23]:
```
#function that returns random age from age range

def fix_age(x):
    if x=='over 65':
        return random.randint(66,78)  #78 is avg life expectancy in us
    else:
        lower=int(x[:2])
        upper=int(x[-2:])
        return random.randint(lower,upper)
```

In [24]:
```
df['AgeOfPolicyHolder']=df['AgeOfPolicyHolder'].apply(fix_age)
```

In [25]:
```
#Examine NumberOfCars
df['NumberOfCars'].value_counts()
```

Out[25]:
```
1 vehicle      14315
2 vehicles       709
3 to 4           372
5 to 8            21
more than 8        2
Name: NumberOfCars, dtype: int64
```

In [26]:
```
#generate random numbers for categories with range.
#Only two rows more than 8-enter 9 for these

def fix_num_cars(x):
    if x=='more than 8':
        return 9
    elif x=='3 to 4':
        return random.randint(3,4)
    elif x=='5 to 8':
        return random.randint(5,8)
    else:
        return int(x[0])

df['NumberOfCars']=df['NumberOfCars'].apply(fix_num_cars)
```

In [27]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15419 entries, 0 to 15419
Data columns (total 31 columns):
 #   Column                     Non-Null Count  Dtype 
---  ------                     --------------  ----- 
 0   Month                      15419 non-null  int64 
 1   WeekOfMonth                15419 non-null  int64 
 2   DayOfWeek                  15419 non-null  int64 
 3   Make                       15419 non-null  object
 4   AccidentArea               15419 non-null  object
 5   DayOfWeekClaimed           15419 non-null  int64 
 6   MonthClaimed               15419 non-null  int64 
 7   WeekOfMonthClaimed         15419 non-null  int64 
 8   Sex                        15419 non-null  object
 9   MaritalStatus              15419 non-null  object
 10  Age                        15419 non-null  int64 
 11  Fault                      15419 non-null  object
 12  PolicyType                 15419 non-null  object
 13  VehicleCategory            15419 non-null  object
 14  VehiclePrice               15419 non-null  int64 
 15  FraudFound_P               15419 non-null  int64 
 16  Deductible                 15419 non-null  int64 
 17  DriverRating               15419 non-null  int64 
 18  PastNumberOfClaims         15419 non-null  object
 19  AgeOfVehicle               15419 non-null  int64 
 20  AgeOfPolicyHolder          15419 non-null  int64 
 21  PoliceReportFiled          15419 non-null  object
 22  WitnessPresent             15419 non-null  object
 23  AgentType                  15419 non-null  object
 24  NumberOfSuppliments        15419 non-null  object
 25  AddressChange_Claim        15419 non-null  object
 26  NumberOfCars               15419 non-null  int64 
 27  BasePolicy                 15419 non-null  object
 28  Less30DaysPolicyAccident   15419 non-null  int64 
 29  Less30DaysPolicyClaim      15419 non-null  int64 
 30  VehicleAgeOver7            15419 non-null  int64 
dtypes: int64(17), object(14)
memory usage: 3.8+ MB
```

1.4.2 Data Preprocessing

In [28]:
```
# Separate target variable and features
y=df['FraudFound_P']
X=df.drop(columns='FraudFound_P')
```

In [29]:
```
#One Hot Encode Categorical Variables
X_processed=pd.get_dummies(X)
X_processed.head()
```

Out[29]:

	Month	WeekOfMonth	DayOfWeek	DayOfWeekClaimed	MonthClaimed	WeekOfMonthClaimed	Ag
0	12	5	4	3	1	1	2
1	1	3	4	2	1	4	3
2	10	5	6	5	11	2	4
3	6	2	7	6	7	1	6
4	1	5	2	3	2	2	2

5 rows × 79 columns

In [30]:
```
#train test split
#stratify y to ensure fraud instances are in both test and train sets

X_train,X_test,y_train,y_test=train_test_split(X_processed,y,random_sta
                                                       stratify=y)
```

In [31]:
```
#Scale test data
scaler=StandardScaler()
X_train_scaled=scaler.fit_transform(X_train)
```

In [32]:
```
X_test_scaled=scaler.transform(X_test)
```

Before building any models, it is important to note what should be considered good performance based on the class imbalance in the data. Since only ~6% of the data are classified as fraud, if the model just picked 'not fraud' every time, it would have an accuracy of 96%. This means any model would need to beat this to be better than just guessing that its not fraud.

It will also be important to pay attention to recall, precision, FPR, and FNR to determine what types of errors the model tends to have.

1.4.3 Threshold Selection Considerations

The acceptable false negative and false positive rates will depend on the cost of each of these situations.

False negatives mean there was fraud that was not identified. False positives mean that a claim was flagged as fraud even though it is not fraudulent.

While we want to capture as much fraud as possible, since false claims cost the company money, the time/resources spent investigating flagged claims that turn out not to be fraudulent still costs money. Therefore, we may be willing to accept a certain level of undetected fraud in order to minimize false positives.

If available, I would also want to examine the distribution of cost of fraud - what is the likelihood that fraud we missed was of a high value vs. likelihood it didn't cost a lot.

For the purpose of this project I'm going to make an assumption that the insurance company can accept missing 10% of all fraud. This may be high, but we do not have the cost information to make an informed selection.

2 XGBoost

XGBoost uses a gradient boosting algorithm. It uses asymmetric decision trees in the model and has built in regularization.

We can address the class imbalance by adjusting the scale_pos_weight hyperparameter in XGBoost Classifier, but I also tried training the model with oversampled data. Overall, using class weighting with scale_pos_weight seems to work better, but I left the sections with oversampling for comparison.

We will also make use of early stopping available with XGBoost. This allows us to stop the model with a specific metric does not improve over a specified number of iterations. It then returns the model at the iteration with the best score. This helps to avoid overfitting.

2.1 Define Functions for Model Comparison

Since we are trying to capture as much fraud as possible while preventing false positives from being too high, we will want to look at FNR and FPR at various thresholds for the models we build. Just looking at accuracy, f1, recall, precision can't as clearly tell us the best performance that model could have if thresholds are shifted. I think its best to look at the possible solutions with a validation set (a subset of original training split- not the test set) in order to judge how it will ultimately perform. If we just go based on logloss or AUC, which are the metrics I use to find model solutions, we may not end up with the best performance since we care more about capturing fraud than overall accuracy. In fact the best model does not have the lowest logloss.

The functions below set up an output with all the desired calculations and threshold tuning graphs for easier comparison of models.

In [33]:

```
#Function to calculate FNR and FPR

def calc_fpr_fnr(y_true,y_predicted):
    """
    Function takes in y_true and y_predicted and returns fpr and fnr
    """

    cm=confusion_matrix(y_true,y_predicted)
    not_fraud=cm[0,:].sum()
    fraud=cm[1,:].sum()
    fnr=cm[1,0]/fraud
    fpr=cm[0,1]/not_fraud

    return fpr, fnr
```

In [34]:

```
#write a function to make predictions with adjusted threshold

def thresh_pred(probs,threshold):
    """
    Takes in probabilites predicted by model and threshold value
    Return array of classification predictions based on the threshold
    """
    predictions=np.where(probs>threshold,1,0)
    return predictions
```

In [35]:

```
def select_threshold_fnr(fnr,thresholds,target_fnr):
    """
    returns the threshold value for the target_fnr and idx value
    """
    #search for value >=target_fnr from end of fnr
    target_fnr=target_fnr
    idx=0
    fn=fnr[0]
    while fn >=target_fnr:
        idx+=1
        fn=fnr[idx]

    #now idx is for value to right of target_fnr
    left_fnr=fnr[idx-1]
    right_fnr=fnr[idx]
    left_threshold=thresholds[idx-1]
    right_threshold=thresholds[idx]

    #find threshold corresponding to target_fnr by linear approximation
    ratio=(left_fnr-target_fnr)/(left_fnr-right_fnr)
    target_thresh=left_threshold-(ratio*(left_threshold-right_threshold
    if (left_fnr-target_fnr)<=(target_fnr-right_fnr):
        best_idx=idx-1
    else:
        best_idx=idx
    return target_thresh, best_idx
```

In [36]:
```python
#Function to display all desired metrics and threshold comparisons

def display_metrics(X_test,y_test,y_hat,y_prob):
    """
    Takes in X,y,y_hat,y_prob and displays metrics
    y_prob is array of only probabilities that observation is 1
    """

    #Print Metrics
    fpr,tpr,thresholds=roc_curve(y_test,y_prob)
    print('AUC: {},  logloss: {}'.format(auc(fpr,tpr),log_loss(y_test,y
    print('accuracy: {}'.format(round(accuracy_score(y_test,y_hat),4)))
    print('recall: {}'.format(round(recall_score(y_test,y_hat),4)))
    print('precision: {}'.format(round(precision_score(y_test,y_hat),4)
    print('-------------------')
    print('For a .5 threshold:')
    print(confusion_matrix(y_test,y_hat))
    fpr_5,fnr_5=calc_fpr_fnr(y_test,y_hat)
    print('FPR: {}  FNR: {}'.format(round(fpr_5,4),round(fnr_5,4)))

    #Calculate gmean and corresponding threshold
    gmeans=np.sqrt(tpr*(1-fpr))
    ix=np.argmax(gmeans)
    g_thresh=thresholds[ix]

    #Calculate threshold corresponding to 10% FNR
    fnr=1-tpr
    thresh_10, idx = select_threshold_fnr(fnr, thresholds, .1)

    fig,axes = plt.subplots(1,2,figsize=(15,4))
    ax1=axes[0]
    ax2=axes[1]

    #Graph FPR/FNR vs Threshold
    style = {'alpha':0.5, 'lw':2}

    ax1.plot(thresholds, fpr, color='blue', label='FPR', **style)
    ax1.plot(thresholds, fnr, color='green', label='FNR', **style)
    ax1.plot([g_thresh,g_thresh],[0,1],color='red', label='g-mean thres
             **style)
    ax1.plot([thresh_10,thresh_10],[0,1],color='orange',
             label='10% fnr thresh',
             **style)
    ax1.set_xlim([0.0,1.0])
    ax1.set_ylim([0.0,1.05])
    #ax1.set_xticks(fontsize=12)
    #ax1.set_yticks(fontsize=12)
    ax1.grid(True)
    ax1.set_xlabel('Threshold', fontsize=10)
    ax1.set_ylabel('Error Rate', fontsize=10)
    ax1.set_title('FPR-FNR curves', fontsize=12)
    ax1.legend(loc='lower left', fontsize=10)

    #Graph ROC Curve with threshold
    # plot the roc curve for the model
    ax2.plot([0,1], [0,1], linestyle='--', label='No Skill')
```

```
    ax2.plot(fpr, tpr, label='XBG ADASYN')
    ax2.scatter(fpr[ix], tpr[ix], marker='o', color='blue', label='gmea
    ax2.scatter(fpr[idx], tpr[idx], marker='o', color='red', label='10%
    # axis labels
    ax2.set_xlabel('False Positive Rate')
    ax2.set_ylabel('True Positive Rate')
    ax2.grid(True)
    ax2.set_title('ROC Curve with Thresholds', fontsize=12)
    ax2.legend()

    #print outcomes with alternative thresholds
    print('---------------------')
    print('G-Mean Threshold:')
    print('Best Threshold=%f, G-Mean=%.3f' % (g_thresh, gmeans[ix]))
    y_hat_g=thresh_pred(y_prob,g_thresh)
    print(confusion_matrix(y_test,y_hat_g))
    g_fpr,g_fnr=calc_fpr_fnr(y_test,y_hat_g)
    print('FPR: {}  FNR: {}'.format(round(g_fpr,4),round(g_fnr,4)))
    print('---------------------')
    print('10% FNR Threshold:')
    print('Best Threshold: {}'.format(round(thresh_10,4)))
    y_hat_10=thresh_pred(y_prob,thresh_10)
    print(confusion_matrix(y_test,y_hat_10))
    ten_fpr,ten_fnr=calc_fpr_fnr(y_test,y_hat_10)
    print('FPR: {}  FNR: {}'.format(round(ten_fpr,4),round(ten_fnr,4)))
```

2.2 XGBoost with Weighted Class

In [37]:

```
#split training data into train/validation sets
xg_trainX, xg_testX, xg_trainy, xg_testy = train_test_split(X_train, y_
                                                            random_state
                                                            stratify=y_t
train_set=[(xg_trainX, xg_trainy)]
eval_set=[(xg_testX, xg_testy)]
```

In [84]:

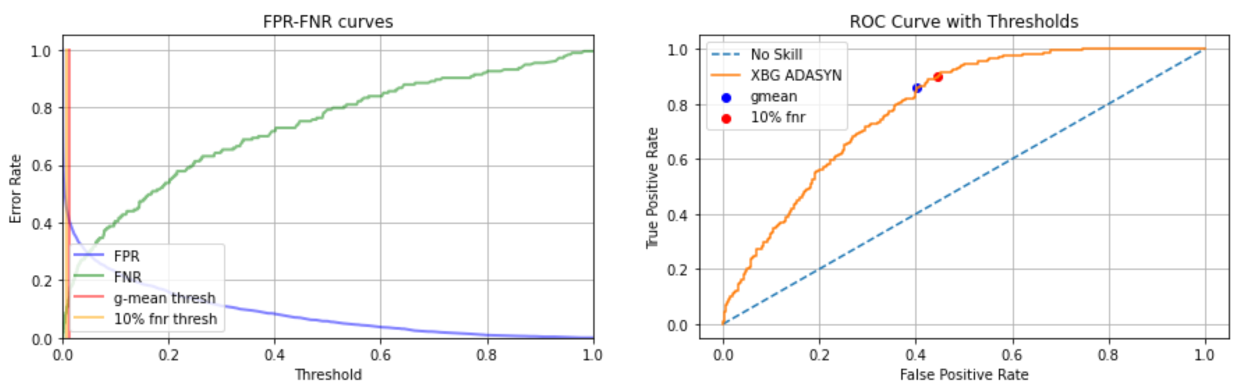
```
xgb_clf=xgb.XGBClassifier(objective='binary:logistic',
                          random_state=123,
                         n_estimators=5000, scale_pos_weight=20, learning_rate=

xgb_clf.fit(
    xg_trainX, xg_trainy,
    early_stopping_rounds=50,
    eval_metric='logloss',
    eval_set=eval_set,
    verbose=True
)
```

```
[0]	validation_0-logloss:0.65952
Will train until validation_0-logloss hasn't improved in 50 rounds.
[1]	validation_0-logloss:0.63375
[2]	validation_0-logloss:0.61414
[3]	validation_0-logloss:0.59776
[4]	validation_0-logloss:0.58392
[5]	validation_0-logloss:0.57302
[6]	validation_0-logloss:0.56530
[7]	validation_0-logloss:0.55839
[8]	validation_0-logloss:0.55375
[9]	validation_0-logloss:0.54847
[10]	validation_0-logloss:0.54520
[11]	validation_0-logloss:0.54194
[12]	validation_0-logloss:0.54010
[13]	validation_0-logloss:0.53659
[14]	validation_0-logloss:0.53504
[15]	validation_0-logloss:0.53374
[16]	validation_0-logloss:0.53280
[17]	validation_0-logloss:0.53046
```

In [140]:

```
y_hat=xgb_clf.predict(xg_testX)
y_prob=xgb_clf.predict_proba(xg_testX)[:,1]

display_metrics(xg_testX,xg_testy,y_hat,y_prob)
```

```
AUC: 0.7884473877851361,  logloss: 0.2605784111173485
accuracy: 0.8987
recall: 0.2081
precision: 0.1875
-------------------
For a .5 threshold:
[[2562  156]
 [ 137   36]]
FPR: 0.0574  FNR: 0.7919
---------------------
G-Mean Threshold:
Best Threshold=0.013045, G-Mean=0.717
[[1624 1094]
 [  25  148]]
FPR: 0.4025  FNR: 0.1445
---------------------
10% FNR Threshold:
Best Threshold: 0.0076
[[1506 1212]
 [  18  155]]
FPR: 0.4459  FNR: 0.104
```



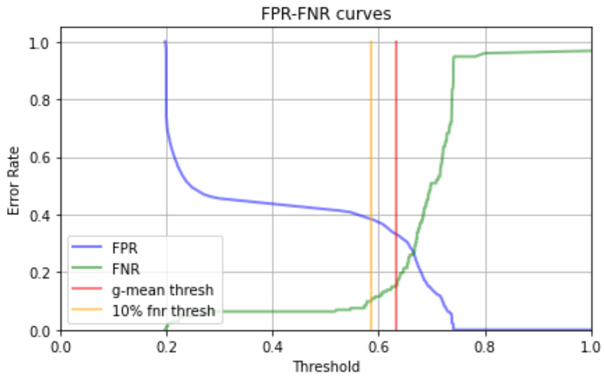
2.2.1 GridSearch with early stopping

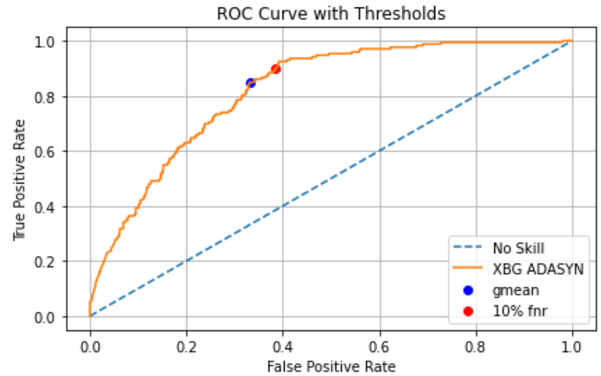
In [144]:

```python
xgb_model=xgb.XGBClassifier(
    objective='binary:logistic',
    random_state=123,
    n_estimators=5000
)

param_grid= {
    'max_depth': [6,10],
    'learning_rate': [.01],
    'gamma': [0,1],
    'reg_lambda': [0,1],
    'scale_pos_weight': [25,35],
    'subsample': [.8,1],
    'colsample_bytree': [1]
}

fit_params = {
    'early_stopping_rounds': 50,
    'eval_metric': 'logloss',
    'eval_set': eval_set,
    'verbose': False
}

gs_xgb = GridSearchCV(xgb_model,param_grid,cv=3,scoring='roc_auc',
                     verbose=0)

gs_xgb.fit(xg_trainX, xg_trainy,**fit_params)

print('best score: {}'.format(gs_xgb.best_score_))
print('best params: {}'.format(gs_xgb.best_params_))
y_hat_test=gs_xgb.predict(xg_testX)
y_hat_prob=gs_xgb.predict_proba(xg_testX)[:,1]
display_metrics(xg_testX,xg_testy,y_hat_test,y_hat_prob)
```

```
best score: 0.8236625593169634
best params: {'colsample_bytree': 1, 'gamma': 1, 'learning_rate': 0.01,
'max_depth': 6, 'reg_lambda': 1, 'scale_pos_weight': 35, 'subsample': 0.
8}
AUC: 0.8196299982561132,  logloss: 0.6089951331868628
accuracy: 0.6036
recall: 0.9364
precision: 0.1249
-------------------
For a .5 threshold:
[[1583 1135]
 [  11  162]]
FPR: 0.4176  FNR: 0.0636
---------------------
G-Mean Threshold:
Best Threshold=0.633048, G-Mean=0.754
[[1817  901]
 [  27  146]]
FPR: 0.3315  FNR: 0.1561
---------------------
10% FNR Threshold:
Best Threshold: 0.5858
```

```
[[1674 1044]
 [  18  155]]
FPR: 0.3841  FNR: 0.104
```



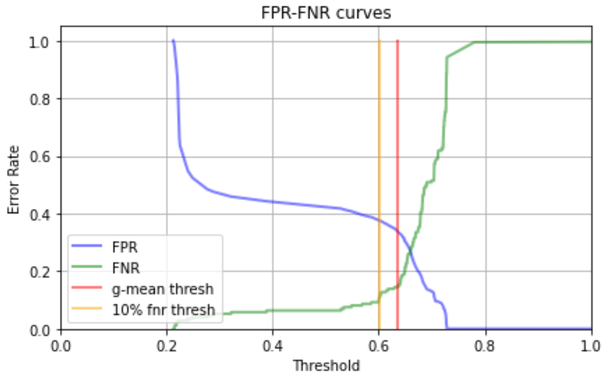


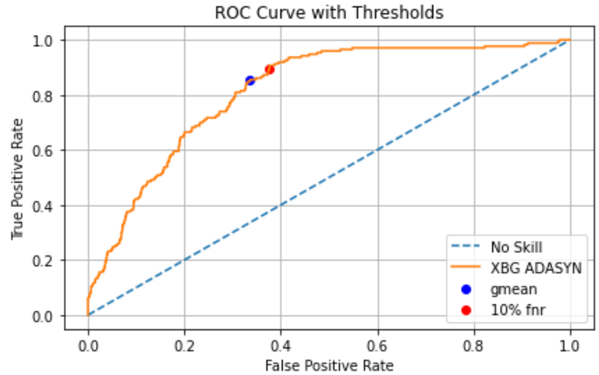
In [145]:

```python
#Refine gridsearch

xgb_model=xgb.XGBClassifier(
    objective='binary:logistic',
    random_state=123,
    n_estimators=5000
)

param_grid= {
    'max_depth': [6,8],
    'learning_rate': [.01,.001],
    'gamma': [1],
    'reg_lambda': [1],
    'scale_pos_weight': [30,32,35],
    'subsample': [1],
    'colsample_bytree': [.8]
}

fit_params = {
    'early_stopping_rounds': 50,
    'eval_metric': 'logloss',
    'eval_set': eval_set,
    'verbose': False
}

gs_xgb1 = GridSearchCV(xgb_model,param_grid,cv=3,scoring='roc_auc',
                   verbose=0)

gs_xgb1.fit(xg_trainX, xg_trainy,**fit_params)

print('best score: {}'.format(gs_xgb1.best_score_))
print('best params: {}'.format(gs_xgb1.best_params_))
y_hat_test=gs_xgb1.predict(xg_testX)
y_hat_prob=gs_xgb1.predict_proba(xg_testX)[:,1]
display_metrics(xg_testX,xg_testy,y_hat_test,y_hat_prob)
```

```
best score: 0.8222553986057412
best params: {'colsample_bytree': 0.8, 'gamma': 1, 'learning_rate': 0.00
1, 'max_depth': 6, 'reg_lambda': 1, 'scale_pos_weight': 35, 'subsample':
1}
AUC: 0.8154372264543379,  logloss: 0.6176079186971358
accuracy: 0.6012
recall: 0.9364
precision: 0.1242
-------------------
For a .5 threshold:
[[1576 1142]
 [  11  162]]
FPR: 0.4202  FNR: 0.0636
---------------------
G-Mean Threshold:
Best Threshold=0.636508, G-Mean=0.753
[[1803  915]
 [  26  147]]
FPR: 0.3366  FNR: 0.1503
---------------------
10% FNR Threshold:
```

```
Best Threshold: 0.6002
[[1694 1024]
 [  18  155]]
FPR: 0.3767  FNR: 0.104
```





These parameters performed slightly better, will a lower FPR with similar FNR. Will still try to refine learning rate, scale_pos_weight, and tree depth.

In [150]:

```python
#Refine gridsearch- try neg_log_loss scoring

xgb_model=xgb.XGBClassifier(
    objective='binary:logistic',
    random_state=123,
    n_estimators=5000
)

param_grid= {
    'max_depth': [6,8],
    'learning_rate': [.01,.001],
    'gamma': [1],
    'reg_lambda': [1],
    'scale_pos_weight': [30,32,35],
    'subsample': [1],
    'colsample_bytree': [.8]
}

fit_params = {
    'early_stopping_rounds': 50,
    'eval_metric': 'logloss',
    'eval_set': eval_set,
    'verbose': False
}

gs_xgb1_5 = GridSearchCV(xgb_model,param_grid,cv=3,scoring='neg_log_los
                    verbose=0)

gs_xgb1_5.fit(xg_trainX, xg_trainy,**fit_params)

print('best score: {}'.format(gs_xgb1_5.best_score_))
print('best params: {}'.format(gs_xgb1_5.best_params_))
y_hat_test=gs_xgb1_5.predict(xg_testX)
y_hat_prob=gs_xgb1_5.predict_proba(xg_testX)[:,1]
display_metrics(xg_testX,xg_testy,y_hat_test,y_hat_prob)
```

```
best score: -0.23787891042901207
best params: {'colsample_bytree': 0.8, 'gamma': 1, 'learning_rate': 0.01,
'max_depth': 8, 'reg_lambda': 1, 'scale_pos_weight': 30, 'subsample': 1}
AUC: 0.7983556423245586,  logloss: 0.25409462682151174
accuracy: 0.9
recall: 0.2543
precision: 0.2157
-------------------
For a .5 threshold:
[[2558  160]
 [ 129   44]]
FPR: 0.0589  FNR: 0.7457
---------------------
G-Mean Threshold:
Best Threshold=0.011708, G-Mean=0.734
[[1612 1106]
 [  17  156]]
FPR: 0.4069  FNR: 0.0983
---------------------
```

AUC appears to result in better FNRs and pick models more suited the identifying fraud.

In [146]:

```
#Refine gridsearch- Try smaller learning rate

xgb_model=xgb.XGBClassifier(
    objective='binary:logistic',
    random_state=123,
    n_estimators=5000
)

param_grid= {
    'max_depth': [5,6],
    'learning_rate': [.0001,.001],
    'gamma': [1],
    'reg_lambda': [1],
    'scale_pos_weight': [35,37],
    'subsample': [1],
    'colsample_bytree': [.8]
}

fit_params = {
    'early_stopping_rounds': 50,
    'eval_metric': 'logloss',
    'eval_set': eval_set,
    'verbose': False
}

gs_xgb2 = GridSearchCV(xgb_model,param_grid,cv=3,scoring='roc_auc',
                   verbose=0)

gs_xgb2.fit(xg_trainX, xg_trainy,**fit_params)

print('best score: {}'.format(gs_xgb1.best_score_))
print('best params: {}'.format(gs_xgb1.best_params_))
y_hat_test=gs_xgb2.predict(xg_testX)
y_hat_prob=gs_xgb2.predict_proba(xg_testX)[:,1]
display_metrics(xg_testX,xg_testy,y_hat_test,y_hat_prob)
```

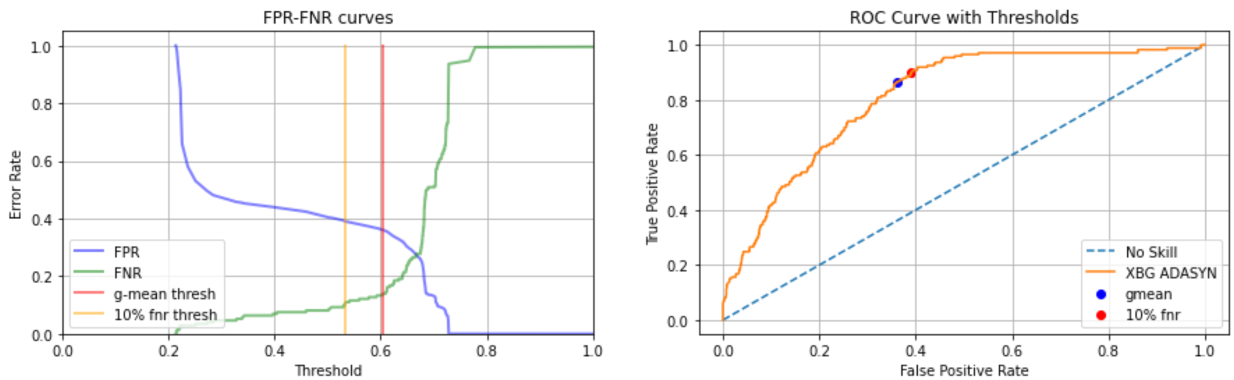
```
best score: 0.8222553986057412
best params: {'colsample_bytree': 0.8, 'gamma': 1, 'learning_rate': 0.00
1, 'max_depth': 6, 'reg_lambda': 1, 'scale_pos_weight': 35, 'subsample':
1}
AUC: 0.8222681587532484,  logloss: 0.6377699272581467
accuracy: 0.5908
recall: 0.9364
precision: 0.1214
-------------------
For a .5 threshold:
[[1546 1172]
 [  11  162]]
FPR: 0.4312  FNR: 0.0636
---------------------
G-Mean Threshold:
Best Threshold=0.610704, G-Mean=0.761
[[1841  877]
 [  26  147]]
FPR: 0.3227  FNR: 0.1503
```

A smaller learning rate with smaller trees performed slightly better in terms of FPR with FNR set to 10%. This is our best model so far.

In [39]:
```
#Refine gridsearch- experiment with regularization

xgb_model=xgb.XGBClassifier(
    objective='binary:logistic',
    random_state=123,
    n_estimators=5000
)

param_grid= {
    'max_depth': [6],
    'learning_rate': [.001],
    'gamma': [3,1],
    'reg_lambda': [3,1],
    'scale_pos_weight': [25,35],
    'subsample': [1],
    'colsample_bytree': [.8]
}

fit_params = {
    'early_stopping_rounds': 50,
    'eval_metric': 'logloss',
    'eval_set': eval_set,
    'verbose': False
}

gs_xgb2 = GridSearchCV(xgb_model,param_grid,cv=3,scoring='roc_auc',
                    verbose=0)

gs_xgb2.fit(xg_trainX, xg_trainy,**fit_params)

print('best score: {}'.format(gs_xgb2.best_score_))
print('best params: {}'.format(gs_xgb2.best_params_))
y_hat_test=gs_xgb2.predict(xg_testX)
y_hat_prob=gs_xgb2.predict_proba(xg_testX)[:,1]
display_metrics(xg_testX,xg_testy,y_hat_test,y_hat_prob)
```

```
best score: 0.8249382196191521
best params: {'colsample_bytree': 0.8, 'gamma': 3, 'learning_rate': 0.00
1, 'max_depth': 6, 'reg_lambda': 3, 'scale_pos_weight': 35, 'subsample':
1}
AUC: 0.8104054749539571,  logloss: 0.6157678469927902
accuracy: 0.615
recall: 0.9191
precision: 0.1264
-------------------
For a .5 threshold:
[[1619 1099]
 [  14  159]]
FPR: 0.4043  FNR: 0.0809
---------------------
G-Mean Threshold:
Best Threshold=0.602761, G-Mean=0.744
[[1737  981]
 [  24  149]]
FPR: 0.3609  FNR: 0.1387
---------------------
10% FNR Threshold:
```

```
Best Threshold: 0.532
[[1656 1062]
 [  18  155]]
FPR: 0.3907  FNR: 0.104
```



After refining our gridsearch, the best XGBoost model has a FPR of ~37% with FNR ~10% at a threshold of .6

2.3 XGBoost with Oversampling

2.3.1 Gridsearch with scaled data [...]

2.3.2 XGBoost with RandomOverSample [...]

2.3.3 XGBoost with SMOTE [...]

2.3.4 XGBoost with ADASYN [...]

2.4 Best XGBoost Model

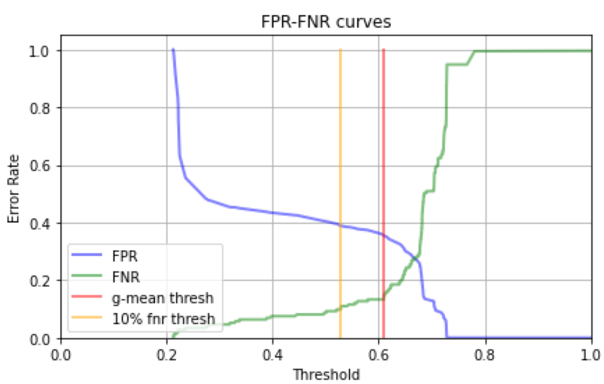
In [40]:

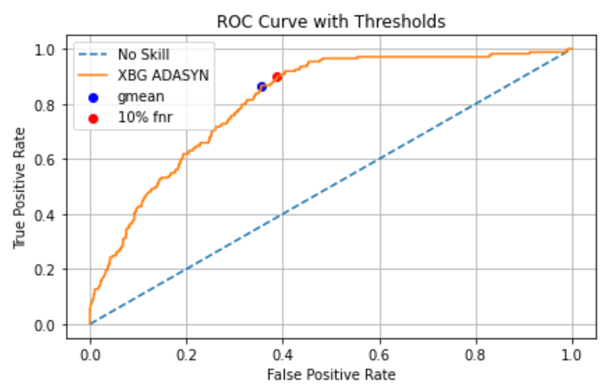
```
xgb_best=xgb.XGBClassifier(
    objective='binary:logistic',
    random_state=123,
    n_estimators=5000,
    scale_pos_weight=35,
    learning_rate=.001,
    colsample_bytree=.8,
    max_depth=6,
    reg_lambda=1,
    gamma=1,
    subsample=1
)

xgb_best.fit(
    xg_trainX, xg_trainy,
    early_stopping_rounds=50,
    eval_metric='logloss',
    eval_set=eval_set,
    verbose=True
)


```

```
[0]	validation_0-logloss:0.69284
Will train until validation_0-logloss hasn't improved in 50 rounds.
[1]	validation_0-logloss:0.69256
[2]	validation_0-logloss:0.69226
[3]	validation_0-logloss:0.69223
[4]	validation_0-logloss:0.69193
[5]	validation_0-logloss:0.69163
[6]	validation_0-logloss:0.69136
[7]	validation_0-logloss:0.69107
[8]	validation_0-logloss:0.69104
[9]	validation_0-logloss:0.69075
[10]	validation_0-logloss:0.69046
[11]	validation_0-logloss:0.69017
[12]	validation_0-logloss:0.68988
[13]	validation_0-logloss:0.68959
[14]	validation_0-logloss:0.68931
[15]	validation_0-logloss:0.68905
[16]	validation_0-logloss:0.68877
[17]	validation_0-logloss:0.68849
```

In [43]:
```
y_hat_test=xgb_best.predict(xg_testX)
y_hat_prob=xgb_best.predict_proba(xg_testX)[:,1]
display_metrics(xg_testX,xg_testy,y_hat_test,y_hat_prob)
```

```
AUC: 0.8112327578506808,  logloss: 0.612395459869171
accuracy: 0.6154
recall: 0.9133
precision: 0.1259
-------------------
For a .5 threshold:
[[1621 1097]
 [  15  158]]
FPR: 0.4036  FNR: 0.0867
---------------------
G-Mean Threshold:
Best Threshold=0.609760, G-Mean=0.748
[[1753  965]
 [  24  149]]
FPR: 0.355  FNR: 0.1387
---------------------
10% FNR Threshold:
Best Threshold: 0.5285
[[1662 1056]
 [  18  155]]
FPR: 0.3885  FNR: 0.104
```





In [47]:
```
#Best threshold is .5285; predict with test data
best_thresh=.5285

y_hat_prob=xgb_best.predict_proba(X_test)[:,1]
y_hat_test=thresh_pred(y_hat_prob,best_thresh)
fpr,tpr,thresholds=roc_curve(y_test,y_hat_prob)
print('AUC: {},  logloss: {}'.format(auc(fpr,tpr),log_loss(y_test,y_hat
print('accuracy: {}'.format(round(accuracy_score(y_test,y_hat_test),4))
print('recall: {}'.format(round(recall_score(y_test,y_hat_test),4)))
print('precision: {}'.format(round(precision_score(y_test,y_hat_test),4
print('--------------------')
print(confusion_matrix(y_test,y_hat_test))
fpr,fnr=calc_fpr_fnr(y_test,y_hat_test)
print('FPR: {}  FNR: {}'.format(round(fpr,4),round(fnr,4)))
```

```
AUC: 0.8045449767304071,  logloss: 0.6040263190459648
accuracy: 0.6329
recall: 0.8831
precision: 0.1281
-------------------
[[2236 1388]
 [  27  204]]
FPR: 0.383  FNR: 0.1169
```

3 CATBoost

3.1 Define Functions for Model Comparison

In [36]:

```
#Function to display all desired metrics and threshold comparisons

def display_metrics_cat(model,val_pool,y_test):
    """
    Takes in model, val_pool, and test set from val_pool and displays m

    """

    #get predictions
    y_hat=model.predict(val_pool)
    y_prob=model.predict_proba(val_pool)[:,1]

    #Print Metrics
    #curve = get_roc_curve(cat_model,val_pool)
    fpr,tpr,thresholds=roc_curve(y_test,y_prob)
    fnr=1-tpr
    #thresholds, fpr = get_fpr_curve(curve = curve)
    #thresholds, fnr = get_fnr_curve(curve = curve)
    print('AUC: {},  logloss: {}'.format(auc(fpr,tpr),log_loss(y_test,y
    print('accuracy: {}'.format(round(accuracy_score(y_test,y_hat),4)))
    print('recall: {}'.format(round(recall_score(y_test,y_hat),4)))
    print('precision: {}'.format(round(precision_score(y_test,y_hat),4)
    print('-------------------')
    print('For a .5 threshold:')
    print(confusion_matrix(y_test,y_hat))
    fpr_5,fnr_5=calc_fpr_fnr(y_test,y_hat)
    print('FPR: {}  FNR: {}'.format(round(fpr_5,4),round(fnr_5,4)))

    #Calculate gmean and corresponding threshold
    gmeans=np.sqrt(tpr*(1-fpr))
    ix=np.argmax(gmeans)
    g_thresh=thresholds[ix]

    #Calculate threshold corresponding to 10% FNR
    thresh_10, idx = select_threshold_fnr(fnr, thresholds, .1)
    thresh_5, idx5 = select_threshold_fnr(fnr, thresholds, .05)
    #thresh10_cat=select_threshold(model,val_pool,FNR=.1)

    fig,axes = plt.subplots(1,2,figsize=(15,4))
    ax1=axes[0]
    ax2=axes[1]

    #Graph FPR/FNR vs Threshold
    style = {'alpha':0.5, 'lw':2}

    ax1.plot(thresholds, fpr, color='blue', label='FPR', **style)
    ax1.plot(thresholds, fnr, color='green', label='FNR', **style)
    ax1.plot([g_thresh,g_thresh],[0,1],color='red', label='g-mean thres
             **style)
    ax1.plot([thresh_10,thresh_10],[0,1],color='orange',
             label='10% fnr thresh',
             **style)
    ax1.set_xlim([0.0,1.0])
    ax1.set_ylim([0.0,1.05])
    #ax1.set_xticks(fontsize=12)
    #ax1.set_yticks(fontsize=12)
```

```
    ax1.grid(True)
    ax1.set_xlabel('Threshold', fontsize=10)
    ax1.set_ylabel('Error Rate', fontsize=10)
    ax1.set_title('FPR-FNR curves', fontsize=12)
    ax1.legend(loc='lower left', fontsize=10)

    #Graph ROC Curve with threshold
    # plot the roc curve for the model
    ax2.plot([0,1], [0,1], linestyle='--', label='No Skill')
    ax2.plot(fpr, tpr, label='XBG ADASYN')
    ax2.scatter(fpr[ix], tpr[ix], marker='o', color='blue', label='gmea
    ax2.scatter(fpr[idx], tpr[idx], marker='o', color='red', label='10%
    # axis labels
    ax2.set_xlabel('False Positive Rate')
    ax2.set_ylabel('True Positive Rate')
    ax2.grid(True)
    ax2.set_title('ROC Curve with Thresholds', fontsize=12)
    ax2.legend()

    #print outcomes with alternative thresholds
    print('---------------------')
    print('G-Mean Threshold:')
    print('Best Threshold=%f, G-Mean=%.3f' % (g_thresh, gmeans[ix]))
    y_hat_g=thresh_pred(y_prob,g_thresh)
    print(confusion_matrix(y_test,y_hat_g))
    g_fpr,g_fnr=calc_fpr_fnr(y_test,y_hat_g)
    print('FPR: {}  FNR: {}'.format(round(g_fpr,4),round(g_fnr,4)))
    print('---------------------')
    print('10% FNR Threshold:')
    print('Best Threshold: {}'.format(round(thresh_10,4)))
    y_hat_10=thresh_pred(y_prob,thresh_10)
    print(confusion_matrix(y_test,y_hat_10))
    ten_fpr,ten_fnr=calc_fpr_fnr(y_test,y_hat_10)
    print('FPR: {}  FNR: {}'.format(round(ten_fpr,4),round(ten_fnr,4)))
    print('---------------------')
    print('5% FNR Threshold:')
    print('Best Threshold: {}'.format(round(thresh_5,4)))
    y_hat_5=thresh_pred(y_prob,thresh_5)
    print(confusion_matrix(y_test,y_hat_5))
    five_fpr,five_fnr=calc_fpr_fnr(y_test,y_hat_5)
    print('FPR: {}  FNR: {}'.format(round(five_fpr,4),round(five_fnr,4)
```

In [37]:

```
#Catboost can handle categorical variables
#We'll use a diff version of X_train that hasn't been ohe'd
#X, y are before ohe

#get list of categorical features in X
dtypes=X.dtypes.reset_index()
cat_features=dtypes[dtypes[0]=='object']['index'].to_list()

#Split into test and train
X_train_cat, X_test_cat, y_train_cat, y_test_cat = train_test_split(
    X, y ,random_state=42, stratify=y
)

#Split training set into train and validation for catboost training

train_X, val_X, train_y, val_y = train_test_split(
    X_train_cat, y_train_cat, random_state=42, stratify=y_train_cat
)
```

In [38]:

```
#create train_pool and validation_pool

train_pool = catboost.Pool(
    data = train_X,
    label = train_y,
    cat_features = cat_features
)

val_pool = catboost.Pool(
    data = val_X,
    label = val_y,
    cat_features=cat_features
)
```

3.2 Build CatBoost Models

3.2.1 Baseline Catboost

Catboost default iterations is 1000 and it automatically calculates a learning rate based on the data set and number of iterations. It will revert back to iteration with the best eval metric score for model output.

It uses log-loss for loss function and calculates log loss as well as any other metrics defined in 'custom_loss' for each iteration (can be seen on the graph).

The scale_pos_weight parameter is used for class imbalances and it is set here to the recommended amount of (sum_pos/sum_neg) (96/4). We will work on tuning this hyperparameter as well as a few others later on.

In [155]:

```
cat_model = catboost.CatBoostClassifier(
    verbose=50,
    #iterations=500,
    #learning_rate=.05,
    custom_loss=['AUC','Recall','Accuracy'],
    train_dir='first',
    scale_pos_weight=24
)

cat_model.fit(train_pool, eval_set = val_pool, verbose=200,
             plot=True)
```

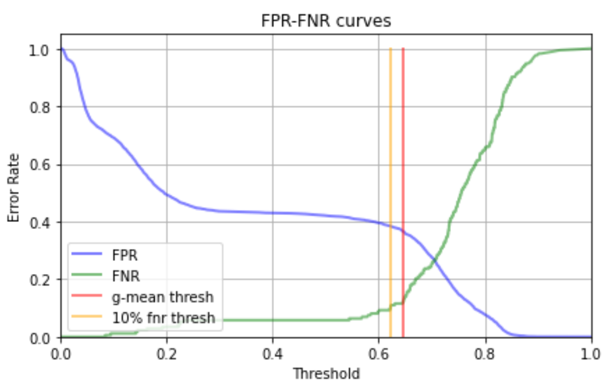
```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))

Learning rate set to 0.05403
0:      learn: 0.6645234        test: 0.6629554 best: 0.6629554 (0)     t
otal: 12.8ms    remaining: 12.8s
200:    learn: 0.3561671        test: 0.4817379 best: 0.4690717 (167)   t
otal: 1.43s     remaining: 5.7s
400:    learn: 0.2248780        test: 0.5667851 best: 0.4690717 (167)   t
otal: 2.97s     remaining: 4.43s
600:    learn: 0.1532430        test: 0.6833042 best: 0.4690717 (167)   t
otal: 4.54s     remaining: 3.02s
800:    learn: 0.1108092        test: 0.8063034 best: 0.4690717 (167)   t
otal: 6.28s     remaining: 1.56s
999:    learn: 0.0821189        test: 0.9274641 best: 0.4690717 (167)   t
otal: 8.06s     remaining: 0us

bestTest = 0.4690717093
bestIteration = 167

Shrink model to first 168 iterations.
```

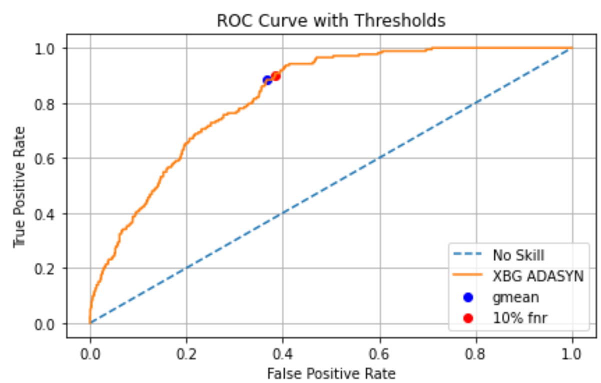
Out[155]:

```
<catboost.core.CatBoostClassifier at 0x7f80dfa9a070>
```

In [157]:
```
1 display_metrics_cat(cat_model,val_pool,val_y)
```

```
AUC: 0.8244522706682489,  logloss: 0.6121886823284021
accuracy: 0.6008
recall: 0.9422
precision: 0.1247
-------------------
For a .5 threshold:
[[1574 1144]
 [  10  163]]
FPR: 0.4209  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.645216, G-Mean=0.749
[[1722  996]
 [  20  153]]
FPR: 0.3664  FNR: 0.1156
---------------------
10% FNR Threshold:
Best Threshold: 0.6221
[[1675 1043]
 [  18  155]]
FPR: 0.3837  FNR: 0.104
```





3.2.2 Catboost built-in grid_search

In [59]:

```
clf = catboost.CatBoostClassifier(
    cat_features = cat_features,
    scale_pos_weight=24,
    verbose=False
)

param_grid = {
    'l2_leaf_reg': [1,5],
    'depth': [4,6]
}

gs_result=clf.grid_search(
    param_grid,
    train_pool,
    stratified=True,
    plot=True,
    cv=3,
    verbose=200
)
gs_result['params']
```

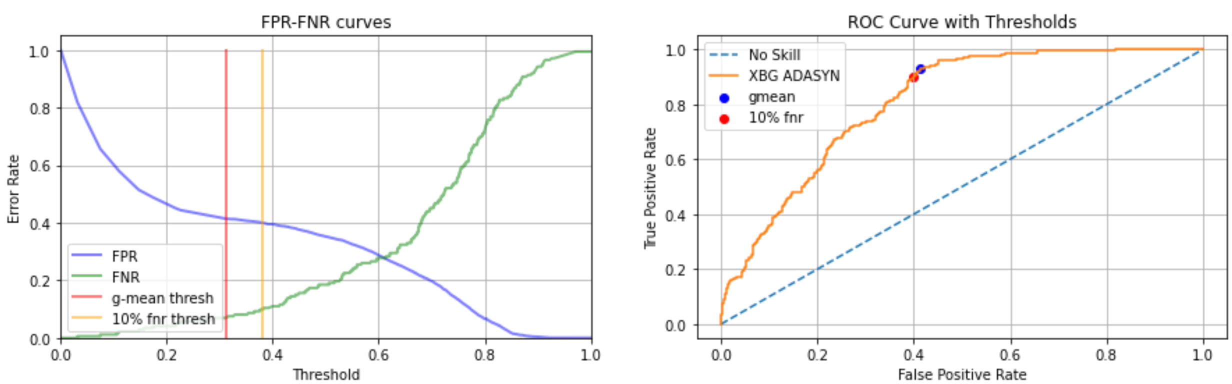
```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))


bestTest = 0.4619986624
bestIteration = 267

0:	loss: 0.4619987 best: 0.4619987 (0)	total: 4.69s	remainin
g: 14.1s

bestTest = 0.4584361725
bestIteration = 277


bestTest = 0.4641567886
bestIteration = 176


bestTest = 0.4650149914
bestIteration = 153

3:	loss: 0.4650150 best: 0.4584362 (1)	total: 23.3s	remainin
g: 0us
Estimating final quality...
```

Out[59]:

```
{'depth': 4, 'l2_leaf_reg': 5}
```

In [60]:
```
display_metrics_cat(clf,val_pool,val_y)
```

```
AUC: 0.807411093672243,  logloss: 0.5351196021036813
accuracy: 0.6579
recall: 0.815
precision: 0.1284
-------------------
For a .5 threshold:
[[1761  957]
 [  32  141]]
FPR: 0.3521  FNR: 0.185
---------------------
G-Mean Threshold:
Best Threshold=0.313214, G-Mean=0.739
[[1594 1124]
 [  13  160]]
FPR: 0.4135  FNR: 0.0751
---------------------
10% FNR Threshold:
Best Threshold: 0.3818
[[1632 1086]
 [  18  155]]
FPR: 0.3996  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.2186
[[1496 1222]
 [   9  164]]
FPR: 0.4496  FNR: 0.052
```

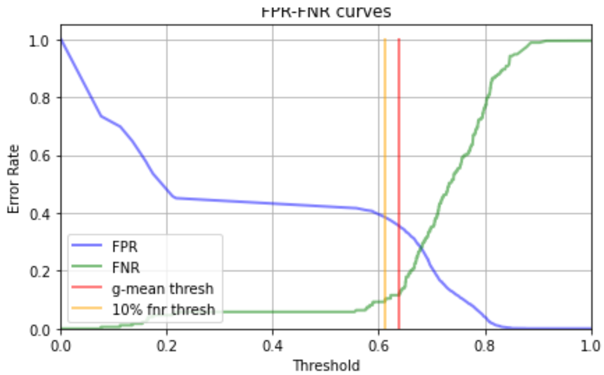


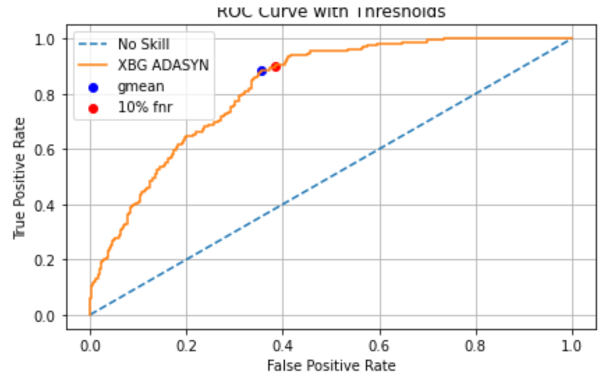
3.2.3 GridSearch for roc_auc

In [52]:

```python
#GridSearch with cat_model for roc_auc

clf = catboost.CatBoostClassifier(
    cat_features = cat_features,
    verbose=False,
)

param_grid = {
    'scale_pos_weight': [20,25,30],
    #'random_strength': [0,1],
    'l2_leaf_reg': [1],
    'depth': [4,6,10],
    'iterations': [1000],
    #'eval_metric': ['Logloss','AUC'],
    'learning_rate': [.01]
}

cat_gs = GridSearchCV(clf, param_grid = param_grid, cv=3,
                              scoring='roc_auc')
cat_gs.fit(train_X,train_y)

print('best score: {}'.format(cat_gs.best_score_))
print('best params: {}'.format(cat_gs.best_params_))

display_metrics_cat(cat_gs,val_pool,val_y)
```

```
best score: 0.8137656471308808
best params: {'depth': 4, 'iterations': 1000, 'l2_leaf_reg': 1, 'learning
_rate': 0.01, 'scale_pos_weight': 20}
AUC: 0.8228168450960626,  logloss: 0.5862809940083122
accuracy: 0.597
recall: 0.9422
precision: 0.1237
-------------------
For a .5 threshold:
[[1563 1155]
 [  10  163]]
FPR: 0.4249  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.639070, G-Mean=0.756
[[1755  963]
 [  21  152]]
FPR: 0.3543  FNR: 0.1214
---------------------
10% FNR Threshold:
Best Threshold: 0.613
[[1676 1042]
 [  18  155]]
FPR: 0.3834  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.2113
[[1475 1243]
 [   9  164]]
FPR: 0.4573  FNR: 0.052
```

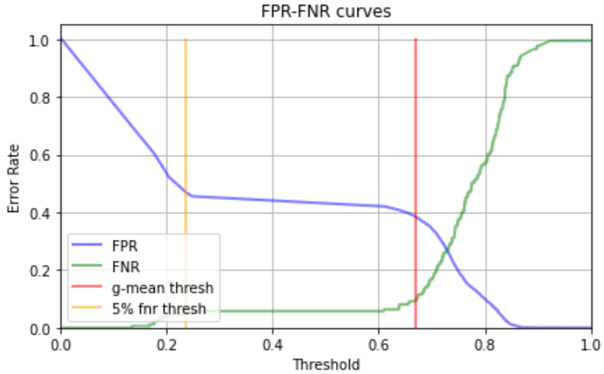

FPR-FNR curves
Error Rate
Threshold
FPR
FNR
g-mean thresh
10% fnr thresh

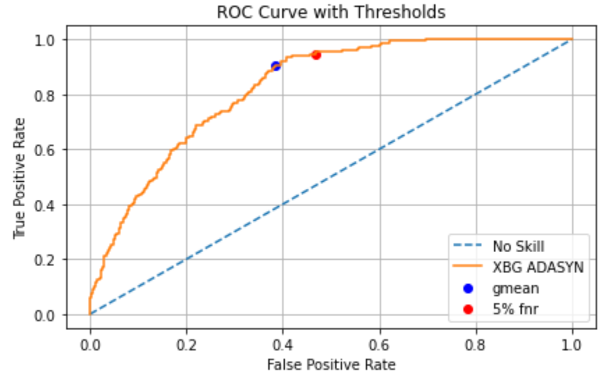

ROC Curve with Thresholds
True Positive Rate
False Positive Rate
No Skill
XBG ADASYN
gmean
10% fnr

In [220]:

```
#GridSearch with cat_model for roc_auc, add l2 regularization parameter

clf = catboost.CatBoostClassifier(
    cat_features = cat_features,
    verbose=False,
)

param_grid = {
    'scale_pos_weight': [20,25,30],
    #'random_strength': [0,1],
    'l2_leaf_reg': [1,5],
    'depth': [4,6],
    'iterations': [1000],
    'eval_metric': ['Logloss','AUC'],
    'learning_rate': [.01]
}

cat_gs = GridSearchCV(clf, param_grid = param_grid, cv=3,
                               scoring='roc_auc')
cat_gs.fit(train_X,train_y)

print('best score: {}'.format(cat_gs.best_score_))
print('best params: {}'.format(cat_gs.best_params_))

display_metrics_cat(cat_gs,val_pool,val_y)
```

```
best score: 0.8156718713890555
best params: {'depth': 4, 'eval_metric': 'Logloss', 'iterations': 1000,
'l2_leaf_reg': 5, 'learning_rate': 0.01, 'scale_pos_weight': 25}
AUC: 0.8249499164210338,  logloss: 0.6643076585476236
accuracy: 0.5918
recall: 0.9422
precision: 0.1223
-------------------
For a .5 threshold:
[[1548 1170]
 [  10  163]]
FPR: 0.4305  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.669730, G-Mean=0.747
[[1673 1045]
 [  17  156]]
FPR: 0.3845  FNR: 0.0983
---------------------
10% FNR Threshold:
Best Threshold: 0.6725
[[1683 1035]
 [  18  155]]
FPR: 0.3808  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.2367
[[1443 1275]
 [   9  164]]
FPR: 0.4691  FNR: 0.052
```

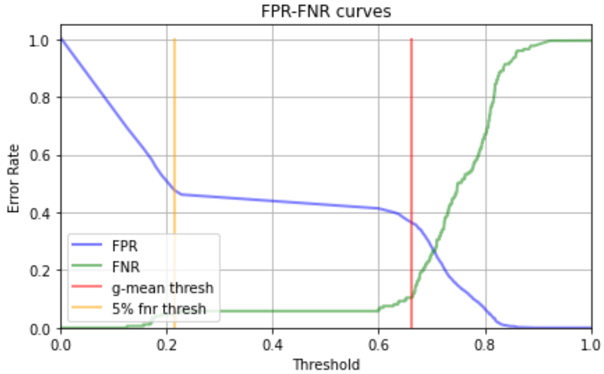

FPR-FNR curves
Error Rate
Threshold
FPR
FNR
g-mean thresh
5% fnr thresh

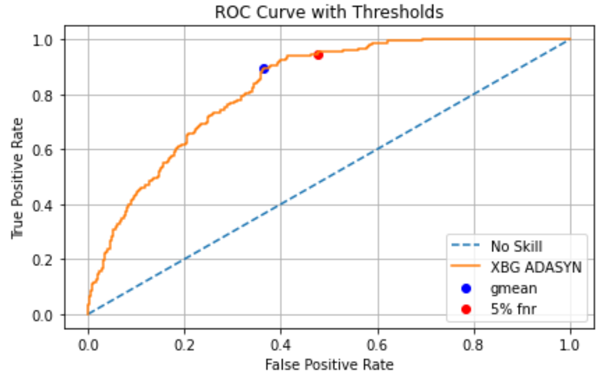

ROC Curve with Thresholds
True Positive Rate
False Positive Rate
No Skill
XBG ADASYN
gmean
5% fnr

In [221]:

```
#GridSearch with cat_model for roc_auc; refine weighting and l2

clf = catboost.CatBoostClassifier(
    cat_features = cat_features,
    verbose=False,
)

param_grid = {
    'scale_pos_weight': [22,25,28],
    #'random_strength': [0,1],
    'l2_leaf_reg': [5,8],
    'depth': [4,6],
    'iterations': [1000],
    'eval_metric': ['Logloss'],
    'learning_rate': [.01]
}

cat_gs1 = GridSearchCV(clf, param_grid = param_grid, cv=3,
                              scoring='roc_auc')
cat_gs1.fit(train_X,train_y)

print('best score: {}'.format(cat_gs1.best_score_))
print('best params: {}'.format(cat_gs1.best_params_))

display_metrics_cat(cat_gs1,val_pool,val_y)
```

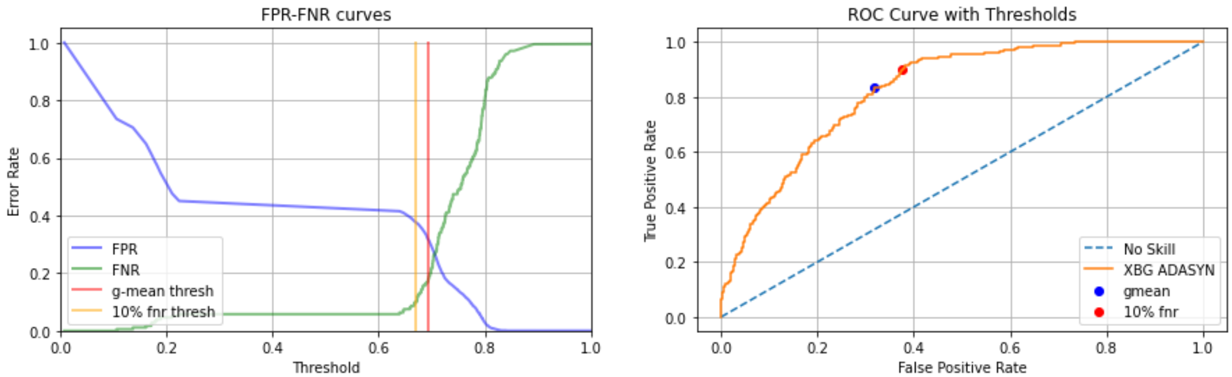
```
best score: 0.8178651989661446
best params: {'depth': 4, 'eval_metric': 'Logloss', 'iterations': 1000,
'l2_leaf_reg': 5, 'learning_rate': 0.01, 'scale_pos_weight': 22}
AUC: 0.8243076556631662,  logloss: 0.6233457838292038
accuracy: 0.5929
recall: 0.9422
precision: 0.1226
-------------------
For a .5 threshold:
[[1551 1167]
 [  10  163]]
FPR: 0.4294  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.663134, G-Mean=0.755
[[1731  987]
 [  19  154]]
FPR: 0.3631  FNR: 0.1098
---------------------
10% FNR Threshold:
Best Threshold: 0.6522
[[1699 1019]
 [  18  155]]
FPR: 0.3749  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.2148
[[1423 1295]
 [   9  164]]
FPR: 0.4765  FNR: 0.052
```


FPR-FNR curves
Error Rate
Threshold
FPR
FNR
g-mean thresh
5% fnr thresh
0.0
0.2
0.4
0.6
0.8
1.0


ROC Curve with Thresholds
True Positive Rate
False Positive Rate
No Skill
XBG ADASYN
gmean
5% fnr
0.0
0.2
0.4
0.6
0.8
1.0

In [62]:
```
#GridSearch with cat_model for roc_auc; look at higher regularization,

clf = catboost.CatBoostClassifier(
    cat_features = cat_features,
    iterations=5000,
    #learning_rate=.01,
    scale_pos_weight=22,
    l2_leaf_reg=5,
    verbose=False
)

param_grid = {
    #'scale_pos_weight': [15,20,22],
    #'random_strength': [0,1],
    #'l2_leaf_reg': [5,8],
    'depth': [4,6],
    'learning_rate': [.001,.01]
}

cat_gs2 = GridSearchCV(clf, param_grid = param_grid, cv=3,
                              scoring='roc_auc')
cat_gs2.fit(train_X,train_y)

print('best score: {}'.format(cat_gs2.best_score_))
print('best params: {}'.format(cat_gs2.best_params_))

display_metrics_cat(cat_gs2,val_pool,val_y)
```

```
best score: 0.8185251821511056
best params: {'depth': 6, 'learning_rate': 0.001}
AUC: 0.826232311245518,  logloss: 0.6286816610148134
accuracy: 0.5901
recall: 0.9422
precision: 0.1218
-------------------
For a .5 threshold:
[[1543 1175]
 [  10  163]]
FPR: 0.4323  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.692498, G-Mean=0.753
[[1852  866]
 [  30  143]]
FPR: 0.3186  FNR: 0.1734
---------------------
10% FNR Threshold:
Best Threshold: 0.671
[[1698 1020]
 [  18  155]]
FPR: 0.3753  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.2105
[[1424 1294]
 [   9  164]]
FPR: 0.4761  FNR: 0.052
```



3.2.4 GridSearch for f1_score

[...]

3.3 Best CatBoost Model

Two models performed very similarly, so we will run the test data against both of them at their 10% FNR thresholds to see which one performs better on the unseen data.

In [41]:
```
#Define both models

cat_best1 = catboost.CatBoostClassifier(
    cat_features = cat_features,
    learning_rate=.01,
    iterations=1000,
    depth=4,
    scale_pos_weight=22,
    l2_leaf_reg=5,
    random_strength=1,
    custom_loss=['AUC','Recall','Accuracy']
)

cat_best2 = catboost.CatBoostClassifier(
    cat_features = cat_features,
    learning_rate=.001,
    iterations=5000,
    depth=6,
    scale_pos_weight=22,
    l2_leaf_reg=5,
    random_strength=1,
    custom_loss=['AUC','Recall','Accuracy']
)
```

In [69]:
```
cat_best1.fit(train_pool, eval_set = val_pool, verbose=100,
                plot=True)
```

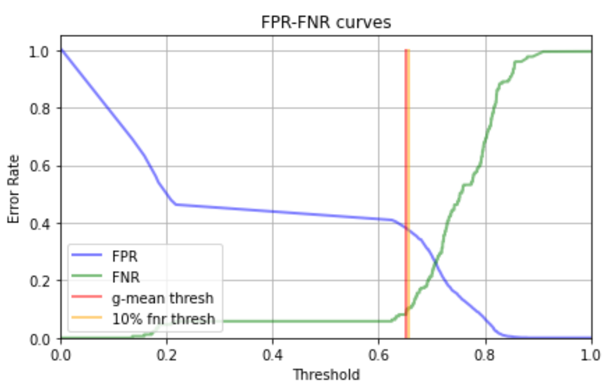
```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))

0:      learn: 0.6880709        test: 0.6879192 best: 0.6879192 (0)     t
otal: 7.3ms     remaining: 7.29s
100:    learn: 0.5226882        test: 0.5326425 best: 0.5326425 (100)   t
otal: 466ms     remaining: 4.15s
200:    learn: 0.4851890        test: 0.5016050 best: 0.5016050 (200)   t
otal: 908ms     remaining: 3.61s
300:    learn: 0.4706581        test: 0.4909785 best: 0.4909785 (300)   t
otal: 1.42s     remaining: 3.31s
400:    learn: 0.4617818        test: 0.4859080 best: 0.4859080 (400)   t
otal: 1.9s      remaining: 2.83s
500:    learn: 0.4559916        test: 0.4834718 best: 0.4834718 (500)   t
otal: 2.39s     remaining: 2.38s
600:    learn: 0.4511818        test: 0.4822575 best: 0.4822575 (600)   t
otal: 2.89s     remaining: 1.92s
700:    learn: 0.4457893        test: 0.4802597 best: 0.4801956 (696)   t
otal: 3.4s      remaining: 1.45s
800:    learn: 0.4414855        test: 0.4792926 best: 0.4792643 (798)   t
otal: 3.88s     remaining: 963ms
900:    learn: 0.4347307        test: 0.4777331 best: 0.4776872 (897)   t
otal: 4.38s     remaining: 482ms
999:    learn: 0.4280728        test: 0.4764063 best: 0.4763350 (948)   t
otal: 4.89s     remaining: 0us

bestTest = 0.4763350263
bestIteration = 948

Shrink model to first 949 iterations.
```

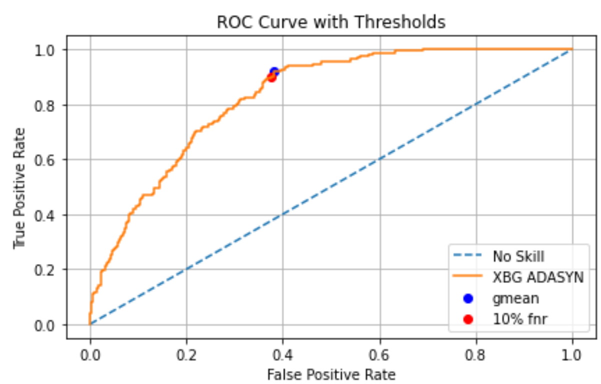
Out[69]:
```
<catboost.core.CatBoostClassifier at 0x7fbfca22cb80>
```

It appears the model is potentially underfitting, but it performs better at 1000 iterations than if we let it run for longer. If you look at AUC, it is slightly overfitting.

In [70]:
```
1 display_metrics_cat(cat_best1,val_pool,val_y)
```

```
AUC: 0.8283909028655037,  logloss: 0.6253155173024246
accuracy: 0.5918
recall: 0.9422
precision: 0.1223
-------------------
For a .5 threshold:
[[1548 1170]
 [  10  163]]
FPR: 0.4305  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.651585, G-Mean=0.754
[[1681 1037]
 [  15  158]]
FPR: 0.3815  FNR: 0.0867
---------------------
10% FNR Threshold:
Best Threshold: 0.6553
[[1697 1021]
 [  18  155]]
FPR: 0.3756  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.2085
[[1416 1302]
 [   9  164]]
FPR: 0.479  FNR: 0.052
```





In [42]:
```
cat_best2.fit(train_pool, eval_set = val_pool, verbose=1000,
                plot=True)
```

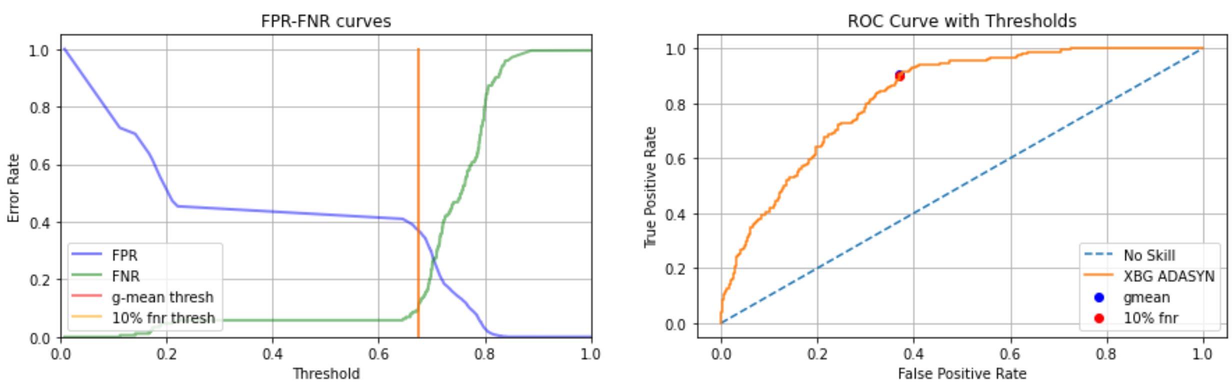
```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))

0:      learn: 0.6926143        test: 0.6925869 best: 0.6925869 (0)     t
otal: 68ms      remaining: 5m 39s
1000:   learn: 0.5127539        test: 0.5250545 best: 0.5250545 (1000)  t
otal: 6.61s     remaining: 26.4s
2000:   learn: 0.4746314        test: 0.4984342 best: 0.4984342 (2000)  t
otal: 13.5s     remaining: 20.2s
3000:   learn: 0.4573812        test: 0.4899498 best: 0.4899498 (3000)  t
otal: 20.8s     remaining: 13.9s
4000:   learn: 0.4443706        test: 0.4857017 best: 0.4857017 (4000)  t
otal: 28.8s     remaining: 7.19s
4999:   learn: 0.4348474        test: 0.4835186 best: 0.4835186 (4999)  t
otal: 37s       remaining: 0us

bestTest = 0.4835185705
bestIteration = 4999
```

Out[42]:
```
<catboost.core.CatBoostClassifier at 0x7fcd3252ad60>
```

This model also appears to potentially be underfitting, but it performs better than letting it fun to the minimum logloss. While taking the iteration with best AUC also does not perform as well, the best performance appears to be somewhere in between where AUC begins to overfit and while logloss is slightly underfit.

In [67]:
```
display_metrics_cat(cat_best2,val_pool,val_y)
```

```
AUC: 0.8261897774204937,  logloss: 0.6285505896562519
accuracy: 0.5905
recall: 0.9422
precision: 0.1219
-------------------
For a .5 threshold:
[[1544 1174]
 [  10  163]]
FPR: 0.4319  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.674519, G-Mean=0.756
[[1711 1007]
 [  17  156]]
FPR: 0.3705  FNR: 0.0983
---------------------
10% FNR Threshold:
Best Threshold: 0.6749
[[1713 1005]
 [  18  155]]
FPR: 0.3698  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.2107
[[1433 1285]
 [   9  164]]
FPR: 0.4728  FNR: 0.052
```



In [71]:

```
#Test both models on the training data

#cat1 Best threshold is .6352; predict with test data
cat1_thresh=.6352
y_hat_prob=cat_best1.predict_proba(X_test_cat)[:,1]
y_hat_test=thresh_pred(y_hat_prob,cat1_thresh)
fpr,tpr,thresholds=roc_curve(y_test_cat,y_hat_prob)
print('Cat1 Results:')
print('AUC: {},  logloss: {}'.format(auc(fpr,tpr),log_loss(y_test_cat,y
print('accuracy: {}'.format(round(accuracy_score(y_test_cat,y_hat_test)
print('recall: {}'.format(round(recall_score(y_test_cat,y_hat_test),4))
print('precision: {}'.format(round(precision_score(y_test_cat,y_hat_tes
print('-------------------')
print(confusion_matrix(y_test_cat,y_hat_test))
fpr,fnr=calc_fpr_fnr(y_test_cat,y_hat_test)
print('FPR: {}  FNR: {}'.format(round(fpr,4),round(fnr,4)))
print('-------------------')

#cat2 Best threshold is .671; predict with test data
cat2_thresh=.671
print('Cat2 Results:')
y_hat_prob=cat_best2.predict_proba(X_test_cat)[:,1]
y_hat_test=thresh_pred(y_hat_prob,cat2_thresh)
fpr,tpr,thresholds=roc_curve(y_test_cat,y_hat_prob)
print('AUC: {},  logloss: {}'.format(auc(fpr,tpr),log_loss(y_test_cat,y
print('accuracy: {}'.format(round(accuracy_score(y_test_cat,y_hat_test)
print('recall: {}'.format(round(recall_score(y_test_cat,y_hat_test),4))
print('precision: {}'.format(round(precision_score(y_test_cat,y_hat_tes
print('-------------------')
print(confusion_matrix(y_test_cat,y_hat_test))
fpr,fnr=calc_fpr_fnr(y_test_cat,y_hat_test)
print('FPR: {}  FNR: {}'.format(round(fpr,4),round(fnr,4)))
```

```
Cat1 Results:
AUC: 0.8188913735271351,  logloss: 0.6105354895181015
accuracy: 0.6241
recall: 0.8831
precision: 0.1255
-------------------
[[2202 1422]
 [  27  204]]
FPR: 0.3924  FNR: 0.1169
-------------------
Cat2 Results:
AUC: 0.8189833529237504,  logloss: 0.6129091758346167
accuracy: 0.6506
recall: 0.8615
precision: 0.1314
-------------------
[[2309 1315]
 [  32  199]]
FPR: 0.3629  FNR: 0.1385
```

Results from both models are fairly similar. Shifting cat2's threshold slightly lower results in the same results except for 1 less false positive (but that threshold shift is based on the test data).

Ultimately, I'd choose cat2, because it has similar FNR and slightly less FPR.

4 XGBoost vs. CatBoost

Catboost and XGBoost both produced models with similar performance. Below are the results of predictions with the test data set for both models. Of note, these are two different test data sets, since catboost handles categorical data and those were not one hot encoded before conducting train_test_split.

XGBoost Best Model:

> Threshold: .5285
> AUC: 0.8045
> accuracy: 0.6329
> recall: 0.8831
> precision: 0.1281
> FPR: 0.383 FNR: 0.1169

CatBoost Best Model:

> Threshold: .671
> AUC: 0.816
> accuracy: 0.6477
> recall: 0.8615
> precision: 0.1305
> FPR: 0.3659 FNR: 0.1385

I will select the CatBoost model since it does a better job at limiting FPR. It also performed better on the validation set that was used to tune the thresholds (where each FNR was set to 10%). Even if the threshold were adjusted on the test data so that FNR was at 10%, it would still outperform the XGBoost model in preventing FPs.

5 Feature Importances

5.0.1 Prediction Value Changes

Catboost provides Prediction Value Changes through importance values for each feature. These show how much the average prediction changes if the feature value is changed. These are normalized and sum to 100.

Fault has the highest importance by a significant amount followed by BasePolicy, VehicleCategory, and PolicyType.

In [43]:
```
pvc=np.array(cat_best2.get_feature_importance(prettified=True))
pvc
```

Out[43]:
```
array([['Fault', 56.372053621640895],
       ['BasePolicy', 8.99476318903155],
       ['VehicleCategory', 7.652009036190619],
       ['PolicyType', 5.204852321633231],
       ['AddressChange_Claim', 2.798442251383974],
       ['PastNumberOfClaims', 1.8950151229950705],
       ['Deductible', 1.5599212444583261],
       ['Make', 1.5383760104694022],
       ['NumberOfSuppliments', 1.5204471753253173],
       ['MonthClaimed', 1.4468874846770767],
       ['DayOfWeek', 1.3231444994709463],
       ['MaritalStatus', 1.0191426172898708],
       ['VehiclePrice', 0.9321602980337282],
       ['AgeOfPolicyHolder', 0.8740873299546149],
       ['Month', 0.8614656207498679],
       ['DriverRating', 0.8399248044756408],
       ['WeekOfMonthClaimed', 0.7334765567798813],
       ['Age', 0.711753281387135],
       ['WeekOfMonth', 0.7064322128819209],
       ['AgentType', 0.6383916953011762],
       ['AgeOfVehicle', 0.6135708488558459],
       ['Sex', 0.5662758755611337],
       ['DayOfWeekClaimed', 0.483897851530311],
       ['NumberOfCars', 0.21854758768192364],
       ['PoliceReportFiled', 0.2123985169040892],
       ['AccidentArea', 0.17047704878229442],
       ['VehicleAgeOver7', 0.09123784551830673],
       ['Less30DaysPolicyAccident', 0.008905085405547996],
       ['Less30DaysPolicyClaim', 0.007370128928347089],
       ['WitnessPresent', 0.004572836702327776]], dtype=object)
```

5.0.2 Loss Function Change

Catboost also provides a metric that gives an approximated difference between the loss function value with and without the feature. The higher the positive difference, the more important the feature is to the model.

Fault also tops this list, followed by NumberOfCars, BasePolicy, and Vehicle Category. Interestingly, NumberofCars has a much lower importance when considering prediction value changes.

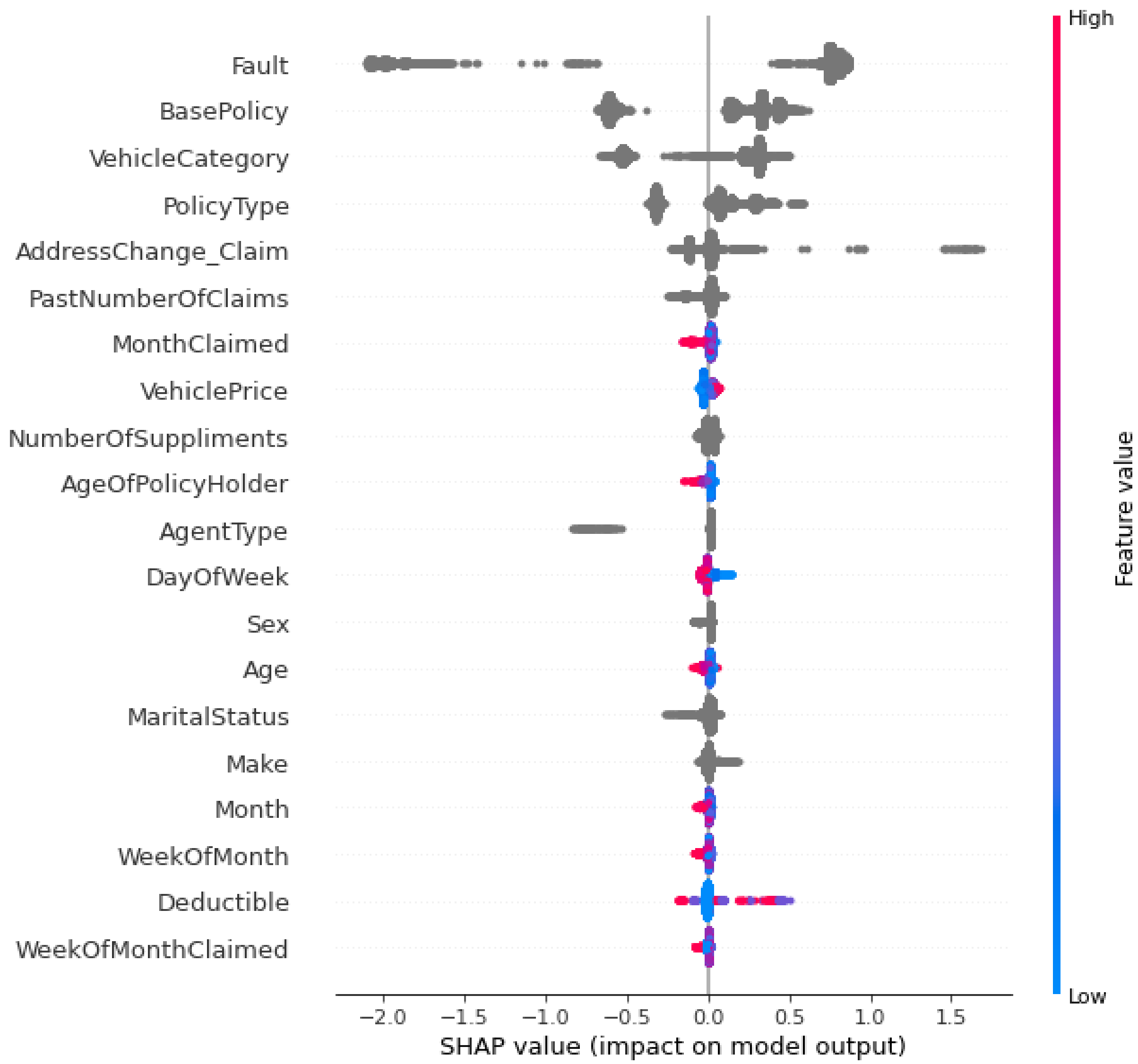
In [73]:
```python
np.array(cat_best2.get_feature_importance(
    train_pool,
    'LossFunctionChange',
    prettified=True
))
```

Out[73]:
```
array([['Fault', 0.09671625876268086],
       ['NumberOfCars', 0.016503400928953787],
       ['BasePolicy', 0.011293021125023598],
       ['VehicleCategory', 0.007500823512664509],
       ['PolicyType', 0.005429615198207626],
       ['Age', 0.005226491830372622],
       ['AddressChange_Claim', 0.004365275105131229],
       ['DriverRating', 0.0040332235215572165],
       ['Deductible', 0.0024875646870862835],
       ['MonthClaimed', 0.0020595499489448055],
       ['AgentType', 0.0015872705274533143],
       ['WeekOfMonth', 0.00150266506432184],
       ['VehiclePrice', 0.0014450892958549992],
       ['DayOfWeek', 0.0012700951005240602],
       ['Make', 0.0012538366739873559],
       ['WeekOfMonthClaimed', 0.0012219968735926643],
       ['AgeOfPolicyHolder', 0.0011402122951284694],
       ['PastNumberOfClaims', 0.0010910281142522728],
       ['Month', 0.0009782435827668707],
       ['AgeOfVehicle', 0.0009690585538549568],
       ['NumberOfSuppliments', 0.0008201602010310761],
       ['MaritalStatus', 0.000740781761663913],
       ['Sex', 0.0006479440101258005],
       ['AccidentArea', 0.0004293302087190846],
       ['DayOfWeekClaimed', 0.00040769153512976164],
       ['PoliceReportFiled', 0.00022946711606380843],
       ['VehicleAgeOver7', 0.00010951851452722927],
       ['Less30DaysPolicyClaim', 4.2011912186445954e-05],
       ['Less30DaysPolicyAccident', 1.3539901580261748e-05],
       ['WitnessPresent', 4.113629244406614e-06]], dtype=object)
```

5.0.3 Shap Values

Shap values show the impact on the model output (log odds) of each feature. The summary plot plot the shap value, impact of the feature, for each observation, so we can see groups of very large positive or negative values. Here we can visually see how much positive and negative impact Fault had on the prediction values of most of the observations in the training data. This shows that the model places a heavier importance on this feature. BasePolicy, VehicleCategory, and PolicyType also had greater importance for many of the predictions.

In [74]:
```
explainer = shap.TreeExplainer(cat_best2)
shap_values = explainer.shap_values(train_pool)
shap.summary_plot(shap_values, train_X)
```



Shap dependence plots show the shap value plotted against the value of the feature for each observation. These color of the marker is also shows the value of another feature to indicate if there are any interactions between those features.

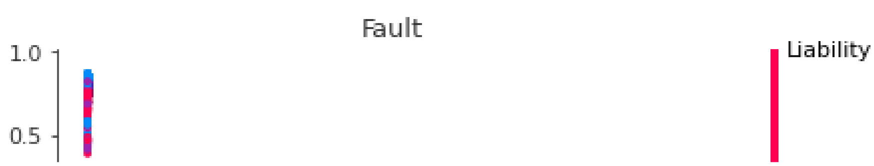
We see in the Fault dependence plot that observations where the policy holder was at fault all have strong positive impacts on the prediction. Third party fault claims all had negative impacts. The model has found that most fraud is committed when the policy holders are at fault and therefore this is an important distinguishing feature.

Furthermore, the color groupings in Base Policy, Policy Type, and Address Claim Change show that there is some level of interaction between these features and Fault. For example, fraud would be more or less likely when the policy holder is at fault and has a certain type of base policy than if they just had that base policy alone.

In [79]:

```
features=['Fault','BasePolicy','VehicleCategory','PolicyType','AddressC

for f in features:

    shap.dependence_plot(f, shap_values, train_X, title=f)
```

```
/Users/fitz/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sha
p/plots/_scatter.py:636: MatplotlibDeprecationWarning:

Passing parameters norm and vmin/vmax simultaneously is deprecated since
3.3 and will become an error two minor releases later. Please pass vmin/v
max directly to the norm when creating it.

/Users/fitz/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sha
p/plots/_scatter.py:712: MatplotlibDeprecationWarning:

Passing the fontdict parameter of _set_ticklabels() positionally is depre
cated since Matplotlib 3.3; the parameter will become keyword-only two mi
nor releases later.
```



5.0.4 Permutation Importance

Permutation importance evaluates the decrease in model score when a single feature is randomly shuffled in the data set (breaking its tie to the other the label). The more a score changes when the feature is permuted, the more importance it has to the model. I used the test data for this calculation to help highlight what features were important in the predictions and see if they were the same as those highlighted by the other metrics.

Fault is still the most important feature followed by BasePolicy, AddressChange_Claim, and VehicleCategory.

In [81]:

```
from sklearn.inspection import permutation_importance
```

In [95]:
```
result = permutation_importance(cat_best2, X_test_cat, y_test_cat, n_re
                                          random_state=0)
means=result.importances_mean
results=list(zip(val_X.columns,means))
pd.DataFrame(results,columns=['Feature','Importance']).sort_values('Imp
```

Out[95]:

	Feature	Importance
11	Fault	0.069027
26	BasePolicy	0.045214
24	AddressChange_Claim	0.003917
13	VehicleCategory	0.001686
16	DriverRating	0.000285
0	Month	0.000233
9	MaritalStatus	0.000233
1	WeekOfMonth	0.000156
7	WeekOfMonthClaimed	0.000156
2	DayOfWeek	0.000078
4	AccidentArea	0.000052
23	NumberOfSuppliments	0.000026
19	AgeOfPolicyHolder	0.000026
18	AgeOfVehicle	0.000026
15	Deductible	0.000026
6	MonthClaimed	0.000026
17	PastNumberOfClaims	0.000000
21	WitnessPresent	0.000000
27	Less30DaysPolicyAccident	0.000000
28	Less30DaysPolicyClaim	0.000000
29	VehicleAgeOver7	0.000000
12	PolicyType	-0.000026
10	Age	-0.000026
20	PoliceReportFiled	-0.000026
25	NumberOfCars	-0.000026
8	Sex	-0.000104
14	VehiclePrice	-0.000130
5	DayOfWeekClaimed	-0.000130
3	Make	-0.000337
22	AgentType	-0.000519

6 Train a Simpler Model

Now that we have determined which features contribute most to the model's prediction, we will try building models that use only the most important features to see if the model performs better on test data when it is simplified.

We will build models that have the top 3 to all 30 features progressively adding features according to their prediction value change.

In [39]:
```python
def try_diff_features(X,y,features):
    """
    X - all observations
    y - all targets
    features - list of features to try (subset of X features)

    Runs catboost model on data with just selected features.

    Outputs the FPR and FNR from the validation set, the
    threshold selected for a 10% FNR, and the FPR and FNR from test set
    """

    #Select desired features
    X_top=X[features]

    #get list of categorical features in X
    dtypes=X_top.dtypes.reset_index()
    cat_features_10=dtypes[dtypes[0]=='object']['index'].to_list()

    #Split into test and train
    X_train_10, X_test_10, y_train_10, y_test_10 = train_test_split(
        X_top, y ,random_state=42, stratify=y
    )

    #Split training set into train and validation for catboost training

    train_X_10, val_X_10, train_y_10, val_y_10 = train_test_split(
        X_train_10, y_train_10, random_state=42, stratify=y_train_10
    )

    #create train_pool and validation_pool

    train_pool_10 = catboost.Pool(
        data = train_X_10,
        label = train_y_10,
        cat_features = cat_features_10
    )

    val_pool_10 = catboost.Pool(
        data = val_X_10,
        label = val_y_10,
        cat_features=cat_features_10
    )

    cat_top10 = catboost.CatBoostClassifier(
        cat_features = cat_features_10,
        learning_rate=.001,
        iterations=5000,
        depth=6,
        scale_pos_weight=22,
        l2_leaf_reg=5,
        random_strength=1,
        custom_loss=['AUC','Recall','Accuracy']
    )
    cat_top10.fit(train_pool_10, eval_set = val_pool_10, verbose=False,
            plot=False)
```

```
    y_prob=cat_top10.predict_proba(val_pool)[:,1]

    fpr,tpr,thresholds=roc_curve(val_y_10,y_prob)
    fnr=1-tpr

    #Calculate threshold corresponding to 10% FNR, make predictions
    thresh_10, idx = select_threshold_fnr(fnr, thresholds, .1)

    y_hat_10=thresh_pred(y_prob,thresh_10)

    ten_fpr,ten_fnr=calc_fpr_fnr(val_y_10,y_hat_10)

    #Make Predictions on the test data
    y_hat_prob_test=cat_top10.predict_proba(X_test_10)[:,1]
    y_hat_test_data=thresh_pred(y_hat_prob_test,thresh_10)
    test_fpr, test_fnr=calc_fpr_fnr(y_test_10,y_hat_test_data)

    return ten_fpr, ten_fnr, thresh_10, test_fpr, test_fnr
```

In [44]:

```
pvc[:,0]
```

Out[44]:

```
array(['Fault', 'BasePolicy', 'VehicleCategory', 'PolicyType',
       'AddressChange_Claim', 'PastNumberOfClaims', 'Deductible', 'Make',
       'NumberOfSuppliments', 'MonthClaimed', 'DayOfWeek',
       'MaritalStatus', 'VehiclePrice', 'AgeOfPolicyHolder', 'Month',
       'DriverRating', 'WeekOfMonthClaimed', 'Age', 'WeekOfMonth',
       'AgentType', 'AgeOfVehicle', 'Sex', 'DayOfWeekClaimed',
       'NumberOfCars', 'PoliceReportFiled', 'AccidentArea',
       'VehicleAgeOver7', 'Less30DaysPolicyAccident',
       'Less30DaysPolicyClaim', 'WitnessPresent'], dtype=object)
```

In [45]:

```
#Loop through including top 3 to all 30 features

results=[]
#List of features in order of prediction value change importance
features=['Fault', 'BasePolicy', 'VehicleCategory', 'PolicyType',
       'AddressChange_Claim', 'PastNumberOfClaims', 'Deductible', 'Make
       'NumberOfSuppliments', 'MonthClaimed', 'DayOfWeek',
       'MaritalStatus', 'VehiclePrice', 'AgeOfPolicyHolder', 'Month',
       'DriverRating', 'WeekOfMonthClaimed', 'Age', 'WeekOfMonth',
       'AgentType', 'AgeOfVehicle', 'Sex', 'DayOfWeekClaimed',
       'NumberOfCars', 'PoliceReportFiled', 'AccidentArea',
       'VehicleAgeOver7', 'Less30DaysPolicyAccident',
       'Less30DaysPolicyClaim', 'WitnessPresent']

for f in range(3,31):
    selected_feats=features[:f]
    val_fpr,val_fnr,thresh,test_fpr,test_fnr=try_diff_features(X,y,sele
    results.append([val_fpr,val_fnr,thresh,test_fpr,test_fnr])

```

In [87]:
```
pd.DataFrame(results,columns=['val_FPR','val_FNR','Threshold',
                              'test_FPR','test_FNR'])
```

Out[87]:

	val_FPR	val_FNR	Threshold	test_FPR	test_FNR
0	0.434143	0.115607	0.309814	0.419702	0.069264
1	0.434143	0.115607	0.335746	0.419702	0.069264
2	0.200883	0.387283	0.704120	0.194536	0.437229
3	0.355776	0.161850	0.699331	0.342991	0.173160
4	0.373068	0.144509	0.689935	0.367274	0.134199
5	0.370861	0.104046	0.691475	0.362583	0.112554
6	0.377116	0.104046	0.682270	0.368929	0.108225
7	0.377483	0.104046	0.682791	0.366722	0.099567
8	0.371965	0.104046	0.684904	0.363411	0.103896
9	0.379691	0.104046	0.680496	0.369205	0.099567
10	0.373804	0.104046	0.685553	0.362859	0.099567

6.0.1 GridSearch with 19 feature model

In [45]:

```
#Catboost can handle categorical variables
#We'll use a diff version of X_train that hasn't been ohe'd
#X, y are before ohe

#Select desired features
features=pvc[:19,0]
X_top=X[features]

#get list of categorical features in X
dtypes=X_top.dtypes.reset_index()
cat_features_10=dtypes[dtypes[0]=='object']['index'].to_list()

#Split into test and train
X_train_10, X_test_10, y_train_10, y_test_10 = train_test_split(
    X_top, y ,random_state=42, stratify=y
)

#Split training set into train and validation for catboost training

train_X_10, val_X_10, train_y_10, val_y_10 = train_test_split(
    X_train_10, y_train_10, random_state=42, stratify=y_train_10
)

#create train_pool and validation_pool

train_pool_10 = catboost.Pool(
    data = train_X_10,
    label = train_y_10,
    cat_features = cat_features_10
)

val_pool_10 = catboost.Pool(
    data = val_X_10,
    label = val_y_10,
    cat_features=cat_features_10
)
```

In [53]:

```
#GridSearch with cat_model for roc_auc

cat_top10 = catboost.CatBoostClassifier(
    cat_features = cat_features_10,
    learning_rate=.001,
    iterations=5000,
    depth=6,
    scale_pos_weight=22,
    l2_leaf_reg=5,
    random_strength=1,
    verbose=False
)

param_grid = {
    'scale_pos_weight': [20,25,30],
    'l2_leaf_reg': [1,3],
    'depth': [4,6,10]
}

cat_gs_10 = GridSearchCV(cat_top10, param_grid = param_grid, cv=3,
                               scoring='roc_auc')
cat_gs_10.fit(train_X_10,train_y_10)

print('best score: {}'.format(cat_gs_10.best_score_))
print('best params: {}'.format(cat_gs_10.best_params_))

display_metrics_cat(cat_gs_10,val_pool_10,val_y_10)
```

```
best score: 0.8205859459735354
best params: {'depth': 6, 'l2_leaf_reg': 3, 'scale_pos_weight': 20}
AUC: 0.825528376441365,  logloss: 0.5987115744020525
accuracy: 0.5891
recall: 0.9422
precision: 0.1216
-------------------
For a .5 threshold:
[[1540 1178]
 [  10  163]]
FPR: 0.4334  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.660754, G-Mean=0.758
[[1745  973]
 [  19  154]]
FPR: 0.358  FNR: 0.1098
---------------------
10% FNR Threshold:
Best Threshold=0.6547
```

In [56]:
```
cat_top10 = catboost.CatBoostClassifier(
    cat_features = cat_features_10,
    learning_rate=.001,
    iterations=5000,
    depth=6,
    scale_pos_weight=20,
    l2_leaf_reg=3,
    random_strength=1,
    verbose=False
)

cat_top10.fit(train_pool_10, eval_set = val_pool_10, verbose=2000,
             plot=True)

#top19 Best threshold is .6547; predict with test data
cat10_thresh=.6547
print('Cat Top 10 Results:')
y_hat_prob=cat_top10.predict_proba(X_test_10)[:,1]
y_hat_test=thresh_pred(y_hat_prob,cat10_thresh)
fpr,tpr,thresholds=roc_curve(y_test_10,y_hat_prob)
print('AUC: {},  logloss: {}'.format(auc(fpr,tpr),log_loss(y_test_10,y_
print('accuracy: {}'.format(round(accuracy_score(y_test_10,y_hat_test),
print('recall: {}'.format(round(recall_score(y_test_10,y_hat_test),4)))
print('precision: {}'.format(round(precision_score(y_test_10,y_hat_test
print('-------------------')
print(confusion_matrix(y_test_10,y_hat_test))
fpr,fnr=calc_fpr_fnr(y_test_10,y_hat_test)
print('FPR: {}  FNR: {}'.format(round(fpr,4),round(fnr,4)))
```

```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))

0:      learn: 0.6925772        test: 0.6926081 best: 0.6926081 (0)     t
otal: 8.37ms    remaining: 41.8s
2000:   learn: 0.4760947        test: 0.5012428 best: 0.5012428 (2000)  t
otal: 11.1s     remaining: 16.6s
4000:   learn: 0.4486712        test: 0.4905771 best: 0.4905771 (4000)  t
otal: 23.4s     remaining: 5.85s
4999:   learn: 0.4399788        test: 0.4885994 best: 0.4885994 (4999)  t
otal: 29.9s     remaining: 0us

bestTest = 0.4885993806
bestIteration = 4999

Cat Top 10 Results:
AUC: 0.8165847213860459,  logloss: 0.5848117916427418
accuracy: 0.6571
recall: 0.8701
precision: 0.1346
-------------------
[[2332 1292]
 [  30  201]]
FPR: 0.3565  FNR: 0.1299
```

7 Model Selection

Ultimately, a catboost model that uses 19 of the 30 features produces the best results, although the difference isn't significant. The test data (using a threshold meant to capture 90% of the fraud) was able to flag 87% of fraud while also flagging slightly less than 36% of the non-fraud as fraud. These are not ideal outcomes, but depending on the cost of fraud vs. the cost of investigating claims that turn out not to be fraudulent, the threshold could be shifted to optimize the overall cost.

Although not in this notebook, models were tested using Logistic Regression, Random Forests, and Adaboost as well. Random Over Sampling, SMOTE, and ADASYN were also tested for dealing with the class imbalance. These models are available in the 'all_models_notebook' also in this repository. Gradient boosting with class weighting performed the best overall and so that was the focus of this notebook.

Also I found that the best catboost models tended to appear to be underfitting to logloss but overfitting to AUC. Increasing iterations did not improve the models and reducing iterations to fit to AUC did not improve performance either. 5000 iterations tended to work the best and typically meant that a model was selected that was somewhere between the best solution for AUC and Logloss.

In [49]:

```
#Prepare data/train pool/val pool for final model
#Select desired features
features=['Fault', 'BasePolicy', 'VehicleCategory', 'PolicyType',
       'AddressChange_Claim', 'PastNumberOfClaims', 'Deductible', 'Make
       'MonthClaimed', 'NumberOfSuppliments', 'DayOfWeek',
       'MaritalStatus', 'Month', 'DriverRating', 'VehiclePrice',
       'AgeOfPolicyHolder', 'Age', 'WeekOfMonth', 'WeekOfMonthClaimed']
X_final=X[features]

#get list of categorical features in X
dtypes=X_final.dtypes.reset_index()
cat_features_final=dtypes[dtypes[0]=='object']['index'].to_list()

#Split into test and train
X_train_, X_test_, y_train_, y_test_ = train_test_split(
    X_final, y ,random_state=42, stratify=y
)

#Split training set into train and validation for catboost training

train_X_, val_X_, train_y_, val_y_ = train_test_split(
    X_train_, y_train_, random_state=42, stratify=y_train_
)

#create train_pool and validation_pool

train_pool_ = catboost.Pool(
    data = train_X_,
    label = train_y_,
    cat_features = cat_features_final
)

val_pool_ = catboost.Pool(
    data = val_X_,
    label = val_y_,
    cat_features=cat_features_final
)
```

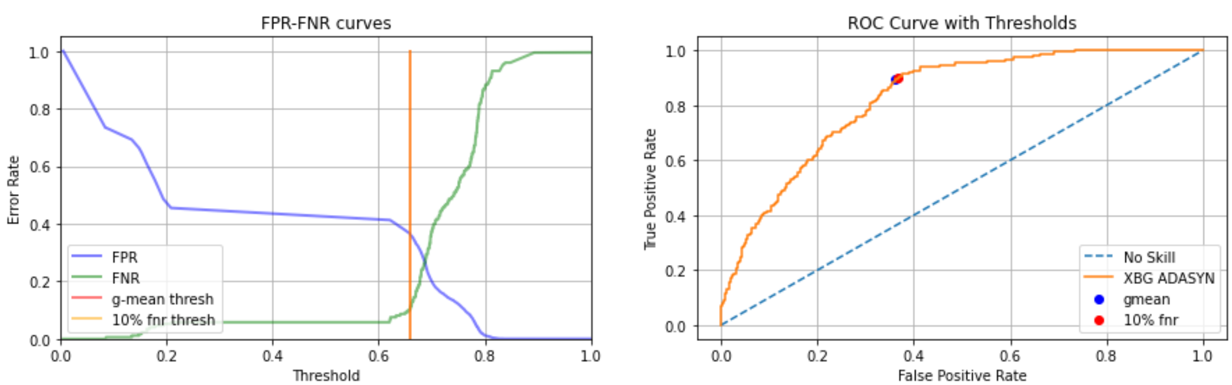
In [50]:
```
#Final Model

final_model = catboost.CatBoostClassifier(
    cat_features = cat_features_final,
    learning_rate=.001,
    iterations=5000,
    depth=6,
    scale_pos_weight=20,
    l2_leaf_reg=3,
    custom_loss=['AUC','Recall','Accuracy'],
    random_strength=1,
    verbose=False
)

final_model.fit(train_pool_, eval_set = val_pool_, verbose=2000,
                plot=True)
```

```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))

0:      learn: 0.6925285        test: 0.6925258 best: 0.6925258 (0)     t
otal: 10.9ms    remaining: 54.4s
2000:   learn: 0.4756569        test: 0.5007243 best: 0.5007243 (2000)  t
otal: 13s       remaining: 19.4s
4000:   learn: 0.4474269        test: 0.4900054 best: 0.4900054 (4000)  t
otal: 27.9s     remaining: 6.98s
4999:   learn: 0.4374264        test: 0.4881340 best: 0.4880745 (4957)  t
otal: 35.8s     remaining: 0us

bestTest = 0.4880744965
bestIteration = 4957

Shrink model to first 4958 iterations.
```

Out[50]:
```
<catboost.core.CatBoostClassifier at 0x7fcd1d6b2ee0>
```

In [47]:
```
display_metrics_cat(final_model,val_pool_10,val_y_10)
plt.savefig('thresh_plots.png',facecolor='w')
```

```
AUC: 0.8261302300654595,  logloss: 0.59900044942059
accuracy: 0.5891
recall: 0.9422
precision: 0.1216
-------------------
For a .5 threshold:
[[1540 1178]
 [  10  163]]
FPR: 0.4334  FNR: 0.0578
---------------------
G-Mean Threshold:
Best Threshold=0.659860, G-Mean=0.756
[[1735  983]
 [  19  154]]
FPR: 0.3617  FNR: 0.1098
---------------------
10% FNR Threshold:
Best Threshold: 0.6583
[[1722  996]
 [  18  155]]
FPR: 0.3664  FNR: 0.104
---------------------
5% FNR Threshold:
Best Threshold: 0.1939
[[1397 1321]
 [   9  164]]
FPR: 0.486  FNR: 0.052
```

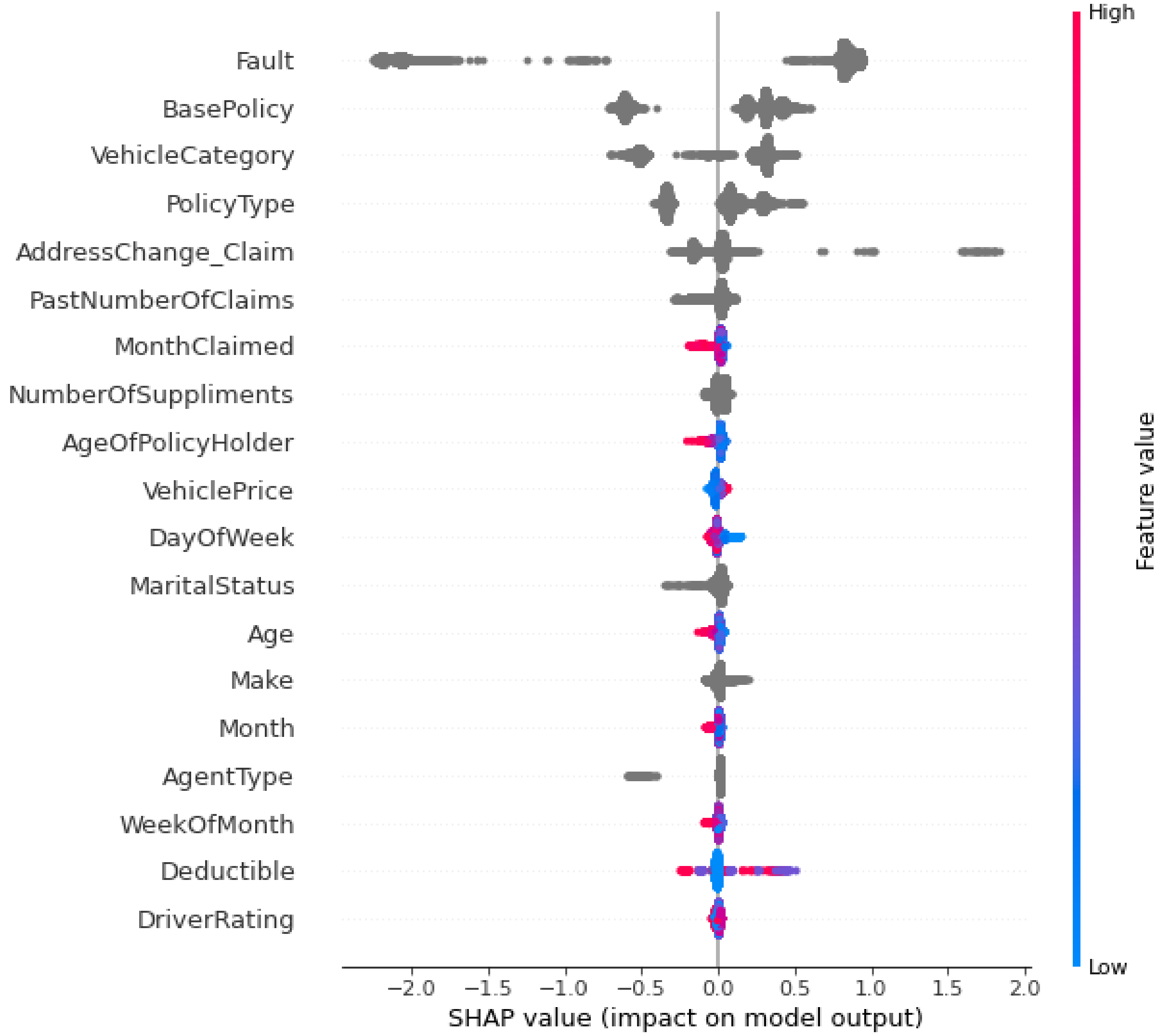


In [94]:
```
#Best threshold is .6583; predict with test data
fnr10_thresh=.6583
print('Final Model Test Results:')
y_hat_prob=final_model.predict_proba(X_test_)[:,1]
y_hat_test=thresh_pred(y_hat_prob,fnr10_thresh)
fpr,tpr,thresholds=roc_curve(y_test_,y_hat_prob)
print('AUC: {},  logloss: {}'.format(auc(fpr,tpr),log_loss(y_test_,y_ha
print('accuracy: {}'.format(round(accuracy_score(y_test_,y_hat_test),4)
print('recall: {}'.format(round(recall_score(y_test_,y_hat_test),4)))
print('precision: {}'.format(round(precision_score(y_test_,y_hat_test),
print('-------------------')
print(confusion_matrix(y_test_,y_hat_test))
fpr,fnr=calc_fpr_fnr(y_test_,y_hat_test)
print('FPR: {}  FNR: {}'.format(round(fpr,4),round(fnr,4)))
```

```
Final Model Test Results:
AUC: 0.8179106581424461,  logloss: 0.5828731040321506
accuracy: 0.6612
recall: 0.8658
precision: 0.1356
-------------------
[[2349 1275]
 [  31  200]]
FPR: 0.3518  FNR: 0.1342
```

Shap values for the final model show similar feature importances as the best catboost model with all the features included. Fault still has the most impact, followed by BasePolicy, VehicleCategory, PolicyType, and AddressChange_Claim.

In [48]:
```
explainer = shap.TreeExplainer(final_model)
shap_values = explainer.shap_values(train_pool_10)
shap.summary_plot(shap_values, train_X_10)
plt.savefig('shap_summary.png',facecolor='w')
```



<Figure size 432x288 with 0 Axes>

8 Example Cost Optimization

To give a rough idea of how a company could choose an acceptable FNR, we'll assume the following:

> -Company receives 4 million claims a year
> (Allstate has ~16 million customers, assume 12 million have car insurance,
> Americans file a claim once every three years on average)

> -Fraudulent claims cost 2000 dollars on average

> -Try various costs for the claim investigation process

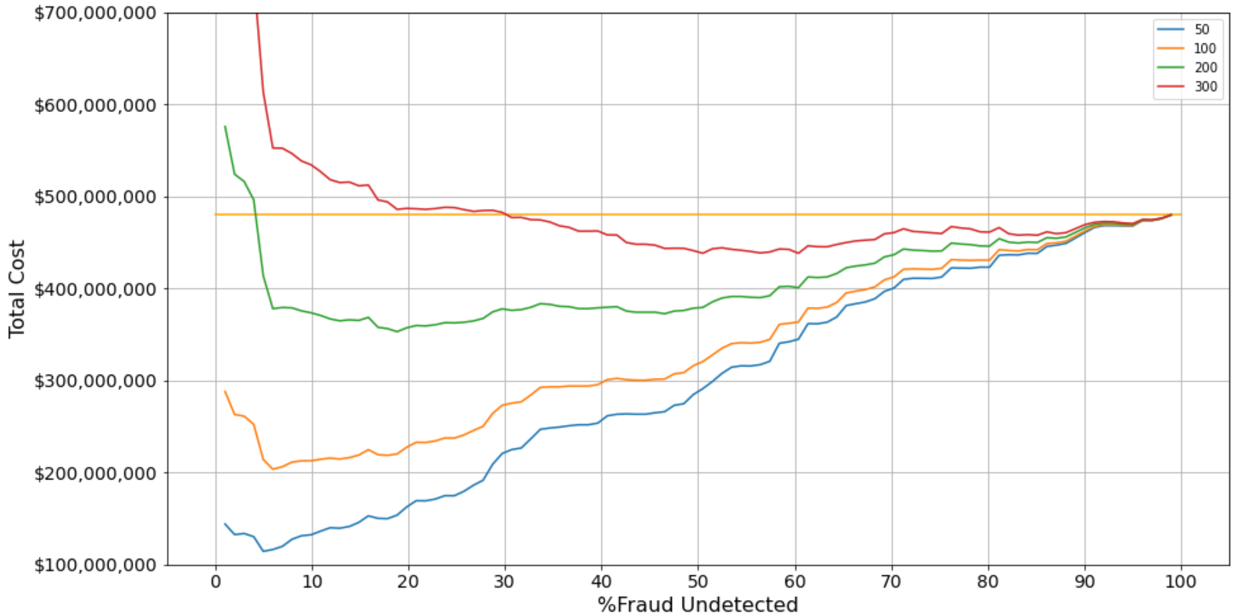
In [51]:

```
def cost_predict(target_fnr,fnr,thresholds,y_prob,y_true,cost_inv,cost_

    thresh,idx=select_threshold_fnr(fnr,thresholds,target_fnr)

    y_pred=thresh_pred(y_prob,thresh)
    fpr_,fnr_=calc_fpr_fnr(y_true,y_pred)

    #cost investigate non_fraud
    cost_non_fraud=legit*fpr_*cost_inv

    #cost non_detected fraud
    cost_undetected=fraud*fnr_*cost_fraud

    #cost to investigate detected fraud
    cost_detected=fraud*(1-fnr_)*cost_inv

    total_cost=cost_non_fraud+cost_undetected+cost_detected

    return total_cost


```

In [55]:

```
#plot cost curve for all fraud/investigation activity for varying inves

legit=4000000*.94
fraud=4000000*.06

y_hat_prob_val=final_model.predict_proba(val_X_)[:,1]
fpr,tpr,thresholds=roc_curve(val_y_,y_hat_prob_val)
fnr=1-tpr
y_hat_prob=final_model.predict_proba(X_test_)[:,1]

def cost_plot(cost_inv,cost_fraud,y_hat_prob,fnr,thresholds):
    fnr_tests=np.linspace(.01,.99,100)
    cost_results=[]

    for f in fnr_tests:

        cost=cost_predict(
            f,fnr,thresholds,y_hat_prob,y_test_,cost_inv,cost_fraud
        )
        cost_results.append(cost)

    return cost_results

fig,ax=plt.subplots(figsize=(15,8))
ax.plot([0,100],[480000000,480000000],color='orange')
fnr_tests=np.linspace(.01,.99,100)

for c in [50,100,200,300]:
    cost_results=cost_plot(c,2000,y_hat_prob,fnr,thresholds)

    ax.plot(100*fnr_tests,cost_results,label=c)

ax.set_title(
    'Undetected Fraud vs Total Fraud Cost (Fraud \${} per claim)'.forma
        2000),
    fontsize=16, y=1.05
)
ax.grid(True,which='both')
ax.set_xlabel('%Fraud Undetected', fontsize=16)
ax.set_ylabel('Total Cost', fontsize=16)
ticks=ax.get_yticks().tolist()[1:-3]
ax.set_yticks(ticks)
ylabels=['$'+'{:,.0f}'.format(y) for y in ticks]
ax.set_yticklabels(ylabels,fontdict={'fontsize':14})
xt=list(range(0,110,10))
ax.set_xticks(xt)
ax.set_xticklabels(xt,fontdict={'fontsize':14})
ax.legend()
ax.set_ylim(100000000,700000000)

plt.savefig('cost_curve.png',facecolor='w')
```

Undetected Fraud vs Total Fraud Cost (Fraud $2000 per claim)



This plot shows us that the usefulness of this model depends on the average cost of fraud and the cost to investigate claims. If the cost to investigate is too high compared to the average cost of fraud, it may not save the company money to flag fraud. Lower costs to investigate in relation to cost of fraud tend to have monetary benefit even if large percentages of actual fraud are missed. This would have to be evaluated carefully before implementation.

9 Conclusions

The model produced in this project can identify approximately 90% of fraudulent claims while flagging only 36% of non-fraudulent claims as potential fraud.

The most important characteristics of a claim to this prediction are whether the claimant or a third-party is at fault, the base policy, vehicle category, policy type, and whether there was an address change.

9.1 Next Steps

To optimize this model for use, it would be important to understand the cost of fraud vs. the cost to investigate a claim. This would help determine an acceptable level of fraud to miss in order to reduce the number of false positives.

The data for this project was also very general in the way it grouped values into ranges. More specific information on the claims could help improve the model. It would also be useful to examine more claim observations.

Lastly, some fraud is more expensive than other fraud. If the cost of the fraud was available it could be useful to group fraud labels into 'high-cost' and 'low-cost' fraud and focus on predicting fraud that is high-cost. This could be achieved by conducting multiclass classification or by only trying to classify a claim as 'high-cost fraud' or 'not high cost fraud'. One could still examine if the low-cost fraud ends up getting frequently flagged by the model as a result of targeting high-cost fraud.