Insurance Purchasing Analysis ¶

Use classification techniques to identify the potential purchasers based on a Kaggle dataset from the real case.

In [1]:

```
!pip install imblearn
```

```
Waiting for a Spark session to start...
Spark Initialization Done! ApplicationId = app-20210429235128-0000
KERNEL_ID = 8830535d-f4b7-4ce1-ba36-8a407a064e9f
Collecting imblearn
  Downloading imblearn-0.0-py2.py3-none-any.whl (1.9 kB)
Collecting imbalanced-learn
  Downloading imbalanced_learn-0.8.0-py3-none-any.whl (206 kB)
     |████████████████████████████████| 206 kB 12.4 MB/s eta 0:00:01
Collecting scikit-learn>=0.24
  Downloading scikit_learn-0.24.2-cp37-cp37m-manylinux2010_x86_64.whl (22.
3 MB)
     |████████████████████████████████| 22.3 MB 11.1 MB/s eta 0:00:01
Collecting scipy>=0.19.1
  Downloading scipy-1.6.3-cp37-cp37m-manylinux1_x86_64.whl (27.4 MB)
     |████████████████████████████████| 27.4 MB 39.5 MB/s eta 0:00:01
Collecting joblib>=0.11
  Downloading joblib-1.0.1-py3-none-any.whl (303 kB)
     |████████████████████████████████| 303 kB 42.6 MB/s eta 0:00:01
Collecting numpy>=1.13.3
  Downloading numpy-1.20.2-cp37-cp37m-manylinux2010_x86_64.whl (15.3 MB)
     |████████████████████████████████| 15.3 MB 27.7 MB/s eta 0:00:01
Collecting threadpoolctl>=2.0.0
  Downloading threadpoolctl-2.1.0-py3-none-any.whl (12 kB)
ERROR: tensorflow 2.1.0 has requirement scipy==1.4.1; python_version >=
 "3", but you'll have scipy 1.6.3 which is incompatible.
Installing collected packages: joblib, threadpoolctl, numpy, scipy, scikit
-learn, imbalanced-learn, imblearn
Successfully installed imbalanced-learn-0.8.0 imblearn-0.0 joblib-1.0.1 nu
mpy-1.20.2 scikit-learn-0.24.2 scipy-1.6.3 threadpoolctl-2.1.0
```

Load the data from IBM cloud

In [2]:

```
import ibmos2spark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Insurance_Analysis").getOrCreate()
# @hidden_cell
credentials = {
    'endpoint': 'https://s3-api.us-geo.objectstorage.service.networklayer.com',
    'service_id': 'iam-ServiceId-3cf147f6-84c7-43ab-b522-6b0af7c5567d',
    'iam_service_endpoint': 'https://iam.cloud.ibm.com/oidc/token',
    'api_key': 'd0zPXxornqV9DL2e8xcLDoA-G8G60kfeGrAIDkidaVGq'
}

configuration_name = 'os_1cb3a8f78c2c4a129c3c68cea1ce623c_configs'
cos = ibmos2spark.CloudObjectStorage(sc, credentials, configuration_name, 'bluemix_cos'
)
caravan_insurance_raw2 = spark.read.csv(cos.url('caravan-insurance_2.csv', 'advanceddat
asciencecapstone-donotdelete-pr-vdy06vb49sgqdo'), header=True, inferSchema=True)
```

In [3]:

```
# Import Libraries
import pyspark.sql.functions as F
from pyspark.sql.functions import avg, col, concat, desc, lit, min, max, split, udf, co
untDistinct, sum, count,array,explode
from pyspark.sql.types import IntegerType, DoubleType
from pyspark.sql import Window
from pyspark.sql.types import Row

from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler, MinMaxScaler,ChiSqSelec
tor, PCA,OneHotEncoder
from pyspark.ml.classification import LogisticRegression,GBTClassifier,LinearSVC, Rando
mForestClassifier, NaiveBayes, FMClassifier
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.linalg import Vectors,DenseVector
from pyspark.ml.evaluation import MulticlassClassificationEvaluator,BinaryClassificatio
nEvaluator

import numpy as np
import pandas as pd
from imblearn.over_sampling import SMOTE

import matplotlib.pyplot as plt
import matplotlib.dates as mdates
%matplotlib inline
import seaborn as sns
```

Data Preprocessing and Cleaning

Here we checked missing/duplicated values and outliers (if any).

In [4]:

```
#caravan_insurance_pdf=caravan_insurance_raw.drop('ORIGIN')
caravan_insurance_pdf=caravan_insurance_raw2
caravan_insurance_pdf=caravan_insurance_pdf.withColumnRenamed('CARAVAN','label')
caravan_insurance_pdf.describe().toPandas()
caravan_insurance_pdf_copy=caravan_insurance_pdf.withColumnRenamed('CARAVAN','label')
```

In [5]:

```
# process duplicate and null value
#Note: If we have any missing values, we need to do imputation. For the dataset used he
re, it is N/A.
caravan_insurance_pdf=caravan_insurance_pdf.dropDuplicates(subset=[c for c in caravan_i
nsurance_pdf.columns if c != 'ID'])
caravan_insurance_pdf=caravan_insurance_pdf.na.drop()
caravan_insurance_pdf=caravan_insurance_pdf.drop('ID')
# check sample size for different classes
print(caravan_insurance_pdf.count(),caravan_insurance_pdf.filter(col('label') == 1).cou
nt(),caravan_insurance_pdf.filter(col('label') == 0).count())
```

```
8950 578 8372
```

In [6]:

```
# print the dataset schema
caravan_insurance_pdf.printSchema()
```

```
root
 |-- ORIGIN: string (nullable = true)
 |-- MOSTYPE: integer (nullable = true)
 |-- MAANTHUI: integer (nullable = true)
 |-- MGEMOMV: integer (nullable = true)
 |-- MGEMLEEF: integer (nullable = true)
 |-- MOSHOOFD: integer (nullable = true)
 |-- MGODRK: integer (nullable = true)
 |-- MGODPR: integer (nullable = true)
 |-- MGODOV: integer (nullable = true)
 |-- MGODGE: integer (nullable = true)
 |-- MRELGE: integer (nullable = true)
 |-- MRELSA: integer (nullable = true)
 |-- MRELOV: integer (nullable = true)
 |-- MFALLEEN: integer (nullable = true)
 |-- MFGEKIND: integer (nullable = true)
 |-- MFWEKIND: integer (nullable = true)
 |-- MOPLHOOG: integer (nullable = true)
 |-- MOPLMIDD: integer (nullable = true)
 |-- MOPLLAAG: integer (nullable = true)
 |-- MBERHOOG: integer (nullable = true)
 |-- MBERZELF: integer (nullable = true)
 |-- MBERBOER: integer (nullable = true)
 |-- MBERMIDD: integer (nullable = true)
 |-- MBERARBG: integer (nullable = true)
 |-- MBERARBO: integer (nullable = true)
 |-- MSKA: integer (nullable = true)
 |-- MSKB1: integer (nullable = true)
 |-- MSKB2: integer (nullable = true)
 |-- MSKC: integer (nullable = true)
 |-- MSKD: integer (nullable = true)
 |-- MHHUUR: integer (nullable = true)
 |-- MHKOOP: integer (nullable = true)
 |-- MAUT1: integer (nullable = true)
 |-- MAUT2: integer (nullable = true)
 |-- MAUT0: integer (nullable = true)
 |-- MZFONDS: integer (nullable = true)
 |-- MZPART: integer (nullable = true)
 |-- MINKM30: integer (nullable = true)
 |-- MINK3045: integer (nullable = true)
 |-- MINK4575: integer (nullable = true)
 |-- MINK7512: integer (nullable = true)
 |-- MINK123M: integer (nullable = true)
 |-- MINKGEM: integer (nullable = true)
 |-- MKOOPKLA: integer (nullable = true)
 |-- PWAPART: integer (nullable = true)
 |-- PWABEDR: integer (nullable = true)
 |-- PWALAND: integer (nullable = true)
 |-- PPERSAUT: integer (nullable = true)
 |-- PBESAUT: integer (nullable = true)
 |-- PMOTSCO: integer (nullable = true)
 |-- PVRAAUT: integer (nullable = true)
 |-- PAANHANG: integer (nullable = true)
 |-- PTRACTOR: integer (nullable = true)
 |-- PWERKT: integer (nullable = true)
 |-- PBROM: integer (nullable = true)
 |-- PLEVEN: integer (nullable = true)
 |-- PPERSONG: integer (nullable = true)
 |-- PGEZONG: integer (nullable = true)
 |-- PWAOREG: integer (nullable = true)
 |-- PBRAND: integer (nullable = true)
```

```
 |-- PZEILPL: integer (nullable = true)
 |-- PPLEZIER: integer (nullable = true)
 |-- PFIETS: integer (nullable = true)
 |-- PINBOED: integer (nullable = true)
 |-- PBYSTAND: integer (nullable = true)
 |-- AWAPART: integer (nullable = true)
 |-- AWABEDR: integer (nullable = true)
 |-- AWALAND: integer (nullable = true)
 |-- APERSAUT: integer (nullable = true)
 |-- ABESAUT: integer (nullable = true)
 |-- AMOTSCO: integer (nullable = true)
 |-- AVRAAUT: integer (nullable = true)
 |-- AAANHANG: integer (nullable = true)
 |-- ATRACTOR: integer (nullable = true)
 |-- AWERKT: integer (nullable = true)
 |-- ABROM: integer (nullable = true)
 |-- ALEVEN: integer (nullable = true)
 |-- APERSONG: integer (nullable = true)
 |-- AGEZONG: integer (nullable = true)
 |-- AWAOREG: integer (nullable = true)
 |-- ABRAND: integer (nullable = true)
 |-- AZEILPL: integer (nullable = true)
 |-- APLEZIER: integer (nullable = true)
 |-- AFIETS: integer (nullable = true)
 |-- AINBOED: integer (nullable = true)
 |-- ABYSTAND: integer (nullable = true)
 |-- label: integer (nullable = true)
```

Exploratory Data Analysis

Based on the visualization, the main types of insurance purchasers were family with grown-ups and average family. The main age group concentrated on 40-50 years old.
Numerical & categoical features both were showing correlation with their kind only.

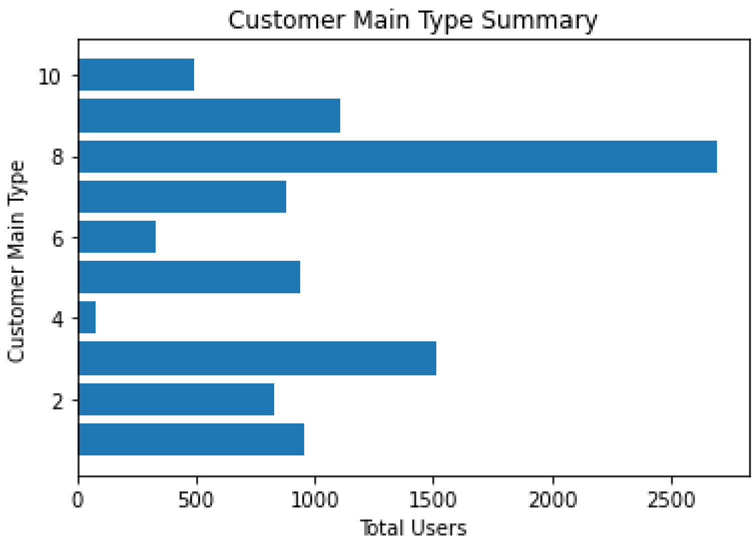
In [7]:

```
df1=caravan_insurance_pdf_copy.groupby('MOSTYPE').agg(F.countDistinct('ID')).sort(F.col
('count(ID)').desc()) #.show(50,False)
df2=caravan_insurance_pdf_copy.filter(col('label') == 0).groupby('MOSTYPE','label').agg
(F.countDistinct('ID')).sort(F.col('count(ID)').desc()) #.show(50,False)
df3=caravan_insurance_pdf_copy.filter(col('label') == 1).groupby('MOSTYPE','label').agg
(F.countDistinct('ID')).sort(F.col('count(ID)').desc()).show(50,False)
#print(df3)
```

```
+-------+-----+---------+
|MOSTYPE|label|count(ID)|
+-------+-----+---------+
|33     |1    |80       |
|8      |1    |72       |
|38     |1    |38       |
|39     |1    |37       |
|3      |1    |33       |
|12     |1    |28       |
|36     |1    |27       |
|1      |1    |26       |
|6      |1    |26       |
|13     |1    |25       |
|10     |1    |23       |
|37     |1    |19       |
|9      |1    |17       |
|11     |1    |16       |
|35     |1    |13       |
|32     |1    |12       |
|34     |1    |12       |
|2      |1    |11       |
|41     |1    |11       |
|31     |1    |11       |
|29     |1    |7        |
|25     |1    |6        |
|24     |1    |6        |
|22     |1    |6        |
|30     |1    |5        |
|7      |1    |5        |
|23     |1    |4        |
|4      |1    |3        |
|26     |1    |2        |
|20     |1    |2        |
|5      |1    |2        |
|27     |1    |1        |
+-------+-----+---------+
```

In [8]:

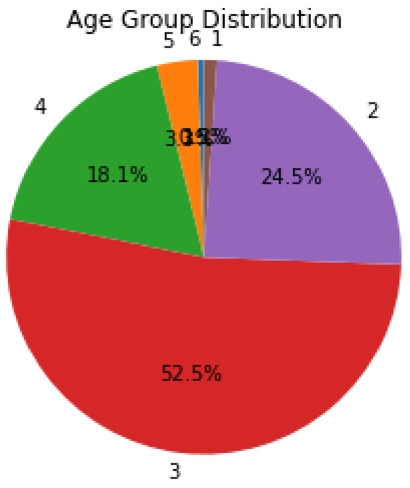
```
# Check the distribution for customer main type
main_count=caravan_insurance_pdf_copy.groupby('MOSHOOFD').agg(F.countDistinct('ID')).so
rt(F.col('MOSHOOFD').desc()).toPandas()
#print(main_count.head(5))
plt.barh(main_count['MOSHOOFD'],main_count['count(ID)'])
plt.ylabel('Customer Main Type')
plt.xlabel('Total Users')
plt.title('Customer Main Type Summary')
plt.show()
```



Customer Main Type List:
1 Successful hedonists
2 Driven Growers
3 Average Family
4 Career Loners
5 Living well
6 Cruising Seniors
7 Retired and Religeous
8 Family with grown ups
9 Conservative families
10 Farmers

In [9]:

```
# Check the distribution for age groups
age_count=caravan_insurance_pdf_copy.groupby('MGEMLEEF').agg(F.countDistinct('ID')).sor
t(F.col('MGEMLEEF').desc()).toPandas()
fig1, ax1 = plt.subplots()
plt.pie(age_count['count(ID)'],labels=age_count['MGEMLEEF'], autopct='%1.1f%%',shadow=F
alse, startangle=90)
plt.axis('equal')
#plt.ylabel('Customer Main Type')
#plt.xlabel('Total Users')
plt.title('Age Group Distribution')
plt.show()
```

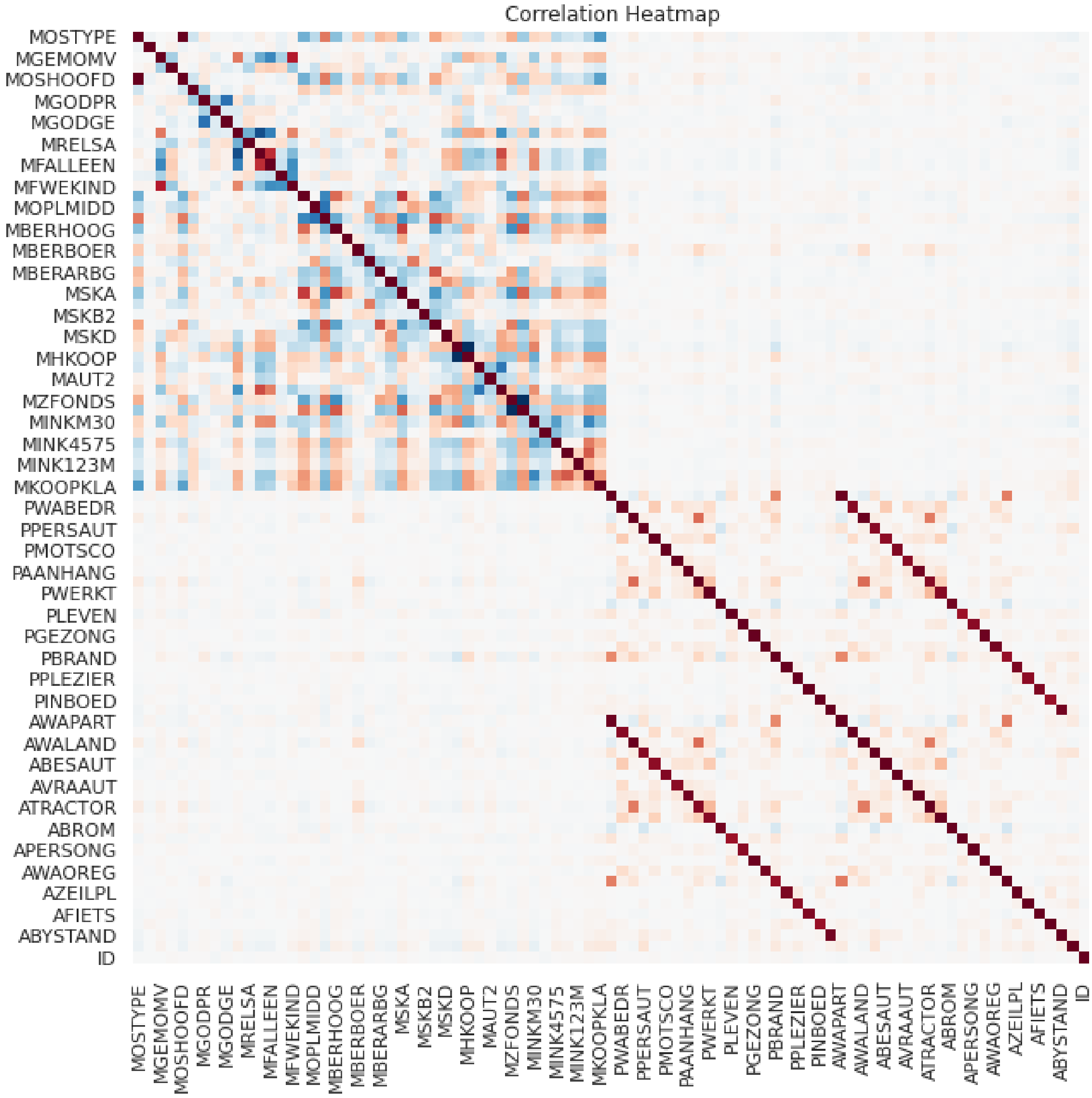


Age Group List:
1 20-30 years
2 30-40 years
3 40-50 years
4 50-60 years
5 60-70 years
6 70-80 years

In [10]:

```
# Get correlations
sns.set(rc={'figure.figsize':(10,10)},font_scale=1)
df=caravan_insurance_pdf_copy.toPandas().drop('ORIGIN',axis=1)
sns.heatmap(df.corr(),cmap='RdBu_r',cbar=None,ax=plt.axes())
plt.axes().set_title('Correlation Heatmap')
plt.show()
```

```
/opt/ibm/conda/miniconda/lib/python/site-packages/ipykernel/__main__.py:5:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version,
a new instance will always be created and returned.  Meanwhile, this warni
ng can be suppressed, and the future behavior ensured, by passing a unique
label to each axes instance.
```

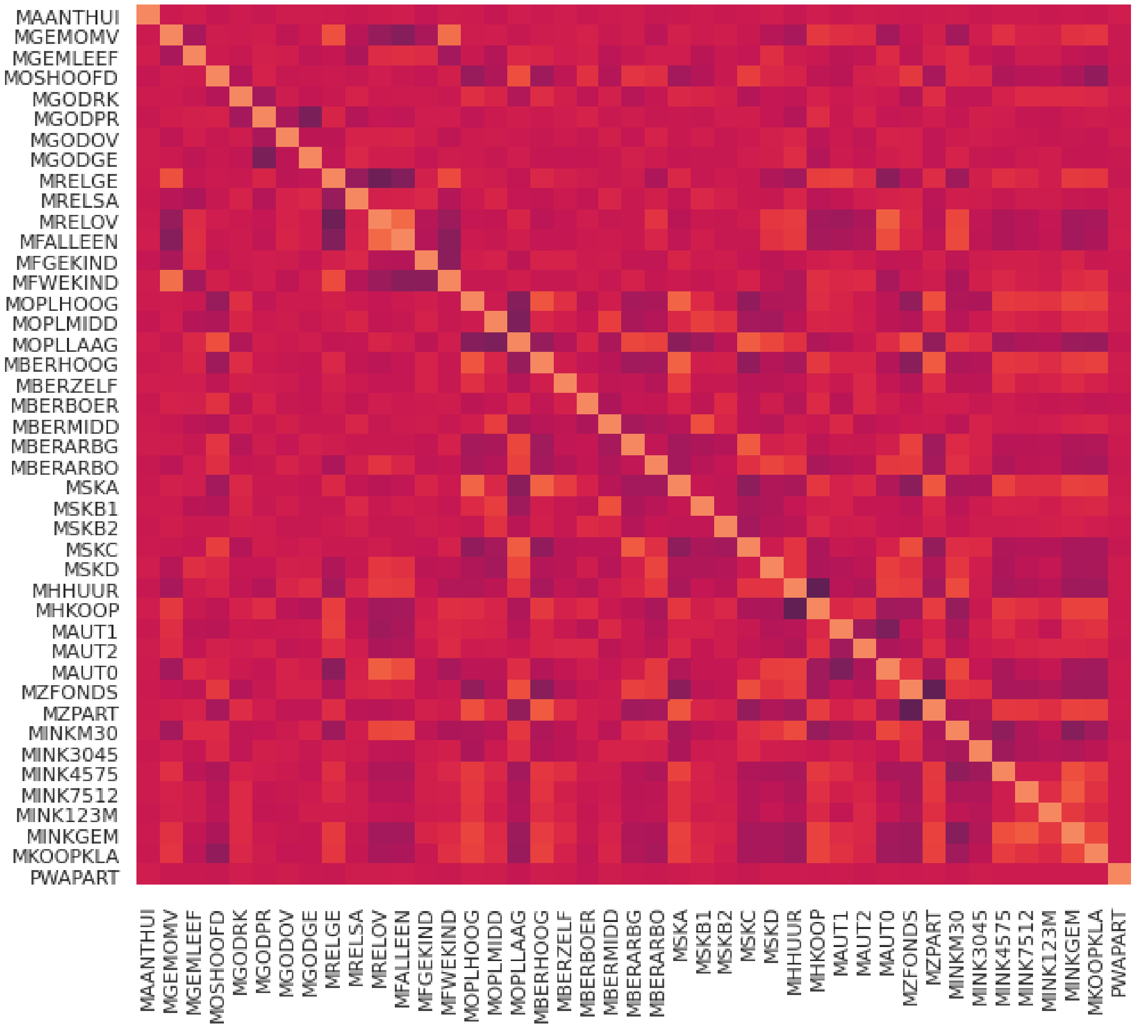


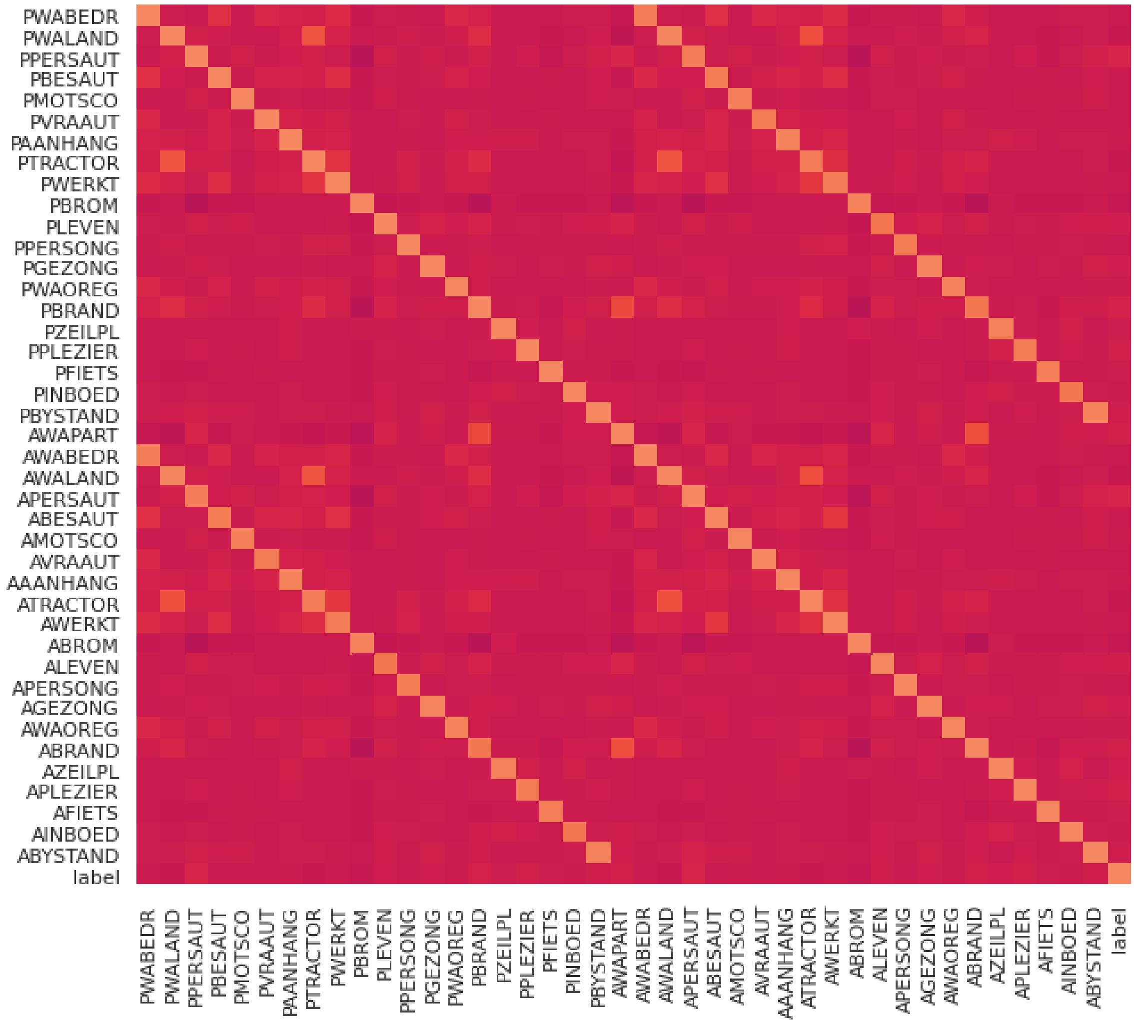
In [11]:

```
f,axs=plt.subplots(2,1,figsize=(10,20))
sns.heatmap(df.iloc[:,1:44].corr(),ax=axs[0],vmin=-2, vmax=2,cbar=None)
sns.heatmap(df.iloc[:,44:-1].corr(),ax=axs[1],vmin=-2, vmax=2,cbar=None)
```

Out[11]:

<matplotlib.axes._subplots.AxesSubplot at 0x7ff13c0cf890>


MAANTHUI
MGEMOMV
MGEMLEEF
MOSHOOFD
MGODRK
MGODPR
MGODOV
MGODGE
MRELGE
MRELSA
MRELOV
MFALLEEN
MFGEKIND
MFWEKIND
MOPLHOOG
MOPLMIDD
MOPLLAAG
MBERHOOG
MBERZELF
MBERBOER
MBERMIDD
MBERARBG
MBERARBO
MSKA
MSKB1
MSKB2
MSKC
MSKD
MHHUUR
MHKOOP
MAUT1
MAUT2
MAUT0
MZFONDS
MZPART
MINKM30
MINK3045
MINK4575
MINK7512
MINK123M
MINKGEM
MKOOPKLA
PWAPART


PWABEDR
PWALAND
PPERSAUT
PBESAUT
PMOTSCO
PVRAAUT
PAANHANG
PTRACTOR
PWERKT
PBROM
PLEVEN
PPERSONG
PGEZONG
PWAOREG
PBRAND
PZEILPL
PPLEZIER
PFIETS
PINBOED
PBYSTAND
AWAPART
AWABEDR
AWALAND
APERSAUT
ABESAUT
AMOTSCO
AVRAAUT
AAANHANG
ATRACTOR
AWERKT
ABROM
ALEVEN
APERSONG
AGEZONG
AWAOREG
ABRAND
AZEILPL
APLEZIER
AFIETS
AINBOED
ABYSTAND
label

Training/test Set Split & Imbalanced Data Processing

As data is imbalanced, we can use re-sampling techniques (over-sampling, under-sampling, combined).
Here I chose:

- Over-Sampling (SMOTE)

In [12]:

```
#Filter_VarianceThreshold=0
var = caravan_insurance_pdf.select([F.variance(col) for col in caravan_insurance_pdf.co
lumns]).toPandas() #collect()
caravan_insurance_pdf_2 = caravan_insurance_pdf.select([col for col_id, col in enumerat
e(caravan_insurance_pdf.columns) if var['var_samp('+col+')'][0]!=0])

# split the data to get training and test sets
#train_org, test = caravan_insurance_pdf_2.randomSplit([0.7, 0.3], seed=42)
# instead, we use the default training and test set
train_org=caravan_insurance_pdf_2.filter(col("ORIGIN")=="train").drop("ORIGIN")
test=caravan_insurance_pdf_2.filter(col("ORIGIN")=="test").drop("ORIGIN")
print("training set, label=1",train_org.filter(col("label")=="1").count(),"training se
t, label=0",train_org.filter(col("label")=="0").count())
print("test set, label=1",test.filter(col("label")=="1").count(),"test set, label=0",te
st.filter(col("label")=="0").count())
```

```
training set, label=1 340 training set, label=0 4880
test set, label=1 238 test set, label=0 3492
```

In [13]:

```
#SMOTE with imblearn
y_base=train_org.select('label').toPandas()
base=train_org.drop('label').toPandas()
sm_trainX , sm_trainY = SMOTE(random_state=42).fit_resample(base,y_base)
train_sm=pd.concat([sm_trainX,sm_trainY],axis=1)
#print(train_sm)
```

In [14]:

```
# simplified over-sampling method
major_df=train_org.filter(col("label")==0)
minor_df=train_org.filter(col("label")==1)
ratio=int(major_df.count()/minor_df.count())
print(major_df.count())
print(minor_df.count())
print("ratio: {}".format(ratio))
#Oversampling
a = range(ratio)
# duplicate the minority rows
oversampled_df = minor_df.withColumn("dummy", explode(array([lit(x) for x in a]))).drop
('dummy')
print(oversampled_df.filter(col("label") == 1).count())

# combine both oversampled minority rows and previous majority rows
train_oversampled = major_df.unionAll(oversampled_df)

print(train_oversampled.count())
#oversampled_df_2.take(5)

# #Undersampling
# reduced_majority_df=major_df.sample(False, 1/ratio)
# Undersampled_df=reduced_majority_df.unionAll(minor_df)
# Undersampled_df.count()
```

```
4880
340
ratio: 14
4760
9640
```

In [15]:

```
#training sets prepared in different ways, here I only trained SMOTE
train_base=train_org
# train_simpOS=train_oversampled
train_SMOTE=spark.createDataFrame(train_sm)
#print(train_SMOTE.filter(col("label") == 1).take(5))
```

Feature Engineering

I chose the target mean encoding method instead of one-hot encoding for the variable 'MOSTYPE' since we already have 86 variables and 'MOSTYPE' has 41 distinct values.
Some other technics used in building the pipeline: VectorAssembler,StandardScaler, etc.

In [16]:

```
print(train_org.columns[43:-1])
```

```
['PWAPART', 'PWABEDR', 'PWALAND', 'PPERSAUT', 'PBESAUT', 'PMOTSCO', 'PVRAA
UT', 'PAANHANG', 'PTRACTOR', 'PWERKT', 'PBROM', 'PLEVEN', 'PPERSONG', 'PGE
ZONG', 'PWAOREG', 'PBRAND', 'PZEILPL', 'PPLEZIER', 'PFIETS', 'PINBOED', 'P
BYSTAND', 'AWAPART', 'AWABEDR', 'AWALAND', 'APERSAUT', 'ABESAUT', 'AMOTSC
O', 'AVRAAUT', 'AAANHANG', 'ATRACTOR', 'AWERKT', 'ABROM', 'ALEVEN', 'APERS
ONG', 'AGEZONG', 'AWAOREG', 'ABRAND', 'AZEILPL', 'APLEZIER', 'AFIETS', 'AI
NBOED', 'ABYSTAND']
```

In [17]:

```
# # pipeline with onehot encoding
# columns_num=train_org.columns[43:-1]
# columns_cat=["MOSTYPE", "MOSHOOFD"]
# columns_all=["MOSTYPE_ONE", "MOSHOOFD_ONE","features_num_scaled"]
# encoder = OneHotEncoder(inputCols=columns_cat,
#                          outputCols=["MOSTYPE_ONE", "MOSHOOFD_ONE"])
# assembler_num = VectorAssembler(inputCols=columns_num, outputCol='features_num',handl
eInvalid = 'skip')
# scalers = StandardScaler(inputCol='features_num', outputCol='features_num_scaled')
# assembler_all = VectorAssembler(inputCols=columns_all, outputCol='features',handleInv
alid = 'skip')
# full_pipeline = Pipeline(stages=[encoder,assembler_num,scalers,assembler_all,model])
```

In [18]:

```
# Build feature engineering pipeline
# columns_num=train_org.columns[43:-1]
# columns_cat=train_org.columns[:43]
# assembler_num = VectorAssembler(inputCols=columns_num, outputCol='features_num',handl
eInvalid = 'skip')
# scalers = StandardScaler(inputCol='features_num', outputCol='features_num_scaled')
# assembler_all = VectorAssembler(inputCols=columns_all, outputCol='features',handleInv
alid = 'skip')
#selector = ChiSqSelector(fpr=0.05,featuresCol='features_scaled',outputCol="features")#
numTopFeatures=84

# process categorical variables - target mean encoding
def target_mean_encoding(df, col, target):
    """
    :param df: pyspark.sql.dataframe
        dataframe to apply target mean encoding
    :param col: str list
        list of columns to apply target encoding
    :param target: str
        target column
    :return:
        dataframe with target encoded columns
    """
    target_encoded_columns_list = []
    for c in col:
        means = df.groupby(F.col(c)).agg(F.mean(target).alias(f"{c}_mean_encoding"))
        dict_ = means.toPandas().to_dict()
        target_encoded_columns = [F.when(F.col(c) == v, encoder)
                                  for v, encoder in zip(dict_[c].values(),
                                                        dict_[f"{c}_mean_encoding"].val
ues())]
        target_encoded_columns_list.append(F.coalesce(*target_encoded_columns).alias(f"
{c}_mean_encoding"))
    return df, df.select(*df.columns, *target_encoded_columns_list)


# function apply on spark inputs
# train_base
train_target_encoded_1 = target_mean_encoding(train_base, col=['MOSTYPE', 'MOSHOOFD'],
target='label')
train_encoded_col_1=train_target_encoded_1[1].drop('MOSTYPE', 'MOSHOOFD')

#train_SMOTE
train_target_encoded_2 = target_mean_encoding(train_SMOTE, col=['MOSTYPE', 'MOSHOOFD'],
target='label')
train_encoded_col_2=train_target_encoded_2[1].drop('MOSTYPE', 'MOSHOOFD')

# since this part is not in the pipeline,
test_target_encoded = target_mean_encoding(test, col=['MOSTYPE', 'MOSHOOFD'], target='l
abel')
test_encoded_withlabel=test_target_encoded[1].drop('MOSTYPE', 'MOSHOOFD')
```

In [19]:

```
#print(train_encoded_col_2.columns)
print('numerical features:', train_encoded_col_2.columns[41:-3])
print('categorical features:', train_encoded_col_2.columns[:41]+train_encoded_col_2.col
umns[-2:])
```

```
numerical features: ['PWAPART', 'PWABEDR', 'PWALAND', 'PPERSAUT', 'PBESAU
T', 'PMOTSCO', 'PVRAAUT', 'PAANHANG', 'PTRACTOR', 'PWERKT', 'PBROM', 'PLEV
EN', 'PPERSONG', 'PGEZONG', 'PWAOREG', 'PBRAND', 'PZEILPL', 'PPLEZIER', 'P
FIETS', 'PINBOED', 'PBYSTAND', 'AWAPART', 'AWABEDR', 'AWALAND', 'APERSAU
T', 'ABESAUT', 'AMOTSCO', 'AVRAAUT', 'AAANHANG', 'ATRACTOR', 'AWERKT', 'AB
ROM', 'ALEVEN', 'APERSONG', 'AGEZONG', 'AWAOREG', 'ABRAND', 'AZEILPL', 'AP
LEZIER', 'AFIETS', 'AINBOED', 'ABYSTAND']
categorical features: ['MAANTHUI', 'MGEMOMV', 'MGEMLEEF', 'MGODRK', 'MGODP
R', 'MGODOV', 'MGODGE', 'MRELGE', 'MRELSA', 'MRELOV', 'MFALLEEN', 'MFGEKIN
D', 'MFWEKIND', 'MOPLHOOG', 'MOPLMIDD', 'MOPLLAAG', 'MBERHOOG', 'MBERZEL
F', 'MBERBOER', 'MBERMIDD', 'MBERARBG', 'MBERARBO', 'MSKA', 'MSKB1', 'MSKB
2', 'MSKC', 'MSKD', 'MHHUUR', 'MHKOOP', 'MAUT1', 'MAUT2', 'MAUT0', 'MZFOND
S', 'MZPART', 'MINKM30', 'MINK3045', 'MINK4575', 'MINK7512', 'MINK123M',
'MINKGEM', 'MKOOPKLA', 'MOSTYPE_mean_encoding', 'MOSHOOFD_mean_encoding']
```

In [20]:

```
columns_num=train_encoded_col_2.columns[41:-3]
columns_col=train_encoded_col_2.columns[:41]+train_encoded_col_2.columns[-2:]
assembler_num = VectorAssembler(inputCols=columns_num, outputCol='features_num',handleI
nvalid = 'skip')
scalers = StandardScaler(inputCol='features_num', outputCol='features_num_scaled')
assembler_col = VectorAssembler(inputCols=columns_col, outputCol='features_col',handleI
nvalid = 'skip')
columns_all=['features_col','features_num']
assembler_all = VectorAssembler(inputCols=columns_all, outputCol='features',handleInval
id = 'skip')
#selector = ChiSqSelector(fpr=0.05,featuresCol='features_scaled',outputCol="features")#
numTopFeatures=84
```

Modeling

To get better results, different classification techniques were applied.

- Logistic Regression
- Linear SVC
- Naive Bayes Classifier
- Random Forest Classifier
- FM Classifier

In [21]:

```
# if you will use tree model and chi-square selector at the same time, choose another f
unction
def fit_model(df,model, paramGrid = None):
    # Model fitting with selected model and paramgric(optional)
    # Input: model, paramgrid
    # Output: fitted model, prediction on validation set
    full_pipeline = Pipeline(stages=[assembler_num,scalers,assembler_col,assembler_all,
model])
    if paramGrid != None:
        crossval_1 = CrossValidator(estimator=full_pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=MulticlassClassificationEvaluator(),#labelCol="CARA
VAN"
                          numFolds=5)
        fitmodel = crossval_1.fit(df)
    else:

        fitmodel = full_pipeline.fit(df)

    results = fitmodel.transform(test_encoded_withlabel)

    return fitmodel, results

def fit_model_tree_withchisquare (model, paramGrid = None):
    # Model fitting with selected model and paramgric(optional)
    # Input: model, paramgrid
    # Output: fitted model, prediction on validation set
    feature_pipeline=Pipeline(stages=[assembler, scalers,selector])
    FeatureModel = feature_pipeline.fit(train_encoded_withlabel)
    train_feature_processed = FeatureModel.transform(train_encoded_withlabel)
    # convert sparse vector to dense vector
    data_modeling = train_feature_processed.select("label", "features")
    rdd = data_modeling.rdd.map(lambda x: Row(label=x[0],features=DenseVector(x[1].toAr
ray()))
                     if (len(x)>1 and hasattr(x[1], "toArray"))
                     else Row(label=None, features=DenseVector([])))
    data_modeling_Dense = sqlContext.createDataFrame(rdd)


    if paramGrid != None:
        crossval_1 = CrossValidator(estimator=model, #data was procsssed, otherwise a w
hole pipeline can be put here instead
                          estimatorParamMaps=paramGrid,
                          evaluator=MulticlassClassificationEvaluator(),
                          numFolds=5)
        fitmodel = crossval_1.fit(data_modeling_Dense)
    else:

        fitmodel = model.fit(data_modeling_Dense)

    # transform test data first

    test_feature_processed=FeatureModel.transform(test_encoded_withlabel)
    test_processed = test_feature_processed.select("label", "features")
    rdd2 = test_processed.rdd.map(lambda x: Row(label=x[0],features=DenseVector(x[1].to
Array()))
                     if (len(x)>1 and hasattr(x[1], "toArray"))
                     else Row(label=None, features=DenseVector([])))
    test_processed_Dense = sqlContext.createDataFrame(rdd2)
```

```
    results = fitmodel.transform(test_processed_Dense)

    return fitmodel, results
```

In [22]:

```
#Evaluate the model on test set
def val_evaluation_imbalanced(results,df):
    # Input: prediction results
    # Output: accuracy, precision and recall score
    predictionAndLabels = results.select(['prediction', 'label']\
                                      ).withColumn('label',col('label').cast(DoubleType
())).rdd

    metrics = MulticlassMetrics(predictionAndLabels)
    cm=metrics.confusionMatrix().toArray()


    # Use confusion matrix to calculate evaluation metrics
    # accuracy: (TP+TN)/Total Predictions
    # For class 1:
    # precision: TP/(TP + FP)
    # recall: TP/(TP + FN)
    # f1 score: 2*(Recall * Precision) / (Recall + Precision)
    accuracy=(cm[0][0]+cm[1][1])/cm.sum()
    precision_1=(cm[1][1])/(cm[0][1]+cm[1][1])
    recall_1=(cm[1][1])/(cm[1][0]+cm[1][1])
    f1 = MulticlassClassificationEvaluator().evaluate(results)
    # For class 0:
    # precision: TN/(TN + FN)
    # recall: TN/(TN + FP)
    precision_0=(cm[0][0])/(cm[1][0]+cm[0][0])
    recall_0=(cm[0][0])/(cm[0][1]+cm[0][0])
    ratio=df.filter(col("label")=="1").count()/df.count()
    precision=ratio*precision_1+(1-ratio)*precision_0
    recall=ratio*recall_1+(1-ratio)*recall_0
    return(round(f1,2), round(accuracy,2),round(precision,2),round(recall,2),cm)

# # Evaluate the model on test set
# def val_evaluation(results):
#     # Input: prediction results
#     # Output: accuracy, precision and recall score
#     predictionAndLabels = results.select(['prediction', 'label']\
#                                        ).withColumn('label',col('label').cast(DoubleTy
pe())).rdd

#     metrics = MulticlassMetrics(predictionAndLabels)
#     cm=metrics.confusionMatrix().toArray()


#     # Use confusion matrix to calculate evaluation metrics
#     # accuracy: (TP+TN)/Total Predictions
#     # precision: TP/(TP + FP)
#     # recall: TP/(TP + FN)
#     # f1 score: 2*(Recall * Precision) / (Recall + Precision)
#     accuracy=(cm[0][0]+cm[1][1])/cm.sum()
#     precision=(cm[1][1])/(cm[0][1]+cm[1][1])
#     recall=(cm[1][1])/(cm[1][0]+cm[1][1])
#     f1 = MulticlassClassificationEvaluator().evaluate(results)
#     return(round(f1,2), round(accuracy,2),round(precision,2),round(recall,2),cm)
```

In [23]:

```
# use default params to run algorithms first
lr = LogisticRegression()
lsvc = LinearSVC()
nb = NaiveBayes()
rf = RandomForestClassifier()
gbt = GBTClassifier()
fm = FMClassifier()
#models=list([lr,lsvc,nb,rf,gbt,fm])
```

In [24]:

```
print("LogisticRegression parameters:\n"+ lr.explainParams()+ "\n" )
print("*****************************************************************")
print("LinearSVC parameters:\n"+ lsvc.explainParams()+ "\n")
print("*****************************************************************")
print("NaiveBayes parameters:\n"+ nb.explainParams()+ "\n")
print("*****************************************************************")
print("RandomForest parameters:\n"+ rf.explainParams()+ "\n")
print("*****************************************************************")
print("GBT parameters:\n"+ gbt.explainParams()+ "\n")
print("*****************************************************************")
print("FMC parameters:\n"+ fm.explainParams()+ "\n")
```

```
LogisticRegression parameters:
aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alp
ha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty.
(default: 0.0)
family: The name of family which is a description of the label distributio
n to be used in the model. Supported options: auto, binomial, multinomial
(default: auto)
featuresCol: features column name. (default: features)
fitIntercept: whether to fit an intercept term. (default: True)
labelCol: label column name. (default: label)
lowerBoundsOnCoefficients: The lower bounds on coefficients if fitting und
er bound constrained optimization. The bound matrix must be compatible wit
h the shape (1, number of features) for binomial regression, or (number of
classes, number of features) for multinomial regression. (undefined)
lowerBoundsOnIntercepts: The lower bounds on intercepts if fitting under b
ound constrained optimization. The bounds vector size must beequal with 1
for binomial regression, or the number oflasses for multinomial regressio
n. (undefined)
maxIter: max number of iterations (>= 0). (default: 100)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities.
Note: Not all models output well-calibrated probability estimates! These p
robabilities should be treated as confidences, not precise probabilities.
(default: probability)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (defaul
t: rawPrediction)
regParam: regularization parameter (>= 0). (default: 0.0)
standardization: whether to standardize the training features before fitti
ng the model. (default: True)
threshold: Threshold in binary classification prediction, in range [0, 1].
If threshold and thresholds are both set, they must match.e.g. if threshol
d is p, then thresholds must be equal to [1-p, p]. (default: 0.5)
thresholds: Thresholds in multi-class classification to adjust the probabi
lity of predicting each class. Array must have length equal to the number
of classes, with values > 0, excepting that at most one value may be 0. Th
e class with largest value p/t is predicted, where p is the original proba
bility of that class and t is the class's threshold. (undefined)
tol: the convergence tolerance for iterative algorithms (>= 0). (default:
1e-06)
upperBoundsOnCoefficients: The upper bounds on coefficients if fitting und
er bound constrained optimization. The bound matrix must be compatible wit
h the shape (1, number of features) for binomial regression, or (number of
classes, number of features) for multinomial regression. (undefined)
upperBoundsOnIntercepts: The upper bounds on intercepts if fitting under b
ound constrained optimization. The bound vector size must be equal with 1
for binomial regression, or the number of classes for multinomial regressi
on. (undefined)
weightCol: weight column name. If this is not set or empty, we treat all i
nstance weights as 1.0. (undefined)

*****************************************************************
LinearSVC parameters:
aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)
featuresCol: features column name. (default: features)
fitIntercept: whether to fit an intercept term. (default: True)
labelCol: label column name. (default: label)
maxIter: max number of iterations (>= 0). (default: 100)
predictionCol: prediction column name. (default: prediction)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (defaul
t: rawPrediction)
```

```
regParam: regularization parameter (>= 0). (default: 0.0)
standardization: whether to standardize the training features before fitti
ng the model. (default: True)
threshold: The threshold in binary classification applied to the linear mo
del prediction.  This threshold can be any real number, where Inf will mak
e all predictions 0.0 and -Inf will make all predictions 1.0. (default: 0.
0)
tol: the convergence tolerance for iterative algorithms (>= 0). (default:
1e-06)
weightCol: weight column name. If this is not set or empty, we treat all i
nstance weights as 1.0. (undefined)

*****************************************************************
NaiveBayes parameters:
featuresCol: features column name. (default: features)
labelCol: label column name. (default: label)
modelType: The model type which is a string (case-sensitive). Supported op
tions: multinomial (default), bernoulli and gaussian. (default: multinomia
l)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities.
Note: Not all models output well-calibrated probability estimates! These p
robabilities should be treated as confidences, not precise probabilities.
(default: probability)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (defaul
t: rawPrediction)
smoothing: The smoothing parameter, should be >= 0, default is 1.0 (defaul
t: 1.0)
thresholds: Thresholds in multi-class classification to adjust the probabi
lity of predicting each class. Array must have length equal to the number
of classes, with values > 0, excepting that at most one value may be 0. Th
e class with largest value p/t is predicted, where p is the original proba
bility of that class and t is the class's threshold. (undefined)
weightCol: weight column name. If this is not set or empty, we treat all i
nstance weights as 1.0. (undefined)

*****************************************************************
RandomForest parameters:
bootstrap: Whether bootstrap samples are used when building trees. (defaul
t: True)
cacheNodeIds: If false, the algorithm will pass trees to executors to matc
h instances with nodes. If true, the algorithm will cache node IDs for eac
h instance. Caching can speed up training of deeper trees. Users can set h
ow often should the cache be checkpointed or disable it by setting checkpo
intInterval. (default: False)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint
(-1). E.g. 10 means that the cache will get checkpointed every 10 iteratio
ns. Note: this setting will be ignored if the checkpoint directory is not
set in the SparkContext. (default: 10)
featureSubsetStrategy: The number of features to consider for splits at ea
ch tree node. Supported options: 'auto' (choose automatically for task: If
numTrees == 1, set to 'all'. If numTrees > 1 (forest), set to 'sqrt' for c
lassification and to 'onethird' for regression), 'all' (use all features),
'onethird' (use 1/3 of the features), 'sqrt' (use sqrt(number of feature
s)), 'log2' (use log2(number of features)), 'n' (when n is in the range
(0, 1.0], use n * number of features. When n is in the range (1, number of
features), use n features). default = 'auto' (default: auto)
featuresCol: features column name. (default: features)
impurity: Criterion used for information gain calculation (case-insensitiv
e). Supported options: entropy, gini (default: gini)
labelCol: label column name. (default: label)
```

```
leafCol: Leaf indices column name. Predicted leaf index of each instance i
n each tree by preorder. (default: )
maxBins: Max number of bins for discretizing continuous features.  Must be
>=2 and >= number of categories for any categorical feature. (default: 32)
maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf nod
e; depth 1 means 1 internal node + 2 leaf nodes. (default: 5)
maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. If
too small, then 1 node will be split per iteration, and its aggregates may
exceed this size. (default: 256)
minInfoGain: Minimum information gain for a split to be considered at a tr
ee node. (default: 0.0)
minInstancesPerNode: Minimum number of instances each child must have afte
r split. If a split causes the left or right child to have fewer than minI
nstancesPerNode, the split will be discarded as invalid. Should be >= 1.
(default: 1)
minWeightFractionPerNode: Minimum fraction of the weighted sample count th
at each child must have after split. If a split causes the fraction of the
total weight in the left or right child to be less than minWeightFractionP
erNode, the split will be discarded as invalid. Should be in interval [0.
0, 0.5). (default: 0.0)
numTrees: Number of trees to train (>= 1). (default: 20)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities.
Note: Not all models output well-calibrated probability estimates! These p
robabilities should be treated as confidences, not precise probabilities.
(default: probability)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (defaul
t: rawPrediction)
seed: random seed. (default: -3233801645059874302)
subsamplingRate: Fraction of the training data used for learning each deci
sion tree, in range (0, 1]. (default: 1.0)
thresholds: Thresholds in multi-class classification to adjust the probabi
lity of predicting each class. Array must have length equal to the number
of classes, with values > 0, excepting that at most one value may be 0. Th
e class with largest value p/t is predicted, where p is the original proba
bility of that class and t is the class's threshold. (undefined)
weightCol: weight column name. If this is not set or empty, we treat all i
nstance weights as 1.0. (undefined)

*****************************************************************
GBT parameters:
cacheNodeIds: If false, the algorithm will pass trees to executors to matc
h instances with nodes. If true, the algorithm will cache node IDs for eac
h instance. Caching can speed up training of deeper trees. Users can set h
ow often should the cache be checkpointed or disable it by setting checkpo
intInterval. (default: False)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint
(-1). E.g. 10 means that the cache will get checkpointed every 10 iteratio
ns. Note: this setting will be ignored if the checkpoint directory is not
set in the SparkContext. (default: 10)
featureSubsetStrategy: The number of features to consider for splits at ea
ch tree node. Supported options: 'auto' (choose automatically for task: If
numTrees == 1, set to 'all'. If numTrees > 1 (forest), set to 'sqrt' for c
lassification and to 'onethird' for regression), 'all' (use all features),
'onethird' (use 1/3 of the features), 'sqrt' (use sqrt(number of feature
s)), 'log2' (use log2(number of features)), 'n' (when n is in the range
(0, 1.0], use n * number of features. When n is in the range (1, number of
features), use n features). default = 'auto' (default: all)
featuresCol: features column name. (default: features)
impurity: Criterion used for information gain calculation (case-insensitiv
e). Supported options: variance (default: variance)
```

```
labelCol: label column name. (default: label)
leafCol: Leaf indices column name. Predicted leaf index of each instance i
n each tree by preorder. (default: )
lossType: Loss function which GBT tries to minimize (case-insensitive). Su
pported options: logistic (default: logistic)
maxBins: Max number of bins for discretizing continuous features.  Must be
>=2 and >= number of categories for any categorical feature. (default: 32)
maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf nod
e; depth 1 means 1 internal node + 2 leaf nodes. (default: 5)
maxIter: max number of iterations (>= 0). (default: 20)
maxMemoryInMB: Maximum memory in MB allocated to histogram aggregation. If
too small, then 1 node will be split per iteration, and its aggregates may
exceed this size. (default: 256)
minInfoGain: Minimum information gain for a split to be considered at a tr
ee node. (default: 0.0)
minInstancesPerNode: Minimum number of instances each child must have afte
r split. If a split causes the left or right child to have fewer than minI
nstancesPerNode, the split will be discarded as invalid. Should be >= 1.
(default: 1)
minWeightFractionPerNode: Minimum fraction of the weighted sample count th
at each child must have after split. If a split causes the fraction of the
total weight in the left or right child to be less than minWeightFractionP
erNode, the split will be discarded as invalid. Should be in interval [0.
0, 0.5). (default: 0.0)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities.
Note: Not all models output well-calibrated probability estimates! These p
robabilities should be treated as confidences, not precise probabilities.
(default: probability)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (defaul
t: rawPrediction)
seed: random seed. (default: -4139645477772115438)
stepSize: Step size (a.k.a. learning rate) in interval (0, 1] for shrinkin
g the contribution of each estimator. (default: 0.1)
subsamplingRate: Fraction of the training data used for learning each deci
sion tree, in range (0, 1]. (default: 1.0)
thresholds: Thresholds in multi-class classification to adjust the probabi
lity of predicting each class. Array must have length equal to the number
of classes, with values > 0, excepting that at most one value may be 0. Th
e class with largest value p/t is predicted, where p is the original proba
bility of that class and t is the class's threshold. (undefined)
validationIndicatorCol: name of the column that indicates whether each row
is for training or for validation. False indicates training; true indicate
s validation. (undefined)
validationTol: Threshold for stopping early when fit with validation is us
ed. If the error rate on the validation input changes by less than the val
idationTol, then learning will stop early (before `maxIter`). This paramet
er is ignored when fit without validation is used. (default: 0.01)
weightCol: weight column name. If this is not set or empty, we treat all i
nstance weights as 1.0. (undefined)

*****************************************************************
FMC parameters:
factorSize: Dimensionality of the factor vectors, which are used to get pa
irwise interactions between variables (default: 8)
featuresCol: features column name. (default: features)
fitIntercept: whether to fit an intercept term. (default: True)
fitLinear: whether to fit linear term (aka 1-way term) (default: True)
initStd: standard deviation of initial coefficients (default: 0.01)
labelCol: label column name. (default: label)
maxIter: max number of iterations (>= 0). (default: 100)
```

```
miniBatchFraction: fraction of the input data set that should be used for
one iteration of gradient descent (default: 1.0)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities.
Note: Not all models output well-calibrated probability estimates! These p
robabilities should be treated as confidences, not precise probabilities.
(default: probability)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (defaul
t: rawPrediction)
regParam: regularization parameter (>= 0). (default: 0.0)
seed: random seed. (default: -1289994653638339637)
solver: The solver algorithm for optimization. Supported options: gd, adam
W. (Default adamW) (default: adamW)
stepSize: Step size to be used for each iteration of optimization (>= 0).
(default: 1.0)
thresholds: Thresholds in multi-class classification to adjust the probabi
lity of predicting each class. Array must have length equal to the number
of classes, with values > 0, excepting that at most one value may be 0. Th
e class with largest value p/t is predicted, where p is the original proba
bility of that class and t is the class's threshold. (undefined)
tol: the convergence tolerance for iterative algorithms (>= 0). (default:
1e-06)
```

In [25]:

```
lrmodel_smote,lrresults_smote=fit_model(train_encoded_col_2,lr,None)
print("LogisticRegressionClassifier_SMOTE: f1_score, accuracy,precision,recall", val_ev
aluation_imbalanced(lrresults_smote,test_encoded_withlabel)[0:4])
lsvcmodel_smote,lsvcresults_smote=fit_model(train_encoded_col_2,lsvc,None)
print("LinearSVC_SMOTE: f1_score, accuracy,precision,recall", val_evaluation_imbalanced
(lsvcresults_smote,test_encoded_withlabel)[0:4])
nbmodel_smote,nbresults_smote=fit_model(train_encoded_col_2,nb,None)
print("NaiveBayesClassifier_SMOTE: f1_score, accuracy,precision,recall", val_evaluation
_imbalanced(nbresults_smote,test_encoded_withlabel)[0:4])
rfmodel_smote,rfresults_smote=fit_model(train_encoded_col_2,rf,None)
print("RandomForestClassifier_SMOTE: f1_score, accuracy,precision,recall", val_evaluati
on_imbalanced(rfresults_smote,test_encoded_withlabel)[0:4])
#print("confusion metrics",val_evaluation(rfresults_smote,test)[4])
gbtmodel_smote,gbtresults_smote=fit_model(train_encoded_col_2,gbt,None)
print("GBTClassifier_SMOTE: f1_score, accuracy,precision,recall", val_evaluation_imbala
nced(gbtresults_smote,test_encoded_withlabel)[0:4])
fmmodel_smote,fmresults_smote=fit_model(train_encoded_col_2,fm,None)
print("FMClassifierClassifier_SMOTE: f1_score, accuracy,precision,recall", val_evaluati
on_imbalanced(fmresults_smote,test_encoded_withlabel)[0:4])
```

```
LogisticRegressionClassifier_SMOTE: f1_score, accuracy,precision,recall
(0.9, 0.93, 0.88, 0.93)
LinearSVC_SMOTE: f1_score, accuracy,precision,recall (0.9, 0.94, 0.88, 0.9
4)
NaiveBayesClassifier_SMOTE: f1_score, accuracy,precision,recall (0.77, 0.6
9, 0.91, 0.69)
RandomForestClassifier_SMOTE: f1_score, accuracy,precision,recall (0.91,
0.92, 0.89, 0.92)
GBTClassifier_SMOTE: f1_score, accuracy,precision,recall (0.9, 0.91, 0.88,
0.91)
FMClassifierClassifier_SMOTE: f1_score, accuracy,precision,recall (0.83,
0.78, 0.9, 0.78)
```

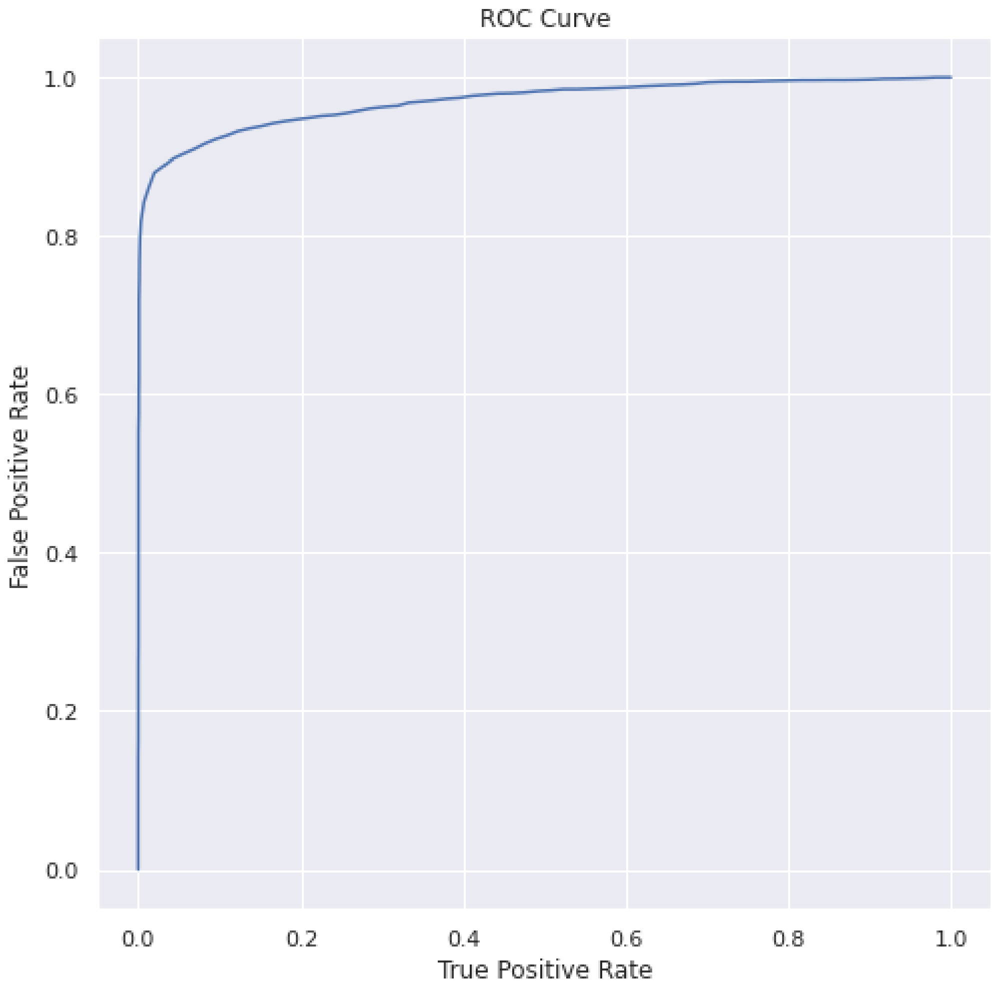
Model Tuning

Based on the performance for different algorithms, Logistics Regression and Random Forest were selected for further investigation.

Logistic Regression

In [26]:

```
# lr model ROC curve
trainingSummary = lrmodel_smote.stages[-1].summary
roc = trainingSummary.roc.toPandas()
plt.gcf().set_size_inches(8, 8)
plt.plot(roc['FPR'],roc['TPR'])
plt.ylabel('False Positive Rate')
plt.xlabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
print('Training set area Under ROC: ' + str(trainingSummary.areaUnderROC))
```




```
Training set area Under ROC: 0.9695633734211234
```

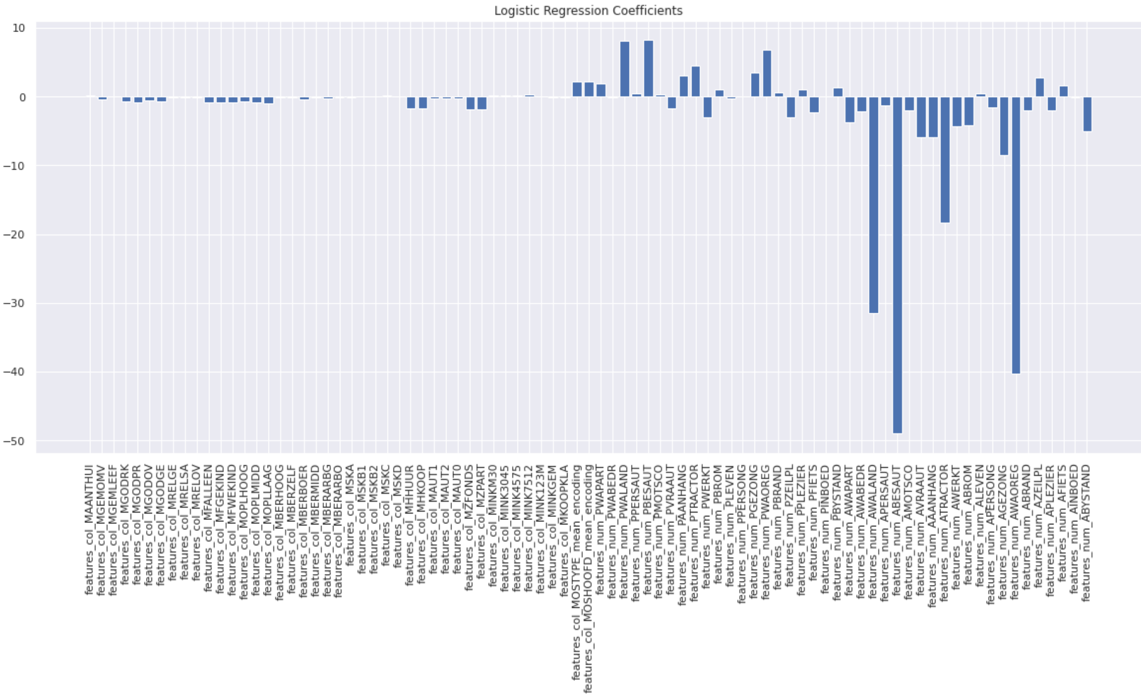
In [27]:

```
# Extract feature names from the original data
dict_feats = lrresults_smote.schema['features'].metadata['ml_attr']['attrs']['numeric']
list_feats = np.array([x['name'] for x in dict_feats])
print(list_feats )
```

```
['features_col_MAANTHUI' 'features_col_MGEMOMV' 'features_col_MGEMLEEF'
 'features_col_MGODRK' 'features_col_MGODPR' 'features_col_MGODOV'
 'features_col_MGODGE' 'features_col_MRELGE' 'features_col_MRELSA'
 'features_col_MRELOV' 'features_col_MFALLEEN' 'features_col_MFGEKIND'
 'features_col_MFWEKIND' 'features_col_MOPLHOOG' 'features_col_MOPLMIDD'
 'features_col_MOPLLAAG' 'features_col_MBERHOOG' 'features_col_MBERZELF'
 'features_col_MBERBOER' 'features_col_MBERMIDD' 'features_col_MBERARBG'
 'features_col_MBERARBO' 'features_col_MSKA' 'features_col_MSKB1'
 'features_col_MSKB2' 'features_col_MSKC' 'features_col_MSKD'
 'features_col_MHHUUR' 'features_col_MHKOOP' 'features_col_MAUT1'
 'features_col_MAUT2' 'features_col_MAUT0' 'features_col_MZFONDS'
 'features_col_MZPART' 'features_col_MINKM30' 'features_col_MINK3045'
 'features_col_MINK4575' 'features_col_MINK7512' 'features_col_MINK123M'
 'features_col_MINKGEM' 'features_col_MKOOPKLA'
 'features_col_MOSTYPE_mean_encoding'
 'features_col_MOSHOOFD_mean_encoding' 'features_num_PWAPART'
 'features_num_PWABEDR' 'features_num_PWALAND' 'features_num_PPERSAUT'
 'features_num_PBESAUT' 'features_num_PMOTSCO' 'features_num_PVRAAUT'
 'features_num_PAANHANG' 'features_num_PTRACTOR' 'features_num_PWERKT'
 'features_num_PBROM' 'features_num_PLEVEN' 'features_num_PPERSONG'
 'features_num_PGEZONG' 'features_num_PWAOREG' 'features_num_PBRAND'
 'features_num_PZEILPL' 'features_num_PPLEZIER' 'features_num_PFIETS'
 'features_num_PINBOED' 'features_num_PBYSTAND' 'features_num_AWAPART'
 'features_num_AWABEDR' 'features_num_AWALAND' 'features_num_APERSAUT'
 'features_num_ABESAUT' 'features_num_AMOTSCO' 'features_num_AVRAAUT'
 'features_num_AAANHANG' 'features_num_ATRACTOR' 'features_num_AWERKT'
 'features_num_ABROM' 'features_num_ALEVEN' 'features_num_APERSONG'
 'features_num_AGEZONG' 'features_num_AWAOREG' 'features_num_ABRAND'
 'features_num_AZEILPL' 'features_num_APLEZIER' 'features_num_AFIETS'
 'features_num_AINBOED' 'features_num_ABYSTAND']
```

In [28]:

```
# Get coefficients
lr_corr = lrmodel_smote.stages[-1].coefficients
plt.gcf().set_size_inches(20, 8)
plt.bar(list_feats,lr_corr)
plt.xticks(rotation='vertical')
plt.title('Logistic Regression Coefficients')
plt.show()
```



In [29]:

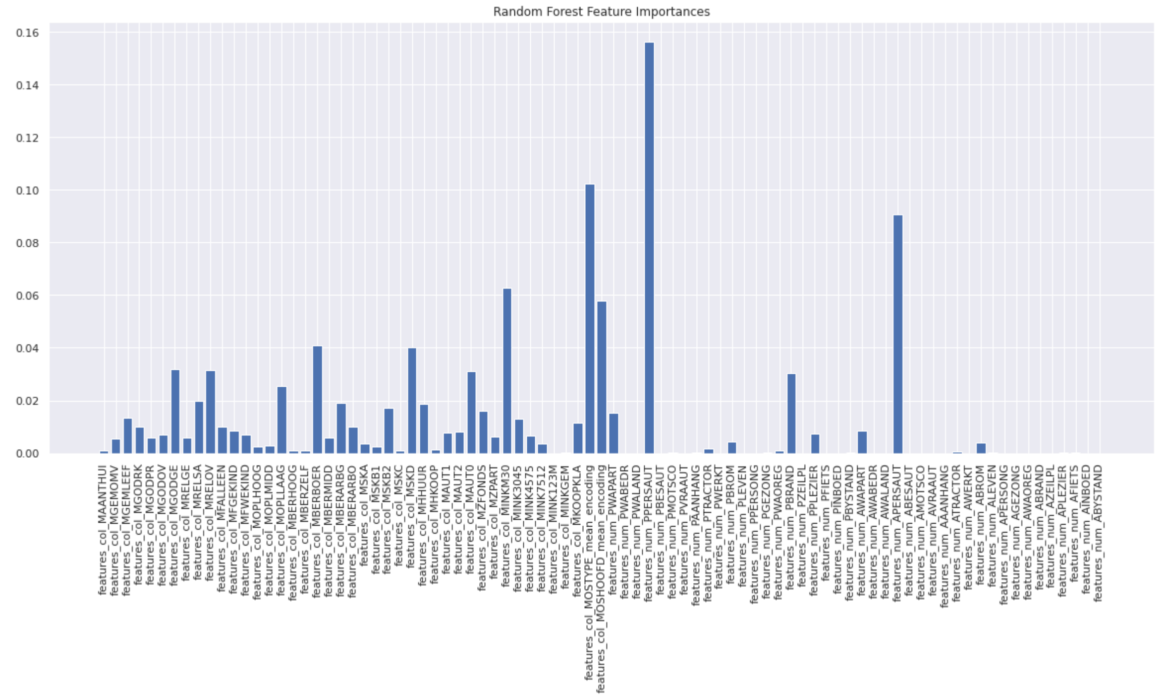
```
# Get best threshold for current lr model
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-Measure)').head()
bestThreshold = fMeasure.where(fMeasure['F-Measure'] == maxFMeasure['max(F-Measure)'])
 \
    .select('threshold').head()['threshold']
print(bestThreshold)
```

```
0.6196875477178173
```

Random Forest

In [30]:

```
# Extract feature importance from rfmodel
featImportances = np.array(rfmodel_smote.stages[-1].featureImportances)
plt.gcf().set_size_inches(20, 8)
plt.bar(list_feats,featImportances)
plt.xticks(rotation='vertical')
plt.title('Random Forest Feature Importances')
plt.show()
```



In [31]:

```
# tune the hyper-parameters for the RF model
# tuning order: n_estimators, max_leaf_models/max_depth/min_samle_split and min_sample_
leaf , tune 'subsmaple' & 'learning rate'
rf_paramGrid = ParamGridBuilder().addGrid(rf.numTrees, [10, 30, 50,70]).addGrid(rf.maxD
epth, [5,10,20]).build()
rfmodel_smote_tuned,rfresults_smote_tuned=fit_model(train_encoded_col_2,rf,rf_paramGrid
)
```

In [32]:

```
bestPipeline = rfmodel_smote_tuned.bestModel
bestRFModel = bestPipeline.stages[-1]
bestParams = bestRFModel.extractParamMap()
print(bestParams) #max depth 20, numtrees 30
```

```
{Param(parent='RandomForestClassifier_b9428446742f', name='bootstrap', doc
='Whether bootstrap samples are used when building trees.'): True, Param(p
arent='RandomForestClassifier_b9428446742f', name='cacheNodeIds', doc='If
false, the algorithm will pass trees to executors to match instances with
nodes. If true, the algorithm will cache node IDs for each instance. Cachi
ng can speed up training of deeper trees. Users can set how often should t
he cache be checkpointed or disable it by setting checkpointInterval.'): F
alse, Param(parent='RandomForestClassifier_b9428446742f', name='checkpoint
Interval', doc='set checkpoint interval (>= 1) or disable checkpoint (-1).
E.g. 10 means that the cache will get checkpointed every 10 iterations. No
te: this setting will be ignored if the checkpoint directory is not set in
the SparkContext.'): 10, Param(parent='RandomForestClassifier_b9428446742
f', name='featureSubsetStrategy', doc="The number of features to consider
for splits at each tree node. Supported options: 'auto' (choose automatica
lly for task: If numTrees == 1, set to 'all'. If numTrees > 1 (forest), se
t to 'sqrt' for classification and to 'onethird' for regression), 'all' (u
se all features), 'onethird' (use 1/3 of the features), 'sqrt' (use sqrt(n
umber of features)), 'log2' (use log2(number of features)), 'n' (when n is
in the range (0, 1.0], use n * number of features. When n is in the range
(1, number of features), use n features). default = 'auto'"): 'auto', Para
m(parent='RandomForestClassifier_b9428446742f', name='featuresCol', doc='f
eatures column name.'): 'features', Param(parent='RandomForestClassifier_b
9428446742f', name='impurity', doc='Criterion used for information gain ca
lculation (case-insensitive). Supported options: entropy, gini'): 'gini',
Param(parent='RandomForestClassifier_b9428446742f', name='labelCol', doc
='label column name.'): 'label', Param(parent='RandomForestClassifier_b942
8446742f', name='leafCol', doc='Leaf indices column name. Predicted leaf i
ndex of each instance in each tree by preorder.'): '', Param(parent='Rando
mForestClassifier_b9428446742f', name='maxBins', doc='Max number of bins f
or discretizing continuous features.  Must be >=2 and >= number of categor
ies for any categorical feature.'): 32, Param(parent='RandomForestClassifi
er_b9428446742f', name='maxDepth', doc='Maximum depth of the tree. (>= 0)
E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2 leaf no
des.'): 20, Param(parent='RandomForestClassifier_b9428446742f', name='maxM
emoryInMB', doc='Maximum memory in MB allocated to histogram aggregation.
If too small, then 1 node will be split per iteration, and its aggregates
may exceed this size.'): 256, Param(parent='RandomForestClassifier_b942844
6742f', name='minInfoGain', doc='Minimum information gain for a split to b
e considered at a tree node.'): 0.0, Param(parent='RandomForestClassifier_
b9428446742f', name='minInstancesPerNode', doc='Minimum number of instance
s each child must have after split. If a split causes the left or right ch
ild to have fewer than minInstancesPerNode, the split will be discarded as
invalid. Should be >= 1.'): 1, Param(parent='RandomForestClassifier_b94284
46742f', name='minWeightFractionPerNode', doc='Minimum fraction of the wei
ghted sample count that each child must have after split. If a split cause
s the fraction of the total weight in the left or right child to be less t
han minWeightFractionPerNode, the split will be discarded as invalid. Shou
ld be in interval [0.0, 0.5).'): 0.0, Param(parent='RandomForestClassifier
_b9428446742f', name='numTrees', doc='Number of trees to train (>= 1).'):
70, Param(parent='RandomForestClassifier_b9428446742f', name='predictionCo
l', doc='prediction column name.'): 'prediction', Param(parent='RandomFore
stClassifier_b9428446742f', name='probabilityCol', doc='Column name for pr
edicted class conditional probabilities. Note: Not all models output well-
calibrated probability estimates! These probabilities should be treated as
confidences, not precise probabilities.'): 'probability', Param(parent='Ra
ndomForestClassifier_b9428446742f', name='rawPredictionCol', doc='raw pred
iction (a.k.a. confidence) column name.'): 'rawPrediction', Param(parent
='RandomForestClassifier_b9428446742f', name='seed', doc='random seed.'):
-3233801645059874302, Param(parent='RandomForestClassifier_b9428446742f',
name='subsamplingRate', doc='Fraction of the training data used for learni
ng each decision tree, in range (0, 1].'): 1.0}
```

In [33]:

```
rf_best = RandomForestClassifier(numTrees=30, maxDepth=20)
rfmodel_smote_tuned,rfresults_smote_tuned=fit_model(train_encoded_col_2,rf_best,None)
print("RandomForestClassifier_SMOTE: f1_score, accuracy,precision,recall", val_evaluati
on_imbalanced(rfresults_smote_tuned,test_encoded_withlabel)[0:4])
```

```
RandomForestClassifier_SMOTE: f1_score, accuracy,precision,recall (0.91,
0.93, 0.89, 0.93)
```

Some ideas for further model improvement:

Use a combined re-smapling method - SMOTE & TomekLinks
Use algorithms with the base estimator (e.g.Easy Ensemble Classifier, RUS Boost Classifier)
Train the model with different feature selection methods (e.g. variance threshold filter, feature selection based on the feature importance in LR/RF model)
Tune the model in a more refined way (e.g use the range() function instead of a given list)