SMOTE (using K-means)

Predicting Airline Delays

W261 Spring 2020

Presentation Date: April 16th, 2020

Team 20: Diana Iftimie, Shaji K Kunjumohamed, Navya Sandadi & Shobha Sankar

Load the data

In [3]:

```
# Load the data into dataframe
airlines = spark.read.option("header", "true").parquet(f"dbfs:/mnt/mids-w2
```

In [4]:

```
# Check number of records
print("Number of rows in original dataset:", airlines.count())
```

```
Number of rows in original dataset: 31746841
```

Clean Data

In [6]:

```
# Remove entries where diverted = 1, cancelled = 1, dep_delay = Null, and
airlines = airlines.where('DIVERTED != 1') \
                    .where('CANCELLED != 1') \
                    .filter(airlines['DEP_DELAY'].isNotNull())

print("Number of rows in cleaned dataset:", airlines.count())
```

```
Number of rows in cleaned dataset: 31174076
```

Import Dependencies

In [8]:

```
from pyspark.sql import functions as F
from pyspark.sql.types import StructType, StructField, StringType
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StringIndexer
from pyspark.sql.functions import broadcast
from pyspark.ml.linalg import DenseVector
from pyspark.ml import Pipeline
from pyspark.sql import Row


import math
import random
```

Create Outcome Variable

DEP_DEL30

In [10]:

```
# Generate outcome variable
def CreateNewDepDelayOutcome(data, thresholds):
  for threshold in thresholds:
    data = data.withColumn('DEP_DEL' + str(threshold), (data['DEP_DELAY']
  return data

airlines = CreateNewDepDelayOutcome(airlines, [30])
```

Feature Selection

In [12]:

```
outcomeName = 'DEP_DEL30'
numFeatureNames = ['YEAR', 'MONTH', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'CRS_DE
catFeatureNames = ['OP_UNIQUE_CARRIER', 'ORIGIN', 'DEST']
joiningFeatures = ['FL_DATE'] # Features needed to join with the holidays

airlines = airlines.select([outcomeName] + numFeatureNames + catFeatureNam
```

Split the data into train and test set

In [14]:

```
# Helper function to split the dataset into train, val, test
def SplitDataset(airlines):
  # Split airlines data into train, dev, test
  test = airlines.where('Year = 2019') # held out
  train, val = airlines.where('Year != 2019').randomSplit([7.0, 1.0], 6)

  # Select a mini subset for the training dataset (~2000 records)
  mini_train = train.sample(fraction=0.0001, seed=6)

  print("mini_train size = " + str(mini_train.count()))
  print("train size = " + str(train.count()))
  print("val size = " + str(val.count()))
  print("test size = " + str(test.count()))

  return (mini_train, train, val, test)

mini_train, train, val, test = SplitDataset(airlines)
```

```
mini_train size = 2071
train size = 20915342
val size = 2990502
test size = 7268232
```

EDA for data imbalance

In [16]:

```
# Count of flights delayed vs. not delayed in training dataset
display(train.groupby('DEP_DEL30').count())
```

DEP_DEL30	count
1	2389667

DEP_DEL30	count
0	18525675

Vectorize the training dataset

In [18]:

```
# String Indexer for categorical variables
indexers = [StringIndexer(inputCol=f, outputCol=f+"_idx", handleInvalid="k
pipeline = Pipeline(stages=indexers)
indexed = pipeline.fit(train).transform(train)
```

In [19]:

```
# Prep Vector assembler
va = VectorAssembler(inputCols = numFeatureNames + [f + "_idx" for f in ca

# Build a pipeline
pipeline = Pipeline(stages= indexers + [va])
pipelineModel = pipeline.fit(train)

# Vectorize
pos_vectorized = pipelineModel.transform(train)
vectorized = pos_vectorized.select('features', outcomeName).withColumn('la
```

Filter the minority data set and convert into feature vector

In [21]:

```
# Filter the minority data
minority_data = vectorized[vectorized.label == 1]

# Select the feature vectors of minority data
featureVect = minority_data.select('features')
```

K-means

In [23]:

```
from pyspark.ml.clustering import KMeans

# Train a k-means model on minority feature vectors
kmeans = KMeans().setK(1000).setSeed(1)
model = kmeans.fit(featureVect)
predict = model.transform(featureVect)
```

In [24]:

```
# Visualize the distribution of data points in each cluster
display(predict.groupBy('prediction').count().orderBy('prediction'))
```

prediction	count
0	3115
1	2188

prediction	count
2	2233
3	602
4	1574
5	1716
6	1967
7	1013
8	1218

In [25]:

```
# Re-order the columns in the dataframe
predict = predict.select(['prediction','features'])
predict.show(10)
```

```
+----------+--------------------+
|prediction|            features|
+----------+--------------------+
|        65|[2018.0,6.0,1.0,5...|
|       421|[2018.0,6.0,1.0,5...|
|        65|[2018.0,6.0,1.0,5...|
|        72|[2018.0,6.0,1.0,5...|
|       253|[2018.0,6.0,1.0,5...|
|        72|[2018.0,6.0,1.0,5...|
|        72|[2018.0,6.0,1.0,5...|
|       421|[2018.0,6.0,1.0,5...|
|       421|[2018.0,6.0,1.0,5...|
|         3|[2018.0,6.0,1.0,5...|
+----------+--------------------+
only showing top 10 rows
```

SMOTE

In [27]:

```
# HELPER FUNCTIONS

# Calculate the Euclidean distance between two feature vectors
def euclidean_distance(row1, row2):
        distance = 0.0
        for i in range(len(row1)-1):
                distance += (row1[i] - row2[i])**2
        return math.sqrt(distance)


# Locate the nearest neighbors
def get_neighbors(train, test_row, num_neighbors):
        distances = list()
        for train_row in train:
                dist = euclidean_distance(test_row, train_row)
                distances.append((train_row, dist))
        distances.sort(key=lambda tup: tup[1])
        neighbors = list()
```

```
        for i in range(num_neighbors):
                neighbors.append(distances[i+1][0])
        return neighbors


# Generate synthetic records
def synthetic(list1, list2):
    synthetic_records = []
    for i in range(len(list1)):
      synthetic_records.append(round(list1[i] + ((list2[i]-list1[i])*rando
    return synthetic_records
```

In [28]:

```
# Convert the k-means predictions dataframe into rdd, find nearest neighbo
smote_rdd = predict.rdd.map(lambda x: (x[0], [list(x[1])])) \
                       .reduceByKey(lambda x,y: x+y) \
                       .flatMap(lambda x: [(n, get_neighbors(x[1], n, 7))
                       .flatMap(lambda x: [synthetic(x[0],n) for n in x[1]
                       .map(lambda x: Row(features = DenseVector(x), label
                       .cache()
```

In [29]:

```
# Convert the synthetic data into a dataframe
augmentedData_DF = smote_rdd.toDF()
```

In [30]:

```
# Combine the original dataset with the synthetic data
smote_data = vectorized.unionAll(augmentedData_DF)
```

```
/databricks/spark/python/pyspark/sql/dataframe.py:1503: DeprecationWarni
ng: Deprecated in 2.0, use union instead.
  warnings.warn("Deprecated in 2.0, use union instead.", DeprecationWarn
ing)
```

In [31]:

```
# EDA of data balance after applying SMOTE
display(smote_data.groupBy('label').count())
```

label	count
0	18529111
1	19117147

Save balanced dataset as columns & reverse string indexing

Reverse Vector Assembler

In [34]:

```
# Reverse Vector Assembler
from pyspark.ml.linalg import Vectors

def vectorToDF(df):
```

```
    def extract(row):
        return (row.label, ) + tuple(row.features.toArray().tolist())

    extracted_df = df.rdd.map(extract).toDF(['label'])

    # Rename Columns
    extracted_df = extracted_df.withColumnRenamed("label","DEP_DEL30") \
                               .withColumnRenamed("_2","YEAR") \
                               .withColumnRenamed("_3","MONTH") \
                               .withColumnRenamed("_4","DAY_OF_MONTH") \
                               .withColumnRenamed("_5","DAY_OF_WEEK") \
                               .withColumnRenamed("_6","CRS_DEP_TIME") \
                               .withColumnRenamed("_7","CRS_ARR_TIME") \
                               .withColumnRenamed("_8","CRS_ELAPSED_TIME") \
                               .withColumnRenamed("_9","DISTANCE") \
                               .withColumnRenamed("_10","DISTANCE_GROUP") \
                               .withColumnRenamed("_11","OP_UNIQUE_CARRIER_i
                               .withColumnRenamed("_12","ORIGIN_idx") \
                               .withColumnRenamed("_13","DEST_idx") \

    return extracted_df
```

In [35]:

```
smoted_train_cols = vectorToDF(smote_data)
smoted_train_cols.show(5)
```

```
+---------+------+-----+------------+-----------+------------+----------
--+----------------+--------+--------------+---------------------+------
----+--------+
DEP_DEL30|  YEAR|MONTH|DAY_OF_MONTH|DAY_OF_WEEK|CRS_DEP_TIME|CRS_ARR_TIM
E|CRS_ELAPSED_TIME|DISTANCE|DISTANCE_GROUP|OP_UNIQUE_CARRIER_idx|ORIGIN_
idx|DEST_idx|
+---------+------+-----+------------+-----------+------------+----------
--+----------------+--------+--------------+---------------------+------
----+--------+
        0|2018.0|  7.0|         1.0|        7.0|         3.0|      536.
0|           213.0|  1558.0|           7.0|                  2.0|
3.0|    23.0|
        0|2018.0|  7.0|         1.0|        7.0|         3.0|      618.
0|           255.0|  1846.0|           8.0|                  2.0|
5.0|     1.0|
        0|2018.0|  7.0|         1.0|        7.0|        10.0|      508.
0|           178.0|  1222.0|           5.0|                  8.0|
7.0|     8.0|
        0|2018.0|  7.0|         1.0|        7.0|        10.0|      736.
0|           266.0|  1972.0|           8.0|                  2.0|
6.0|    25.0|
        0|2018.0|  7.0|         1.0|        7.0|        13.0|      812.
0|           299.0|  2296.0|          10.0|                  2.0|
5.0|     9.0|
+---------+------+-----+------------+-----------+------------+----------
--+----------------+--------+--------------+---------------------+------
----+--------+
```

```
only showing top 5 rows
```

Reverse StringIndexer

In [37]:

```
# Create lookup table for OP_UNIQUE_CARRIER
carrier_index_lookup = indexed['OP_UNIQUE_CARRIER', 'OP_UNIQUE_CARRIER_idx
display(carrier_index_lookup.orderBy('OP_UNIQUE_CARRIER_idx'))
```

OP_UNIQUE_CARRIER	OP_UNIQUE_CARRIER_idx
WN	0.0
DL	1.0
AA	2.0
OO	3.0
UA	4.0
EV	5.0
B6	6.0
AS	7.0
NK	8.0
MQ	9.0

In [38]:

```
# Create lookup table for ORIGIN
origin_index_lookup = indexed['ORIGIN', 'ORIGIN_idx'].distinct()
display(origin_index_lookup.orderBy('ORIGIN_idx'))
```

ORIGIN	ORIGIN_idx
ATL	0.0
ORD	1.0
DFW	2.0
DEN	3.0
LAX	4.0
SFO	5.0
PHX	6.0
LAS	7.0
IAH	8.0
CLT	9.0

In [39]:

```
# Create lookup table for DEST
dest_index_lookup = indexed['DEST', 'DEST_idx'].distinct().orderBy('DEST_i
display(dest_index_lookup.orderBy('DEST_idx'))
```

DEST	DEST_idx
ATL	0.0
ORD	1.0
DFW	2.0
DEN	3.0
LAX	4.0
SFO	5.0
PHX	6.0
LAS	7.0
IAH	8.0
CLT	9.0

In [40]:

```
# Map OP_UNIQUE_CARRIER to OP_UNIQUE_CARRIER_idx
smoted_train_cols_carrier = smoted_train_cols.join(broadcast(carrier_index
                                                   (smoted_train_cols.

smoted_train_cols_carrier = smoted_train_cols_carrier.drop('OP_UNIQUE_CARR
```

In [41]:

```
# Map ORIGIN to ORIGIN_idx
smoted_train_cols_origin = smoted_train_cols_carrier.join(broadcast(origin
                                                   (smoted_train_cols_

smoted_train_cols_origin = smoted_train_cols_origin.drop('ORIGIN_idx')
```

In [42]:

```
# Map DEST to DEST_idx
smoted_train_cols_dest = smoted_train_cols_origin.join(broadcast(dest_inde
                                                   (smoted_train_cols_

smoted_train_cols_dest = smoted_train_cols_dest.drop('DEST_idx')
```

In [43]:

```
# Perform an action as the transformations are lazily evaluated
# Check the number of records in smoted dataset
smoted_train_cols_dest.count()
```

Out[30]: 37646258

Save the dataset to parquet

In [45]:

```
# Write train & val data to parquet for easier EDA
def WriteAndRefDataToParquet(data, dataName):
  # Write data to parquet format (for easier EDA)
  data.write.mode('overwrite').format("parquet").save("dbfs/user/team20/fi
```

```
    # Read data back directly from disk
    return spark.read.option("header", "true").parquet(f"dbfs/user/team20/fi
```

In [46]:
```
smoted_train_kmeans = WriteAndRefDataToParquet(smoted_train_cols_dest, 'sm
```

Load the smoted dataset to dataframe

In [48]:
```
# Load the data into dataframe
smoted_train_kmeans = spark.read.option("header", "true").parquet(f"dbfs/u
```

EDA of balanced vs. unbalanced train dataset

Delayed vs. Not Delayed

In [51]:
```
display(train.groupby('DEP_DEL30').count())
```

DEP_DEL30	count
1	2389904
0	18528016

In [52]:
```
display(smoted_train_kmeans.groupby('DEP_DEL30').count())
```

DEP_DEL30	count
0	18528591
1	19120805

Filtering out delayed data within train and smoted_train & plot graphs to observe the distribution of data

In [54]:
```
# Filter only delayed data
train_delay = train.filter(train.DEP_DEL30 == 1)
smoted_train_kmeans_delay = smoted_train_kmeans.filter(smoted_train_kmeans
```

1. OP_UNIQUE_CARRIER

In [56]:
```
display(train_delay.groupby('OP_UNIQUE_CARRIER').count().orderBy('OP_UNIQU
```

OP_UNIQUE_CARRIER	count
9E	25881
AA	330450

OP_UNIQUE_CARRIER	count
AS	46677
B6	167217
DL	271971
EV	171761
F9	57960
G4	11690

In [57]:

```
display(smoted_train_kmeans_delay.groupby('OP_UNIQUE_CARRIER').count().ord
```

OP_UNIQUE_CARRIER	count
9E	172172
AA	2791490
AS	424440
B6	1362673
DL	2230117
EV	1366414
F9	409360
G4	70871
HA	105543
MQ	473036

2. ORIGIN

In [59]:

```
display(train_delay.groupby('ORIGIN').count().orderBy('ORIGIN'))
```

ORIGIN	count
ABE	1002
ABI	431
ABQ	7384
ABR	179
ABY	375
ACK	435
ACT	405
ACV	796
ACY	1391
ADK	48

In [60]:
```
display(smoted_train_kmeans_delay.groupby('ORIGIN').count().orderBy('ORIGI
```

ORIGIN	count
ABE	8930
ABI	3552
ABQ	58744
ABR	1378
ABY	2425
ACK	1714
ACT	3334
ACV	6477
ACY	6095
ADK	446

3. DEST

In [62]:
```
display(train_delay.groupby('DEST').count().orderBy('DEST'))
```

DEST	count
ABE	1129
ABI	447
ABQ	8579
ABR	181
ABY	419
ACK	318
ACT	486
ACV	736
ACY	1847
ADK	17

In [63]:
```
display(smoted_train_kmeans_delay.groupby('DEST').count().orderBy('DEST'))
```

DEST	count
ABE	8871
ABI	3310
ABQ	68814
ABR	2283
ABY	2330

DEST	count
ACK	2405
ACT	4255
ACV	5911

4. DISTANCE_GROUP

In [65]:

```
display(train_delay.groupby('DISTANCE_GROUP').count().orderBy('DISTANCE_GR
```

DISTANCE_GROUP	count
1	272033
2	556876
3	463212
4	376688
5	279703
6	105115
7	116969
8	56829
9	41769
10	69037

In [66]:

```
display(smoted_train_kmeans_delay.groupby('DISTANCE_GROUP').count().orderB
```

DISTANCE_GROUP	count
1.0	2176284
2.0	4453813
3.0	3705302
4.0	3013154
5.0	2239037
6.0	840003
7.0	937540
8.0	455057
9.0	334563
10.0	552925

5. DISTANCE

In [68]:

```
display(train_delay.groupby('DISTANCE').count().orderBy('DISTANCE'))
```

DISTANCE	count
31.0	193
41.0	57
49.0	4
55.0	9
66.0	451
67.0	3861
68.0	541
69.0	592
70.0	114
72.0	1

In [69]:

```
display(smoted_train_kmeans_delay.groupby('DISTANCE').count().orderBy('DIS
```

DISTANCE	count
31.0	1557
39.0	1
41.0	481
47.0	1
49.0	6
50.0	1
51.0	1
53.0	1
54.0	1
55.0	53

6. YEAR

In [71]:

```
display(train_delay.groupby('YEAR').count().orderBy('YEAR'))
```

YEAR	count
2015	569783
2016	523015
2017	563278
2018	733917

In [72]:

```
display(smoted_train_kmeans_delay.groupby('YEAR').count().orderBy('YEAR'))
```

YEAR	count
2015.0	4024343
2016.0	4705309
2017.0	5084982
2018.0	5306171

7. MONTH

In [74]:

```
display(train_delay.groupby('MONTH').count().orderBy('MONTH'))
```

MONTH	count
1	194728
2	168354
3	191881
4	183062
5	207763
6	260509
7	273783
8	252067
9	144900
10	149856

In [75]:

```
display(smoted_train_kmeans_delay.groupby('MONTH').count().orderBy('MONTH'
```

MONTH	count
1.0	1074803
2.0	1395159
3.0	1467596
4.0	1550438
5.0	1797861
6.0	2252574
7.0	2604608
8.0	2143867
9.0	1368995
10.0	1198907

8. DAY_OF_MONTH

In [77]:
```
display(train_delay.groupby('DAY_OF_MONTH').count().orderBy('DAY_OF_MONTH'
```

DAY_OF_MONTH	count
1	77807
2	82478
3	77944
4	71801
5	78198
6	77431
7	75740
8	81400
9	84622
10	77883

In [78]:
```
display(smoted_train_kmeans_delay.groupby('DAY_OF_MONTH').count().orderBy(
```

DAY_OF_MONTH	count
1.0	295695
2.0	521346
3.0	603523
4.0	628204
5.0	663615
6.0	673201
7.0	675969
8.0	690372
9.0	694044
10.0	667129

9. DAY_OF_WEEK

In [80]:
```
display(train_delay.groupby('DAY_OF_WEEK').count().orderBy('DAY_OF_WEEK'))
```

DAY_OF_WEEK	count
1	380944
2	326799
3	322929
4	379551
5	393101

DAY_OF_WEEK	count
6	253326
7	333343

In [81]:

```
display(smoted_train_kmeans_delay.groupby('DAY_OF_WEEK').count().orderBy('
```

DAY_OF_WEEK	count
1.0	1852321
2.0	2931734
3.0	3376284
4.0	3720690
5.0	3370253
6.0	2353390
7.0	1516133