In [18]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import os


from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import average_precision_score
from sklearn import tree
import xgboost
import shap
import sklearn.metrics as metrics
import graphviz
```

In [2]:
```
print(os.getcwd())
os.chdir('D:/OneDrive/ASU/Humana_Case_Competition')
print(os.getcwd())
```

```
C:\Users\Jinhang Jiang
D:\OneDrive\ASU\Humana_Case_Competition
```

Read Data

In [3]:
```
humana = pd.read_csv('Train_Dummy.csv')
```

In [5]:
```
humana.head()
```

Out[5]:

	person_id_syn	transportation_issues	est_age	age_group	smoker_current_ind	sm
0	0002MOb79ST17bLYAe46eIc2	0	62	2	1.0	
1	0004cMOS6bTLf34Y7AIca8f3	0	59	1	1.0	
2	000536M9O3ST98LaYaeA29Ia	1	63	2	0.0	
3	0009bMO9SfTLYe77A51I4ac3	0	75	3	0.0	
4	000M7OeS66bTL8bY89Aa16Ie	0	51	1	1.0	

5 rows × 1882 columns

In [6]:
```
holdout = pd.read_csv('Test_Dummy.csv')
```

In [7]:
```
holdout.head()
```

Out[7]:

	person_id_syn	transportation_issues	est_age	age_group	smoker_current_ind	smo
0	000M289dOSbe8dTL75c71YAI	2	68	2	1	
1	000b16MOSTLY7A637698c5l3	2	65	2	0	
2	0011MOdcfS9188T8aLYA3dla	2	67	2	0	
3	001MO8SaT6dL8ae755cYA3dl	2	76	3	0	
4	001MOS3a40Tc5L1534YAel40	2	65	2	0	

5 rows × 1882 columns

In [8]:
```
print("Training:",humana.shape,", Testing:", holdout.shape)
```

```
Training: (69572, 1882) , Testing: (17681, 1882)
```

In [5]:
```
humana=humana.dropna(axis='columns')
```

In [6]:
```
humana.head()
```

Out[6]:

	person_id_syn	transportation_issues	src_platform_cd	sex_cd	est_age	smoker_cu
0	0002MOb79ST17bLYAe46eIc2	0	EM	F	62	
1	0004cMOS6bTLf34Y7AIca8f3	0	EM	F	59	
2	000536M9O3ST98LaYaeA29la	1	EM	F	63	
3	0009bMO9SfTLYe77A51l4ac3	0	EM	M	75	
4	000M7OeS66bTL8bY89Aa16le	0	EM	M	51	

5 rows × 695 columns

In [4]:
```
label = humana['transportation_issues']
data = humana.drop(['person_id_syn','transportation_issues'], axis = 1)
data= data.fillna(data.mean())
```

Encoding Variables

In [9]:
```
data_dict = data.to_dict(orient='records')
```

In [99]:
```
# DictVectorizer
from sklearn.feature_extraction import DictVectorizer
# instantiate a Dictvectorizer object for X
dv_X = DictVectorizer(sparse=False)
# sparse = False makes the output is not a sparse matrix

# apply dv_X on X_dict
X_encoded = dv_X.fit_transform(data_dict)
X_encoded = pd.DataFrame(X_encoded)
# show X_encoded
X_encoded
```

Out[99]:

	0	1	2	3	4	5	6	7	8	9	...	1687	1688	1689	1690	1691	1692
0	0.0	0.0	0.0	0.0	1.0	1.162658	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	1.0	1.155124	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	1.0	0.333333	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	1.0	0.250000	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	1.0	0.083333	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
69567	0.0	0.0	0.0	0.0	1.0	1.250000	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
69568	0.0	0.0	0.0	0.0	1.0	1.668168	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
69569	0.0	0.0	0.0	0.0	1.0	0.666667	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
69570	0.0	0.0	0.0	0.0	1.0	0.116657	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
69571	0.0	0.0	0.0	0.0	1.0	0.288073	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0

69572 rows × 1697 columns

Split data, convert to xgboost.DMatrix

In [7]:
```
X_train, X_test, y_train, y_test = train_test_split(data, label, test_size = 0.2, random_s
d_train = xgboost.DMatrix(X_train, label=y_train)
d_test = xgboost.DMatrix(X_test, label=y_test)
```

Cross Validation

In [8]:
```
params = {
    # Parameters that we are going to tune.
    'max_depth':4,
    'min_child_weight': 1,
    'eta':0.05,
    'subsample': 0.9,
    'colsample_bytree': 0.4,
    'objective':'binary:logistic',
    "eval_metric": ["auc","logloss"],
    'gamma':5,
    "base_score": np.mean(y_train),
    'scale_pos_weight':1,
    'tree_method': "hist",
    'lambda': 80,
    'alpha': 0,
    'grow_policy': 'lossguide',
    'max_bin':1000,
    'num_parallel_tree':1
}
```

In [21]:
```
# current cv score
%time cv_results = xgboost.cv(params,d_train,num_boost_round=3000,seed=42,nfold=5,metrics=

cv_results
```

```
Wall time: 8min 17s
```

Out[21]:

	train-auc-mean	train-auc-std	test-auc-mean	test-auc-std
0	0.682317	0.001639	0.674400	0.009431
1	0.716434	0.001797	0.708675	0.005283
2	0.728344	0.002515	0.721136	0.003905
3	0.729347	0.002666	0.721819	0.004045
4	0.729166	0.002555	0.721189	0.004319
...	...	...	...	...
301	0.791739	0.001177	0.747037	0.005536
302	0.791861	0.001180	0.747034	0.005500
303	0.791967	0.001203	0.747049	0.005477
304	0.792066	0.001215	0.747032	0.005464
305	0.792193	0.001220	0.747090	0.005456

306 rows × 4 columns

In [72]:
```
# Max_Depth / min_child_weight
gridsearch_params = [
    (max_depth, min_child_weight)
    for max_depth in range(4,12)
    for min_child_weight in range(1,3)
]

# Define initial best params and MAE
max_auc = 0.001
best_params = None
for max_depth, min_child_weight in gridsearch_params:
    print("CV with max_depth={}, min_child_weight={}".format(
                             max_depth,
                             min_child_weight))
    # Update our parameters
    params['max_depth'] = max_depth
    params['min_child_weight'] = min_child_weight
    # Run CV
    cv_results = xgboost.cv(
        params,
        d_train,
        num_boost_round=3000,
        seed=42,
        nfold=5,
        metrics={'auc'},
        early_stopping_rounds=20
    )
    # Update best AUC
    mean_auc = cv_results['test-auc-mean'].max()
    boost_rounds = cv_results['test-auc-mean'].idxmax()
    print("\tAUC {} for {} rounds".format(mean_auc, boost_rounds))
    if mean_auc > max_auc:
        max_auc = mean_auc
        best_params = (max_depth,min_child_weight)

print("Best params: {}, {}, AUC: {}".format(best_params[0], best_params[1], max_auc))
```

```
CV with max_depth=4, min_child_weight=1
        AUC 0.733268 for 11 rounds
CV with max_depth=4, min_child_weight=2
        AUC 0.7332424000000001 for 12 rounds
CV with max_depth=5, min_child_weight=1
        AUC 0.7307572 for 12 rounds
CV with max_depth=5, min_child_weight=2
        AUC 0.7305996 for 9 rounds
CV with max_depth=6, min_child_weight=1
        AUC 0.7221287999999999 for 7 rounds
CV with max_depth=6, min_child_weight=2
        AUC 0.7230958 for 7 rounds
CV with max_depth=7, min_child_weight=1
        AUC 0.7152972 for 6 rounds
CV with max_depth=7, min_child_weight=2
        AUC 0.7141366 for 7 rounds
CV with max_depth=8, min_child_weight=1
        AUC 0.7068496 for 7 rounds
CV with max_depth=8, min_child_weight=2
```

```
        AUC 0.7060690000000001 for 6 rounds
CV with max_depth=9, min_child_weight=1
        AUC 0.6949455999999999 for 7 rounds
CV with max_depth=9, min_child_weight=2
        AUC 0.6998168 for 4 rounds
CV with max_depth=10, min_child_weight=1
        AUC 0.6856842000000001 for 5 rounds
CV with max_depth=10, min_child_weight=2
        AUC 0.6893614 for 7 rounds
CV with max_depth=11, min_child_weight=1
        AUC 0.6799682 for 7 rounds
CV with max_depth=11, min_child_weight=2
        AUC 0.681673 for 4 rounds


---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-72-00d73cdef7a9> in <module>
     34         best_params = (max_depth,min_child_weight)
     35
---> 36 print("Best params: {}, {}, AUC: {}".format(best_params[0], best_params[1
], max_auc))

TypeError: 'NoneType' object is not subscriptable
```

In [76]:
```
# CV for subsample, colsample
gridsearch_params = [
    (subsample, colsample)
    for subsample in [i/10. for i in range(3,11)]
    for colsample in [i/10. for i in range(3,11)]
]

max_auc = 0.001
best_params = None
# We start by the largest values and go down to the smallest
for subsample, colsample in reversed(gridsearch_params):
    print("CV with subsample={}, colsample={}".format(
                             subsample,
                             colsample))
    # We update our parameters
    params['subsample'] = subsample
    params['colsample_bytree'] = colsample
    # Run CV
    cv_results = xgboost.cv(
        params,
        d_train,
        num_boost_round=3000,
        seed=42,
        nfold=5,
        metrics={'auc'},
        early_stopping_rounds=20
    )
    # Update best score
    mean_auc = cv_results['test-auc-mean'].max()
    boost_rounds = cv_results['test-auc-mean'].idxmax()
    print("\tAUC {} for {} rounds".format(mean_auc, boost_rounds))
    if mean_auc > max_auc:
        max_auc = mean_mae
        best_params = (subsample,colsample)
print("Best params: {}, {}, AUC: {}".format(best_params[0], best_params[1], max_auc))
```

```
CV with subsample=1.0, colsample=1.0
        AUC 0.7380396 for 20 rounds
CV with subsample=1.0, colsample=0.9
        AUC 0.7395216 for 18 rounds
CV with subsample=1.0, colsample=0.8
        AUC 0.7379863999999999 for 19 rounds
CV with subsample=1.0, colsample=0.7
        AUC 0.7381412 for 21 rounds
CV with subsample=1.0, colsample=0.6
        AUC 0.7391922 for 21 rounds
CV with subsample=1.0, colsample=0.5
        AUC 0.738558 for 15 rounds
CV with subsample=1.0, colsample=0.4
        AUC 0.7384006000000001 for 25 rounds
CV with subsample=1.0, colsample=0.3
        AUC 0.7379743999999999 for 18 rounds
CV with subsample=0.9, colsample=1.0
        AUC 0.7366931999999999 for 14 rounds
CV with subsample=0.9, colsample=0.9
```

In [117]:
```
# eta cv
%time
# This can take some time…
max_auc = 0.001
best_params = None
for eta in [0.5, .1, .05, .01]:
    print("CV with eta={}".format(eta))
    # We update our parameters
    params['eta'] = eta

    # Run and time CV
    %time cv_results = xgboost.cv(params,d_train,num_boost_round=3000,seed=42,nfold=5,metr

    # Update best score
    mean_auc = cv_results['test-auc-mean'].max()
    boost_rounds = cv_results['test-auc-mean'].idxmax()
    print("\tAUC {} for {} rounds\n".format(mean_auc, boost_rounds))
    if mean_auc > max_auc:
        max_auc = mean_auc
        best_params = eta
print("Best params: {}, AUC: {}".format(best_params, max_auc))
```

```
Wall time: 0 ns
CV with eta=0.5
Wall time: 1min 23s
        AUC 0.7348878 for 9 rounds

CV with eta=0.1
Wall time: 2min 54s
        AUC 0.7438035999999999 for 69 rounds

CV with eta=0.05
Wall time: 5min 21s
        AUC 0.7449472 for 187 rounds

CV with eta=0.01
Wall time: 21min 10s
        AUC 0.7465132 for 876 rounds

Best params: 0.01, AUC: 0.7465132
```

In [120]:
```
# gamma cv
%time
# This can take some time…
max_auc = 0.001
best_params = None
for gamma in range(0,11):
    print("CV with gamma={}".format(gamma))
    # We update our parameters
    params['gamma'] = gamma

    # Run and time CV
    %time cv_results = xgboost.cv(params,d_train,num_boost_round=3000,seed=42,nfold=5,metr

    # Update best score
    mean_auc = cv_results['test-auc-mean'].max()
    boost_rounds = cv_results['test-auc-mean'].idxmax()
    print("\tgamma {} for {} rounds\n".format(mean_auc, boost_rounds))
    if mean_auc > max_auc:
        max_auc = mean_auc
        best_params = gamma
print("Best params: {}, AUC: {}".format(best_params, max_auc))
```

```
Wall time: 0 ns
CV with gamma=0
Wall time: 5min 4s
        gamma 0.7447266000000001 for 187 rounds

CV with gamma=1
Wall time: 5min 11s
        gamma 0.7449472 for 187 rounds

CV with gamma=2
Wall time: 6min 33s
        gamma 0.7452806000000001 for 236 rounds

CV with gamma=3
Wall time: 5min 15s
        gamma 0.7452108 for 203 rounds

CV with gamma=4
Wall time: 6min 2s
        gamma 0.7453942 for 210 rounds

CV with gamma=5
Wall time: 5min 50s
        gamma 0.7456396000000001 for 187 rounds

CV with gamma=6
Wall time: 5min 33s
        gamma 0.7453147999999999 for 189 rounds

CV with gamma=7
Wall time: 5min 13s
        gamma 0.7450448 for 187 rounds

CV with gamma=8
```

```
Wall time: 5min 21s
        gamma 0.7454401999999999 for 192 rounds

CV with gamma=9
Wall time: 5min 43s
        gamma 0.745478 for 219 rounds

CV with gamma=10
Wall time: 6min 25s
        gamma 0.745482 for 203 rounds

Best params: 5, AUC: 0.7456396000000001
```

In [8]:
```
# lambda cv
%time
# This can take some time…
max_auc = 0.001
best_params = None
for lam in range(70,95,5):
    print("CV with lambda={}".format(lam))
    # We update our parameters
    params['lambda'] = lam

    # Run and time CV
    %time cv_results = xgboost.cv(params,d_train,num_boost_round=3000,seed=42,nfold=5,metr

    # Update best score
    mean_auc = cv_results['test-auc-mean'].max()
    boost_rounds = cv_results['test-auc-mean'].idxmax()
    print("\tlambda {} for {} rounds\n".format(mean_auc, boost_rounds))
    if mean_auc > max_auc:
        max_auc = mean_auc
        best_params = lam
print("Best params: {}, AUC: {}".format(best_params, max_auc))
```

```
Wall time: 0 ns
CV with lambda=70
Wall time: 12min 24s
        lambda 0.7464752000000001 for 274 rounds

CV with lambda=75
Wall time: 11min 50s
        lambda 0.7464544 for 256 rounds

CV with lambda=80
Wall time: 11min 59s
        lambda 0.7470898 for 305 rounds

CV with lambda=85
Wall time: 9min 25s
        lambda 0.747023 for 293 rounds

CV with lambda=90
Wall time: 7min 46s
        lambda 0.7468312000000001 for 271 rounds

Best params: 80, AUC: 0.7470898
```

Train Model

In [9]:
```
# base_score = mean(label) can help in this case since we have a fairly large dataset


model = xgboost.train(params, d_train, 5000, evals = [(d_test, "test")], verbose_eval=100,
```

```
[0]     test-auc:0.70192        test-logloss:0.40276
Multiple eval metrics have been passed: 'test-logloss' will be used for early stopping.

Will train until test-logloss hasn't improved in 30 rounds.
[100]   test-auc:0.74801        test-logloss:0.35578
[200]   test-auc:0.74909        test-logloss:0.35470
[300]   test-auc:0.74972        test-logloss:0.35443
Stopping. Best iteration:
[275]   test-auc:0.74970        test-logloss:0.35439
```

In [28]:
```
xgb = XGBClassifier(**params)
xgb.fit(X_train, y_train)
```

Out[28]:
```
XGBClassifier(alpha=0, base_score=0.1480855957022477, booster='gbtree',
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=0.4,
              eta=0.05, eval_metric=['auc', 'logloss'], gamma=5, gpu_id=-1,
              grow_policy='lossguide', importance_type='gain',
              interaction_constraints='', lambda=80, learning_rate=0.0500000007,
              max_bin=1000, max_delta_step=0, max_depth=4, min_child_weight=1,
              missing=nan, monotone_constraints='()', n_estimators=100,
              n_jobs=0, num_parallel_tree=1, random_state=0, reg_alpha=0,
              reg_lambda=80, scale_pos_weight=1, subsample=0.9,
              tree_method='hist', ...)
```

In [32]:
```
params
```

Out[32]:
```
{'max_depth': 4,
 'min_child_weight': 1,
 'eta': 0.05,
 'subsample': 0.9,
 'colsample_bytree': 0.4,
 'objective': 'binary:logistic',
 'eval_metric': ['auc', 'logloss'],
 'gamma': 5,
 'base_score': 0.1480855957022477,
 'scale_pos_weight': 1,
 'tree_method': 'hist',
 'lambda': 80,
 'alpha': 0,
 'grow_policy': 'lossguide',
 'max_bin': 1000,
 'num_parallel_tree': 1}
```

In [29]:
```
# make predictions for test data
y_pred = xgb.predict_proba(X_test)
predictions = [round(value) for value in y_pred[:,1]]


# evaluate predictions
accuracy = accuracy_score(y_test, predictions)

#laucpr
print("Predict test set... ")
#test_prediction = DecisionTree.predict(X_test)
score = average_precision_score(y_test, predictions)

#auc_roc
fpr, tpr, threshold = metrics.roc_curve(y_test, y_pred[:,1])
roc_auc = metrics.auc(fpr, tpr)



print("Accuracy: %.2f%%" % (accuracy * 100.0),
      'area under the precision-recall curve test set: {:.6f}'.format(score),
     "roc:",roc_auc,)
```

```
Predict test set...
Accuracy: 86.24% area under the precision-recall curve test set: 0.170974 roc: 0.747996
5614281193
```

Accuracy, aucpr, auc

In [9]:
```
from sklearn.metrics import average_precision_score
print("Predict test set... ")
test_prediction = model.predict(xgboost.DMatrix(X_test))
score = average_precision_score(y_test, test_prediction)
print('area under the precision-recall curve test set: {:.6f}'.format(score))
```

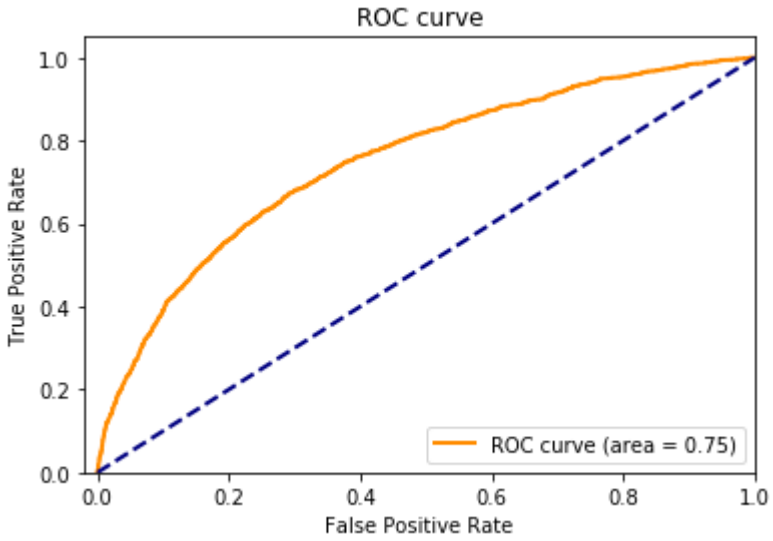
```
Predict test set...
area under the precision-recall curve test set: 0.354189
```

In [10]:
```
fpr, tpr, threshold = metrics.roc_curve(y_test, test_prediction)
roc_auc = metrics.auc(fpr, tpr)
print(roc_auc)
```

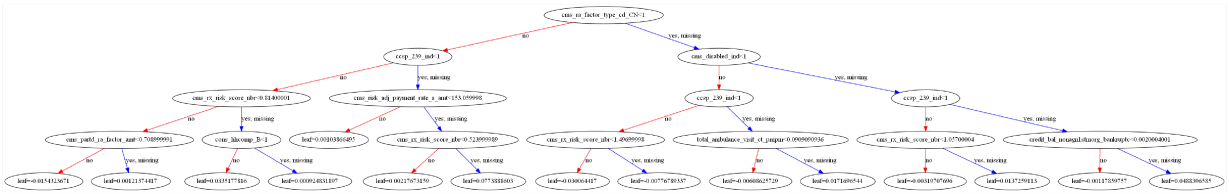
```
0.7497151630755545
```

ROC Plot

In [26]:
```
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange',
         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([-0.02, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve')
plt.legend(loc="lower right")
plt.show()
```

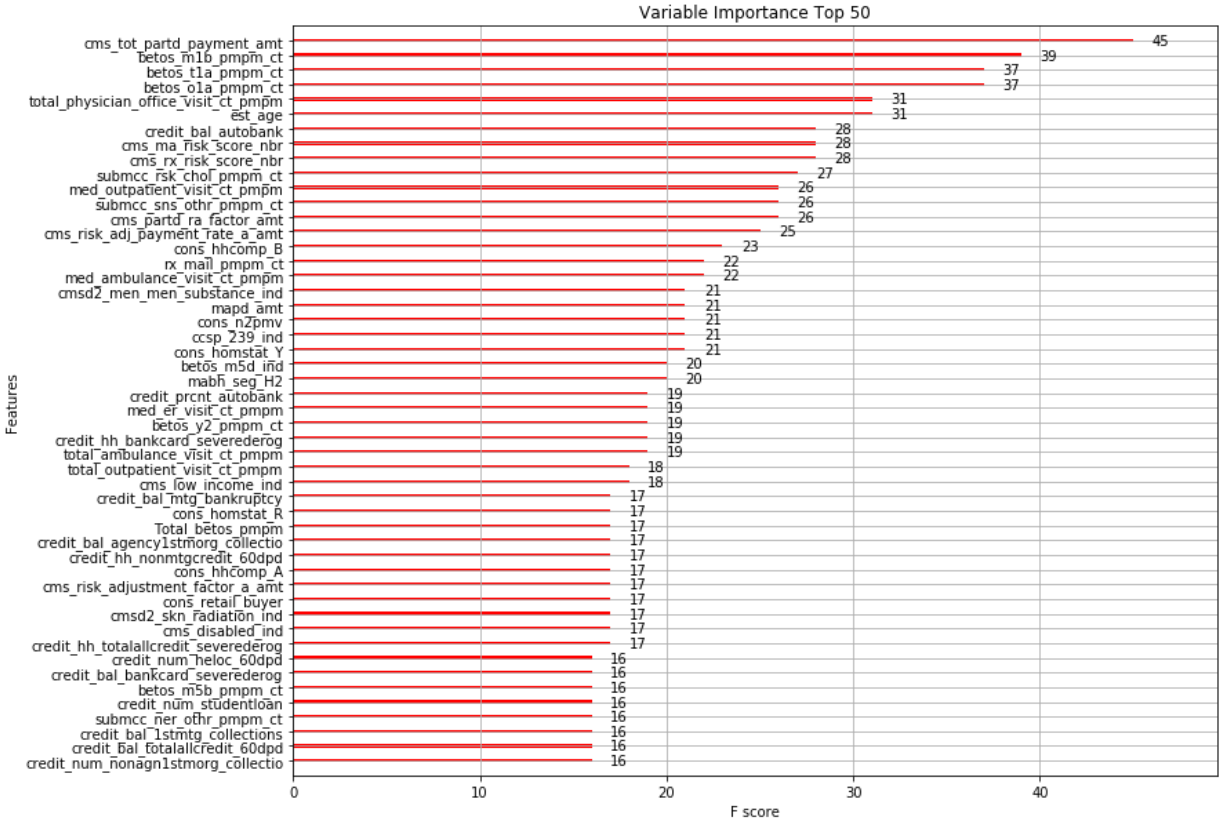


In [20]:
```
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 40,150
fig = xgboost.plot_tree(model, num_trees=2,max_depth=4)
fig = plt.gcf()
fig.set_dpi(150)
fig.savefig('xgbtree.png')
```

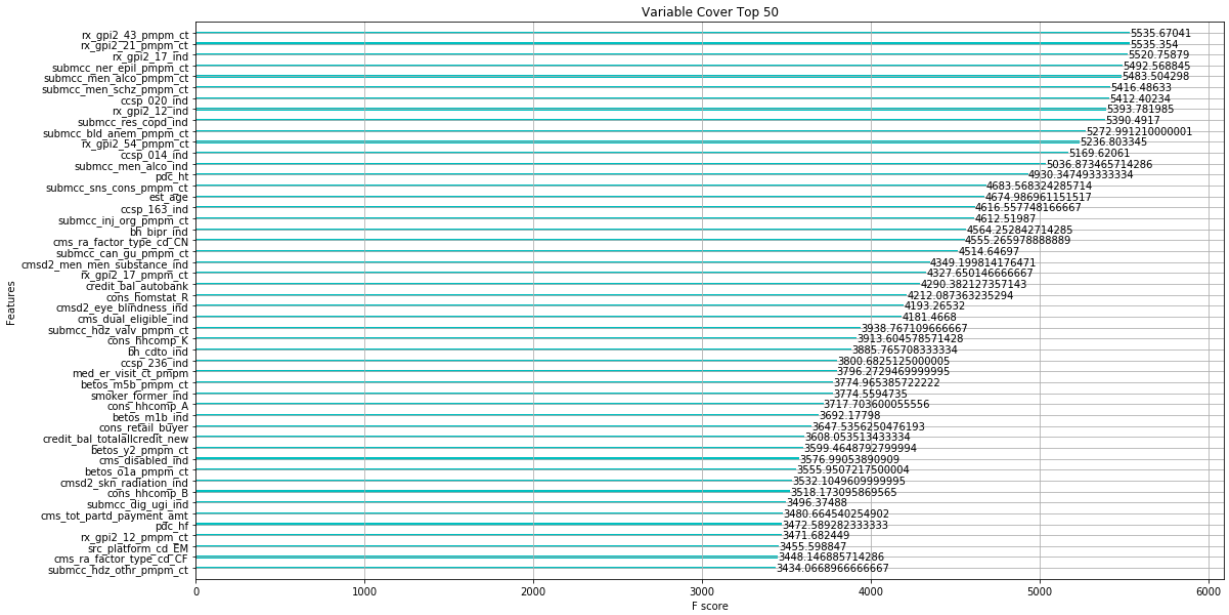


Variables Selection

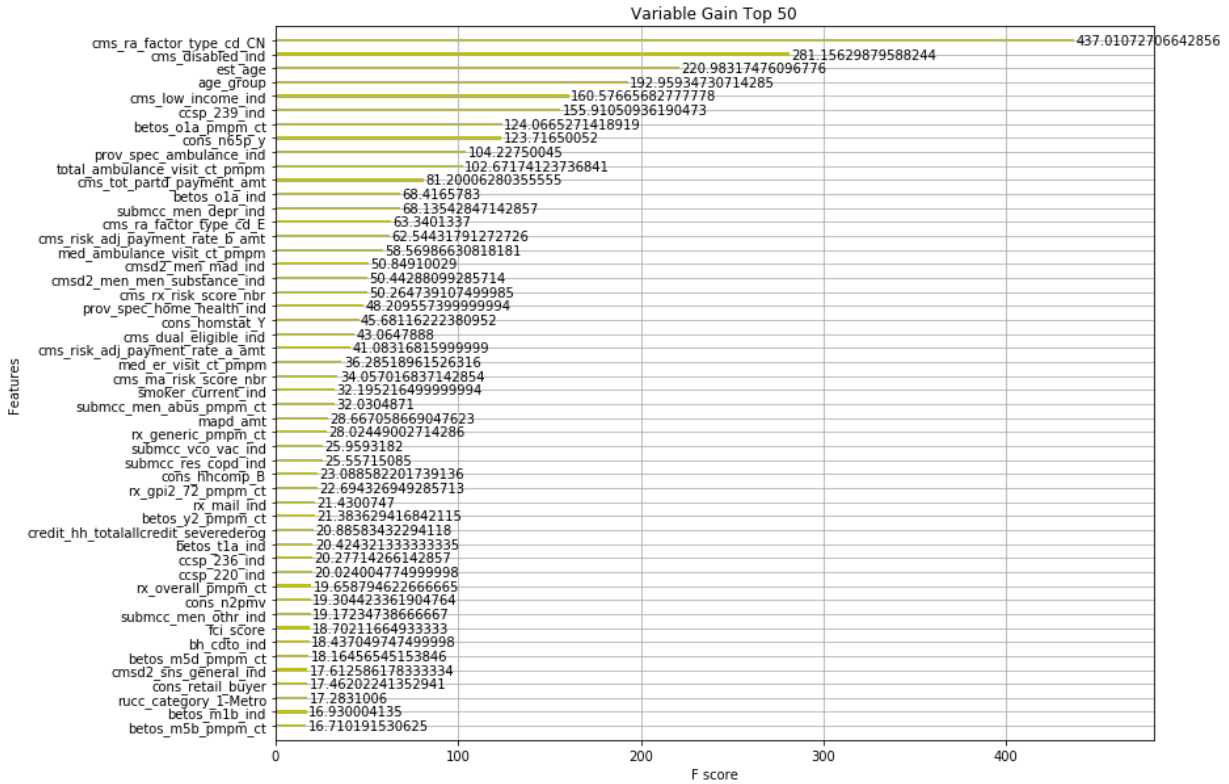
In [17]:
```
xgboost.plot_importance(model, max_num_features=50,color='red')
plt.title("Variable Importance Top 50")
plt.rcParams['figure.figsize'] = (28,10)
plt.show()
```



In [36]:
```
xgboost.plot_importance(model, importance_type="cover",max_num_features=50, color = 'c')
plt.title('Variable Cover Top 50')
plt.rcParams['figure.figsize'] = (28,10)
plt.show()
```



In [30]:
```
xgboost.plot_importance(model, importance_type="gain",max_num_features=50, color = 'y')
plt.title('Variable Gain Top 50')
plt.rcParams['figure.figsize'] = (28,10)
plt.show()
```



In [ ]:
```

```

Get important variables out

In [31]:
```
Feature_Importance = pd.DataFrame.from_dict(model.get_score(),orient='index')
Feature_Importance_cover = pd.DataFrame.from_dict(model.get_score(importance_type="cover")
Feature_Importance_gain = pd.DataFrame.from_dict(model.get_score(importance_type="gain"),o
```

In [32]:
```
Feature_Importance.columns=['Importance']
Feature_Importance_cover.columns=['Cover']
Feature_Importance_gain.columns=['Gain']
```

In [33]:
```
print('Importance:',len(Feature_Importance),"; Cover:",len(Feature_Importance_cover),
      "; Gain:", len(Feature_Importance_gain))
```

```
Importance: 376 ; Cover: 376 ; Gain: 376
```

In [34]:
```
VariableSelection = Feature_Importance.merge(
    Feature_Importance_cover, left_index=True, right_index=True)
```

In [35]:
```
VariableSelection = VariableSelection.merge(
    Feature_Importance_gain, left_index=True, right_index=True)
```

In [36]:
```
VariableSelection
```

Out[36]:

	Importance	Cover	Gain
betos_o1a_pmpm_ct	37	3550.438160	124.066527
cms_low_income_ind	18	3011.415755	160.576657
cms_risk_adj_payment_rate_b_amt	11	1439.558214	62.544318
credit_hh_totalallcredit_severederog	17	1862.375507	20.885834
cmsd2_men_mad_ind	10	3941.071406	50.849100
...	...	...	...
hedis_cmc_ldc_c_control_Y	1	3680.504880	7.269076
ccsp_062_ind	1	3528.826420	6.274889
submcc_hdz_arrh_ind	1	1453.488400	7.490728
submcc_men_schz_pmpm_ct	1	5373.187990	6.152776
submcc_rsk_fh/ho_pmpm_ct	1	2742.234130	7.009776

376 rows × 3 columns

In [37]:
```
#VariableSelection.to_csv('VariablesNew.csv')
```

In [96]:
```
cols=list(VariableSelection.index)
```

In [107]:
```
Dummy_396=humana[cols]
Dummy_396['transportation_issues']=humana['transportation_issues']
Dummy_396.index=humana['person_id_syn']
Dummy_396.head()
#Dummy_396.to_csv('Dummy_396.csv')
```

```
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
```

predict

In [118]:
```
TEST = pd.read_csv('Test_Dummy.csv')
```

In [119]:
```
TEST.head()
ID = TEST['person_id_syn']
```

In [112]:
```
TEST=TEST[list(Dummy_396.columns)]
```

In [115]:
```
TEST_dm = xgboost.DMatrix(TEST.drop(['person_id_syn','transportation_issues'], axis=1))
```

In [116]:
```
TEST_pred = model.predict(TEST_dm)
print(TEST_pred)
```

```
[0.4231314  0.05022658 0.13446388 ... 0.20961092 0.05612351 0.3281207 ]
```

In [120]:
```
DATA = pd.DataFrame({"ID": ID, "Score":TEST_pred})
#DATA['Transportation_Issues'].astype(int)
DATA['RANK'] = DATA['Score'].rank().astype(int)
DATA.head()
```

Out[120]:

	ID	Score	RANK
0	000M289dOSbe8dTL75c71YAI	0.423131	16848
1	000b16MOSTLY7A637698c5I3	0.050227	2662
2	0011MOdcfS9188T8aLYA3dIa	0.134464	10990
3	001MO8SaT6dL8ae755cYA3dI	0.057107	3930
4	001MOS3a40Tc5L1534YAeI40	0.347057	16177

In [121]:
```
DATA.to_csv('xgboost0.749_Oct5.csv', index=False)
```