Predicting Airline Departure Delays

W261 Spring 2020

Presentation Date: April 16th, 2020

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

I. Question Formulation & Introduction

As we've all probably experienced at some point in our lives, air travel is never easy. Whether you're the person getting on a flight traveling around the world, the folks in the air traffic control towers orchestrating incoming and outgoing flights, or the airports and airlines trying their best to effectively coordinate flights at every hour of every day of every year, so much can go wrong. The delays alone are enough to completely derail anyone's plans and trigger a cascading effect of consequences down the line as delays continue to stack up on top of each other over the course of time. And the biggest problem is that these delays often occur when we least expect them and at the worst possible times.

Delays are costly for airlines and their passengers. A 2010 study commissioned by the Federal Aviation Administration estimated that flight delays cost the airline industry $8 billion a year, much of it due to increased spending on crews, fuel and maintenance. They cost passengers even more, nearly $17 billion.

To attempt to solve this problem, we introduce the *Airline Delays* dataset, a dataset of US domestic flights from 2015 to 2019 collected by the Bureau of Transportation Statistics for the purpose of studying airline delays. For this analysis, we will primarily use this dataset to study the nature of airline delays in the United States over the last few years, with the ultimate goal of developing models for predicting significant flight departure delays (30 minutes or more) in the United States.

In developing such models, we seek to answer the core question, **"Given known information prior to a flight's departure, can we predict departure delays and identify the likely causes of such delays?"**. In the last few years, about 11% of all US domestic flights resulted in significant delays, and answering these questions can truly help us to understand why such delays happen. In doing so, not only can airlines and airports start to identify likely causes and find ways to mitigate them and save both time and money, but air travelers also have the potential to better prepare for likely delays and possibly even plan for different flights in order to reduce their chance of significant delay.

To effectively investigate this question and produce a practically useful model, we will aim to develop a model that performs better than a baseline model that predicts the majority class of 'no delay' every time (this would have an accuracy of 89%). Having said that, we have been informed by our instructors that the state of the art is 85% accuracy, but will proceed to also prioritize model interpretability along side model performance

metrics to help address our core question. Given the classification nature of this problem, we will concentrate on improving metrics such as precision, recall, F1, area under ROC, and area under PR curve, over our baseline model. We will also concentrate on producing models that can explain what features of flights known prior to departure time can best predict departure delays and from these, attempt to best infer possible causes of departure delays.

In [3]:
```
# Pyspark SQL libraries
from pyspark.sql import functions as F
from pyspark.sql.functions import col
from pyspark.sql.functions import when
from pyspark.sql.types import StructType, StructField, StringType, Integer
from pyspark.sql.functions import udf
from pyspark.sql import Window

# Pyspark ML libraries
import pyspark.ml.pipeline as pl
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.classification import LinearSVC
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.feature import Bucketizer, StringIndexer, VectorIndexer, V
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.classification import RandomForestClassifier

from pyspark.mllib.evaluation import MulticlassMetrics

# Other python librarires
from dateutil.relativedelta import relativedelta, SU, MO, TU, WE, TH, FR,
import pandas as pd
import datetime as dt
import ast
import random
```

In [4]:
```
%sh
pip install plotly --upgrade
```

```
Collecting plotly
  Downloading https://files.pythonhosted.org/packages/15/90/918bccb0ca60
dc6d126d921e2c67126d75949f5da777e6b18c51fb12603d/plotly-4.6.0-py2.py3-no
ne-any.whl (7.1MB)
Collecting retrying>=1.3.3 (from plotly)
  Downloading https://files.pythonhosted.org/packages/44/ef/beae4b4ef809
02f22e3af073397f079c96969c69b2c7d52a57ea9ae61c9d/retrying-1.3.3.tar.gz
Requirement already satisfied, skipping upgrade: six in /databricks/cond
a/envs/databricks-ml/lib/python3.7/site-packages (from plotly) (1.12.0)
Building wheels for collected packages: retrying
  Building wheel for retrying (setup.py): started
  Building wheel for retrying (setup.py): finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/d7/a9/33/acc7b709e2a35caa
```

```
7d4cae442f6fe6fbf2c43f80823d46460c
Successfully built retrying
Installing collected packages: retrying, plotly
Successfully installed plotly-4.6.0 retrying-1.3.3
```

In [5]:

```
from plotly.offline import plot
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
```

II. EDA & Discussion of Challenges

Dataset Introduction

The Bureau of Transporation Statistics provides us with a wide variety of features relating to each flight in the *Airline Delays* dataset. These features range from features about the scheduled flight such as the planned departure, arrival, and elapsed times, the planned distance, the carrier and airport information, information regarding the causes of certain delays for the entire flight, as well as the amounts of delay (for both flight departure and arrival), among many other features.

Given that for this analysis, we will be concentrating on predicting and identifying the likely causes of departure delays before any such delay happens, we will primarily concentrate on our EDA, feature engineering, and model development using features of flights that would be known at inference time. We will choose the inference time to be 6 hours prior to the scheduled departure time of a flight. Realistically speaking, providing someone with a notice that a flight will likely be delayed 6 hours in advance is likely a sufficient amount of time to let people prepare for such a delay to reduce the cost of the departure delay, if it occurs. Such features that fit this criterion include those that are related to:

- **Time of year / Flight Date**
  - `Year` : The year in which the flight occurs (range: [2015, 2019])
  - `Month` : A numerical indicator for the month in which the flight occurs (range: [1, 12], 1 corresponds to January)
  - `Day_Of_Month` : The day of the month in which the flight occurs (range: [1, 31])
  - `Day_Of_Week` : A numerical indiciator for the day of the week in which the flight occurs (range: [1, 7], 1 corresponds to Monday)
- **Scheduled Departure & Arrival Times**
  - `CRS_Dep_Time` : The scheduled departure time of the flight (range: (0, 2400], 100 corresponds to 1AM departure time)
  - `CRS_Arr_Time` : The scheduled arrival time of the flight (range: (0, 2400], 100 corresonds to 1AM arrival time)
- **Planned Elapsed Times & Distances**

    - `CRS_Elapsed_Time` : The scheduled elapsed time (in minutes) of the flight (continuous variable, 60 corresponds to 1 hour elapsed time)
    - `Distance` : The planned distance (in miles) for the flight distance from origin to destination airports (continuous variable, e.g. 2475 miles)
    - `Distance_Group` : A binned version of the `Distance` variable into integer bins (range: [1, 11], e.g. 2475 miles maps to a distance group of 10)
- **Airline Carrier**
    - `Op_Unique_Carrier` : A shortcode denoting the airline carrier that operated the flight (categorical, 19 distinct carriers, e.g. 'AS' corresponds to Alaska Airlines, more mappings of airlines codes can be found here: https://www.bts.gov/topics/airlines-and-airports/airline-codes)
- **Origin & Destination Airports**
    - `Origin` : A shortcode denoting the origin airport from which the plane took off (categorical, 364 distinct airports, e.g. 'SFO' corresponds to San Francisco International Airport, more mappings of airport codes can be found here: https://www.bts.gov/topics/airlines-and-airports/world-airport-codes)
    - `Dest` : A shortcode denoting the destination airport at which the plane landed (categorical, 364 distinct airports, same in construct as `Origin`)

We will prioritize working with these features in the next few sections. Additionally, we will use the variable `Dep_Delay` (which describes the amount of departure delay in minutes) to define our outcome variable for "significiant" departure delays (i.e. delays of 30 minutes or more). These significant delays will be encoded in the variable `Dep_Del30` , a 0/1 indicator for whether the flight was delayed, which we will append to the dataset below. Finally, we will focus our analysis to the subset of flights that are not diverted, are not cancelled, and have non-null values for departure delays to ensure that we can accurately predict departure delays for flights. This subset will still leave us with a significant number of records to work with for the purpose of training and model development.

Below are a few example flights taken from the *Airline Delays* dataset that we will use for our analysis.

In [7]:
```
def LoadAirlineDelaysData():
  # Read in original dataset
  airlines = spark.read.option("header", "true").parquet(f"dbfs:/mnt/mids-

  # Filter to datset with entries where diverted != 1, cancelled != 1, and
  airlines = airlines.where('DIVERTED != 1') \
                     .where('CANCELLED != 1') \
                     .filter(airlines['DEP_DELAY'].isNotNull())
  return airlines

# Generate other Departure Delay outcome indicators for n minutes
def CreateNewDepDelayOutcome(data, thresholds):
  for threshold in thresholds:
    if ('Dep_Del' + str(threshold) in data.columns):
      print('Dep_Del' + str(threshold) + " already exists")
      continue
```

```
    data = data.withColumn('Dep_Del' + str(threshold), (data['Dep_Delay']
  return data

# Load data & define outcome variable
airlines = LoadAirlineDelaysData()
airlines = CreateNewDepDelayOutcome(airlines, [30])

print(airlines.columns)

# Filter dataset to variables of interest
outcomeName = 'Dep_Del30'
numFeatureNames = ['Year', 'Month', 'Day_Of_Month', 'Day_Of_Week', 'Distan
contNumFeatureNames = ['CRS_Dep_Time', 'CRS_Arr_Time', 'CRS_Elapsed_Time',
catFeatureNames = ['Op_Unique_Carrier', 'Origin', 'Dest']
joiningFeatures = ['FL_Date'] # Features needed to join with the holidays
airlines = airlines.select([outcomeName] + numFeatureNames + contNumFeatur

# Display a small sample of flight records
display(airlines.select([outcomeName] + numFeatureNames + contNumFeatureNa
```

Dep_Del30	Year	Month	Day_Of_Month	Day_Of_Week	Distance_Group	CRS_Dep_Tim
1	2019	6	20	4	1	224
0	2019	6	21	5	1	162
0	2019	6	21	5	2	165
0	2019	6	1	6	5	180
0	2019	6	2	7	3	111
0	2019	6	3	1	3	81
1	2019	6	19	3	1	144
0	2019	6	28	5	8	101
0	2019	6	14	5	4	63
0	2019	6	24	1	3	152

Note that because we are interested in predicting departure delays for future flights, we will define our test set to be the entirety of flights from the year 2019 and use the years 2015-2018 for feature engineering and model development (training & validation sets). This way, we will simulate the conditions for training a model that will predict departure delays for future flights. This will leave about 23% of the data for testing and the remaining 77% for training & validation.

Additionally, in this section and the next section on feature engineering, we'll mainly be operating on distinct columns of the dataset at any given time. In order to help with scalability of our explorations and preprocessing, we will save these dataset splits to parquet files in the cluster, as parquet is optimized for column-wise storage and thus will help improve the performance of our column-wise analysis of the *Airlines Delays* dataset. We will also focus our EDA on the union of training & validation sets, to ensure our decisions are not influenced by the test set.

In [9]:

```
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)

# 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/f

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

# Split dataset into training/validation/test; use mini_train to help with
mini_train, train, val, test = SplitDataset(airlines)

# Save and reload datasets for more efficient EDA & feature engineering
mini_train = WriteAndRefDataToParquet(mini_train, 'mini_train')
train = WriteAndRefDataToParquet(train, 'train')
val = WriteAndRefDataToParquet(val, 'val')
test = WriteAndRefDataToParquet(test, 'test')
train_and_val = WriteAndRefDataToParquet(train.union(val), 'train_and_val'
```

```
mini_train size = 2055
train size = 20916140
val size = 2989704
test size = 7268232
```

Prospective Models for the Departure Delay Classification Task

Before we go into detail on our core EDA tasks, we'd like to introduce the models we will be considering in section IV to motivate our discussion. At a high level, we will be considering the set of models that include the following:

- Decision Trees
- Logistic Regression
- Naive Bayes
- Support Vector Machines

Given that our task for this analysis is to classify flights as delayed (1) or not delayed (0), we want to ensure that the models we consider are well suited for classification tasks, which all four of these models are good at. Additionally, since we'll be interested in looking at explainable models to help inform why certain flights are delayed over others, we consider Decision Trees, Logistic Regression, and Naive Bayes explicitly for this task,

whereas Support Vector Machines are not as well-suited for explainability. We will discuss these models in more detail in section IV when we discuss our algorithm exploration. With that said, we'll keep these models in mind as we explore our core EDA tasks.

EDA Task #1: Binning Continuous Features

Among the features that we have to consider for predicting departure delays, we have 4 that are relatively continuous and can take on thousands of different values: `CRS_Dep_Time`, `CRS_Arr_Time`, `CRS_Elapsed_Time`, and `Distance`. Let's primarily focus on the variable `CRS_Elapsed_Time` to motivate this discussion and we'll generalize it to the remaining 3 variables. Below is a plot showing the bulk of the distribution for the feature `CRS_Elapsed_Time`.

In [12]:
```
# Helper Function for plotting distinct values of feature on X and number
def MakeRegBarChart(full_data_dep, outcomeName, var, orderBy, barmode, xty
  if (orderBy):
      d = full_data_dep.select(var, outcomeName).groupBy(var, outcomeName)
  else:
    d = full_data_dep.select(var, outcomeName).groupBy(var, outcomeName).c

  t1 = go.Bar(
    x = d[d[outcomeName] == 0.0][var],
    y = d[d[outcomeName] == 0.0]["count"],
    name=outcomeName + " = " + str(0.0)
  )
  t2 = go.Bar(
    x = d[d[outcomeName] == 1.0][var],
    y = d[d[outcomeName] == 1.0]["count"],
    name=outcomeName + " = " + str(1.0)
  )

  l = go.Layout(
    barmode=barmode,
    title="Flight Counts by " + var + " & " + outcomeName,
    xaxis=dict(title=var, type=xtype, range=xrange),
    yaxis=dict(title="Number of Flights", range=yrange)
  )
  fig = go.Figure(data=[t1, t2], layout=l)
  fig.show()
  return fig

var = 'CRS_Elapsed_Time'
fig = MakeRegBarChart(train_and_val, outcomeName, var, orderBy=var, barmod
```

From the plot above, we can see the continuous nature of `CRS_Elapsed_Time`, with the majortiy of flights ranging between 50 to 200 minutes. Note that each of the taller bars correspond to flight durations at the 5-minute markers (60, 65, 70, etc), as these are more "convenient" flight durations that airlines tend to define when scheduling their flights.

Now, if we consider the possible values that `CRS_Elapsed_Time` can intrinsically take on, it can be any number of minutes that the flight is scheduled to take; these could be values such as 60 minutes, 65 minutes, 120 minutes, even going onto 400 minutes in some cases, as evident above. Conceptually speaking, some of these times are drasticaly different (60 minutes vs. 400 minutes), while others are similar enough that they would be considered the same flight duration (60 minutes vs. 65 minutes). Because of this, these flight times could be aggregated into bins in order to group similar flight durations. For example, we could aggregate the `CRS_Elapsed_Time` into 1-hour blocks, which will produce a distribution such as the one shown below:

In [14]:
```
# Augments the provided dataset for the given variable with binned/bucketi
# versions of that variable, as defined by splits parameter
# Column name suffixed with '_bin' will be the bucketized column
# Column name suffixed with '_binlabel' will be the nicely-named version c
def BinValuesForEDA(df, var, splits, labels):
  # Bin values
  bucketizer = Bucketizer(splits=splits, inputCol=var, outputCol=var + "_b
  df_buck = bucketizer.setHandleInvalid("keep").transform(df)

  # Add label column for binned values
  bucketMaps = {}
  bucketNum = 0
  for l in labels:
    bucketMaps[bucketNum] = l
    bucketNum = bucketNum + 1

  callnewColsUdf = udf(lambda x: bucketMaps[x], StringType())
  return df_buck.withColumn(var + "_binlabel", callnewColsUdf(F.col(var +

# Make plot with binned version of CRS_Elapsed_Time
d = BinValuesForEDA(train_and_val, var,
                    splits = [float("-inf"), 60, 120, 180, 240, 300, 360,
                    labels = ['0-1 hours', '1-2 hours', '2-3 hours', '3-4
fig = MakeRegBarChart(d, outcomeName, var + "_binlabel", orderBy=var + "_b
```

By doing this kind of binning, we can see that the same general shape of the distribution is preserved, albeit at a coarser level, which removes some of the extra information that was present in the original variable. But doing this kind of aggregation has its benefits in terms of reducing the noise from the signal when it comes to modeling.

In the case of Logistic Regression, if we had referred to the original `CRS_Elapsed_Time`, we would estimate a single coefficient for the variable, which would tell us the effect that adding 1 minute to the elapsed time would have on the probability that a flight is delayed. However, if we were to bin this variable and treat the result as a categorical variable in our regression, the model would estimate a coefficient for all but one of the bins, which would tell us more detailed information about the effect of a flight having a certain kind of duration (e.g. 1-2 hours). By comparison to the coefficient we'd estimate on the raw `CRS_Elapsed_Time`, this would be a much more meaningful estimate to use to understand the underlying factors for departure delays. This will require the algorithm to have to learn more coefficients than if the original

`CRS_Elapsed_Time` were to be used, but to answer our core question, it seems to be a worthwhile cost if we proceed with Logistic Regression as our model of choice.

For the case of Decision Trees, binning the `CRS_Elapsed_Time` will actually help to improve the scalability of the algorithm. If we are to use the raw `CRS_Elapsed_Time`, the decision tree algorithm will have to consider every possible split for the feature (as we'll see later, there are in fact 646 distinct values for the `CRS_Elapsed_Time` feature, meaning that the algorithm would have to consider 645 different splits when finding the best split). However, if we bin the feature as shown above, the number of splits the algorithm needs to consider drops to just 6, which is a large reduction in the amount of work the algorithm needs to do to find the best split, which will help with the scalability of the algorithm.

The benefits that come with binning `CRS_Elapsed_Time` can really extend to all our continuous variables. The `Distance_Group` already takes care of this for the feature `Distance` and in section III, we will take care bin the remaining continuous variable depending on the situation.

EDA Task #2: Ordering Categorical Features

Among the features we have to consider, there are three that are categorical in nature, namely `Op_Unique_Carrier`, `Origin`, and `Dest`. While some features such as `Op_Unique_Carrier` have only a few distinct values (19 in total), others such as `Origin` and `Dest` have many more distinct values (364 each). While these categorical features are meaningful to us, they can introduce some added difficulties to our models for training.

In the case of Support Vector Machines, each of these categorical features will have to be encoded with 1-hot encoding, where we generate a sparse vector of a length equal to the number of unique values in the categorical feature. For `Op_Unique_Carrier`, this would lead to the algorithm to have to consider a 19-dimensional space. Take it to `Origin` or `Dest` and these features alone will require considering a 364-dimensional space, which makes it difficult for the algorithm to scale.

Similarly for Logistic Regression, for a categorical feature with n distinct values, this will require estimating $n - 1$ unique coefficients--while these coefficients can be meaningful to us, the sheer number can be overwhelming to estimate from a scalability perspective.

And with Decision Trees, the issue comes when considering the number of splits. If we take the `Op_Unique_Carrier` feature, because there are 19 values with no implicit order to them, when the Decision Tree algorithm considers all possible splits, it will have to consider not 18 splits, but $2^{k-1} - 1 = 2^{19-1} - 1 = 262,143$ possible splits. Go even further to a categorical variable like `Origin` and the number becomes massive ($1.87 * 10^{109}$ different splits to consider), which is computationally prohibitive to the algorithm as it will need to consider every single possible split to find the best split for the feature.

In order to address these issues, we'd really want to provide some sort of ordering and thus a ranking for each distinct value of our categorical features. Let's consider this in the context of our 'smaller' categorical variable, `Op_Unique_Carrier`. In its raw form, the distinct categories are in no way comparable (how does one compare 'DL' (Delta Airlines) to 'AS' (Alaska Airlines) in a meaningful way?). However, these categories could be compared using some intrinsic property of the category. Given that we're interested in using the information in `Op_Unique_Carrier` to predict our outcome `Dep_Del30`, we can evaluate what our average outcome is (or really the probability of a significant departure delay) for each variable and use this measure as a means of ordering the distinct categories. This ordering is shown in the plot below:

In [17]:

```
# Helper function that plot the probability of outcome on the x axis, the
# With entries for each distinct value of the feature as separate bars.
def MakeProbBarChart(full_data_dep, var, xtype, numDecimals):
  # Filter out just to rows with delays or no delays
  d_delay = full_data_dep.select(var, outcomeName).filter(F.col(outcomeNam
  d_nodelay = full_data_dep.select(var, outcomeName).filter(F.col(outcomeN

  # Join tables to get probabilities of departure delay for each table
  probs = d_delay.join(d_nodelay, d_delay[var] == d_nodelay[var]) \
            .select(d_delay[var], (d_delay["count"]).alias("DelayCount"),
                    (d_delay["count"] / (d_delay["count"] + d_nodelay["co

  # Join back with original data to get 0/1 labeling with probablities of
  d = full_data_dep.select(var, outcomeName).groupBy(var, outcomeName).cou
  d = d.join(probs, full_data_dep[var] == probs[var]) \
       .select(d[var], d[outcomeName], d["count"], probs["Prob_" + outcome
       .orderBy("Prob_" + outcomeName, outcomeName).toPandas()
  d = d.round({'Prob_' + outcomeName: numDecimals})

  t1 = go.Bar(
    x = d[d[outcomeName] == 0.0]["Prob_" + outcomeName],
    y = d[d[outcomeName] == 0.0]["count"],
    name=outcomeName + " = " + str(0.0),
    text=d[d[outcomeName] == 0.0][var]
  )
  t2 = go.Bar(
    x = d[d[outcomeName] == 1.0]["Prob_" + outcomeName],
    y = d[d[outcomeName] == 1.0]["count"],
    name=outcomeName + " = " + str(1.0),
    text=d[d[outcomeName] == 1.0][var]
  )

  l = go.Layout(
    barmode='stack',
    title="Flight Counts by " + "Prob_" + outcomeName + " & " + outcomeNam
    xaxis=dict(title="Prob_" + outcomeName + " (Note: axis type = " + xtyp
    yaxis=dict(title="Number of Flights")
  )
  fig = go.Figure(data=[t1, t2], layout=l)
  fig.show()
  return fig

# Plot Carrier and outcome with bar plots of probability on x axis
fig = MakeProbBarChart(airlines, "Op_Unique_Carrier", xtype='category', nu
```

By ordering the airline carriers by this average outcome (`Prob_Dep_Del30`), we can not only begin to compare the airlines (Alaska Airlines seems to be better than Delta Airlines by a small margin), but we can significantly reduce the number of splits to consider, from 262,143 possible splits to just 18 for `Op_Unique_Carrier` when it comes to the Decision Tree algorithm. Even further, if we assign numerical ranks, we have the potential to convert this categorical feature into a numerical feature (by assigning 1 to the highest ranked airline, 'HA' (Hawaiian Airlines), and 19 to the lowest ranked airline 'B6' (Jet Blue)), which helps to reduce the workload for both Logistic Regression and Support Vector Machines.

This data transformation essentially describes the application of Breiman's Theorem, which we can apply to all our categorical features, even the ones we feature engineer. Note that any ranking we generate for our categorical features will need to be generated based on our training set and separately applied to the test set, to ensure the ranking isn't in any way influenced by the data in our test set. We will proceed to apply this to all our categorical features in section III.

EDA Task #3: Balancing the Dataset

For our final EDA task, let's consider our outcome variable `Dep_Del30` . To ensure that our model is able to adequately predict whether a flight will be delayed, we need to make sure there is a good representation of both classes in the training set. However, as is evident in the pie chart below, we clearly have an over-representation of the no-delay category (`Dep_Del30 == 0`).

In [20]:
```
# Look at balance for outcome in training
display(train_and_val.groupBy(outcomeName).count())
```

Dep_Del30	count
1	2732080
0	21173764

What this chart tells us is that a simple model that always predicts no departure delay will have a high accuracy of around 89% (assuming our test set has a similar distribution). But of course, simply predicting the majority class is meaningless when it comes to answering our core question--all it really tells us is that most flights are not delayed.

But the problem extends further. Even if we develop a model that doesn't always predict no departure delay as a simple baseline model would, there is still a major problem of bias if we leave the dataset unbalanced. Namely, if the dataset is unbalanced, the model will still favor the majority class and learn features of majority class well at the expense of the minority class, which will come at a cost to model performance. Regardless of which of our four algorithms we'll concentrate on, this data imbalance is a concern that

we'll need to address. There are a variety of methods that we can use to deal with this data imbalance (including majority class undersampling, minority class oversampling, SMOTE, majority class splitting), which we will discuss the theory, implementation, and scalability concerns at the end of section III. Any method that will balance the dataset will help to reduce bias, coming at a cost to inducing more variance in the model. However, for the purpose of developing a model that can help inform the likely causes of departure delays, this is a worthwhile tradeoff.

Further EDA

During the investigation of our dataset, we did a deep dive to examine all of our in-scope variables from the original dataset to understand the nature of the dataset and help inform our decision making. To see more plots and discussion, please refer to our more extensive EDA linked below. We will also explore some more of these plots in the next section.

https://dbc-b1c912e7-d804.cloud.databricks.com/?o=7564214546094626#notebook/3895804345790408/command/3895804345790409

III. Feature Engineering

With the EDA discussion from the previous section in mind, we now proceed with applying the feature engineering described in the previous section. We will first look at summary statistics and check for missing values for each of our features for predicting departure delays. We'll then look to binning our numerical features, adding additional features via interactions, bringing in additional datasets, and looking at aggregated statistics. For the categorical features in the dataset (both original and those developed from an feature transformations), we will apply Breiman's Theorem to order these features and transform them into numerical features. Finally, we'll take a closer look at the data preprocessing techniques we will explore for balancing our dataset.

Summary Statistics & Missing Values Assessment

In the table shown below, we can see a set of summary statistics for each base feature, including general summary statistics (count, mean, standard deviation, min, max), as well as the number of distinct values and the number of null/missing values in the training & validation set. Based on the results shown below, all values appear to fall into the expected ranges based on the definitions of these features.

In [25]:

```
def GetStats(df):
  allCols = [outcomeName] + numFeatureNames + contNumFeatureNames + catFea

  # Get summary stats for the full training dataset
  summaryStats = df.select(allCols).describe().select(['summary'] + allCol

  # Get number of distinct values for each column in full training dataset
  distinctCount = df.select(allCols) \
                              .agg(*(F.countDistinct(F.col(c)).alias(c) f
```

```
                                    .withColumn('summary', F.lit('distinct coun
                                    .select(['summary'] + allCols)

    # Get number of nulls for each column in full training dataset
    nullCount = df.select([F.count(F.when(F.isnan(c) | F.col(c).isNull(), c)
                  .withColumn('summary', F.lit('null count')) \
                  .select(['summary'] + allCols)

    # Union all statistics to single display
    res = summaryStats.union(distinctCount.union(nullCount))
    display(res)

GetStats(train_and_val)
```

summary	Dep_Del30	Year	Month	Day_O
count	23905844	23905844	23905844	23
mean	0.11428502587066158	2016.5862149439274	6.558258725356026	15.76760197
stddev	0.31815713565262305	1.1456243075500832	3.3955166330828477	8.78039702
min	0	2015	1	
max	1	2018	12	
distinct count	2	4	12	
null count	0	0	0	

The only odd value seems to be the minimum value for `CRS_Elapsed_Time` that takes on a value of -99.0. Upon closer inspection, there's no true indication for why this datapoint is negative, except that it is likely a mistake in the dataset, since the difference in the departure and arrival times is 76 minutes, which should be the actual value of `CRS_Elapsed_Time`. However, given that we have about 24 million data points to train from, having this single data point be slightly incorrect should not affect the results of our training. For this reason, we'll leave the data point unchanged.

Finally, note that we do not appear to have any null values in our training and validation data and thus we will not need to handle missing values for the purpose of training. However, there is a potential that our test data has missing values and or the features of the test data take on values that were not seen in the training data. Because this is always a possibility at inference time, we will need to make sure our data transformations are robust to such cases--we will evaluate this on a case-by-case basis.

Binning Continuous Numerical Features

As discussed in task #1 of the EDA in section II, one of the transformations we'd like to apply to our continuous numerical features is a binning transformation. In doing so, we can reduce the continuous variables to meaningful increments that will help with interpretability in Logistic Regression and help to reduce the number of splits that needs to be considered by the Decision Tree algorithm. In order to determine reasonable split points, let's evaluate the distributions of each of the continuous variables

`CRS_Dep_Time`, `CRS_Arr_Time`, and `CRS_Elapsed_Time` (note that the continuous feature `Distance` has already been binned via the variable `Distance_Group`, so we will not examine this feature). These distribution are shown below (please zoom in for more detail):

In [28]:
```
fig1 = MakeRegBarChart(train_and_val, outcomeName, 'CRS_Dep_Time', orderBy
```

In [29]:
```
fig2 = MakeRegBarChart(train_and_val, outcomeName, 'CRS_Arr_Time', orderBy
```

From the distributions of values for `CRS_Dep_Time` and `CRS_Arr_Time`, we can see clusters of flights defined by --00 to --59 blocks (these clusters are a remnant of the structure of the 2400-clock, since there are no valid times from --60 to --99). With that said, by analyzing any one of these clusters, it's clear that most flights are scheduled at 5-minute markers (such as 1200, 1205, 1210, etc) for both departure and arrival times. Since there really isn't a big difference between times separated by a couple of 5 minute intervals, we will choose to bin these values by 10-minute increments to ensure we still have enough data granularity to differentiate between populate times such as 1200 and 1230 but not too much granularity to have too many splits to consider (reducing to 10-minute granularity alone reduces the number of splits for a Decision Tree algorithm to consider from at most 1439 to just about 144 split points).

In [31]:
```
fig3 = MakeRegBarChart(train_and_val, outcomeName, 'CRS_Elapsed_Time', ord
```

For the `CRS_Elapsed_Time`, as we saw in the EDA Task 1, we can still capture the general distribution of the scheduled flight duration even when we bin by 1 hour durations, which leaves us with meaningful groups and much fewer splits to consider. For this reason, we will bin `CRS_Elapsed_Time` to 1 hour increments.

All three of the described binning transformations are applied with the code shown below and appended to our original *Airline Delays* dataset. Note that this binning operation is independently applied to each record in the original dataset, using the binning we decided based on the training dataset. Also note that by using the `Bucketizer` 'keep' flag, we explicitly choose to bin any invalid values into a special bucket, which will allow our models to be resiliant even in the event of encountering values that go beyond the limits defined. Note that for simplicity, we extend the `CRS_Elapsed_Time` bins to negative and positive infinity to ensure all values are binned properly. All new binned features will be suffixed with `_bin`, except for `Distance_Group`, which is already defined. A few examples of the new columns are shown below.

In [33]:
```
# Augments the provided dataset for the given feature/variable with a binn
# of that variable, as defined by splits parameter
# Column name suffixed with '_bin' will be the binned column
```

```
def BinFeature(df, featureName, splits):
  if (featureName + "_bin" in df.columns):
    print("Variable '" + featureName + "_bin' already exists")
    return df

  # Generate binned column for feature
  bucketizer = Bucketizer(splits=splits, inputCol=featureName, outputCol=f
  df_bin = bucketizer.setHandleInvalid("keep").transform(df)
  df_bin = df_bin.withColumn(featureName + "_bin", df_bin[featureName + "_
  return df_bin

# Bin numerical features in entire airlines dataset
# Note that splits are not based on test set but are applied to test set (
airlines = BinFeature(airlines, 'CRS_Dep_Time', splits = [i for i in range
airlines = BinFeature(airlines, 'CRS_Arr_Time', splits = [i for i in range
airlines = BinFeature(airlines, 'CRS_Elapsed_Time', splits = [float("-inf"
binFeatureNames = ['CRS_Dep_Time_bin', 'CRS_Arr_Time_bin', 'CRS_Elapsed_Ti

display(airlines.select(contNumFeatureNames + binFeatureNames + ['Distance
```

CRS_Dep_Time	CRS_Arr_Time	CRS_Elapsed_Time	Distance	CRS_Dep_Time_bin	CRS_
1449	2324	335.0	2588.0	144	
1445	1800	375.0	2475.0	144	
1450	2322	332.0	2475.0	145	
1105	1228	83.0	369.0	110	
2316	157	161.0	1129.0	231	
1804	2019	135.0	749.0	180	

Interacting Features

During our in-depth EDA, we investigated a few interactions based on intuition. From our own experiences, we believed that certain temporal interaction terms and airport-related interactions would intuitively be indicative of a flight that is more likely to experience a departure delay.

Let's consider the features `Month` and `Day_Of_Month`. On their own, we may be able to capture aggregated information about flights delays on a particular day of the month or a particular month on its own. But if we interact the two, we get the concept of `Day_Of_Year`, and upon closer inspection, there do appear to be certain days of the year that experience more traffic and more delays. Take the stacked bar chart comparing the ratio of delayed and no-delay flights shown below for `Day_Of_Year`, which is the concatenation of `Month` and `Day_Of_Month`. From this chart, we can see that there is a higher probability of departure delays for dates in the summer, as well as near the end of December and beginning of January, but not on holidays in that time frame (e.g. December 25th - Christmas Day). Because there does appear to be some relationship between `Day_Of_Year` and the likelihood of a departure delay, we choose to add this as one of our interaction terms.

In [35]:
```
# Plot that demonstrates the probability of a departure delay, given the d
var = "Day_Of_Year"
d = train_and_val.select("Month", "Day_Of_Month", outcomeName) \
                 .withColumn(var, F.concat(F.col('Month'), F.lit('-'), F.c
                 .groupBy(var, "Month", "Day_Of_Month", outcomeName).count
                 .orderBy("Month", "Day_Of_Month") \
                 .toPandas()
display(d)
```

Day_Of_Year	Month	Day_Of_Month	Dep_Del30	count
1-1	1	1	0	49601
1-1	1	1	1	8211
1-2	1	2	0	54668
1-2	1	2	1	11590
1-3	1	3	1	11886
1-3	1	3	0	52368
1-4	1	4	1	10967
1-4	1	4	0	50166
1-5	1	5	1	11546
1-5	1	5	0	51417

Along a similar vein of thinking, we decided to add the interaction of `CRS_Dep_Time` and `Day_Of_Week` along with the interaction of `CRS_Arr_Time` and `Day_Of_Week` to produce `Dep_Time_Of_Week` and `Arr_Time_Of_Week` respectively. We discovered that there are certain times of day that encounter a higher likelihood of departure delays, depending on the day of the week these times fall on. But to ensure there are not too many categories in this new categorical feature, we will interact the binned times with the `Day_Of_Week` for each interaction.

Finally, we considered that there may be some inherent relationship between the origin airport and the destination airport with respect to departure delays that may be well captured via an interaction between the two features. Namely, during our EDA, we saw that more popular flights (such as between SFO (San Francisco) and LAX (Los Angeles)) also saw higher levels of departure delays, whereas less popular flights (such as between SEA (Seattle) and PHX (Pheonix)) saw lower levels of departure delays. Thus, we chose to interact `Origin` and `Dest` to form the categorical feature `Origin_Dest` for our dataset. All the interactions discussed are added using the provided code and a few examples are shown below:

In [37]:
```
# Given a tuple specifying the two features to interact and the new name c
# Creates an interaction term corresponding to each tuple
def AddInteractions(df, featurePairsAndNewFeatureNames):
  for (feature1, feature2, newName) in featurePairsAndNewFeatureNames:
    if (newName in df.columns):
      print("Variable '" + newName + "' already exists")
      continue
```

```
    # Generate interaction feature (concatenation of two features)
    df = df.withColumn(newName, F.concat(F.col(feature1), F.lit('-'), F.co

  return df

# Make interaction features on airlines dataset
# Note that interactions are independentently defined for each record
interactions = [('Month', 'Day_Of_Month', 'Day_Of_Year'),
                ('Origin', 'Dest', 'Origin_Dest'),
                ('Day_Of_Week', 'CRS_Dep_Time_bin', 'Dep_Time_Of_Week'),
                ('Day_Of_Week', 'CRS_Arr_Time_bin', 'Arr_Time_Of_Week')]
intFeatureNames = [i[2] for i in interactions]
airlines = AddInteractions(airlines, interactions)

display(airlines.select(['Month', 'Day_Of_Month', 'Day_Of_Year', 'Day_Of_W
```

Month	Day_Of_Month	Day_Of_Year	Day_Of_Week	CRS_Dep_Time_bin	Dep_Time_Of_W
6	1	6-1	6	144	6
6	1	6-1	6	144	6
6	1	6-1	6	145	6
6	1	6-1	6	110	6
6	1	6-1	6	231	6
6	25	6-25	2	180	2

Adding a Holiday Feature

With the `Day_Of_Year` variable in mind, we'd noted during the EDA that the likelihood of a departure delay, as well as the number of flights occuring on a given day of the year seemed to have some relationship with the commonly celebrated holidays in the United States. Take Christmas Day, which always falls on December 25th. If we examine the plot shown previously, there generally appeared to be a lower probability of delay on Christmas Day and Christmas Eve, with higher probabilities immediately before and after those two days. Intuitively, one would expect a higher probability of departure delay as people try to get home for the holidays or leave promptly after the holidays are over; but on the day of holidays, people generally stay home (and thus experience less probability of departure delay).

There are a variety of cases where holidays and days near holidays seem to reflect something about the likelihood of a flight experiencing a departure delay. For this reason, we attempted to capture this information by joining a dataset of government holidays to our original *Airlines Delays* dataset and construct the `Holiday` feature. This feature is a categorical feature indicating whether a flight occurs before, after, or on a holiday, or is not in any way related to a holiday. Because the *Holidays* dataset is much smaller compared to the *Airline Delays* dataset and will fit in memory, we will join this dataset to *Airline Delays* via a broadcast join after it has been prepared with before and after limits for specific government holidays. Note that the limits for whether a flight

occurs before or after a holiday is dependent on the day of the week in addition to government holidays that are near the flight date. This feature construction is done in the following code with a few examples shown below with new feature:

In [39]:

```
def AddHolidayFeature(df):
  if ('Holiday' in df.columns):
      print("Variable 'Holiday' already exists")
      return df

  # Import dataset of government holidays
  holiday_df_raw = spark.read.csv("dbfs:/user/shajikk@ischool.berkeley.edu
  holiday_df_raw.columns = ['ID', 'FL_DATE', 'Holiday']

  # Get limits for a given date when we'll likely see "near holiday" condi
  # This is more ofr the purpose of helping to capture likely long-weekend
  # that could influence departure delays near holidays
  def get_limits(date):
    given_date = dt.datetime.strptime(date,'%Y-%m-%d')
    this_day = given_date.strftime('%a')

    lastSun = given_date + relativedelta(weekday=SU(-1))
    lastMon = given_date + relativedelta(weekday=MO(-1))
    lastTue = given_date + relativedelta(weekday=TU(-1))
    lastWed = given_date + relativedelta(weekday=WE(-1))
    lastThu = given_date + relativedelta(weekday=TH(-1))
    lastFri = given_date + relativedelta(weekday=FR(-1))
    lastSat = given_date + relativedelta(weekday=SA(-1))
    thisSun = given_date + relativedelta(weekday=SU(1))
    thisMon = given_date + relativedelta(weekday=MO(1))
    thisTue = given_date + relativedelta(weekday=TU(1))
    thisWed = given_date + relativedelta(weekday=WE(1))
    thisThu = given_date + relativedelta(weekday=TH(1))
    thisFri = given_date + relativedelta(weekday=FR(1))
    thisSat = given_date + relativedelta(weekday=SA(1))

    if this_day == 'Sun' : prev, nxt = lastFri, thisMon
    if this_day == 'Mon' : prev, nxt = lastFri, thisMon
    if this_day == 'Tue' : prev, nxt = lastFri, thisTue
    if this_day == 'Wed' : prev, nxt = lastFri, thisWed
    if this_day == 'Thu' : prev, nxt = lastWed, thisSun
    if this_day == 'Fri' : prev, nxt = lastThu, thisMon
    if this_day == 'Sat' : prev, nxt = lastFri, thisMon

    return prev.strftime("%Y-%m-%d"), prev.strftime('%a'), nxt.strftime("%

  # Construct a holiday dataframe for days that fall before, after, or on
  # Consider holidays for all years 2015-2019 individually
  holiday_df = pd.DataFrame()
  for index, row in holiday_df_raw.iterrows():
      prev, prev_day, nxt, nxt_day = get_limits(row['FL_DATE'])
      line = pd.DataFrame({"ID": 0, "date": prev, "holiday" : 'before'}, i
      holiday_df = holiday_df.append(line, ignore_index=False)

      line = pd.DataFrame({"ID": 0, "date": row['FL_DATE'], "holiday" : 'h
      holiday_df = holiday_df.append(line, ignore_index=False)

      line = pd.DataFrame({"ID": 0, "date": nxt, "holiday" : 'after'}, ind
      holiday_df = holiday_df.append(line, ignore_index=False)
```

```
  holiday_df = holiday_df.sort_index().reset_index(drop=True).drop("ID",

  # Convert holidays pandas dataframe to a pyspark dataframe for easier jo
  schema = StructType([StructField("FL_DATE", StringType(), True), StructF
  holiday = spark.createDataFrame(holiday_df, schema)

  # Add new holiday/no holiday column to dataset
  return df.join(F.broadcast(holiday), df.FL_Date == holiday.FL_DATE, how=

# Add holidays indicator to airlines dataset
# Note that holidays are known well in advance of a flight and are not spe
holFeatureNames = ['Holiday']
airlines = AddHolidayFeature(airlines)
display(airlines.select('Year', 'Month', 'Day_Of_Month', 'Day_Of_Week', 'H
```

Year	Month	Day_Of_Month	Day_Of_Week	Holiday
2019	6	19	3	not holiday
2019	5	7	2	not holiday
2019	5	11	6	not holiday
2018	6	19	2	not holiday
2018	12	31	1	holiday
2018	12	12	3	not holiday
2018	11	14	3	not holiday
2017	6	23	5	not holiday
2017	6	19	1	not holiday
2015	7	31	5	not holiday

Adding an Origin Activity Feature

Another intuition based feature that we believed would affect the likelihood of departure delay would be the amount of airline traffic an airport was experiencing around the time that departure was scheduled. Given that busier airports are likely to have busier air traffic control towers and more planes lined up to depart (not to mention more traffic inside of the airport itself as people try to get to their flights), it seemed likely that the amount of activity happening at the origin airport would be indicative of the likelihood of a departure delay. With the `Origin_Activity` feature, we attempt to define a numerical metric that aggregates the "amount of traffic" occuring at a given airport during a specific scheduled departure time block on a given day of the year. This aggregated dataset can then be joined back to the *Airline Delays* dataset via a broadcast join, which gives each flight record a count of the scheduled flight activity that would be occuring in the same departure time block and origin airport. This feature is generated below and a few examples are displayed below:

In [41]:

```
def AddOriginActivityFeature(df):
  if ('Origin_Activity' in df.columns):
      print("Variable 'Origin_Activity' already exists")
```

```
    return df

  # Construct a flight bucket attribute to group flights occuring on the s
  # Compute aggregated statistics for these flight buckets
  df = df.withColumn("flightbucket", F.concat_ws("-", F.col("Year"), F.col
  originActivityAgg = df.groupBy("flightbucket").count()

  # Join aggregated statistics back to original dataframe
  df = df.join(F.broadcast(originActivityAgg), df.flightbucket == originAc
         .drop(originActivityAgg.flightbucket) \
         .drop('flightbucket') \
         .withColumnRenamed('count', 'Origin_Activity')
  return df

# Add OriginActivity feature
# Note that the scheduled origin activity is known well in advance of a fl
orgFeatureNames = ['Origin_Activity']
airlines = AddOriginActivityFeature(airlines)
display(airlines.select('Year', 'Month', 'Day_Of_Month', 'CRS_Dep_Time_bin
```

Year	Month	Day_Of_Month	CRS_Dep_Time_bin	Origin	Origin_Activity
2019	6	1	74	PDX	5
2019	6	4	94	MAF	1
2019	6	14	223	LAS	3
2019	6	3	73	PDX	2
2019	6	22	123	ORD	3
2019	6	17	142	SYR	1

Applying Breiman's Theorem to Categorical Features

As discussed in the previous secition in EDA task #2, in our original *Airline Delays* dataset, we have three categorical features to consider, and with the addition of our interaction terms and our holiday feature, we have a total of 8 categorical features to consider for training our model. While some of these categorical features have few distinct values, some of them, especially our interaction terms like `Origin_Dest`, can have a very large number of distinct values. Depending on the model we choose following our algorithm exploration section, these large number of distinct values can be cause for concern with respect to the scalability of our algoirthms. For SVMs, this can lead to very large one-hot encoded vectors. For Logistic Regression, this can lead to many (if not too many) unique coefficients to estimate. And with Decision Trees, too many categories can lead to an inordinate number of possible splits for the algorithm to consider and finding the "best" split would be computationally prohibitive.

However, as we saw in our EDA task, we do have a way of ordering the categories in each categorical feature in a more meaningful way by applying Breiman's Theorem to each of our categorical features. Let's consider again one of our original categorical features `Op_Unique_Carrier` that we'd explored in EDA task #2. The cateogries by themselves, do not have any implicit ordering. Yet, using these distinct categories, we

can develop aggregated statistics on the outcome variable `Dep_Del30` to understand how some categories compare to others and rank them--this is the idea behind Breiman's Theorem. Below, we define a function for generating such "Breiman Ranks" given a training dataset, with the example shown for the `Op_Unique_Carrier` feature.

In [44]:
```
# Applies Breiman's Theorem to the categorical feature
# Generates the ranking of the categories in the provided categorical feat
# Orders the categories by the average outcome ascending, from integer 1 t
# Note that this should only be run on the training data
def GenerateBreimanRanks(df, catFeatureName, outcomeName):
  window = Window.orderBy('avg(' + outcomeName + ')')
  breimanRanks = df.groupBy(catFeatureName).avg(outcomeName) \
                   .sort(F.asc('avg(' + outcomeName + ')')) \
                   .withColumn(catFeatureName + "_brieman", F.row_number()
  return breimanRanks

# Generate example Breiman ranks for Op_Unique_Carrier
exBreimanRanks = GenerateBreimanRanks(train_and_val, 'Op_Unique_Carrier',
display(exBreimanRanks)
```

Op_Unique_Carrier	avg(Dep_Del30)	Op_Unique_Carrier_brieman
HA	0.033542917204872985	1
AS	0.06905222712211065	2
DL	0.08529200285416323	3
US	0.08998934214794334	4
YX	0.11116389081807818	5
AA	0.11117867242701317	6
OO	0.11333882967506877	7
WN	0.11723854777220556	8
MQ	0.122757807728906	9
EV	0.12610044862283	10

As evident in the output above, we can generate aggregated statistics on the outcome variable for each distinct carrier based ont the sample records present in the dataset for that carrier. Using these aggregated statistics, we can order the categories and assign a ranking from 1 to n, where n is the number of distinct categories (in this case 19). This ranking is saved in the new feature `Op_Unique_Carrier_brieman` . Given that the table of Breiman ranks is small and can fit in memory (even for "larger" categorical features like our interaction terms), we can take these training-set-generated Breiman ranks and independently apply them to the entire *Airline Delays* dataset via a broadcast join, similar to the ones shown previously. This code is shown below. In doing so, we effectively transform our categorical feature into a numerical feature, which reduces the need for 1-hot encoding in SVM as well as the number of coefficients to estimate and splits to consider in the Logistic Regression and Decision Tree algorithms respectively. Note that to properly handle unseen values, we will encode unseen categorical features

with a -1 Brieman rank, as is common practice when transforming categorical features to numerical form.

In [46]:
```
# Using the provided Breiman's Ranks, applies Breiman's Theorem to the cat
# and creates a column in the original table using the mapping in breimanR
# Note that this effectively transforms the categorical feature to a numer
# The new column will be the original categorical feature name, suffixed w
def ApplyBreimansTheorem(df, breimanRanks, catFeatureName, outcomeName):
  if (catFeatureName + "_brieman" in df.columns):
    print("Variable '" + catFeatureName + "_brieman" + "' already exists")
    return df

  res = df.join(F.broadcast(breimanRanks), catFeatureName, how='left') \
          .drop(breimanRanks['avg(' + outcomeName + ')']) \
          .fillna(-1, [catFeatureName + "_brieman"])
  return res
```

With these two functions, we can apply Breiman's Theorem to each of our 8 categorical features separately and generate new columns numerical that will be suffixed with `_brieman` . Note that the Breiman ranks will always be generated by referring to the training dataset only. Once these are generated, they can be applied to any dataset, as the ranks are depending on the training dataset and should not be influenced by our test set. This is done below (note that we'll split the dataset again to ensure that our training dataset has all new features that were generated in the previous sections).

In [48]:
```
# Regenerate splits for Breiman ranking prep (so have context of new
# features added that require application of Breiman's Theorem)
mini_train, train, val, test = SplitDataset(airlines)
```

```
mini_train size = 2129
train size = 21526836
val size = 3076829
test size = 7498430
```

In [49]:
```
# Apply breiman ranking to all datasets, based on ranking developed from t
breimanRanksDict = {} # save to apply later as needed
featuresToApplyBreimanRanks = catFeatureNames + intFeatureNames + holFeatu
for feature in featuresToApplyBreimanRanks:
  # Get ranks for feature, based on training data only
  ranksDict = GenerateBreimanRanks(train, feature, outcomeName)
  breimanRanksDict[feature] = ranksDict

  # Apply Breiman's theorem & do feature transformation for all datasets
  mini_train = ApplyBreimansTheorem(mini_train, ranksDict, feature, outcom
  train = ApplyBreimansTheorem(train, ranksDict, feature, outcomeName)
  val = ApplyBreimansTheorem(val, ranksDict, feature, outcomeName)
  test = ApplyBreimansTheorem(test, ranksDict, feature, outcomeName)
  airlines = ApplyBreimansTheorem(airlines, ranksDict, feature, outcomeNam

briFeatureNames = [entry + "_brieman" for entry in breimanRanksDict]

# Show examples of Breiman features
selectCols = []
```

```
for feature in featuresToApplyBreimanRanks:
  selectCols.append(feature)
  selectCols.append(feature + "_brieman")
display(train.select(selectCols).take(10))
```

Op_Unique_Carrier	Op_Unique_Carrier_brieman	Origin	Origin_brieman	Dest	Dest_brie
AA	5	CLT	217	ATL	
AA	5	CLT	217	RDU	
HA	1	HNL	7	ITO	
DL	3	FAR	116	MSP	
AA	5	MCO	301	MIA	
HA	1	HNL	7	OGG	
OO	7	BJI	44	MSP	
AA	5	TUL	88	DFW	
AA	5	SAT	120	DFW	
DL	3	CLT	217	ATL	

With the application of Breiman's Theorem to our dataset, we now have a series of new features that we will leverage in section V for our chosen algorithm implementation, which are listed below. Note that we have checkpointed this work and saved the updated dataset to parquet format to be read from prior to model training (to avoid needing to re-compute these features).

In [51]:
```
print("All Available Features:")
print("------------------------")
print(" - Numerical Features: \t\t", numFeatureNames) # numerical features
print(" - Cont. Numerical Features:\t", contNumFeatureNames) # numerical f
print(" - Categorical Features: \t", catFeatureNames) # categorical featur
print(" - Binned Features: \t\t", binFeatureNames) # numerical features
print(" - Interaction Features: \t", intFeatureNames) # categorical featur
print(" - Holiday Feature: \t\t", holFeatureNames) # categorical features
print(" - Origin Activity Feature: \t", orgFeatureNames) # numerical featu
print(" - Breiman Ranked Features: \t", briFeatureNames) # numerical featu
```

```
All Available Features:
------------------------
 - Numerical Features:           ['Year', 'Month', 'Day_Of_Month', 'Day_
Of_Week', 'Distance_Group']
 - Cont. Numerical Features:     ['CRS_Dep_Time', 'CRS_Arr_Time', 'CRS_E
lapsed_Time', 'Distance']
 - Categorical Features:         ['Op_Unique_Carrier', 'Origin', 'Dest']
 - Binned Features:              ['CRS_Dep_Time_bin', 'CRS_Arr_Time_bi
n', 'CRS_Elapsed_Time_bin']
 - Interaction Features:         ['Day_Of_Year', 'Origin_Dest', 'Dep_Tim
e_Of_Week', 'Arr_Time_Of_Week']
 - Holiday Feature:              ['Holiday']
 - Origin Activity Feature:      ['Origin_Activity']
 - Breiman Ranked Features:      ['Op_Unique_Carrier_brieman', 'Origin_b
```

```
rieman', 'Dest_brieman', 'Day_Of_Year_brieman', 'Origin_Dest_brieman',
'Dep_Time_Of_Week_brieman', 'Arr_Time_Of_Week_brieman', 'Holiday_briema
n']
```

Balancing the Training Dataset

In the EDA task #3 from the previous section, we saw that our entire *Airline Delays* dataset is highly imbalanced, with only 11% of the flight data constituting flights with departure delays. As discussed previously, in order to develop a useful model that isn't biased by the majority class in the training dataset. To address this dataset imbalance issue, we will concentrate on two primary methods: SMOTE and Majority Class Splitting.

SMOTE (Synthetic Minority Over-sampling Technique)

A dataset is imbalanced if the classes are not approximately equally represented. Training a machine learning model with an imbalanced dataset causes the model to develop a certain bias towards the majority class. To tackle the issue of class imbalance, Synthetic Minority Over-sampling Technique (SMOTE) was introduced by Chawla et al. in 2002.(Chawla, Nitesh V., et al. “SMOTE: synthetic minority over-sampling technique.” Journal of artificial intelligence research16 (2002): 321–357).

Under-sampling of the majority class or/and over-sampling of the minority class have been proposed as a good means of increasing the sensitivity of a classifier to the minority class. However, under-sampling the majority class samples could potentially lead to loss of important information. Also, over-sampling the minority class could lead to overfitting. The reason is fairly straightforward. Consider the effect on the decision regions in feature space when minority over-sampling is done by replication (sampling with replacement). With replication, the decision region that results in a classification decision for the minority class can actually become smaller and more specific as the minority samples in the region are replicated. This is the opposite of the desired effect.

SMOTE provides a new approach to over-sampling. It is an approach in which the minority class is over-sampled by creating “synthetic” examples rather than by over-sampling with replacement. This approach is inspired by a technique that proved successful in handwritten character recognition (Ha & Bunke, 1997). They created extra training data by performing certain operations on real data. In their case, operations like rotation and skew were natural ways to perturb the training data. SMOTe generates synthetic examples in a less application-specific manner, by operating in “feature space” rather than “data space”. The minority class is over-sampled by taking each minority class sample and introducing synthetic examples along the line segments joining the k nearest neighbors. Our implementation currently uses seven nearest neighbors.

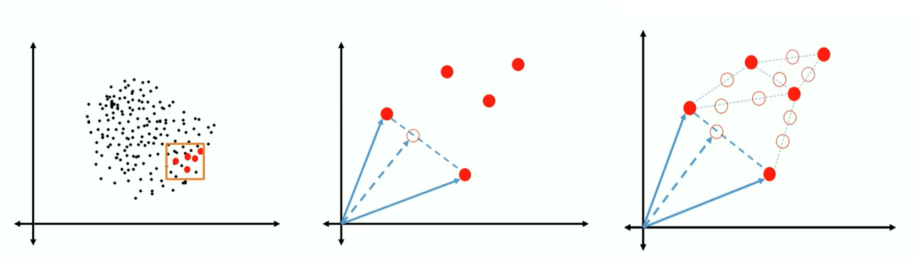
Synthetic samples are generated in the following way:

- Take the difference between the feature vector (of the sample) under consideration and the feature vector of its nearest neighbor(s).

- Multiply this difference by a random number between 0 and 1 to scale the difference.
- Add the scaled difference to the feature vector under consideration.

The diagrams below highlight the steps of capturing the region of k-nearest neighbors for a given datapoint (in orange), connecting the datapoint under consideration to is k-nearest neighbors (also in orange) via the blue dotted lines in feature space, and generating a white synthetic datapoint along these blue dotted lines.

Visualizing SMOTE



Source: https://www.youtube.com/watch?v=FheTDyCwRdE

By following these steps, we can generate a new random point along the line segment between two specific feature vectors to be a new synthetic datapoint. This approach effectively forces the decision region of the minority class to become more general. The synthetic examples cause the classifier to create larger and less specific decision regions (that contain nearby minority class points), rather than smaller and more specific regions. More general regions are now learned for the minority class samples rather than those being subsumed by the majority class samples around them. SMOTE provides more related minority class samples to learn from, thus allowing a learner to carve broader decision regions, leading to more coverage of the minority class. The effect is that models, in theory, will generalize better.

Deviations from the original paper

1. The number of minority class samples from our training set were approximately two million. However, running K Nearest Neighbors on each of these ~2M samples is not scalable as KNN needs to store the list of ~2M feature vectors in memory. We considered and implemented two approaches to address this scalability challenge:

   > i. Find KNN of each minority sample (feature vector) from a random sample (0.005%) of all the minority sample (feature vectors). This will produce a list of feature vectors small enough to fit in memory.
   >
   > ii. Create clusters of minority sample data using K-means algorithm, run KNN on each cluster in parallel and generate synthetic data for each cluster. This approach uses the entire training data. We split the data into 1000 clusters.

> Out of the above two approaches, we found the second approach took far less time to run (~2.5 hrs vs. 24+ hrs). Also, when we compared the distribution of minority samples from the original training set to all the minority samples after applying SMOTE, the data generated by the second approach matched the original feature distributions of the training set better than the data generated by the first approach. Thus, for the remainder of this analysis, we will proceed with the second approach for balancing our dataset using SMOTE.

2. The original paper shows that a combination of over-sampling the minority class using SMOTE and under-sampling the majority class can achieve better classifier performance (in ROC space) than plain under-sampling the majority class. We tried random under-sampling with SMOTE but it did not give us better results compared to creating synthetic data without under-sampling the majority class. There are existing "strategic" methods for under-sampling such as Near-Miss, Edited Nearest Neighbors Rule and One-Sided Selection. However, these are all implementions in imblearn library which converts the feature vector into numpy array. We cannot use these approaches directly due to scalability issues. For future work, we will try to find a scalable solution for a "strategic" under-sampling technique, similar to what we did with creating synthetic data in (1) using K-means.

We have provided an additional notebook with the full implementation of SMOTE for this project, which can be found here:

https://dbc-b1c912e7-d804.cloud.databricks.com/?o=7564214546094626#notebook/2791835342809045/command/814519033153637

Implementing SMOTE at Scale

The code below provides a summary of the functions we generated for SMOTEing our training dataset. These functions are documented below for reference and have been applied via the notebook mentioned previously.

In [58]:

```
# HELPER FUNCTIONS

from pyspark.ml.clustering import KMeans

# Train a k-means model on the minority samples (feature vectors)
def kmeans_model(featureVect, k):
  """
  Function to run k-means algorithm.
    arg:
        featureVect - (dataframe) feature vectors of minority samples
        k - (int) number of clusters
    returns:
        smote_rdd - (dataframe) predictions of k-means for each minority s
  """
  kmeans = KMeans().setK(k).setSeed(1)
  model = kmeans.fit(featureVect)
  predict = model.transform(featureVect)
  return predict
```

```
# Calculate the Euclidean distance between two feature vectors
def euclidean_distance(row1, row2):
  """
  Function to calculate Euclidean distance.
   arg:
       row1, row2 - (list) feature vector
   returns:
       distance - (float) euclidean distance between two feature vectors
  """
  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):
  """
  Function to calculate nearest neighbors.
   arg:
       train - (list) list of feature vectors from which to find nearest n
       test_row - (list) feature vector under consideration whose nearest
       num_neighbors - (int) number of nearest neighbors
   returns:
       neighbors - (list) nearest 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):
  """
  Function to generate synthetic data.
   arg:
       list1, list2 - (list) feature vectors from which synthetic data poi
   returns:
       synthetic_records - (list) synthetic data
  """
  synthetic_records = []
  for i in range(len(list1)):
    synthetic_records.append(round(list1[i] + ((list2[i]-list1[i])*random.
  return synthetic_records


# RUNNING SMOTE AT SCALE
# Convert the k-means predictions dataframe into rdd, find nearest neighbo
def SmoteSampling(predict, k):
  """
  Function to create an rdd of sunthetic data.
    arg:
        predict - (dataframe) k-means predictions
```

```
        k - (int) number of nearest neighbors
    returns:
        smote_rdd - (rdd) synthetic data in the form of rows of feature ve
    """
    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, k)
                           .flatMap(lambda x: [synthetic(x[0],n) for n in x[
                           .map(lambda x: Row(features = DenseVector(x), lab
                           .cache()
    return smote_rdd
```

Using SMOTEd Training Data

We applied our version of the SMOTE algorithm on the original subset of the training data we have worked with preivously. Since the feature engineering described in earlier in this section has not been applied to our SMOTEd training data, we will apply the same feature engineering steps here as an additional step before being able to use the SMOTEd data for training. The result will also be saved to parquet format to help with training moving forward.

In [60]:

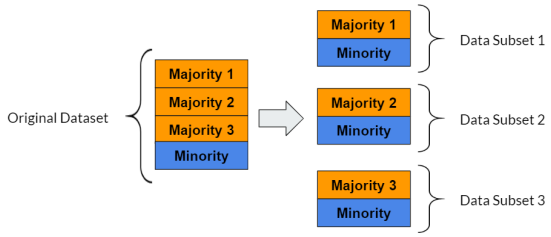
```
# Apply Feature Engineering described above to smoted training data
# Provided Breiman ranks dict should be based on unsmoted training data
def ApplyFeatureEngineeringToSmotedTrainingData(df, breimanRanksDict):
  # Select features to ensure features use pascal casing
  outcomeName = 'Dep_Del30'
  numFeatureNames = ['Year', 'Month', 'Day_Of_Month', 'Day_Of_Week', 'Dist
  contNumFeatureNames = ['CRS_Dep_Time', 'CRS_Arr_Time', 'CRS_Elapsed_Time
  catFeatureNames = ['Op_Unique_Carrier', 'Origin', 'Dest']
  df = df.select([outcomeName] + numFeatureNames + contNumFeatureNames + c


  # Bin Continuous Features
  df = BinFeature(df, 'CRS_Dep_Time', splits = [i for i in range(0, 2400 +
  df = BinFeature(df, 'CRS_Arr_Time', splits = [i for i in range(0, 2400 +
  df = BinFeature(df, 'CRS_Elapsed_Time', splits = [float("-inf")] + [i fo

  # Add Interaction Features
  interactions = [('Month', 'Day_Of_Month', 'Day_Of_Year'),
                  ('Origin', 'Dest', 'Origin_Dest'),
                  ('Day_Of_Week', 'CRS_Dep_Time_bin', 'Dep_Time_Of_Week'),
                  ('Day_Of_Week', 'CRS_Arr_Time_bin', 'Arr_Time_Of_Week')]
  df = AddInteractions(df, interactions)
  intFeatureNames = [i[2] for i in interactions]

  # Add FL_Date column (for holiday feature generation)
  df = df.withColumn('FL_Date', F.concat_ws("-", F.col("Year"), F.col("Mon

  # Add Holiday Feature
  df = AddHolidayFeature(df)
  holFeatureNames = ['Holiday']

  # Add Origin Activity Feature
  df = AddOriginActivityFeature(df)

  # Apply Breiman Ranking
```

```
  featuresToApplyBreimanRanks = catFeatureNames + intFeatureNames + holFea
  for feature in featuresToApplyBreimanRanks:
    # Apply Breiman's method & do feature transformation for all datasets
    df = ApplyBreimansTheorem(df, breimanRanksDict[feature], feature, outc

  return df

# Read prepared data from parquet for training
def ReadDataFromParquet(dataName):
  # Read data back directly from disk
  return spark.read.option("header", "true").parquet(f"dbfs/user/team20/fi

# Prep Smoted training data
train_smoted = ReadDataFromParquet('smoted_train_kmeans')
train_smoted = ApplyFeatureEngineeringToSmotedTrainingData(train_smoted, b
train_smoted = WriteAndRefDataToParquet(train_smoted, 'augmented1_smoted_t
```

Majority Class Splitting

Another method for balancing the training dataset is with the use of we call the "Majority Class Splitting" technique, which is described in the paper "New Applications of Ensembles of Classifiers" by Barandela et. al (http://marmota.dlsi.uji.es/WebBIB/papers/2003/paa-2.pdf). Referring to the paper, this is a dataset balancing approach where instead of oversampling the minority class to balance the dataset (similar to what we did with SMOTE), we take an majority lass undersampling approach to get a data subset that is balanced between majority and minority classes.

However, unlike the traditional majority class undersampling techniques, majority class splitting will generate multiple balanced subsets of the original dataset. Let's consider a hypothetical case where the ratio of majority to minority classes is 3:1. With the majority class splitting approach, we will take the majority class and randomly split the majority class into 3 parts, where each part is approximately the same size as the minority class. For each of these majority class samples, we will join the sample back with the full minority class. This will in turn generate 3 balanced subsets of the original dataset, each of which contains the full minority class. This technique is depicted in the diagram below:



In the case of the training dataset for the *Airline Delays*, we have a ratio of 7:1 for the majority and minority class, which will generate 7 subsets of data using the majority splitting technique. While this is a possible dataset balance approach for us to use in model training, this approach is best-suited for ensemble approaches, where each model in the ensemble is assigned one balanced subset of data for training. With this

approach, each model in the ensemble will have a balanced dataset to learn from, reducing bias in the indiviual models, but no majority class data is lost as would be the case for traditional undersampling techniques. We will explore the use of majority class splitting when we explore ensemble approaches to predicting departure dleays in section V of the report.

IV. Algorithm Exploration

Considerations for Algorithm Exploration

In our original problem statement introduced in section I, we pointed out that the goal of this analysis is to develop a model that is able to predict significant departure delays (30 minutes or more); this inherently is a classification task, where our positive class is where `Dep_Del30 == 1` (delay) and our negative class is `Dep_Del30 == 0` (no delay). As we introduced near the start of section II, to accomplish this classification task, we considered a vareity of models suitable for classification problems for the purpose of our algorithm exploration. Based on our experience working with machine learning models in W207, we decided to consider the following models:

- Logistic Regression
- Decision Trees
- Naive Bayes
- Support Vector Machines

In picking these models for our initial exploration, we considered a variety of factors in addition to the requirement that we select a model that is well-suited for our classification task. Our three additional considerations for our algorithm selection included the following:

- the resulting model must be interpretable / explainable
- the algorithm must scale
- the algorithm must handle continuous and categorical variables

Because we want to be able to not only predict whether a significant departure delay occurs for a given flight, but also determine the underlying factors that may lead to departure delays, we were also interested in looking at explainable models, thus the consideration of Decision Trees, Logistic Regression, and Naive Bayes. Additionally, given the size of our *Airline Delays* dataset, we wanted to make sure that the models we considered could handle large datasets, such as Decision Trees and Naive Bayes. Finally, given that our dataset includes both numerical and categorical features, we wanted to make sure that we chose a model that was well suited for handling both numerical and categorical features, which all four of these models are able to do so. While not all four of these models satisfy all three of our additional conditions, we decided for the sake of exploration to look at all four regardless.

Applying our Candidate Algorithms

For the sake of the algorithm exploration, we decided to keep things fairly simple and work off of the original dataset without any major feature engineering or dataset balancing applied to the dataset. That is, for all four algorithms, we will consider all 12 relevant features for predicting departure delays described in section I and our custom-made outcome variable `Dep_Del30` .

Note that because we are not balancing the dataset for this exploration, simply taking a baseline model which predicts the outcome to be 'no delay' (the majority class) at inference time will achieve a high accuracy of about 89%, which practically speaking, is not a useful model as it would fail to identify any true positives. Regardless, for simplicity of this algorithm exploration, we will make use of the original unbalanced, un-feature engineered dataset for high-level exploration. We will also fix any relevant hyperparameters and will not look at exploring any tuning of these hyperparameters.

In the next few sections of code, we prepare the dataset for our algorithm exploration, as well as our model training and evaluation functions. Note that we will be considering a variety of metrics, including accuracy, precision, recall, F1-score, as well as aggregated metrics such as area under the curve, and the full confusion matrix. This will allow us to understand the full story of how these models perform in a baseline scenario.

In [66]:
```
# Define outcome & features to use in model development
# numFeatureNames & contNumFeatureNAmes are continuous features
# catFeatureNames are categorical features
outcomeName = 'Dep_Del30'
numFeatureNames = ['Year', 'Month', 'Day_Of_Month', 'Day_Of_Week', 'Distan
contNumFeatureNames = ['CRS_Dep_Time', 'CRS_Arr_Time', 'CRS_Elapsed_Time',
catFeatureNames = ['Op_Unique_Carrier', 'Origin', 'Dest']

# subset the dataset to the features in numFeatureNames, contNumFeatureNam
mini_train_algo = mini_train.select([outcomeName] + numFeatureNames + cont
train_algo = train.select([outcomeName] + numFeatureNames + contNumFeature
val_algo = val.select([outcomeName] + numFeatureNames + contNumFeatureName
```

In [67]:
```
# This function is for training all four of our candidate models in baseli
# Only support vector machines (svm) use one hot encoding for the categori
def train_model(df,algorithm,categoricalCols,continuousCols,labelCol,svmfl

    indexers = [ StringIndexer(inputCol=cat, outputCol= cat + "_indexed",

    # If it is svm do hot encoding
    if svmflag == True:
      encoder = OneHotEncoderEstimator(inputCols=[indexer.getOutputCol() f
                                 outputCols=["{0}_encoded".format(indexer.
      assembler = VectorAssembler(inputCols=encoder.getOutputCols() + cont
    # else skip it
    else:
      assembler = VectorAssembler(inputCols = continuousCols + [cat + "_in

    # choose the appropriate model constrution, depending on the algorithm
    if algorithm == 'lr':
      lr = LogisticRegression(labelCol = outcomeName, featuresCol="feature
```

```
    pipeline = Pipeline(stages=indexers + [assembler,lr])

  elif algorithm == 'dt':
    dt = DecisionTreeClassifier(labelCol = outcomeName, featuresCol = "f
    pipeline = Pipeline(stages=indexers + [assembler,dt])

  elif algorithm == 'nb':
    # set the CRS_Elapsed_Time to 0 if its negative
    df = df.withColumn('CRS_Elapsed_Time', when(df['CRS_Elapsed_Time'] <

    nb = NaiveBayes(labelCol = outcomeName, featuresCol = "features", sm
    pipeline = Pipeline(stages=indexers + [assembler,nb])

  elif algorithm == 'svm':
    svc = LinearSVC(labelCol = outcomeName, featuresCol = "features", ma
    pipeline = Pipeline(stages=indexers + [encoder,assembler,svc])

  else:
    pass

  tr_model=pipeline.fit(df)

  return tr_model
```

In [68]:

```
# Model Evaluation
# This function takes predictions dataframe and outcomeName, evaluates the
# and calculates the scores for multiple metrics.
# If the returnval is true it will return the values otherwise it will pri
# Predictions must have two columns: prediction & label
def EvaluateModelPredictions(predict_df, dataName=None, outcomeName='label

  if not ReturnVal :
    print("Model Evaluation - " + dataName)
    print("-----------------------------")

  evaluation_df = (predict_df \
                   .withColumn('metric', F.concat(F.col(outcomeName), F.
                   .count() \
                   .toPandas() \
                   .assign(label = lambda df: df.metric.map({'1-1.0': 'T
  metric = evaluation_df.set_index("label").to_dict()['count']

  # Default missing metrics to 0
  for key in ['TP', 'TN', 'FP', 'FN']:
    metric[key] = metric.get(key, 0)

      # Compute metrics
  total = metric['TP'] + metric['FN'] + metric['TN'] + metric['FP']
  accuracy = 0 if (total == 0) else (metric['TP'] + metric['TN'])/total
  precision = 0 if ((metric['TP'] + metric['FP']) == 0) else metric['TP'
  recall = 0 if ((metric['TP'] + metric['FN']) == 0) else metric['TP'] /
  fscore = 0 if (precision+recall) == 0 else 2 * precision * recall /(pr

  # Compute Area under ROC & PR Curve
  evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPredict
  areaUnderROC = evaluator.evaluate(predict_df, {evaluator.metricName: "
  areaUnderPRC = evaluator.evaluate(predict_df, {evaluator.metricName: "
```

```
    if ReturnVal : return {'Accuracy' : round(accuracy, 7),
                           'Precision' : round(precision, 7),
                           'Recall' : round(recall, 7),
                           'f-score' : round(fscore, 7),
                           'areaUnderROC' : round(areaUnderROC, 7),
                           'AreaUnderPRC' : round(areaUnderPRC, 7),
                           'metric' : metric}

    print("Accuracy = {}, Precision = {}, Recall = {}, f-score = {}, AreaU
        round(accuracy, 7), round(precision, 7),round(recall, 7),round(fsc
    print("\nConfusion Matrix:\n", pd.DataFrame.from_dict(metric, orient='

# This function takes a trained model, generates predictions,
# And calls model evaluation function to evaluate the predictions
def PredictAndEvaluate(model, data, dataName, outcomeName):
  predictions = model.transform(data)
  EvaluateModelPredictions(predictions, dataName, outcomeName)
```

In [69]:

```
# Train the model using the "train" dataset and evaluate against the "val"
algorithms = ['lr','dt','nb','svm']
for algorithm in algorithms:
  newnumFeatureNames = numFeatureNames + contNumFeatureNames

  # Train on training data & evaluate validation data
  tr_model = train_model(train_algo, algorithm, catFeatureNames, newnumFea
  PredictAndEvaluate(tr_model, val_algo, 'val with ' + algorithm, outcomeN
```

```
Model Evaluation - val with lr
------------------------------
Accuracy = 0.8855706, Precision = 0, Recall = 0.0, f-score = 0, AreaUnde
rROC = 0.6288745, AreaUnderPRC = 0.1603632

Confusion Matrix:
       count
FN    341760
TN   2644885
TP         0
FP         0

Model Evaluation - val with dt
------------------------------
Accuracy = 0.88558, Precision = 0.5224359, Recall = 0.0009539, f-score =
0.0019043, AreaUnderROC = 0.6010569, AreaUnderPRC = 0.1523681

Confusion Matrix:
       count
TP       326
FN    341434
TN   2644587
FP       298

Model Evaluation - val with nb
------------------------------
```

```
Accuracy = 0.5849453, Precision = 0.1540864, Recall = 0.5851328, f-score
= 0.2439358, AreaUnderROC = 0.4069267, AreaUnderPRC = 0.0894408

Confusion Matrix:
       count
TP   199975
FN   141785
TN  1547049
FP  1097836

Model Evaluation - val with svm
-----------------------------
Accuracy = 0.8856789, Precision = 0, Recall = 0.0, f-score = 0, AreaUnde
rROC = 0.5232673, AreaUnderPRC = 0.1273751

Confusion Matrix:
       count
FN   341395
TN  2644886
TP        0
FP        0
```

Analyzing Candidate Algorithm Results & Choosing our Algorithm

Based on the modeling results displayed above for all four of our candidate models, we observed the following:

- Both Logistic Regression and SVM had high accuracy but failed to predict any of the true positives. Both of these models also failed to predict any false positives. Based on this we concluded both the models were predicting "no delay" all the time.
- Naive Bayes identified the most number of true positives, but it also had roughly 5 times as many false positives. It had an accuracy slightly better than a coin toss.
- Decision Tree identified some true positives, it identified a similar set of false positives. It also demonstrated good accuracy and precision. Its recall rate wasn't very high. Overall emperically, this model looked much more balanced fit for our further analysis.

Theoretically speaking, Logistic Regression is very easy to model, interpret, and train. The main limitation of Logistic Regression is the multi-collinearity present among some of the features. Logisitc regression is also susceptible to overfitting due to imbalanced data.

Support Vector Machines on the other hand is not suitable for large datasets. As these algorithms use hyperplanes to separate the two classes in feature space, it is much harder to interpret them and larger data sets require a much longer time to process in relation to our other candidate models.

With Naive Bayes, this model is fairly simple to process and scales really well. Much like Logistic Regression, Naive Bayes does makes the naive assumption that the features under consieration are independent of each other, which is not true in many cases, including for the our scenario.

By comparison to all three of these algorithms, the Decision Tree algorithm seems to be the most promising for our scenario, especially from a theoretical perspective. Compared to the other candidate algorithms, we saw the following benefits (note that all four the the requirements described previously are satisfied by the Decision Tree algorithm):

- the algorithm is highly interpretable, given that the decision rules can be easily interpretted by anyone
- they require little-to-no data preparation during pre-processing function to successfully (can work with both categorical & numerical features)
- the algorithm also doesn't require normalization or scaling of our features and can handle null or unknown values gracefully
- they can still easily benefit from certain data preparation, such as Breiman's theorem applications & binning of numerical features
- they automatically do feature selection and can highlight feature importance using information gain metrics
- they can automatically generate relevant interaction terms
- the algorithm requires very little hyperparameter tuning
- inference is fast & explainable
- the algorithm itself can be parallelize at training, which can help with scalability

With that said, one of the consequences that do come with Decision Trees is that they can tend to overfit as they try to memorize the data, leading to an increase in variance in the model (which is the case with the results of our candidate model shown above). While this might be a limitation to make us pause and reconsider this algorithm, Decision Trees can easily be extended to the Random Forest algorithm to help with the bias-variance tradeoff that may be present in a single decision tree. Thus, for these reasons, we will continue our analysis using **Decision Trees** as our algorithm of choice and look further to extending to Random Forests and other ensembles of trees approaches in the next section.

V. Algorithm Implementation

With Decision Trees as our chosen algorithm, we will proceed to explore the algorithm in a toy example, apply it to our feature engineered dataset, and expand on the basic algorithm to random forests and ensembles of random forests.

Toy Example: Decision Trees

Given that in the previous section, we decided on Decision Trees as our algorithm of choice, we will now proceed to describe the math behind the algorithm with our toy example.

Dataset

For the toy example, we will leverage a toy dataset for motivating the algorithm explanation, which consists of a 10 records from the original *Airline Delays* dataset and includes our outcome variable `Dep_Del30`, 2 numerical features `Day_Of_Week` and `CRS_Dep_Time`, as well as 2 categorical features `Origin` and `Op_Unique_Carrier`. These are displayed below:

In [73]:
```
# Load the toy example dataset
toy_dataset = spark.read.option("header", "true").parquet(f"dbfs/user/team
display(toy_dataset)
```

Dep_Del30	Day_Of_Week	CRS_Dep_Time	Origin	Op_Unique_Carrier
0	5	1735	HOU	B6
0	2	2035	SMF	WN
1	6	1509	DCA	OH
0	6	1500	SDF	WN
1	3	2210	HOU	WN
1	3	1755	MCO	WN
0	2	1425	DEN	OO
0	5	615	SJC	DL
0	6	1940	DEN	AA
1	7	1425	MSP	EV

Introduction to Decision Trees

Decision trees predict the label (or class) by evaluating a set of rules that follow an IF-THEN-ELSE pattern in a question and answer style. The questions are the nodes, and the answers (true or false) are the branches in the tree to the child nodes, thus construct a tree-like structure of questions and answers. A decision tree model estimates the minimum number of true/false questions needed, to assess the probability of making a correct decision.

For this analysis, we leveraged the CART algorithm (Classification and Regression Trees). The Decision Tree algorithm is a greedy algorithm that considers all features to select the best feature and the best split point for that feature at each given node in the tree. Initially, we have a root node for the tree. The root node receives the entire training set as input and all subsequent nodes receive a subset of rows as input. Each node asks a true/false question about one of the features using a threshold and in response, the dataset is split into two subsets. The subsets become input to the child nodes that are

added to the tree for the next level of splitting. The goal of the algorithm is to produce the purest distribution of labels at each leaf node in the tree, using the training data.

If a node contains examples of only a single type of label, it has 100% purity and becomes a leaf node. The subset of data at the leaf node doesn't need to be split any further. On the other hand, if a node still contains mixed labels in its data subset, the decision tree algorithm chooses another question and threshold, based on which the subset is split further. The trick to building an effective tree is to decide which feature to select at each node and the best threshold for that feature. To do this, we need to quantify how well a feature and threshold can split the dataset at each step of the algorithm.

Entropy

Entropy is a measure of disorder in the dataset. It characterizes the (im)purity of an arbitrary collection of examples. In decision trees, at each node, we split the data and try to group together samples that belong in the same class. The objective is to maximize the purity of the groups each time a new child node of the tree is created. The goal is to decrease the entropy as much as possible at each split. Entropy ranges between 0 and 1, where an entropy of 0 indicates a pure set (i.e the subset of observations contains only one label).

Gini Impurity and Information Gain

We quantify the amount of uncertainity at a single node by a metric called the gini impurity. We can quantify how much a split reduces the uncertainity by using a metric called the information gain. Information gain is the expected reduction in entropy caused by partitioning the examples according to a given feature and threshold. These two metrics are used to select the best feature and threshold at each split point. The best feature reduces the uncertainity the most. Given the feature and threshold, the algorithm recursively builds the tree at each of the new child nodes. This process continues until all the nodes are pure or we reach a stopping criteria (such as a minimum number of examples).

Mathematical definition of entropy

The general formula for entropy is:

$$E = \sum_i -p_i \log_2 p_i$$

where p_i is the frequentist probability of elements in class i.

Since our outcome variable `Dep_Del30` is binary, all the observations in our toy dataset fall into one of two classes (`0` or `1`). Suppose we have N observations in the dataset. Let's assume that n observations belong to label `1` and $m = N - n$ observations belong to label `0` . p and q, the ratios of elements of each label in the dataset are given by:

$$p = \frac{n}{N}$$

$$q = \frac{m}{N} = 1 - p$$

Thus, entropy is given by the following equation:

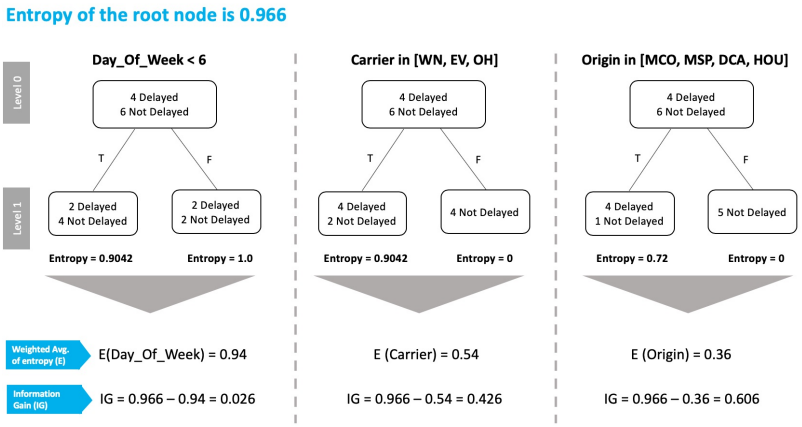
$$E = -p\log_2(p) - q\log_2(q)$$

Entropy at Level 0

In our toy dataset, we have ten observations. Four of them have label `1` and six of them have label `0`. Thus, entropy at the root node is given by:

$$Entropy = -\frac{4}{10}\log_2(\frac{4}{10}) - \frac{6}{10}\log_2(\frac{6}{10}) = 0.966$$

In this case, our entropy is close to 1, as we have a distribution close to 50/50 for the observations belonging to each class. Given the entropy at the root node, we can use this information to grow the next level in the tree.



Entropy at Level 1

The data subset that goes down each branch of the tree has its own entropy value. We can calculate the expected entropy for each possible attribute. This is the degree to which the entropy would change if we branch on a particular feature. We calculate the weighted entropy of the split by adding the entropies of the two child nodes, weighted by the proportion of examples from the parent node that ended up at that child.

Weighted entropy calculations

$$E(DayOfWeek) = -\frac{6}{10}\log_2(0.9042) - \frac{4}{10}\log_2(1) = 0.94$$

$$E(Carrier) = -\frac{6}{10}\log_2(0.9042) - \frac{4}{10}\log_2(0) = 0.54$$

$$E(Origin) = -\frac{5}{10}\log_2(0.72) - \frac{5}{10}\log_2(0) = 0.36$$

Information Gain at Level 1

Information gain gives the number of bits of information gained about the dataset by choosing a specific feature and threshold as the first branch of the decision tree, and is calculated as:

$$IG = Entropy(Parent) - WeightedEntropy(ChildNodes)$$

Based on the information gain calculations shown in the diagram above, the highest information gain is 0.606 when we use the feature `Origin` with the decision rule that the `Origin` i in the set of airports ["MCO", "MSP", "DCA", "HOU"]. Thus, among the features and split points considered in this toy example, the best feature to split on is `Origin` at level 1. This procedure can continue recursively until the appropriate stopping criteria is achieved.

Evaluating Models

Before going into training decision trees and analyzing the results they give, let us consider the evaluation metrics that we'll use to evaluate whether these models are "good" models.

As we noted in previous sections, while accuracy may be an intuitive metric for us to use to evaluate our model performance, it is not necessarily the best metric use. It's especially an issue if our dataset is imbalanced, because if we predict the majority class, 89% of the time, we'd be right. In which case, we'd have to concentrate on metrics that tell us the whole story like area under the curve & our confusion matrix.

However, since we'll focus on training our models on balanced datasets (using either SMOTE or Majority Class Splitting), accuracy becomes a more valid metric. With that said, we will still look at other metrics as well, including precision, recall, f-score, area under the curve (including AUROC and AUPRC), as well as the confusion matrix to ensure we get the full story of how our models perform. By having all the metrics, we can choose which metric to prioritize depending on the business case of interest. If we were to prioritize recall, for example, this would ensure a model that is great for helping airlines and airports prepare for delays from a resource perspective. By comparison, if we were to prioritize precision, this would help passengers better plan for delays and not miss their flights. At the end of the day, it depends on what use case we want to prioritize and for these reasons, we'll evaluate all metrics in aggregate as we proceed through this section.

Training Decision Tree on SMOTEd (Balanced) Training Dataset

For our first step towards modeling departure delays, we trained a decision tree model using all the feature engineering described in prior sections on the SMOTEd (balanced) training dataset. One of the best characteristics of a decision tree is its interpretability. From the printout of the model below, we can see that the model chose `CRS_Dep_Time_bin` as the most important feature to split on first, followed by

`Origin_Activity` and `Origin_Dest_brieman`. `Distance` and `Carrier` are considered less important features and these features are chosen further down the tree. At the root node, the decision tree splits on `CRS_Dep_Time_bin` and the threshold chosen is 115.5 (note that the bin 115 corresponds to the 10-minute block 1150, which is approximately noon). Thus, we can infer that a departure time approximately before or after noon along with origin airport can give us information about a departure delay.

We did some hyper-parameter tuning on the decision tree model using maxDepth. This parameter represents the maximum depth the tree is allowed to grow. In general, the deeper we allow the tree to grow, the more complex the model will become because there will be more splits and it captures more information about the data. However, this is one of the root causes of overfitting in decision trees. The model will fit perfectly to the training data but will not be able to generalize well on test set. Yet selecting too low a value for maxDepth will make the model underfit to the data. Thus, selecting the right maxDepth is important to build a good model.

For selecting the optimal hyperparameter value, we tried maxDepth values of 5, 10, 15, 20, 30, 50 and 100. The Accuracy does not increase much after maxDepth = 30 and Area Under ROC (AUROC) for the validation set is highest for maxDepth = 15. Since, we can easily overfit the data using a higher maxDepth, we select maxDepth = 15 for our decision tree model. Note however that even for our model trained with maxDepth = 15, the model performs much better on the SMOTEd training data used for training, but less so on the original training data and the held-out validation set. In fact, we see fairy good performance across most all our metrics for the data used for training, but see a definite drop, especially in terms of precision, F-score, and Area Under PRC. These outcomes do seem to suggest that the single decision tree likely overfit to the training data.

In [81]:

```
# Read prepared data from parquet for training
def ReadDataFromParquet(dataName):
  # Read data back directly from disk
  return spark.read.option("header", "true").parquet(f"dbfs/user/team20/fi

def ReadDataFromParquet1(dataName):
  # Read data back directly from disk
  return spark.read.option("header", "true").parquet(f"dbfs:/user/team20/f

airlines = ReadDataFromParquet('augmented')
mini_train = ReadDataFromParquet('augmented_mini_train')
train = ReadDataFromParquet('augmented_train')
val = ReadDataFromParquet('augmented_val')
test = ReadDataFromParquet('augmented_test')
train_smoted = ReadDataFromParquet1('augmented2_smoted_train_kmeans')

########################################
# Define all variables for easy reference #
########################################

# Numerical Variables to use for training
outcomeName = 'Dep_Del30'
numFeatureNames = ['Year', 'Month', 'Day_Of_Month', 'Day_Of_Week', 'Distan
contNumFeatureNames = ['CRS_Dep_Time', 'CRS_Arr_Time', 'CRS_Elapsed_Time',
```

```
catFeatureNames = ['Op_Unique_Carrier', 'Origin', 'Dest']
binFeatureNames = ['CRS_Dep_Time_bin', 'CRS_Arr_Time_bin', 'CRS_Elapsed_Ti
intFeatureNames = ['Day_Of_Year', 'Origin_Dest', 'Dep_Time_Of_Week', 'Arr_
holFeatureNames = ['Holiday']
orgFeatureNames = ['Origin_Activity'] ##
briFeatureNames = ['Op_Unique_Carrier_brieman', 'Origin_brieman', 'Dest_br
```

In [82]:

```
# Encodes a string column of labels to a column of label indices
# Set HandleInvalid to "keep" so that the indexer adds new indexes when it
# Apply string indexer to categorical, binned, and interaction features (a
def PrepStringIndexer(stringfeatureNames):
  return [StringIndexer(inputCol=f, outputCol=f+"_idx", handleInvalid="kee

# Use VectorAssembler() to merge our feature columns into a single vector
# We will not transform the dataset just yet as we will be passing the Vec
def PrepVectorAssembler(numericalFeatureNames, stringFeatureNames):
  return VectorAssembler(inputCols = numericalFeatureNames + [f + "_idx" f

# Trains a simple Decision Tree model
def TrainDecisionTreeModel(trainingData, stages, outcomeName, maxDepth, ma
  dt = DecisionTreeClassifier(labelCol = outcomeName, featuresCol = "featu
  pipeline = Pipeline(stages = stages + [dt])
  dt_model = pipeline.fit(trainingData)
  return dt_model

# Visualize the decision tree model that was trained in text form
# Note that the featureNames need to be in the same order they were provid
# to the vector assembler prior to training the model
def PrintDecisionTreeModel(model, featureNames):
  lines = model.toDebugString.split("\n")
  featuresUsed = set()
  print("\n")

  for line in lines:
    parts = line.split(" ")

    # Replace "feature #" with feature name
    if ("feature" in line):
      featureNumIdx = parts.index("(feature") + 1
      featureNum = int(parts[featureNumIdx])
      parts[featureNumIdx] = featureNames[featureNum] # replace feature nu
      parts[featureNumIdx - 1] = "" # remove word "feature"
      featuresUsed.add(featureNames[featureNum])

    # For cateogrical features, summarize sets of values selected for easi
    if ("in" in parts):
      setIdx = parts.index("in") + 1
      vals = ast.literal_eval(parts[setIdx][:-1])
      vals = list(vals)
      numVals = len(vals)
      if (len(vals) > 5):
        newVals = random.sample(vals, 5)
        newVals = [str(int(d)) for d in newVals]
        newVals.append("...")
        vals = newVals
      parts[setIdx] = str(vals) + " (" + str(numVals) + " total values)"

    line = " ".join(parts)
```

```
    print(line)

  print("\n", "Provided Features: ", featureNames)
  print("\n", "    Used Features: ", featuresUsed)
  print("\n")
```

In [83]:

```
# Prep features to use for decision tree model
featureNames = numFeatureNames + binFeatureNames + orgFeatureNames + briFe
va_base = PrepVectorAssembler(numericalFeatureNames = featureNames, string

# Train, evaluate, & display the model
dt_model = TrainDecisionTreeModel(train_smoted, [va_base], outcomeName, ma
PredictAndEvaluate(dt_model, train_smoted, 'train_smoted', outcomeName)
PredictAndEvaluate(dt_model, train, 'train', outcomeName)
PredictAndEvaluate(dt_model, val, 'val', outcomeName)
PrintDecisionTreeModel(dt_model.stages[-1], featureNames)
```

```
Model Evaluation - train_smoted
------------------------------
Accuracy = 0.680256, Precision = 0.6810461, Recall = 0.6967006, f-score
= 0.6887844, AreaUnderROC = 0.6167985, AreaUnderPRC = 0.6230655

Confusion Matrix:
        count
TP  13321477
FN   5799328
TN  12289752
FP   6238839

Model Evaluation - train
------------------------------
Accuracy = 0.6692392, Precision = 0.1723143, Recall = 0.4942987, f-score
= 0.2555448, AreaUnderROC = 0.5280442, AreaUnderPRC = 0.1332813

Confusion Matrix:
        count
TP   1222207
FN   1250401
TN  13186123
FP   5870685

Model Evaluation - val
------------------------------
Accuracy = 0.6720968, Precision = 0.174178, Recall = 0.4935288, f-score
= 0.2574838, AreaUnderROC = 0.5283781, AreaUnderPRC = 0.1338269

Confusion Matrix:
       count
TP   174762
FN   179345
TN  1891185
```

FP   828591

```
DecisionTreeClassificationModel (uid=DecisionTreeClassifier_04e8ad4e08c
0) of depth 15 with 26923 nodes
  If  CRS_Dep_Time_bin <= 115.5)
   If  Origin_Activity <= 1.5)
    If  Origin_Dest_brieman <= 63.0)
     If  CRS_Dep_Time_bin <= 72.5)
      If  CRS_Elapsed_Time_bin <= 2.5)
       If  Dest_brieman <= 221.5)
        If  Op_Unique_Carrier_brieman <= 17.5)
         If  Op_Unique_Carrier_brieman <= 1.5)
          Predict: 1.0
         Else  Op_Unique_Carrier_brieman > 1.5)
          If  Op_Unique_Carrier_brieman <= 9.5)
           If  Op_Unique_Carrier_brieman <= 6.5)
            If  Dest_brieman <= 141.5)
             If  Op_Unique_Carrier_brieman <= 3.5)
              If  CRS_Dep_Time_bin <= 67.5)
               If  Origin_brieman <= 305.5)
                Predict: 0.0
               Else  Origin_brieman > 305.5)
                Predict: 1.0
              Else  CRS_Dep_Time_bin > 67.5)
               If  Origin_brieman <= 143.5)
                If  Op_Unique_Carrier_brieman <= 2.5)
                 Predict: 1.0
                Else  Op_Unique_Carrier_brieman > 2.5)
                 Predict: 0.0
               Else  Origin_brieman > 143.5)
                If  Origin_brieman <= 228.5)
                 Predict: 1.0
                Else  Origin_brieman > 228.5)
                 Predict: 0.0
             Else  Op_Unique_Carrier_brieman > 3.5)
              If  Dest_brieman <= 77.5)
               If  Origin_brieman <= 85.5)
                If  Origin_brieman <= 75.5)
                 Predict: 1.0
                Else  Origin_brieman > 75.5)
                 Predict: 0.0
               Else  Origin_brieman > 85.5)
                Predict: 1.0
              Else  Dest_brieman > 77.5)
               Predict: 1.0
            Else  Dest_brieman > 141.5)
             If  Op_Unique_Carrier_brieman <= 5.5)
              Predict: 1.0
             Else  Op_Unique_Carrier_brieman > 5.5)
```

```
    If  CRS_Arr_Time_bin <= 64.5)
     If  CRS_Dep_Time_bin <= 53.5)
      If  Day_Of_Month <= 22.5)
       Predict: 1.0
      Else  Day_Of_Month > 22.5)
       Predict: 0.0
     Else  CRS_Dep_Time_bin > 53.5)
      Predict: 0.0
    Else  CRS_Arr_Time_bin > 64.5)
     If  Month <= 10.5)
      Predict: 1.0
     Else  Month > 10.5)
      If  Day_Of_Month <= 27.5)
       Predict: 1.0
      Else  Day_Of_Month > 27.5)
       Predict: 0.0
 Else  Op_Unique_Carrier_brieman > 6.5)
  If  Dest_brieman <= 27.5)
   Predict: 1.0
  Else  Dest_brieman > 27.5)
   If  Dest_brieman <= 58.5)
    If  Op_Unique_Carrier_brieman <= 7.5)
     If  Distance_Group <= 2.5)
      Predict: 0.0
     Else  Distance_Group > 2.5)
      Predict: 1.0
    Else  Op_Unique_Carrier_brieman > 7.5)
     Predict: 1.0
   Else  Dest_brieman > 58.5)
    If  Dest_brieman <= 77.5)
     Predict: 1.0
    Else  Dest_brieman > 77.5)
     Predict: 0.0
Else  Op_Unique_Carrier_brieman > 9.5)
 If  Op_Unique_Carrier_brieman <= 10.5)
  If  Origin_brieman <= 229.5)
   Predict: 1.0
  Else  Origin_brieman > 229.5)
   If  Origin_brieman <= 234.5)
    If  Dest_brieman <= 129.5)
     If  Dest_brieman <= 102.5)
      Predict: 1.0
     Else  Dest_brieman > 102.5)
      Predict: 0.0
    Else  Dest_brieman > 129.5)
     Predict: 1.0
   Else  Origin_brieman > 234.5)
    If  Year <= 2017.5)
     Predict: 1.0
    Else  Year > 2017.5)
     If  Dest_brieman <= 210.5)
```

```
          Predict: 1.0
         Else  Dest_brieman > 210.5)
          If  Origin_brieman <= 303.5)
           Predict: 1.0
          Else  Origin_brieman > 303.5)
           Predict: 0.0
    Else  Op_Unique_Carrier_brieman > 10.5)
     If  Origin_brieman <= 18.5)
      Predict: 1.0
     Else  Origin_brieman > 18.5)
      If  Dest_brieman <= 169.5)
       If  Dest_brieman <= 76.5)
        If  Op_Unique_Carrier_brieman <= 15.5)
         If  CRS_Arr_Time_bin <= 4.5)
          Predict: 0.0
         Else  CRS_Arr_Time_bin > 4.5)
          Predict: 1.0
        Else  Op_Unique_Carrier_brieman > 15.5)
         If  Dest_brieman <= 57.5)
          Predict: 0.0
         Else  Dest_brieman > 57.5)
          Predict: 1.0
       Else  Dest_brieman > 76.5)
        If  Origin_brieman <= 246.5)
         Predict: 0.0
        Else  Origin_brieman > 246.5)
         If  Op_Unique_Carrier_brieman <= 14.5)
          Predict: 0.0
         Else  Op_Unique_Carrier_brieman > 14.5)
          Predict: 1.0
      Else  Dest_brieman > 169.5)
       If  Dest_brieman <= 220.5)
        If  Op_Unique_Carrier_brieman <= 15.5)
         If  Origin_Dest_brieman <= -0.5)
          Predict: 1.0
         Else  Origin_Dest_brieman > -0.5)
          Predict: 0.0
        Else  Op_Unique_Carrier_brieman > 15.5)
         If  Op_Unique_Carrier_brieman <= 16.5)
          Predict: 0.0
         Else  Op_Unique_Carrier_brieman > 16.5)
          Predict: 1.0
       Else  Dest_brieman > 220.5)
        If  Op_Unique_Carrier_brieman <= 11.5)
         If  CRS_Elapsed_Time_bin <= 1.5)
          Predict: 1.0
         Else  CRS_Elapsed_Time_bin > 1.5)
          Predict: 0.0
        Else  Op_Unique_Carrier_brieman > 11.5)
         Predict: 1.0
Else  Op_Unique_Carrier_brieman > 17.5)
```

```
  If  Day_Of_Week <= 6.5)
   If  Month <= 1.5)
    If  Year <= 2015.5)
     If  Day_Of_Month <= 5.5)
      Predict: 0.0
     Else  Day_Of_Month > 5.5)
      Predict: 1.0
    Else  Year > 2015.5)
     Predict: 1.0
   Else  Month > 1.5)
    Predict: 1.0
  Else  Day_Of_Week > 6.5)
   If  Distance_Group <= 2.5)
    Predict: 1.0
   Else  Distance_Group > 2.5)
    If  CRS_Arr_Time_bin <= 77.0)
     Predict: 0.0
    Else  CRS_Arr_Time_bin > 77.0)
     Predict: 1.0
Else  Dest_brieman > 221.5)
 If  CRS_Arr_Time_bin <= 73.5)
  If  Dest_brieman <= 312.5)
   If  Year <= 2017.5)
    Predict: 1.0
   Else  Year > 2017.5)
    If  Origin_brieman <= 181.5)
     If  Dest_brieman <= 231.5)
      Predict: 1.0
     Else  Dest_brieman > 231.5)
      If  Dest_brieman <= 267.5)
       If  Dest_brieman <= 253.5)
        If  Dest_brieman <= 235.5)
         Predict: 0.0
        Else  Dest_brieman > 235.5)
         If  Month <= 11.5)
          Predict: 1.0
         Else  Month > 11.5)
          Predict: 0.0
       Else  Dest_brieman > 253.5)
        Predict: 0.0
      Else  Dest_brieman > 267.5)
       Predict: 1.0
    Else  Origin_brieman > 181.5)
     Predict: 1.0
  Else  Dest_brieman > 312.5)
   If  Op_Unique_Carrier_brieman <= 6.5)
    If  Dest_brieman <= 324.5)
     If  Day_Of_Week <= 4.5)
      Predict: 1.0
     Else  Day_Of_Week > 4.5)
      Predict: 0.0
```

```
 Else  Dest_brieman > 324.5)
  If  Op_Unique_Carrier_brieman <= 5.5)
   Predict: 1.0
  Else  Op_Unique_Carrier_brieman > 5.5)
   If  Origin_brieman <= 260.5)
    Predict: 1.0
   Else  Origin_brieman > 260.5)
    If  Origin_brieman <= 263.5)
     Predict: 0.0
    Else  Origin_brieman > 263.5)
     Predict: 1.0
Else  Op_Unique_Carrier_brieman > 6.5)
 If  Origin_brieman <= 263.5)
  If  Dest_brieman <= 351.0)
   If  Op_Unique_Carrier_brieman <= 7.5)
    If  Distance_Group <= 1.5)
     Predict: 0.0
    Else  Distance_Group > 1.5)
     Predict: 1.0
   Else  Op_Unique_Carrier_brieman > 7.5)
    If  Dest_brieman <= 337.5)
     If  Op_Unique_Carrier_brieman <= 10.5)
      Predict: 1.0
     Else  Op_Unique_Carrier_brieman > 10.5)
      If  Origin_brieman <= 68.5)
       Predict: 1.0
      Else  Origin_brieman > 68.5)
       Predict: 0.0
    Else  Dest_brieman > 337.5)
     If  Distance_Group <= 2.5)
      Predict: 1.0
     Else  Distance_Group > 2.5)
      If  Origin_brieman <= 85.5)
       Predict: 0.0
      Else  Origin_brieman > 85.5)
       Predict: 1.0
  Else  Dest_brieman > 351.0)
   Predict: 1.0
 Else  Origin_brieman > 263.5)
  If  Dest_brieman <= 324.5)
   If  Month <= 6.5)
    Predict: 1.0
   Else  Month > 6.5)
    If  Day_Of_Month <= 13.5)
     Predict: 0.0
    Else  Day_Of_Month > 13.5)
     If  Day_Of_Month <= 25.5)
      If  Day_Of_Week <= 3.5)
       Predict: 1.0
      Else  Day_Of_Week > 3.5)
       Predict: 0.0
```

```
       Else  Day_Of_Month > 25.5)
        Predict: 0.0
    Else  Dest_brieman > 324.5)
     Predict: 1.0
Else  CRS_Arr_Time_bin > 73.5)
 If  Year <= 2017.5)
  If  Origin_brieman <= 312.5)
   If  Origin_brieman <= 85.5)
    If  Origin_brieman <= 75.5)
     If  Op_Unique_Carrier_brieman <= 2.5)
      If  CRS_Elapsed_Time_bin <= 1.5)
       If  Day_Of_Week <= 2.5)
        Predict: 1.0
       Else  Day_Of_Week > 2.5)
        Predict: 0.0
      Else  CRS_Elapsed_Time_bin > 1.5)
       Predict: 1.0
     Else  Op_Unique_Carrier_brieman > 2.5)
      Predict: 1.0
    Else  Origin_brieman > 75.5)
     If  Distance_Group <= 2.5)
      Predict: 1.0
     Else  Distance_Group > 2.5)
      If  CRS_Dep_Time_bin <= 60.5)
       If  Month <= 7.5)
        Predict: 1.0
       Else  Month > 7.5)
        Predict: 0.0
      Else  CRS_Dep_Time_bin > 60.5)
       If  Dest_brieman <= 330.5)
        Predict: 1.0
       Else  Dest_brieman > 330.5)
        Predict: 0.0
   Else  Origin_brieman > 85.5)
    If  Origin_brieman <= 305.5)
     If  Month <= 1.5)
      If  CRS_Arr_Time_bin <= 87.0)
       Predict: 1.0
      Else  CRS_Arr_Time_bin > 87.0)
       If  Day_Of_Month <= 5.5)
        Predict: 0.0
       Else  Day_Of_Month > 5.5)
        Predict: 1.0
     Else  Month > 1.5)
      Predict: 1.0
    Else  Origin_brieman > 305.5)
     If  Day_Of_Week <= 1.5)
      Predict: 0.0
     Else  Day_Of_Week > 1.5)
      Predict: 1.0
  Else  Origin_brieman > 312.5)
```

```
  If  Year <= 2016.5)
   If  Dest_brieman <= 348.5)
    If  Dest_brieman <= 307.5)
     Predict: 1.0
    Else  Dest_brieman > 307.5)
     If  Origin_brieman <= 336.5)
      If  Op_Unique_Carrier_brieman <= 6.5)
       Predict: 1.0
      Else  Op_Unique_Carrier_brieman > 6.5)
       If  CRS_Dep_Time_bin <= 54.5)
        Predict: 1.0
       Else  CRS_Dep_Time_bin > 54.5)
        Predict: 0.0
     Else  Origin_brieman > 336.5)
      Predict: 1.0
   Else  Dest_brieman > 348.5)
    Predict: 1.0
  Else  Year > 2016.5)
   Predict: 1.0
Else  Year > 2017.5)
 If  Op_Unique_Carrier_brieman <= 5.5)
  If  CRS_Elapsed_Time_bin <= 1.5)
   If  Day_Of_Week <= 1.5)
    Predict: 0.0
   Else  Day_Of_Week > 1.5)
    Predict: 1.0
  Else  CRS_Elapsed_Time_bin > 1.5)
   Predict: 1.0
 Else  Op_Unique_Carrier_brieman > 5.5)
  If  Origin_brieman <= 234.5)
   If  Origin_brieman <= 75.5)
    If  CRS_Arr_Time_bin <= 74.5)
     If  Origin_brieman <= 11.5)
      If  Dest_brieman <= 228.5)
       Predict: 1.0
      Else  Dest_brieman > 228.5)
       Predict: 0.0
     Else  Origin_brieman > 11.5)
      Predict: 1.0
    Else  CRS_Arr_Time_bin > 74.5)
     Predict: 1.0
   Else  Origin_brieman > 75.5)
    If  Dest_brieman <= 345.0)
     If  Dest_brieman <= 228.5)
      Predict: 1.0
     Else  Dest_brieman > 228.5)
      If  Dest_brieman <= 267.5)
       Predict: 0.0
      Else  Dest_brieman > 267.5)
       If  Origin_brieman <= 85.5)
        Predict: 0.0
```

```
          Else  Origin_brieman > 85.5)
           Predict: 1.0
         Else  Dest_brieman > 345.0)
          Predict: 1.0
       Else  Origin_brieman > 234.5)
        If  Op_Unique_Carrier_brieman <= 18.5)
         If  Origin_Dest_brieman <= 16.0)
          If  Op_Unique_Carrier_brieman <= 17.5)
           Predict: 1.0
          Else  Op_Unique_Carrier_brieman > 17.5)
           If  Day_Of_Month <= 24.5)
            Predict: 1.0
           Else  Day_Of_Month > 24.5)
            Predict: 0.0
         Else  Origin_Dest_brieman > 16.0)
          Predict: 0.0
        Else  Op_Unique_Carrier_brieman > 18.5)
         If  Origin_brieman <= 282.5)
          Predict: 1.0
         Else  Origin_brieman > 282.5)
          If  CRS_Dep_Time_bin <= 62.5)
           Predict: 1.0
          Else  CRS_Dep_Time_bin > 62.5)
           Predict: 0.0
Else  CRS_Elapsed_Time_bin > 2.5)
 If  Op_Unique_Carrier_brieman <= 6.5)
  If  Dest_brieman <= 83.5)
   If  Origin_brieman <= 230.5)
    If  Month <= 2.5)
     If  CRS_Dep_Time_bin <= 55.5)
      If  Distance_Group <= 3.5)
       If  CRS_Arr_Time_bin <= 80.5)
        Predict: 1.0
       Else  CRS_Arr_Time_bin > 80.5)
        If  Day_Of_Month <= 25.5)
         If  Day_Of_Month <= 7.5)
          If  Day_Of_Month <= 6.5)
           Predict: 0.0
          Else  Day_Of_Month > 6.5)
           Predict: 1.0
         Else  Day_Of_Month > 7.5)
          Predict: 0.0
        Else  Day_Of_Month > 25.5)
         If  Day_Of_Month <= 26.5)
          Predict: 1.0
         Else  Day_Of_Month > 26.5)
          Predict: 0.0
      Else  Distance_Group > 3.5)
       Predict: 1.0
     Else  CRS_Dep_Time_bin > 55.5)
      Predict: 1.0
```

```
 Else  Month > 2.5)
  Predict: 1.0
Else  Origin_brieman > 230.5)
 If  Origin_brieman <= 234.5)
  If  Day_Of_Week <= 6.5)
   If  CRS_Arr_Time_bin <= 102.5)
    If  Day_Of_Month <= 20.5)
     If  Day_Of_Month <= 16.5)
      If  CRS_Dep_Time_bin <= 55.5)
       If  Year <= 2016.5)
        Predict: 1.0
       Else  Year > 2016.5)
        Predict: 0.0
      Else  CRS_Dep_Time_bin > 55.5)
       Predict: 0.0
     Else  Day_Of_Month > 16.5)
      If  Day_Of_Week <= 5.5)
       Predict: 0.0
      Else  Day_Of_Week > 5.5)
       If  Day_Of_Month <= 19.5)
        Predict: 0.0
       Else  Day_Of_Month > 19.5)
        Predict: 1.0
    Else  Day_Of_Month > 20.5)
     If  Day_Of_Month <= 29.5)
      Predict: 0.0
     Else  Day_Of_Month > 29.5)
      If  Month <= 5.5)
       Predict: 0.0
      Else  Month > 5.5)
       If  Month <= 7.5)
        Predict: 1.0
       Else  Month > 7.5)
        Predict: 0.0
   Else  CRS_Arr_Time_bin > 102.5)
    Predict: 1.0
  Else  Day_Of_Week > 6.5)
   If  Month <= 9.5)
    If  Day_Of_Month <= 28.5)
     If  Day_Of_Month <= 13.5)
      Predict: 0.0
     Else  Day_Of_Month > 13.5)
      If  Day_Of_Month <= 15.5)
       Predict: 1.0
      Else  Day_Of_Month > 15.5)
       Predict: 0.0
    Else  Day_Of_Month > 28.5)
     If  Month <= 4.5)
      Predict: 0.0
     Else  Month > 4.5)
      Predict: 1.0
```

```
    Else  Month > 9.5)
     If  Day_Of_Month <= 4.5)
      Predict: 0.0
     Else  Day_Of_Month > 4.5)
      If  Day_Of_Month <= 8.5)
       Predict: 1.0
      Else  Day_Of_Month > 8.5)
       If  Day_Of_Month <= 26.5)
        Predict: 0.0
       Else  Day_Of_Month > 26.5)
        Predict: 1.0
  Else  Origin_brieman > 234.5)
   If  CRS_Dep_Time_bin <= 70.5)
    Predict: 1.0
   Else  CRS_Dep_Time_bin > 70.5)
    If  Op_Unique_Carrier_brieman <= 2.5)
     If  Dest_brieman <= 57.5)
      If  Day_Of_Month <= 7.5)
       If  Year <= 2017.5)
        Predict: 0.0
       Else  Year > 2017.5)
        If  Month <= 2.5)
         Predict: 0.0
        Else  Month > 2.5)
         Predict: 1.0
      Else  Day_Of_Month > 7.5)
       Predict: 0.0
     Else  Dest_brieman > 57.5)
      Predict: 1.0
    Else  Op_Unique_Carrier_brieman > 2.5)
     Predict: 1.0
Else  Dest_brieman > 83.5)
 If  Dest_brieman <= 89.5)
  If  Year <= 2015.5)
   If  CRS_Arr_Time_bin <= 92.5)
    Predict: 1.0
   Else  CRS_Arr_Time_bin > 92.5)
    If  Day_Of_Month <= 19.5)
     Predict: 0.0
    Else  Day_Of_Month > 19.5)
     If  Day_Of_Month <= 21.5)
      Predict: 1.0
     Else  Day_Of_Month > 21.5)
      Predict: 0.0
  Else  Year > 2015.5)
   If  Day_Of_Month <= 2.5)
    If  Month <= 3.5)
     Predict: 0.0
    Else  Month > 3.5)
     Predict: 1.0
   Else  Day_Of_Month > 2.5)
```

```
      Predict: 1.0
    Else  Dest_brieman > 89.5)
     Predict: 1.0
  Else  Op_Unique_Carrier_brieman > 6.5)
   If  Origin_brieman <= 34.5)
    If  Op_Unique_Carrier_brieman <= 7.5)
     If  Distance_Group <= 3.5)
      If  Origin_brieman <= 18.5)
       Predict: 1.0
      Else  Origin_brieman > 18.5)
       If  Dest_brieman <= 100.5)
        If  Day_Of_Month <= 9.5)
         If  Day_Of_Month <= 8.5)
          Predict: 0.0
         Else  Day_Of_Month > 8.5)
          If  Month <= 10.5)
           Predict: 1.0
          Else  Month > 10.5)
           Predict: 0.0
        Else  Day_Of_Month > 9.5)
         Predict: 0.0
       Else  Dest_brieman > 100.5)
        If  Day_Of_Month <= 10.5)
         Predict: 0.0
        Else  Day_Of_Month > 10.5)
         If  Month <= 3.5)
          If  Day_Of_Week <= 4.5)
           If  Day_Of_Month <= 13.5)
            Predict: 0.0
           Else  Day_Of_Month > 13.5)
            Predict: 1.0
          Else  Day_Of_Week > 4.5)
           Predict: 0.0
         Else  Month > 3.5)
          Predict: 1.0
     Else  Distance_Group > 3.5)
      If  Day_Of_Week <= 6.5)
       Predict: 1.0
      Else  Day_Of_Week > 6.5)
       If  Origin_brieman <= 18.5)
        If  Month <= 7.5)
         Predict: 0.0
        Else  Month > 7.5)
         Predict: 1.0
       Else  Origin_brieman > 18.5)
        Predict: 1.0
    Else  Op_Unique_Carrier_brieman > 7.5)
     Predict: 1.0
   Else  Origin_brieman > 34.5)
    If  Origin_brieman <= 263.5)
     If  Dest_brieman <= 167.5)
```

```
If  Origin_brieman <= 179.5)
 If  Origin_brieman <= 66.5)
  If  CRS_Dep_Time_bin <= 60.5)
   If  Month <= 10.5)
    If  Year <= 2015.5)
     If  Month <= 5.5)
      Predict: 0.0
     Else  Month > 5.5)
      Predict: 1.0
    Else  Year > 2015.5)
     Predict: 1.0
   Else  Month > 10.5)
    If  Year <= 2016.5)
     Predict: 1.0
    Else  Year > 2016.5)
     Predict: 0.0
  Else  CRS_Dep_Time_bin > 60.5)
   Predict: 1.0
 Else  Origin_brieman > 66.5)
  If  Origin_brieman <= 171.5)
   Predict: 1.0
  Else  Origin_brieman > 171.5)
   If  Dest_brieman <= 123.5)
    Predict: 1.0
   Else  Dest_brieman > 123.5)
    If  Dest_brieman <= 126.5)
     Predict: 0.0
    Else  Dest_brieman > 126.5)
     Predict: 1.0
Else  Origin_brieman > 179.5)
 If  Dest_brieman <= 94.5)
  If  Dest_brieman <= 75.0)
   If  Origin_brieman <= 181.5)
    If  Day_Of_Week <= 5.5)
     If  Dest_brieman <= 42.5)
      Predict: 0.0
     Else  Dest_brieman > 42.5)
      Predict: 1.0
    Else  Day_Of_Week > 5.5)
     Predict: 0.0
   Else  Origin_brieman > 181.5)
    Predict: 1.0
  Else  Dest_brieman > 75.0)
   If  Origin_brieman <= 196.5)
    If  Origin_brieman <= 181.5)
     If  Dest_brieman <= 90.5)
      Predict: 1.0
     Else  Dest_brieman > 90.5)
      Predict: 0.0
    Else  Origin_brieman > 181.5)
     Predict: 1.0
```

```
                Else  Origin_brieman > 196.5)
                 If  Origin_brieman <= 253.5)
                  If  Op_Unique_Carrier_brieman <= 18.5)
                   Predict: 0.0
                  Else  Op_Unique_Carrier_brieman > 18.5)
                   Predict: 1.0
                 Else  Origin_brieman > 253.5)
                  Predict: 1.0
               Else  Dest_brieman > 94.5)
                If  Origin_brieman <= 181.5)
                 If  CRS_Dep_Time_bin <= 63.5)
                  If  CRS_Dep_Time_bin <= 60.5)
                   Predict: 0.0
                  Else  CRS_Dep_Time_bin > 60.5)
                   Predict: 1.0
                 Else  CRS_Dep_Time_bin > 63.5)
                  If  Distance_Group <= 5.5)
                   If  Year <= 2017.5)
                    Predict: 1.0
                   Else  Year > 2017.5)
                    Predict: 0.0
                  Else  Distance_Group > 5.5)
                   Predict: 1.0
                Else  Origin_brieman > 181.5)
                 If  Origin_brieman <= 230.5)
                  If  CRS_Arr_Time_bin <= 71.5)
                   If  CRS_Dep_Time_bin <= 53.5)
                    Predict: 1.0
                   Else  CRS_Dep_Time_bin > 53.5)
                    Predict: 0.0
                  Else  CRS_Arr_Time_bin > 71.5)
                   Predict: 1.0
                 Else  Origin_brieman > 230.5)
                  If  Dest_brieman <= 129.5)
                   If  Dest_brieman <= 100.5)
                    Predict: 1.0
                   Else  Dest_brieman > 100.5)
                    Predict: 0.0
                  Else  Dest_brieman > 129.5)
                   Predict: 1.0
             Else  Dest_brieman > 167.5)
              If  Dest_brieman <= 339.5)
               If  Origin_brieman <= 53.5)
                If  Distance_Group <= 4.5)

*** WARNING: skipped 1481050 bytes of output ***

                   Predict: 0.0
                  Else  Day_Of_Month > 30.5)
                   Predict: 1.0
                 Else  Month > 9.5)
```

```
      If  Month <= 11.5)
       Predict: 1.0
      Else  Month > 11.5)
       Predict: 0.0
Else  Origin_Dest_brieman > 5621.5)
 If  Year <= 2017.5)
  If  Year <= 2015.5)
   If  Dest_brieman <= 345.0)
    If  Distance_Group <= 2.5)
     If  Op_Unique_Carrier_brieman <= 3.5)
      If  Dest_brieman <= 324.5)
       Predict: 0.0
      Else  Dest_brieman > 324.5)
       Predict: 1.0
     Else  Op_Unique_Carrier_brieman > 3.5)
      Predict: 0.0
    Else  Distance_Group > 2.5)
     If  Day_Of_Month <= 5.5)
      Predict: 1.0
     Else  Day_Of_Month > 5.5)
      If  Day_Of_Month <= 8.5)
       Predict: 0.0
      Else  Day_Of_Month > 8.5)
       Predict: 1.0
   Else  Dest_brieman > 345.0)
    If  Dest_brieman <= 348.5)
     If  Month <= 11.5)
      If  Day_Of_Month <= 19.5)
       Predict: 1.0
      Else  Day_Of_Month > 19.5)
       Predict: 0.0
     Else  Month > 11.5)
      Predict: 1.0
    Else  Dest_brieman > 348.5)
     If  Day_Of_Month <= 22.5)
      If  Day_Of_Month <= 5.5)
       Predict: 1.0
      Else  Day_Of_Month > 5.5)
       Predict: 0.0
     Else  Day_Of_Month > 22.5)
      Predict: 1.0
  Else  Year > 2015.5)
   If  Day_Of_Month <= 25.5)
    If  Day_Of_Month <= 23.5)
     Predict: 1.0
    Else  Day_Of_Month > 23.5)
     If  Month <= 9.5)
      Predict: 0.0
     Else  Month > 9.5)
      If  Month <= 11.5)
       Predict: 1.0
```

```
      Else  Month > 11.5)
       Predict: 0.0
    Else  Day_Of_Month > 25.5)
     If  Month <= 9.5)
      Predict: 1.0
     Else  Month > 9.5)
      If  Day_Of_Month <= 30.5)
       Predict: 1.0
      Else  Day_Of_Month > 30.5)
       If  Year <= 2016.5)
        Predict: 1.0
       Else  Year > 2016.5)
        Predict: 0.0
  Else  Year > 2017.5)
   If  Day_Of_Month <= 29.5)
    If  Day_Of_Month <= 6.5)
     Predict: 1.0
    Else  Day_Of_Month > 6.5)
     If  Dest_brieman <= 345.0)
      If  Dest_brieman <= 144.5)
       If  CRS_Arr_Time_bin <= 174.5)
        Predict: 0.0
       Else  CRS_Arr_Time_bin > 174.5)
        Predict: 1.0
      Else  Dest_brieman > 144.5)
       If  Day_Of_Month <= 28.5)
        Predict: 0.0
       Else  Day_Of_Month > 28.5)
        Predict: 1.0
     Else  Dest_brieman > 345.0)
      If  Day_Of_Month <= 9.5)
       If  Dest_brieman <= 349.5)
        Predict: 0.0
       Else  Dest_brieman > 349.5)
        Predict: 1.0
      Else  Day_Of_Month > 9.5)
       Predict: 1.0
   Else  Day_Of_Month > 29.5)
    If  Day_Of_Month <= 30.5)
     If  Dest_brieman <= 355.5)
      Predict: 0.0
     Else  Dest_brieman > 355.5)
      If  CRS_Arr_Time_bin <= 217.0)
       If  Dest_brieman <= 358.5)
        Predict: 1.0
       Else  Dest_brieman > 358.5)
        Predict: 0.0
      Else  CRS_Arr_Time_bin > 217.0)
       Predict: 0.0
    Else  Day_Of_Month > 30.5)
     Predict: 1.0
```

```
Else  Op_Unique_Carrier_brieman > 17.5)
 If  Origin_Activity <= 6.5)
  If  Origin_Dest_brieman <= 5117.5)
   If  Origin_brieman <= 253.5)
    If  Origin_Activity <= 4.5)
     If  CRS_Arr_Time_bin <= 193.5)
      If  Origin_Dest_brieman <= 4375.0)
       If  CRS_Arr_Time_bin <= 41.0)
        If  Day_Of_Month <= 16.5)
         Predict: 1.0
        Else  Day_Of_Month > 16.5)
         Predict: 0.0
       Else  CRS_Arr_Time_bin > 41.0)
        If  CRS_Elapsed_Time_bin <= 5.5)
         Predict: 0.0
        Else  CRS_Elapsed_Time_bin > 5.5)
         Predict: 1.0
      Else  Origin_Dest_brieman > 4375.0)
       If  Origin_brieman <= 247.5)
        If  Month <= 9.5)
         Predict: 0.0
        Else  Month > 9.5)
         Predict: 1.0
       Else  Origin_brieman > 247.5)
        Predict: 1.0
     Else  CRS_Arr_Time_bin > 193.5)
      If  Year <= 2017.5)
       If  Op_Unique_Carrier_brieman <= 18.5)
        Predict: 0.0
       Else  Op_Unique_Carrier_brieman > 18.5)
        If  Day_Of_Month <= 24.5)
         Predict: 1.0
        Else  Day_Of_Month > 24.5)
         Predict: 0.0
      Else  Year > 2017.5)
       If  Dest_brieman <= 181.5)
        If  Day_Of_Month <= 23.5)
         Predict: 1.0
        Else  Day_Of_Month > 23.5)
         Predict: 0.0
       Else  Dest_brieman > 181.5)
        If  Day_Of_Month <= 25.5)
         Predict: 0.0
        Else  Day_Of_Month > 25.5)
         Predict: 1.0
    Else  Origin_Activity > 4.5)
     If  Distance_Group <= 1.5)
      Predict: 1.0
     Else  Distance_Group > 1.5)
      If  Origin_brieman <= 123.5)
       If  Dest_brieman <= 330.5)
```

```
      Predict: 1.0
     Else  Dest_brieman > 330.5)
      If  Year <= 2017.5)
       Predict: 1.0
      Else  Year > 2017.5)
       Predict: 0.0
    Else  Origin_brieman > 123.5)
     If  Day_Of_Month <= 3.5)
      If  CRS_Arr_Time_bin <= 52.5)
       Predict: 0.0
      Else  CRS_Arr_Time_bin > 52.5)
       Predict: 1.0
     Else  Day_Of_Month > 3.5)
      Predict: 1.0
Else  Origin_brieman > 253.5)
 If  Dest_brieman <= 345.0)
  If  CRS_Arr_Time_bin <= 191.5)
   If  CRS_Dep_Time_bin <= 210.5)
    If  Day_Of_Month <= 17.5)
     If  Origin_brieman <= 326.5)
      Predict: 0.0
     Else  Origin_brieman > 326.5)
      Predict: 1.0
    Else  Day_Of_Month > 17.5)
     If  Origin_brieman <= 299.5)
      Predict: 1.0
     Else  Origin_brieman > 299.5)
      Predict: 0.0
   Else  CRS_Dep_Time_bin > 210.5)
    If  Op_Unique_Carrier_brieman <= 18.5)
     If  CRS_Arr_Time_bin <= 77.0)
      Predict: 0.0
     Else  CRS_Arr_Time_bin > 77.0)
      Predict: 1.0
    Else  Op_Unique_Carrier_brieman > 18.5)
     If  Dest_brieman <= 158.5)
      Predict: 1.0
     Else  Dest_brieman > 158.5)
      Predict: 0.0
  Else  CRS_Arr_Time_bin > 191.5)
   If  CRS_Dep_Time_bin <= 214.5)
    If  Dest_brieman <= 329.5)
     Predict: 0.0
    Else  Dest_brieman > 329.5)
     If  Distance_Group <= 1.5)
      Predict: 0.0
     Else  Distance_Group > 1.5)
      Predict: 1.0
   Else  CRS_Dep_Time_bin > 214.5)
    If  Distance_Group <= 3.5)
     Predict: 1.0
```

```
      Else  Distance_Group > 3.5)
       Predict: 0.0
    Else  Dest_brieman > 345.0)
     If  Year <= 2017.5)
      If  CRS_Arr_Time_bin <= 213.5)
       If  CRS_Dep_Time_bin <= 163.5)
        If  Distance_Group <= 10.5)
         Predict: 1.0
        Else  Distance_Group > 10.5)
         Predict: 0.0
       Else  CRS_Dep_Time_bin > 163.5)
        Predict: 1.0
      Else  CRS_Arr_Time_bin > 213.5)
       If  Day_Of_Month <= 8.5)
        Predict: 0.0
       Else  Day_Of_Month > 8.5)
        If  CRS_Arr_Time_bin <= 220.5)
         Predict: 0.0
        Else  CRS_Arr_Time_bin > 220.5)
         Predict: 1.0
     Else  Year > 2017.5)
      If  Month <= 10.5)
       If  Day_Of_Month <= 12.5)
        If  Day_Of_Month <= 7.5)
         Predict: 0.0
        Else  Day_Of_Month > 7.5)
         Predict: 1.0
       Else  Day_Of_Month > 12.5)
        Predict: 0.0
      Else  Month > 10.5)
       If  CRS_Dep_Time_bin <= 212.5)
        If  Day_Of_Month <= 11.5)
         Predict: 0.0
        Else  Day_Of_Month > 11.5)
         Predict: 1.0
       Else  CRS_Dep_Time_bin > 212.5)
        If  Day_Of_Month <= 4.5)
         Predict: 1.0
        Else  Day_Of_Month > 4.5)
         Predict: 0.0
  Else  Origin_Dest_brieman > 5117.5)
   If  Origin_brieman <= 322.5)
    If  Origin_brieman <= 299.5)
     If  CRS_Arr_Time_bin <= 60.5)
      If  Dest_brieman <= 278.5)
       If  Year <= 2016.5)
        If  Year <= 2015.5)
         Predict: 0.0
        Else  Year > 2015.5)
         Predict: 1.0
       Else  Year > 2016.5)
```

```
    If  Dest_brieman <= 228.5)
     Predict: 1.0
    Else  Dest_brieman > 228.5)
     Predict: 0.0
  Else  Dest_brieman > 278.5)
   If  Day_Of_Month <= 20.5)
    If  CRS_Dep_Time_bin <= 222.5)
     Predict: 0.0
    Else  CRS_Dep_Time_bin > 222.5)
     Predict: 1.0
   Else  Day_Of_Month > 20.5)
    If  CRS_Dep_Time_bin <= 204.5)
     Predict: 0.0
    Else  CRS_Dep_Time_bin > 204.5)
     Predict: 1.0
 Else  CRS_Arr_Time_bin > 60.5)
  If  Origin_Activity <= 3.5)
   Predict: 1.0
  Else  Origin_Activity > 3.5)
   If  CRS_Dep_Time_bin <= 187.5)
    If  CRS_Arr_Time_bin <= 230.5)
     Predict: 1.0
    Else  CRS_Arr_Time_bin > 230.5)
     Predict: 0.0
   Else  CRS_Dep_Time_bin > 187.5)
    Predict: 1.0
Else  Origin_brieman > 299.5)
 If  CRS_Dep_Time_bin <= 200.5)
  If  Dest_brieman <= 217.5)
   If  Dest_brieman <= 112.5)
    Predict: 1.0
   Else  Dest_brieman > 112.5)
    If  Origin_brieman <= 316.5)
     Predict: 0.0
    Else  Origin_brieman > 316.5)
     Predict: 1.0
  Else  Dest_brieman > 217.5)
   If  Day_Of_Month <= 29.5)
    Predict: 1.0
   Else  Day_Of_Month > 29.5)
    Predict: 0.0
 Else  CRS_Dep_Time_bin > 200.5)
  If  Op_Unique_Carrier_brieman <= 18.5)
   If  Dest_brieman <= 240.5)
    If  Day_Of_Month <= 24.5)
     Predict: 1.0
    Else  Day_Of_Month > 24.5)
     Predict: 0.0
   Else  Dest_brieman > 240.5)
    Predict: 0.0
  Else  Op_Unique_Carrier_brieman > 18.5)
```

```
    If  Origin_Activity <= 3.5)
     If  Distance_Group <= 4.5)
      Predict: 1.0
     Else  Distance_Group > 4.5)
      Predict: 0.0
    Else  Origin_Activity > 3.5)
     Predict: 1.0
Else  Origin_brieman > 322.5)
 If  Day_Of_Month <= 14.5)
  If  Year <= 2016.5)
   If  CRS_Dep_Time_bin <= 174.5)
    If  Origin_Dest_brieman <= 5170.5)
     Predict: 1.0
    Else  Origin_Dest_brieman > 5170.5)
     Predict: 0.0
   Else  CRS_Dep_Time_bin > 174.5)
    If  Origin_Dest_brieman <= 5813.0)
     If  CRS_Arr_Time_bin <= 4.5)
      Predict: 1.0
     Else  CRS_Arr_Time_bin > 4.5)
      Predict: 0.0
    Else  Origin_Dest_brieman > 5813.0)
     Predict: 1.0
  Else  Year > 2016.5)
   If  Distance_Group <= 1.5)
    If  Month <= 10.5)
     Predict: 0.0
    Else  Month > 10.5)
     If  Day_Of_Month <= 2.5)
      Predict: 1.0
     Else  Day_Of_Month > 2.5)
      Predict: 0.0
   Else  Distance_Group > 1.5)
    If  Origin_Activity <= 3.5)
     If  CRS_Arr_Time_bin <= 214.5)
      Predict: 1.0
     Else  CRS_Arr_Time_bin > 214.5)
      Predict: 0.0
    Else  Origin_Activity > 3.5)
     If  CRS_Arr_Time_bin <= 0.5)
      Predict: 0.0
     Else  CRS_Arr_Time_bin > 0.5)
      Predict: 1.0
 Else  Day_Of_Month > 14.5)
  If  Origin_Dest_brieman <= 5221.5)
   If  Dest_brieman <= 277.5)
    If  CRS_Dep_Time_bin <= 162.5)
     Predict: 0.0
    Else  CRS_Dep_Time_bin > 162.5)
     If  Month <= 9.5)
      Predict: 0.0
```

```
          Else  Month > 9.5)
           Predict: 1.0
         Else  Dest_brieman > 277.5)
          If  Day_Of_Month <= 25.5)
           Predict: 0.0
          Else  Day_Of_Month > 25.5)
           If  Year <= 2016.5)
            Predict: 0.0
           Else  Year > 2016.5)
            Predict: 1.0
       Else  Origin_Dest_brieman > 5221.5)
        If  Origin_Activity <= 3.5)
         If  Day_Of_Month <= 18.5)
          If  Dest_brieman <= 337.5)
           Predict: 1.0
          Else  Dest_brieman > 337.5)
           Predict: 0.0
         Else  Day_Of_Month > 18.5)
          Predict: 1.0
        Else  Origin_Activity > 3.5)
         If  Origin_brieman <= 332.5)
          Predict: 1.0
         Else  Origin_brieman > 332.5)
          If  Year <= 2015.5)
           Predict: 0.0
          Else  Year > 2015.5)
           Predict: 1.0
Else  Origin_Activity > 6.5)
 If  Origin_Dest_brieman <= 5117.5)
  If  Origin_Activity <= 11.5)
   If  Origin_brieman <= 262.5)
    If  Day_Of_Month <= 22.5)
     If  CRS_Dep_Time_bin <= 162.5)
      If  Day_Of_Month <= 5.5)
       Predict: 1.0
      Else  Day_Of_Month > 5.5)
       Predict: 0.0
     Else  CRS_Dep_Time_bin > 162.5)
      If  Dest_brieman <= 330.5)
       If  Origin_brieman <= 67.5)
        Predict: 0.0
       Else  Origin_brieman > 67.5)
        Predict: 1.0
      Else  Dest_brieman > 330.5)
       If  CRS_Dep_Time_bin <= 224.5)
        Predict: 1.0
       Else  CRS_Dep_Time_bin > 224.5)
        Predict: 0.0
    Else  Day_Of_Month > 22.5)
     If  Origin_Dest_brieman <= 4446.5)
      If  CRS_Arr_Time_bin <= 74.5)
```

```
    If  Day_Of_Month <= 25.5)
     Predict: 0.0
    Else  Day_Of_Month > 25.5)
     Predict: 1.0
   Else  CRS_Arr_Time_bin > 74.5)
    Predict: 1.0
  Else  Origin_Dest_brieman > 4446.5)
   If  Dest_brieman <= 260.5)
    Predict: 1.0
   Else  Dest_brieman > 260.5)
    If  Year <= 2015.5)
     Predict: 1.0
    Else  Year > 2015.5)
     Predict: 0.0
Else  Origin_brieman > 262.5)
 If  Origin_Activity <= 8.5)
  If  CRS_Dep_Time_bin <= 201.5)
   If  CRS_Arr_Time_bin <= 224.5)
    If  CRS_Arr_Time_bin <= 184.5)
     Predict: 0.0
    Else  CRS_Arr_Time_bin > 184.5)
     Predict: 1.0
   Else  CRS_Arr_Time_bin > 224.5)
    Predict: 0.0
  Else  CRS_Dep_Time_bin > 201.5)
   If  Origin_brieman <= 300.5)
    If  Dest_brieman <= 155.5)
     Predict: 1.0
    Else  Dest_brieman > 155.5)
     Predict: 0.0
   Else  Origin_brieman > 300.5)
    Predict: 1.0
 Else  Origin_Activity > 8.5)
  If  Year <= 2017.5)
   If  CRS_Dep_Time_bin <= 177.5)
    If  Day_Of_Month <= 8.5)
     Predict: 0.0
    Else  Day_Of_Month > 8.5)
     Predict: 1.0
   Else  CRS_Dep_Time_bin > 177.5)
    If  Dest_brieman <= 345.0)
     Predict: 1.0
    Else  Dest_brieman > 345.0)
     Predict: 0.0
  Else  Year > 2017.5)
   If  CRS_Elapsed_Time_bin <= 5.5)
    If  CRS_Elapsed_Time_bin <= 3.5)
     Predict: 0.0
    Else  CRS_Elapsed_Time_bin > 3.5)
     Predict: 1.0
   Else  CRS_Elapsed_Time_bin > 5.5)
```

```
     Predict: 0.0
Else  Origin_Activity > 11.5)
 If  Month <= 11.5)
  If  CRS_Elapsed_Time_bin <= 5.5)
   If  CRS_Arr_Time_bin <= 231.5)
    If  Day_Of_Month <= 4.5)
     If  Month <= 9.5)
      Predict: 0.0
     Else  Month > 9.5)
      Predict: 1.0
    Else  Day_Of_Month > 4.5)
     Predict: 1.0
   Else  CRS_Arr_Time_bin > 231.5)
    If  CRS_Elapsed_Time_bin <= 3.5)
     If  Day_Of_Month <= 7.5)
      Predict: 1.0
     Else  Day_Of_Month > 7.5)
      Predict: 0.0
    Else  CRS_Elapsed_Time_bin > 3.5)
     If  Origin_Activity <= 36.5)
      Predict: 0.0
     Else  Origin_Activity > 36.5)
      Predict: 1.0
  Else  CRS_Elapsed_Time_bin > 5.5)
   If  Year <= 2016.5)
    If  Day_Of_Month <= 22.5)
     If  CRS_Dep_Time_bin <= 231.5)
      Predict: 1.0
     Else  CRS_Dep_Time_bin > 231.5)
      Predict: 0.0
    Else  Day_Of_Month > 22.5)
     If  CRS_Dep_Time_bin <= 183.5)
      Predict: 1.0
     Else  CRS_Dep_Time_bin > 183.5)
      Predict: 0.0
   Else  Year > 2016.5)
    If  CRS_Dep_Time_bin <= 183.5)
     Predict: 0.0
    Else  CRS_Dep_Time_bin > 183.5)
     If  Dest_brieman <= 170.5)
      Predict: 1.0
     Else  Dest_brieman > 170.5)
      Predict: 0.0
 Else  Month > 11.5)
  If  Day_Of_Month <= 13.5)
   If  Origin_Activity <= 15.5)
    If  CRS_Dep_Time_bin <= 187.5)
     If  CRS_Arr_Time_bin <= 221.5)
      Predict: 1.0
     Else  CRS_Arr_Time_bin > 221.5)
      Predict: 0.0
```

```
      Else  CRS_Dep_Time_bin > 187.5)
       Predict: 1.0
     Else  Origin_Activity > 15.5)
      If  Dest_brieman <= 249.5)
       If  CRS_Arr_Time_bin <= 211.5)
        Predict: 0.0
       Else  CRS_Arr_Time_bin > 211.5)
        Predict: 1.0
      Else  Dest_brieman > 249.5)
       If  Origin_Dest_brieman <= 4998.5)
        Predict: 0.0
       Else  Origin_Dest_brieman > 4998.5)
        Predict: 1.0
   Else  Day_Of_Month > 13.5)
    If  Dest_brieman <= 73.5)
     Predict: 0.0
    Else  Dest_brieman > 73.5)
     Predict: 1.0
Else  Origin_Dest_brieman > 5117.5)
 If  CRS_Elapsed_Time_bin <= 6.5)
  If  Day_Of_Month <= 13.5)
   If  Origin_Dest_brieman <= 5541.0)
    If  Dest_brieman <= 312.5)
     If  Dest_brieman <= 208.5)
      If  CRS_Dep_Time_bin <= 172.5)
       Predict: 1.0
      Else  CRS_Dep_Time_bin > 172.5)
       Predict: 0.0
     Else  Dest_brieman > 208.5)
      Predict: 1.0
    Else  Dest_brieman > 312.5)
     If  CRS_Arr_Time_bin <= 203.5)
      If  CRS_Arr_Time_bin <= 53.5)
       Predict: 1.0
      Else  CRS_Arr_Time_bin > 53.5)
       Predict: 0.0
     Else  CRS_Arr_Time_bin > 203.5)
      If  Origin_Activity <= 24.5)
       Predict: 1.0
      Else  Origin_Activity > 24.5)
       Predict: 0.0
   Else  Origin_Dest_brieman > 5541.0)
    If  CRS_Arr_Time_bin <= 174.5)
     Predict: 1.0
    Else  CRS_Arr_Time_bin > 174.5)
     If  Origin_brieman <= 260.5)
      Predict: 1.0
     Else  Origin_brieman > 260.5)
      If  Origin_brieman <= 268.5)
       Predict: 0.0
      Else  Origin_brieman > 268.5)
```

```
      Predict: 1.0
   Else  Day_Of_Month > 13.5)
    If  Origin_brieman <= 322.5)
     If  Year <= 2017.5)
      Predict: 1.0
     Else  Year > 2017.5)
      If  Month <= 10.5)
       If  Dest_brieman <= 337.5)
        Predict: 0.0
       Else  Dest_brieman > 337.5)
        Predict: 1.0
      Else  Month > 10.5)
       Predict: 1.0
    Else  Origin_brieman > 322.5)
     If  Origin_Activity <= 16.5)
      Predict: 1.0
     Else  Origin_Activity > 16.5)
      If  Day_Of_Month <= 14.5)
       If  Month <= 10.5)
        Predict: 0.0
       Else  Month > 10.5)
        Predict: 1.0
      Else  Day_Of_Month > 14.5)
       Predict: 1.0
  Else  CRS_Elapsed_Time_bin > 6.5)
   If  CRS_Dep_Time_bin <= 190.5)
    If  Origin_Activity <= 15.5)
     If  Distance_Group <= 10.5)
      If  Origin_Activity <= 9.5)
       If  Year <= 2017.5)
        Predict: 0.0
       Else  Year > 2017.5)
        Predict: 1.0
      Else  Origin_Activity > 9.5)
       If  Day_Of_Month <= 23.5)
        Predict: 0.0
       Else  Day_Of_Month > 23.5)
        Predict: 1.0
     Else  Distance_Group > 10.5)
      If  Month <= 9.5)
       If  Year <= 2016.5)
        Predict: 1.0
       Else  Year > 2016.5)
        Predict: 0.0
      Else  Month > 9.5)
       Predict: 0.0
    Else  Origin_Activity > 15.5)
     If  CRS_Dep_Time_bin <= 180.5)
      If  CRS_Arr_Time_bin <= 203.5)
       If  Day_Of_Month <= 20.5)
        Predict: 0.0
```

```
                Else  Day_Of_Month > 20.5)
                 Predict: 1.0
               Else  CRS_Arr_Time_bin > 203.5)
                Predict: 1.0
              Else  CRS_Dep_Time_bin > 180.5)
               Predict: 0.0
            Else  CRS_Dep_Time_bin > 190.5)
             If  Month <= 11.5)
              If  Day_Of_Month <= 29.5)
               If  Origin_Activity <= 8.5)
                Predict: 0.0
               Else  Origin_Activity > 8.5)
                Predict: 1.0
              Else  Day_Of_Month > 29.5)
               Predict: 0.0
             Else  Month > 11.5)
              If  Origin_brieman <= 300.5)
               If  Year <= 2016.5)
                Predict: 0.0
               Else  Year > 2016.5)
                Predict: 1.0
              Else  Origin_brieman > 300.5)
               Predict: 1.0
```

```
 Provided Features:  ['Year', 'Month', 'Day_Of_Month', 'Day_Of_Week', 'D
istance_Group', 'CRS_Dep_Time_bin', 'CRS_Arr_Time_bin', 'CRS_Elapsed_Tim
e_bin', 'Origin_Activity', 'Op_Unique_Carrier_brieman', 'Origin_briema
n', 'Dest_brieman', 'Day_Of_Year_brieman', 'Origin_Dest_brieman', 'Dep_T
ime_Of_Week_brieman', 'Arr_Time_Of_Week_brieman', 'Holiday_brieman']

     Used Features:  {'Year', 'CRS_Arr_Time_bin', 'CRS_Elapsed_Time_bi
n', 'Day_Of_Month', 'Origin_Activity', 'Origin_Dest_brieman', 'Distance_
Group', 'CRS_Dep_Time_bin', 'Dest_brieman', 'Origin_brieman', 'Month',
'Op_Unique_Carrier_brieman', 'Day_Of_Week'}
```

In [84]:

```
# Hyper parameter tuning to find optimal parameters for the Decision Tree
for max_depth in [5,10,15,20,30,50,100]:
  dt_model = TrainDecisionTreeModel(train_smoted, [va_base], outcomeName,
  print("\nMax Depth:", max_depth)
  PredictAndEvaluate(dt_model, val, 'val', outcomeName)
```

Training Random Forest on Smoted (Balanced) Training Dataset

Decision trees have a tendency to overfit because they memorize the training data. One way to overcome this limitation is to prune the tree to help it generalize better. Another approach is to build a Random Forest (i.e), a forest of random decision trees, where the

"randomness" comes from the random subset of features (without replacement) and observations (with replacement) given to each tree to consider. Multiple trees are generated through the random forest training, where each tree is trained on their own subset of data and features.

At inference time, the inference on a single data point involves taking the results or "votes" from each tree on the classification for that data point and aggregating them as an average of results to be the final prediction given by the random forest. Random forests tend to perform better than decision trees because they can generalize more easily. However, random forests have a bit of a loss in terms of interpretability compared to decision trees, as the decision making becomes a bit more distributed and requires aggregation. Having said that, we can still plot a ranking of feature importances for a random forest to understand which features are given most/least importance.

Below, we train and evaluate a simple random forest model, and rank the importance of each of our considered features, as shown in the barplot below. From this plot, we can see that `CRS_Dep_Time_bin` and `Origin_Activity` are given the highest importance by the full random forest model, meaning that these features gave some of the highest information gain among all the trees in the random forest model. Following this, we see `CRS_Arr_Time_bin`, `Origin_Dest_brieman`, and `Day_Of_Week` are next in importance, with features relating to the time of year, the elapsed time, and the distance traveled being less imortant. From this, we can generally infer that infromation about the origin airport along with the time of the day and the day of the week the flight is scheduled to take off and arrive can give us important information about predicting a departure delay.

With regard to performance on this single random forest model, we see a similar story to the decision tree. That is to say, the model performs quite well on the SMOTEd training data, but seems to fall short in performance on the original training and validation data, although not quite as severely as that seen with the decision tree, suggesting that the random forest algorithm may have helped reduce the amount of overfitting seen in the tree (though some overfitting is still present).

In [86]:
```
# Trains a simple Random Forest model
def TrainRandomForestModel(trainingData, stages, outcomeNames, maxDepth, m
  # Train Model
  rf = RandomForestClassifier(labelCol = outcomeName, featuresCol = "featu
  pipeline = Pipeline(stages = stages + [rf])
  rf_model = pipeline.fit(trainingData)
  return rf_model
```

In [87]:
```
rf_model = TrainRandomForestModel(train_smoted, [va_base], outcomeName, ma
PredictAndEvaluate(rf_model, train_smoted, 'train_smoted', outcomeName)
PredictAndEvaluate(rf_model, train, 'train', outcomeName)
PredictAndEvaluate(rf_model, val, 'val', outcomeName)
```

```
Model Evaluation - train_smoted
```

```
-------------------------------
Accuracy = 0.6622529, Precision = 0.6554999, Recall = 0.7060148, f-score
= 0.6798203, AreaUnderROC = 0.7217311, AreaUnderPRC = 0.7117491

Confusion Matrix:
        count
TP  13499572
FN   5621233
TN  11433848
FP   7094743

Model Evaluation - train
-------------------------------
Accuracy = 0.6550594, Precision = 0.1691789, Recall = 0.5122745, f-score
= 0.2543564, AreaUnderROC = 0.6346952, AreaUnderPRC = 0.1772946

Confusion Matrix:
        count
TP   1266654
FN   1205954
TN  12836393
FP   6220415

Model Evaluation - val
-------------------------------
Accuracy = 0.6542051, Precision = 0.1700237, Recall = 0.5157057, f-score
= 0.2557341, AreaUnderROC = 0.6354687, AreaUnderPRC = 0.1785722

Confusion Matrix:
       count
TP   182615
FN   171492
TN  1828335
FP   891441
```

In [88]:

```
# Plot feature importance of the random forest model
rf_importances =list(rf_model.stages[-1].featureImportances.toArray())
fig = go.Figure([go.Bar(x=featureNames, y=rf_importances)])
fig.update_layout(title_text='Feature Importances in Random Forest Model',
fig.update_yaxes(title_text="Feature Importance")
fig.update_xaxes(title_text="Features")
fig.show()
```

For hyper-parameter tuning our random forest model, we considered the following parameters:

- numTrees:
    - Represents the number of trees in the forest.

> - More trees reduce overfitting but takes longer to train.
> - Values used are 10, 20, 50, 100, 200.

- maxDepth:

> - Represents the maximum depth of each tree in the forest.
> - The deeper the tree, the more splits it has and it captures more information about the data but leads to overfitting, as discussed above.
> - Values used are 5, 10, 15.

The experiments we ran for our random forest model are shown below:

In [90]:
```
# Hyper parameter tuning to find optimal parameters for the random forest
import time

for maxDepth in [5, 10, 15]:
    for numTrees in [10, 20, 50, 100, 200]:
        t0 = time.time()
        # Train a RandomForest model
        rforest_model = TrainRandomForestModel(train_smoted, [va_base], ou
        print("\nmaxDepth:", maxDepth,", numTrees:", numTrees)
        PredictAndEvaluate(rforest_model, val, 'val', outcomeName)
        t1 = time.time()
        print("finish in %f seconds" % (t1-t0))
        print('********************')
```

```
maxDepth: 5 , numTrees: 10
Model Evaluation - val
-----------------------------
Accuracy = 0.5914259, Precision = 0.1661318, Recall = 0.6336136, f-score
= 0.2632422, AreaUnderROC = 0.6474931, AreaUnderPRC = 0.1810566

Confusion Matrix:
       count
TP    224367
FN    129740
TN   1593607
FP   1126169

finish in 118.956747 seconds
********************

maxDepth: 5 , numTrees: 20
Model Evaluation - val
-----------------------------
Accuracy = 0.5989522, Precision = 0.1639272, Recall = 0.6051702, f-score
= 0.2579748, AreaUnderROC = 0.6406965, AreaUnderPRC = 0.1749929

Confusion Matrix:
       count
```

```
TP   214295
FN   139812
TN  1626814
FP  1092962

finish in 291.069050 seconds
********************

maxDepth: 5 , numTrees: 50
Model Evaluation - val
-----------------------------
Accuracy = 0.6057075, Precision = 0.1659364, Recall = 0.601708, f-score
= 0.2601342, AreaUnderROC = 0.6434111, AreaUnderPRC = 0.1777422

Confusion Matrix:
       count
TP   213069
FN   141038
TN  1648805
FP  1070971

finish in 391.879560 seconds
********************

maxDepth: 5 , numTrees: 100
Model Evaluation - val
-----------------------------
Accuracy = 0.6049358, Precision = 0.1640636, Recall = 0.5932359, f-score
= 0.2570408, AreaUnderROC = 0.6400135, AreaUnderPRC = 0.1766616

Confusion Matrix:
       count
TP   210069
FN   144038
TN  1649433
FP  1070343

finish in 631.625718 seconds
********************

maxDepth: 5 , numTrees: 200
Model Evaluation - val
-----------------------------
Accuracy = 0.6050012, Precision = 0.1649875, Recall = 0.5980819, f-score
= 0.2586292, AreaUnderROC = 0.6415828, AreaUnderPRC = 0.1782341

Confusion Matrix:
       count
TP   211785
FN   142322
TN  1647918
```

```
FP  1071858

finish in 912.671784 seconds
********************

maxDepth: 10 , numTrees: 10
Model Evaluation - val
------------------------------
Accuracy = 0.6430697, Precision = 0.1726967, Recall = 0.5535926, f-score
= 0.263266, AreaUnderROC = 0.6477268, AreaUnderPRC = 0.1860668

Confusion Matrix:
       count
TP   196031
FN   158076
TN  1780690
FP   939086

finish in 158.189163 seconds
********************

maxDepth: 10 , numTrees: 20
Model Evaluation - val
------------------------------
Accuracy = 0.6467852, Precision = 0.1720742, Recall = 0.5420876, f-score
= 0.2612273, AreaUnderROC = 0.6440754, AreaUnderPRC = 0.1827624

Confusion Matrix:
       count
TP   191957
FN   162150
TN  1796185
FP   923591

finish in 405.629747 seconds
********************

maxDepth: 10 , numTrees: 50
Model Evaluation - val
------------------------------
Accuracy = 0.6498585, Precision = 0.1728339, Recall = 0.5386988, f-score
= 0.2617039, AreaUnderROC = 0.6454886, AreaUnderPRC = 0.1832027

Confusion Matrix:
       count
TP   190757
FN   163350
TN  1806832
FP   912944

finish in 990.112626 seconds
```

```
********************

maxDepth: 10 , numTrees: 100
Model Evaluation - val
-----------------------------
Accuracy = 0.6491483, Precision = 0.172296, Recall = 0.5377612, f-score
= 0.2609765, AreaUnderROC = 0.6444026, AreaUnderPRC = 0.1843993

Confusion Matrix:
       count
TP   190425
FN   163682
TN  1804981
FP   914795

finish in 2000.537420 seconds
********************

maxDepth: 10 , numTrees: 200
Model Evaluation - val
-----------------------------
Accuracy = 0.650151, Precision = 0.1726988, Recall = 0.5373856, f-score
= 0.2613939, AreaUnderROC = 0.6452626, AreaUnderPRC = 0.1850408

Confusion Matrix:
       count
TP   190292
FN   163815
TN  1808196
FP   911580

finish in 4729.728167 seconds
********************

maxDepth: 15 , numTrees: 10
Model Evaluation - val
-----------------------------
Accuracy = 0.6676015, Precision = 0.1785717, Recall = 0.5237344, f-score
= 0.2663345, AreaUnderROC = 0.6500522, AreaUnderPRC = 0.1894012

Confusion Matrix:
       count
TP   185458
FN   168649
TN  1866671
FP   853105

finish in 609.658666 seconds
********************

maxDepth: 15 , numTrees: 20
```

```
Model Evaluation - val
------------------------------
Accuracy = 0.6673338, Precision = 0.1780915, Recall = 0.5221896, f-score
= 0.2656006, AreaUnderROC = 0.6497931, AreaUnderPRC = 0.1902498

Confusion Matrix:
       count
TP   184911
FN   169196
TN  1866395
FP   853381

finish in 1203.103352 seconds
********************

maxDepth: 15 , numTrees: 50
Model Evaluation - val
------------------------------
Accuracy = 0.6676412, Precision = 0.178449, Recall = 0.5230792, f-score
= 0.2661133, AreaUnderROC = 0.6507497, AreaUnderPRC = 0.1914346

Confusion Matrix:
       count
TP   185226
FN   168881
TN  1867025
FP   852751

finish in 2748.465021 seconds
********************

maxDepth: 15 , numTrees: 100
Model Evaluation - val
------------------------------
Accuracy = 0.6705821, Precision = 0.1788263, Recall = 0.5176938, f-score
= 0.265828, AreaUnderROC = 0.6504246, AreaUnderPRC = 0.1916837

Confusion Matrix:
       count
TP   183319
FN   170788
TN  1877972
FP   841804

finish in 5600.550341 seconds
********************
```

After performing parameter optimization on the random forest, we found that the random forest classifier with 100 trees and maxDepth of 15 performed best with metrics as follows:

- Accuracy of 0.67,
- Precision of 0.17,
- Recall of 0.51,
- AUROC of 0.65.

As expected, performance of the random forest model is better than the performance we generally saw on a single decision tree. But, to improve the performance further, we will try to generalize our models even moreso and try an ensemble of random forests (i.e. a forest of forests), as our next approach.

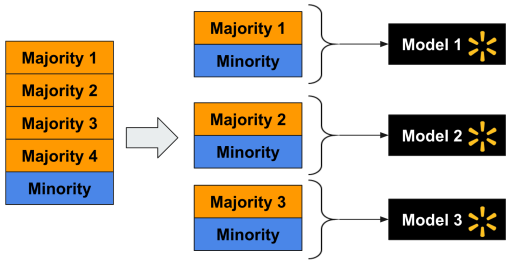
Training Ensemble with Majority Class Splitted (Balanced) Training Dataset

When we considered constructing ensembles of random forests to try to better generalize our models, we took inspiration from a model design known as stacking, which attempts to train an ensemble of models on an unbalanced dataset. The stacking approach we referred to depended on balancing the dataset via the majority class splitting approach, which we described at a high-level in section III. For this reason, we will turn to using majority class splitting to balance our dataset in order to motivate the explanation behind constructing stacked models and then return to using our SMOTEd dataset in a similar stacking approach.

As we discussed previously, the majority class splitting approach involves simply the majority class into $\boldsymbol{N}$ parts and constructing $\boldsymbol{N}$ subsets of the majority class and combining them with the entire subset of the minority class, such that each data subset is balanced, contains the full minority class, and has a $\frac{\mathbf{1}}{\boldsymbol{N}}$th random sample of the majority class. Each of these datasets will be used to train a single model in the first stage of the ensemble. By taking this approach, we can still ensure the training data used for each model is balanced and in the process, we will not lose any information that might have been lost from simply undersampling the majority class, nor will the stage 1 models overfit to the minority class (which would have happened had we oversampled the minority class).

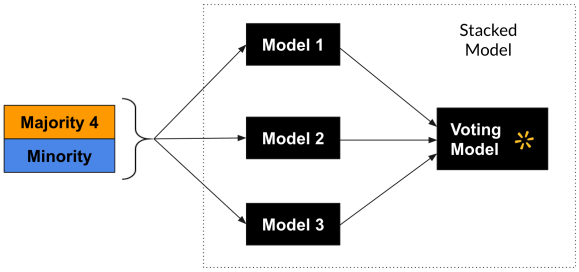
In the diagram below, we depict how the distribution of the datasets are shared amongst the individual random forest models used in the first stage of the ensemble stacking approach. With this approach, each model learns from the dataset independently of each other. The training of these models can be parallized in spark using the python thread pool utility. Note that in the example, we have three stage 1 models, yet the majority class is split into four parts. This fourth majority split will be reserved for the second stage of the ensemble.

Stacking & Majority Class Splitting - Stage 1



In stacking, the algorithm takes the outputs of the sub-models and learns how to combine the input predictions to make a better final prediction. The stacking procedure consists of two levels of models. The first level generates predictions and the second level combines these predictions to generate the final prediction. As a result, the models need to be trained in two stages. The first one is shown above, where we train each individual model in parallel. The second stage is shown below, where we train a voting model that aggregates the predictions of the individual models. The last subset of data is then used for training the Stage 2 voting model, as shown below:

Stacking & Majority Class Splitting - Stage 2



In stage 2, we take the remaining balanced dataset and run inference on stage 1 models and collect their predictions. These predictions are the features for the second level voting model and it is used to train that model. The final model is an assembled pipeline of all these models and can be used for prediction and evaluation purposes. Hence, a stacked ensemble can use first level predictions and conditionally decide to weigh the input predictions of the voting model differently - giving a better performance. This approach works far better than a simple majority classifier or a weighted model.

The steps for stacking can be outlined as below.
(a) Group the **training** data into majority and minority class.
(b) Split the majority class into $N+1$ groups, each group containing same number of data points as that in minority class.
(c) Create $N+1$ datasets for training by combining the each group from (b) with minority class from (a). Each of these groups will be balanced.
(d) Use N datasets from (c) to train the first level classifier. Once the models are generated, use the remaing one dataset from (c) to generate predictions for each of these models. These N predictions are the features for the second level classifier. The target/label value for the second level classifier is the target/label value of this remaining dataset.

(e) Train the second level classifier. A final pipeline can be created by combining the models.

In [98]:

```
# Prepare features for ensemble
def PreprocessForEnsemble(mini_train_data, train_data, val_data, test_data
  target       = ["Dep_Del30"]
  all_features = numFeatureNames + binFeatureNames + orgFeatureNames + bri

  assembler = VectorAssembler(inputCols=all_features, outputCol="features"
  ensemble_pipeline = Pipeline(stages=[assembler])

  tmp_mini_train, tmp_train, tmp_val, tmp_test, tmp_train_smoted = (ensemb
                                                    ensemble_pipeline.fit(tr
                                                    ensemble_pipeline.fit(va
                                                    ensemble_pipeline.fit(te
                                                    ensemble_pipeline.fit(tr
  featureIndexer = VectorIndexer(inputCol="features", outputCol="indexedFe
  featureIndexer_smoted = VectorIndexer(inputCol="features", outputCol="in
  return all_features, featureIndexer, featureIndexer_smoted, tmp_mini_tra


all_ensemble_features, ensemble_featureIndexer, ensemble_featureIndexer_sm

print(all_ensemble_features)
ensemble_mini_train.show(2)
```

```
['Year', 'Month', 'Day_Of_Month', 'Day_Of_Week', 'Distance_Group', 'CRS_
Dep_Time_bin', 'CRS_Arr_Time_bin', 'CRS_Elapsed_Time_bin', 'Origin_Activ
ity', 'Op_Unique_Carrier_brieman', 'Origin_brieman', 'Dest_brieman', 'Da
y_Of_Year_brieman', 'Origin_Dest_brieman', 'Dep_Time_Of_Week_brieman',
'Arr_Time_Of_Week_brieman', 'Holiday_brieman']
+--------------------+-----+
            features|label|
+--------------------+-----+
[2018.0,10.0,1.0,...|    0|
[2018.0,10.0,1.0,...|    0|
+--------------------+-----+
only showing top 2 rows
```

In [99]:

```
# Intermediate checkpoint creation
if False :
  ensemble_val.write.mode('overwrite').format("parquet").save("dbfs:/user/
  ensemble_test.write.mode('overwrite').format("parquet").save("dbfs:/user
```

Balance the dataset, partition for further training

For the purpose of training the ensemble with the majority class splitted approach, we will generate a set of 10 datasets. 9 of them will be used for training the level one random forest classifiers. The last subset will be used to train the level two classfier. This is done below:

In [101…

```
def PrepareDatasetForStacking(train, outcomeName, majClass = 0, minClass =
  # Determine distribution of dataset for each outcome value (zero & one)
  ones, zeros = train.groupBy(outcomeName).count().sort(train[outcomeName]

  # Set number of models & number of datasets (3 more than ratio majority
  # last split use to train level 2 classifier
  num_splits = int(zeros/ones) + 3
  print("Number of splits : " + str(num_splits))

  # Split dataset for training individual modesl and for training the voti
  zero_df = train.filter(outcomeName + ' == ' + str(majClass))
  one_df  = train.filter(outcomeName + ' == ' + str(minClass))

  # get number of values in minority class
  one_df_count = one_df.count()

  zeros_array = zero_df.randomSplit([1.0] * num_splits, 1)
  zeros_array_count = [s.count() for s in zeros_array]
  ones_array = [one_df.sample(False, min(0.999999999999, r/one_df_count),
  ones_array_count = [s.count() for s in ones_array]

  # Array of `num_models` datasets
  # below resampling (shuffling) may not be necessary for random forest.
  # Need to remove it in case of performance issues
  train_group = [a.union(b).sample(False, 0.999999999999, 1) for a, b in z

  # Construct dataset for voting (ensemble) model
  train_combiner = zeros_array[-1].union(ones_array[-1]).sample(False, 0.9

  return (train_combiner, train_group)

# Prepare datasets for stacking

# Input the training set prep-ed for ensemble approach.
train_combiner, train_group = PrepareDatasetForStacking(ensemble_train, 'l
```

```
/databricks/spark/python/pyspark/sql/types.py:1636: DeprecationWarning:

Using or importing the ABCs from 'collections' instead of from 'collecti
ons.abc' is deprecated, and in 3.8 it will stop working

Number of splits : 10
```

In [102…

```
# For non- smoted cases - database is partitioned.
print([[d.groupBy('label').count().toPandas()["count"].to_list()] for d in
 train_combiner.groupBy('label').count().toPandas()["count"].to_list())
```

```
[[[1905441, 1906236]], [[1903204, 1904025]], [[1904958, 1905813]], [[190
7560, 1908290]], [[1906608, 1907372]], [[1901660, 1902453]], [[1905272,
1906105]], [[1905913, 1906699]], [[1903074, 1903916]]] [1905050, 190589
9]
```

In [103…

```
# For non- smoted case
print(ensemble_train_smoted.groupBy('label').count().toPandas()["count"].t
smoted_splits = ensemble_train_smoted.randomSplit([1.0] * 10, 1) # Split t
train_combiner_smoted, train_group_smoted = smoted_splits[-1], smoted_spli
```

```
[18528591, 19120805]
```

Stage 1 : Train first-level classifiers

Each of the first level classifier can be trained parallelly. Concurrency is obtained by using Python's ThreadPool utility, which triggers training of various models over many workers.

In [106…

```
from pyspark.ml.classification import RandomForestClassifier
from multiprocessing.pool import ThreadPool

# allow up to 10 concurrent threads
pool = ThreadPool(10)

# You can increase the timeout for broadcasts. Default is 300 s
spark.conf.set('spark.sql.broadcastTimeout', '900000ms')
spark.conf.get('spark.sql.broadcastTimeout')
```

```
Out[31]: '900000ms'
```

This method does not work well when the clusters are loaded, so may need to revert back to serial training if the setup gives timeout errors.

In [108…

```
# Code for parallel training.
def TrainEnsembleModels_parallel(en_train, featureIndexer, classifier) :
  job = []
  for num, _ in enumerate(en_train):
      print("Create ensemble model : " + str(num))
      # Chain indexer and classifier in a Pipeline
      job.append(Pipeline(stages=[featureIndexer, classifier]))

  return pool.map(lambda x: x[0].fit(x[1]), zip(job, en_train))

# Below is the code for parallel training. (Commented out now)
# Parallel training is not done in databricks environment.
# ensemble_model = TrainEnsembleModels_parallel(train_group, ensemble_feat
#                     # Type of model we can use.
#                     RandomForestClassifier(featuresCol="indexedFeatures"
#                    )
# print("Training done")

# The training is still done serially now to avoid databricks error during
def TrainEnsembleModels(en_train, featureIndexer, classifier) :
  model = []
  for num, train in enumerate(en_train):
      print("Create ensemble model : " + str(num))
      model.append(Pipeline(stages=[featureIndexer, classifier]).fit(train
  return model

ensemble_model = TrainEnsembleModels(train_group, ensemble_featureIndexer,
                    #RandomForestClassifier(featuresCol="indexedFeatures",
                    #RandomForestClassifier(featuresCol="indexedFeatures",
                    # Works best
                    RandomForestClassifier(featuresCol="indexedFeatures",
                   )
```

In [109…

```
# Create check points
if False :
  for i, model in enumerate(ensemble_model):
    model.save("dbfs:/user/team20/finalnotebook/ensemble_model" + str(i) +
```

Visualize feature importance for individual ensembles

Each feature's importance is the average of its importance across all trees in the ensemble. The importance vector is normalized to sum to 1, thus the bars represent the weight of each feature in the individual random forests. From the plots below, we see very similar distributions across all nine random forest models, which means that each of the models learned similar things from the data subsets they were provided. More specifically, we see that `Dep_Time_Of_Week`, `Day_Of_Year`, `Arr_Time_Of_Week`, and `Origin_Dest` were the most important features, which indicates that attributes relating to the departure/arrival time, time of week, time of year, and the origin airport are likely more important for prediction departure delays, especially compared to features like `CRS_Elapsed_Time`, distance, and holidays.

In [111…

```
from collections import defaultdict

def makedict(em, columns, features):
    plot = defaultdict(dict)
    rows = int(len(em)/columns)
    for num, m in enumerate(em):
        plot[num]['importance'] = list(m.stages[-1].featureImportances.toA
        plot[num]['features']   = features
        plot[num]['x_pos']      = int(num/columns)+1
        plot[num]['y_pos']      = num%columns+1
        plot[num]['title']      = "ensemble model {}".format(num)
    return plot, rows, columns

plt, rows, columns = makedict(ensemble_model, columns=3, features=all_ense


fig = make_subplots(rows=rows, cols=columns, subplot_titles=tuple([plt[key

for key, value in plt.items() :
    fig.add_trace(go.Bar(
      x=plt[key]['features'],
      y=plt[key]['importance'],
      #marker_color=list(map(lambda x: px.colors.sequential.thermal[x%12],
      marker_color=list(map(lambda x: px.colors.sequential.thermal[x] if x
                        range(0,len(plt[key]['features'])))),
      name = '',
      showlegend = False,
    ), row=plt[key]['x_pos'], col=plt[key]['y_pos'])

    fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
    fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
    if plt[key]['y_pos'] == 1: fig.update_yaxes(title_text="Feature import
    fig.update_xaxes(tickangle=-45)

fig.update_layout(height=1200, width=1200, title_text="Feature importance
```

```
fig.update_layout(xaxis_tickangle=-45)
fig.show()
```

Construct a new dataset based on the output of base classifiers

Using the random forest classifiers training in the first stage, we'll use the remaining balanced dataset subset to collection predictions from these random forest classifiers and use these predictions to train the voting models. The code below outlines how we'd aggregate these predictions for ingestion to the voting model for training the voting model.

In [113…

```
def do_ensemble_prediction(em, train_en) :
    prediction_array = []
    for num, m in enumerate(em) :
        predictions = em[num].transform(train_en)
        if num == 0 : prediction_array.append(
            predictions.select("label").withColumn('ROW_ID', F.monotonical
        )
        prediction_array.append(predictions
                                .select(F.col("prediction").alias("predict
                                # Create a monotonically increasing row id
                                # so that we can do recursive join based o
                                .withColumn('ROW_ID', F.monotonically_incr
        )
    return prediction_array

ensemble_prediction = do_ensemble_prediction(ensemble_model, train_combine
```

Assemble and transform data for second level training

Join together the individual dataframes to create the final training dataset for level two model.

In [116…

```
from functools import reduce

def assemble_dataframe(prediction_array) :
    # Do a reduction operation using functional programming concepts, iter
    # a dataframe towrds end.
    def do_reduce(df1, df2): return df1.join(df2, "ROW_ID")
    return reduce(do_reduce, prediction_array).drop("ROW_ID")
```

```
def do_transform_final(df) :
    ensemble_columns = df.schema.names
    en_target, en_features = ensemble_columns[0], ensemble_columns[1:]

    assembler = VectorAssembler(inputCols=en_features, outputCol="features
    pipeline = Pipeline(stages=[assembler])
    return (pipeline
            .fit(df)
            .transform(df)
            .select(["features"] + [en_target])
    )

reduced_df = assemble_dataframe(ensemble_prediction)
#reduced_df.show(2)
ensemble_transformed = do_transform_final(reduced_df)
#ensemble_transformed.show(2)
```

In [117…

```
reduced_df.show(2)
```

```
+-----+-----------+-----------+-----------+-----------+-----------+-----------+
-----------+-----------+-----------+-----------+
label|prediction_0|prediction_1|prediction_2|prediction_3|prediction_4|p
rediction_5|prediction_6|prediction_7|prediction_8|
+-----+-----------+-----------+-----------+-----------+-----------+-----------+
-----------+-----------+-----------+-----------+
    0|         0.0|         0.0|         0.0|         0.0|         0.0|         0.0|
0.0|         0.0|         0.0|         0.0|
    0|         0.0|         0.0|         0.0|         0.0|         0.0|         0.0|
0.0|         0.0|         0.0|         0.0|
+-----+-----------+-----------+-----------+-----------+-----------+-----------+
-----------+-----------+-----------+-----------+
only showing top 2 rows
```

Stage 2 : Learn a second-level classifier based on training set from first-level.

For the second stage, we'll train the second level classifier, which will be our voting model. Among our candidate voting models, we will consider Logistic Regression, Support Vector Machines, and Random Forests, to have a diverse set of candidates for our voting models. This is done below:

In [119…

```
def TrainCombiner(data, featureIndexer, classifier):
  # Chain indexer and forest in a Pipeline
  pipeline_ensemble = Pipeline(stages=[featureIndexer, classifier])

  # Train model.  This also runs the indexer.
  return pipeline_ensemble.fit(data)

# Set up VectorIndexer for second level training
ensemble_featureIndexer = VectorIndexer(inputCol="features", outputCol="in
```

In [120…

```
# Logistic Regression
model_trained_ensemble_lr = TrainCombiner(ensemble_transformed, ensemble_f
              LogisticRegression(featuresCol="indexedFeatures", maxIter=10
```

In [121…

```
# Linear SVM
model_trained_ensemble_svm = TrainCombiner(ensemble_transformed, ensemble_
              LinearSVC(featuresCol="indexedFeatures", maxIter=10, regPara
```

In [122…

```
# Random forest
model_trained_ensemble_rf = TrainCombiner(ensemble_transformed, ensemble_f
              RandomForestClassifier(featuresCol="indexedFeatures", maxBin
```

In [123…

```
# Create checkpoint
if False :
  model_trained_ensemble_lr.save("dbfs:/user/team20/finalnotebook/model_tr
  model_trained_ensemble_svm.save("dbfs:/user/team20/finalnotebook/model_t
  model_trained_ensemble_rf.save("dbfs:/user/team20/finalnotebook/model_tr
```

Create final ensemble pipeline

For the final pipeline, we'll stitch together the following pieces:

- The level one models, which are combined into a single parameter
- Level two voting model
- Data for running inference (e.g. validation set)

In [125…

```
# recursively call each of the functions described above to transform the
# This is the final model pipeline.
def FinalEnsmblePipeline(model_comb, model_group, data) :
  return model_comb.transform(
    do_transform_final(
      assemble_dataframe(
        do_ensemble_prediction(model_group, data)
      )
    )
  )
```

Plot Model weights of ensembles

Similar to plotting the feature importances, we can also attempt to plot "model importances" as considered by each of the three voting models we've explored.

In the case of Logistic Regression, these metrics are the coefficients estimated for each of the individual random forest predictions. We ideally don't want to see any of these

coefficients set to 0, as this indicates that the model is ignored and thus the subset of data is ignored. In this case, we see that all 9 of the models are pretty evenly considered. Note that for our Logistic Regression voting model, we chose to use L2 regularization, which allows the weights to get close to zero but not be set to zero (which would be a consequence of using L1 regularization so in this case, we prefer L2 regularization). In this case, we thankfully see that the models are fairly evenly considered by the voting model.

For SVM, the weights show correspond to the coefficients for the separating boundary in "model space" for our delay and no-delay cases. Like for Logistic Regression, we don't want to see any of these weights zeroed out, as that would indicate that the model and the data subset is ignored for training. In this case, we see that the SVM gives a similar distribution for the model importance, as was seen with Logistic Regression.

For Random Forests, the weights we see correspond directly to the feature importances we've shown previously for the individual random forests. In this case though, our "features" are our individual models. Like in the previous plots, these model importances are based on how much the models contributed to the information gain in the voting model, so the higher the importance, the more information gain provided by the individual tree. Once again, we see that all models are being considered, though some more than others (in this case, model 2 is the least).

In [127…

```
from collections import defaultdict

def makedict(em, columns, features, info):
    plot = defaultdict(dict)
    rows = int(len(em)/columns)
    for num, m in enumerate(em):
        plot[num]['importance'] = em[num]
        plot[num]['features']   = features
        plot[num]['x_pos']      = int(num/columns)+1
        plot[num]['y_pos']      = num%columns+1
        plot[num]['title']      = info[num]
    return plot, rows, columns

plt, rows, columns = makedict([
  list(model_trained_ensemble_lr.stages[-1].coefficients.toArray()),
  list(model_trained_ensemble_svm.stages[-1].coefficients.toArray()),
  list(model_trained_ensemble_rf.stages[-1].featureImportances.toArray())]
  columns=3,
  features=[ "model {}".format(s) for s in range(0, 9)],
  info=[
    "Logistic regression - weights of ensembles",
    "SVM - weights of ensembles",
    "Random forest - weights of ensembles",
  ]
)

fig = make_subplots(rows=rows, cols=columns, subplot_titles=tuple([plt[key

for key, value in plt.items() :
    fig.add_trace(go.Bar(
      x=plt[key]['features'],
```

```
        y=plt[key]['importance'],
        marker_color=list(map(lambda x: px.colors.sequential.thermal[x], ran
        name = '',
        showlegend = False,
      ), row=plt[key]['x_pos'], col=plt[key]['y_pos'])

      fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
      fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
      if plt[key]['y_pos'] == 1: fig.update_yaxes(title_text="Feature import
      fig.update_xaxes(tickangle=-45)

fig.update_layout(height=400, width=1200, title_text="Feature importance f
fig.update_layout(xaxis_tickangle=-45)
fig.show()
```

Training Ensemble with SMOTEd (Balanced) Training Dataset

Now that we've developed the stacking approach using the majority class splitting, we can easily extend this to our original SMOTEd dataset. This is showcased in the code below:

In [129…

```
# Train first-level classifiers using random forest, store the resulting m
ensemble_model_smoted = TrainEnsembleModels(train_group_smoted, ensemble_f
                    RandomForestClassifier(featuresCol="indexedFeatures",
                   )
```

```
Create ensemble model : 0
Create ensemble model : 1
Create ensemble model : 2
Create ensemble model : 3
Create ensemble model : 4
```

```
Create ensemble model : 5
Create ensemble model : 6
Create ensemble model : 7
Create ensemble model : 8
```

Similar to how we did previously, we can visualize the feature importance provided by each individual random forest model, as shown in the following plots. From the plots below, we again see very similar feature importance distributions for all nine random forest models, which means that each of the individual models are once again learning similar things from their subsets of data. Specifically, we see that `Origin_Activity`, `CRS_Dep_Time`, `CRS_Arr_Time`, `Day_Of_Week`, and `Origin_Dest` are some of the most important features which indicates that using the SMOTEd dataset, the model appears to learn that departure/arrival time, day of week and the origin airports are likely the most indicative features for predicting departure delays.

In [131…

```
from collections import defaultdict

def makedict(em, columns, features):
    plot = defaultdict(dict)
    rows = int(len(em)/columns)
    for num, m in enumerate(em):
        plot[num]['importance'] = list(m.stages[-1].featureImportances.toA
        plot[num]['features']   = features
        plot[num]['x_pos']      = int(num/columns)+1
        plot[num]['y_pos']      = num%columns+1
        plot[num]['title']      = "ensemble model {}".format(num)
    return plot, rows, columns

plt, rows, columns = makedict(ensemble_model_smoted, columns=3, features=a


fig = make_subplots(rows=rows, cols=columns, subplot_titles=tuple([plt[key

for key, value in plt.items() :
    fig.add_trace(go.Bar(
      x=plt[key]['features'],
      y=plt[key]['importance'],
      #marker_color=list(map(lambda x: px.colors.sequential.thermal[x%12],
      marker_color=list(map(lambda x: px.colors.sequential.thermal[x] if x
                        range(0,len(plt[key]['features'])))),
      name = '',
      showlegend = False,
    ), row=plt[key]['x_pos'], col=plt[key]['y_pos'])

    fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
    fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
    if plt[key]['y_pos'] == 1: fig.update_yaxes(title_text="Feature import
    fig.update_xaxes(tickangle=-45)

fig.update_layout(height=1200, width=1200, title_text="Feature importance
fig.update_layout(xaxis_tickangle=-45)
fig.show()
```

Next, we will proceed to train three voting models for the second stage of stacking, which again includes Logistic Regression, SVM, and Random Forests. We will use the 9 models trained previously as our stage one models for the stacked ensemble.

In [133…

```
# Checkpoint the smoted ensemble model
if False :
  for i, model in enumerate(ensemble_model_smoted):
    model.save("dbfs:/user/team20/finalnotebook/ensemble_model_smoted" + s
```

In [134…

```
# Construct a new data set based on the output of base classifiers
ensemble_prediction_smoted = do_ensemble_prediction(ensemble_model_smoted,
```

In [135…

```
# Assemble and transform data for second level training
reduced_df_smoted = assemble_dataframe(ensemble_prediction_smoted)
ensemble_transformed_smoted = do_transform_final(reduced_df_smoted)
```

In [136…

```
ensemble_featureIndexer_smoted = VectorIndexer(inputCol="features", output
```

In [137…

```
# Learn a second-level classifier based on training set from first-level.

# Logistic Regression
model_trained_ensemble_lr_smoted = TrainCombiner(ensemble_transformed_smot
              LogisticRegression(featuresCol="indexedFeatures", maxIter=10
```

In [138…

```
# Linear SVM
model_trained_ensemble_svm_smoted = TrainCombiner(ensemble_transformed_smo
              LinearSVC(featuresCol="indexedFeatures", maxIter=10, regPara
```

In [139…

```
# Random forest
model_trained_ensemble_rf_smoted = TrainCombiner(ensemble_transformed_smot
              RandomForestClassifier(featuresCol="indexedFeatures", maxBin
```

In [140…

```
# Save and checkpoint the models
if False :
  model_trained_ensemble_lr_smoted.save("dbfs:/user/team20/finalnotebook/m
  model_trained_ensemble_svm_smoted.save("dbfs:/user/team20/finalnotebook/
  model_trained_ensemble_rf_smoted.save("dbfs:/user/team20/finalnotebook/m
```

With the voting models trained, we can again evaluate how these voting models consider each of the individual random forest models. In the case of Logistic Regression and SVM, we see a fairly uniform distribution across all nine individual models, suggesting an event consideration of the entire smoted dataset, similar to what we saw before with majority class splitting. However, there's a stark difference when looking at the model importances for the Random Forest voting model. In this case, model 6 is highly prioritized over the other 8 models, leading to less of a priority given to models 7, 4, 3, 2, 5, and 8, which is slight concerning given that the data learned by these models may be ignored and may lead to a bias towards the data that belongs to model 6.

In [142…

```
from collections import defaultdict

def makedict(em, columns, features, info):
    plot = defaultdict(dict)
    rows = int(len(em)/columns)
    for num, m in enumerate(em):
        plot[num]['importance'] = em[num]
        plot[num]['features']   = features
        plot[num]['x_pos']      = int(num/columns)+1
        plot[num]['y_pos']      = num%columns+1
        plot[num]['title']      = info[num]
    return plot, rows, columns

plt, rows, columns = makedict([
  list(model_trained_ensemble_lr_smoted.stages[-1].coefficients.toArray())
  list(model_trained_ensemble_svm_smoted.stages[-1].coefficients.toArray()
  list(model_trained_ensemble_rf_smoted.stages[-1].featureImportances.toAr
  columns=3,
  features=[ "model {}".format(s) for s in range(0, 9)],
  info=[
    "Logistic regression - weights of ensembles",
    "SVM - weights of ensembles",
    "Random forest - weights of ensembles",
  ]
)

fig = make_subplots(rows=rows, cols=columns, subplot_titles=tuple([plt[key

for key, value in plt.items() :
    fig.add_trace(go.Bar(
      x=plt[key]['features'],
      y=plt[key]['importance'],
      marker_color=list(map(lambda x: px.colors.sequential.thermal[x], ran
      name = '',
      showlegend = False,
    ), row=plt[key]['x_pos'], col=plt[key]['y_pos'])

    fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
```

```
    fig.update_xaxes(categoryorder='total descending', row=plt[key]['x_pos
    if plt[key]['y_pos'] == 1: fig.update_yaxes(title_text="Feature import
    fig.update_xaxes(tickangle=-45)

fig.update_layout(height=400, width=1200, title_text="Feature importance f
fig.update_layout(xaxis_tickangle=-45)
fig.show()
```

Model Evaluation

With the ensemble models for both dataset balancing techniques trained and evaluated at a high-level, we'll now examine the actual performance results of the models against both the validation and the held out test sets. We will consider all of the six performance metrics discussed previously, as well as the full confusion matrix to again understand the nuances of model performance. We'll look at the performance metrics for each of our three voting models and compare them accordingly.

In [144…

```
# Reload voting model from checkpoints
print("Loading model_trained_ensemble_lr_smoted.v2.model")
model_trained_ensemble_lr_smoted_load = pl.PipelineModel.load("dbfs:/user/
print("Loading model_trained_ensemble_svm_smoted.v2.model")
model_trained_ensemble_svm_smoted_load = pl.PipelineModel.load("dbfs:/user
print("Loading model_trained_ensemble_rf_smoted.v2.model")
model_trained_ensemble_rf_smoted_load = pl.PipelineModel.load("dbfs:/user/
print("Loading model_trained_ensemble_lr.v2.model")
model_trained_ensemble_lr_load = pl.PipelineModel.load("dbfs:/user/team20/
print("Loading model_trained_ensemble_svm.v2.model")
model_trained_ensemble_svm_load = pl.PipelineModel.load("dbfs:/user/team20
print("Loading model_trained_ensemble_rf.v2.model")
model_trained_ensemble_rf_load = pl.PipelineModel.load("dbfs:/user/team20/
```

```
Loading model_trained_ensemble_lr_smoted.v2.model
Loading model_trained_ensemble_svm_smoted.v2.model
Loading model_trained_ensemble_rf_smoted.v2.model
Loading model_trained_ensemble_lr.v2.model
Loading model_trained_ensemble_svm.v2.model
Loading model_trained_ensemble_rf.v2.model
```

In [145…

```
# Reload ensemble model from checkpoints
ensemble_model_load = []
for i in range(0,9) :
  print("Loading ensemble_model " + str(i))
  ensemble_model_load.append(pl.PipelineModel.load("dbfs:/user/team20/fina
```

```
Loading ensemble_model 0
Loading ensemble_model 1
Loading ensemble_model 2
Loading ensemble_model 3
Loading ensemble_model 4
Loading ensemble_model 5
Loading ensemble_model 6
Loading ensemble_model 7
Loading ensemble_model 8
```

In [146…

```
# Reload smoted ensemble model from checkpoints
ensemble_model_smoted_load = []
for i in range(0,9) :
  print("Loading ensemble_model " + str(i))
  ensemble_model_smoted_load.append(pl.PipelineModel.load("dbfs:/user/team
```

```
Loading ensemble_model 0
Loading ensemble_model 1
Loading ensemble_model 2
Loading ensemble_model 3
Loading ensemble_model 4
Loading ensemble_model 5
Loading ensemble_model 6
Loading ensemble_model 7
Loading ensemble_model 8
```

In [147…

```
# Reload test and validation data from checkpoints
print("Loading ensemble_test.v1.parquet")
ensemble_test_load = spark.read.option("header", "true").parquet("dbfs:/us
print("Loading ensemble_val.v1.parquet")
ensemble_val_load = spark.read.option("header", "true").parquet("dbfs:/use
```

```
Loading ensemble_test.v1.parquet
Loading ensemble_val.v1.parquet
```

In [148…

```
# Run the evaluation metrics after prediction. Use the model trained with

model_eval_regular = []
```

```
for (l2_name, l2_model, l1_model)  in [
  ("LR", model_trained_ensemble_lr_load, ensemble_model_load),
  ("SVM", model_trained_ensemble_svm_load, ensemble_model_load),
  ("RF", model_trained_ensemble_rf_load, ensemble_model_load),
] :
  for data_name, data in [("test set", ensemble_test_load), ("validation",
      print("Level 2 model type = {}, running on {}".format(l2_name,data_n
      ensemble_test_prediction = FinalEnsmblePipeline(l2_model, l1_model,
      eval = EvaluateModelPredictions(ensemble_test_prediction, dataName=d

      # Collect the evaluation metrics.
      model_eval_regular.append({ 'l2_name' : l2_name, 'data_name' : data_
```

```
Level 2 model type = LR, running on test set
Level 2 model type = LR, running on validation
Level 2 model type = SVM, running on test set
Level 2 model type = SVM, running on validation
Level 2 model type = RF, running on test set
Level 2 model type = RF, running on validation
```

In [149…

```
model_eval = model_eval_regular
headerColor  = 'lightgrey'
rowEvenColor = 'lightgrey'
rowOddColor  = 'white'

fig = go.Figure(data=[go.Table(
  header=dict(
    values=['<b>Run type</b>','<b>Accuracy</b>','<b>Precision</b>','<b>Rec
            '<b>AUROC</b>', '<b>AUPRC</b>', '<b>TP</b>', '<b>TN</b>', '<b>
    line_color='darkslategray',
    fill_color=headerColor,
    align=['left','center'],
    font=dict(color='black', size=13)
  ),
  cells=dict(
    values=[
      [ev['l2_name'] + '<br>(' + ev['data_name'] + ')' for ev in model_eva
      [ev['result']['Accuracy'] for ev in model_eval],
      [ev['result']['Precision'] for ev in model_eval],
      [ev['result']['Recall'] for ev in model_eval],
      [ev['result']['f-score'] for ev in model_eval],
      [ev['result']['areaUnderROC'] for ev in model_eval],
      [ev['result']['AreaUnderPRC'] for ev in model_eval],
      [ev['result']['metric']['TP'] for ev in model_eval],
      [ev['result']['metric']['TN'] for ev in model_eval],
      [ev['result']['metric']['FP'] for ev in model_eval],
      [ev['result']['metric']['FN'] for ev in model_eval],
    ],
    line_color='darkslategray',
    align = ['left'],
    font = dict(color = 'darkslategray', size = 13)
    ))
])
fig.update_layout(width=1400, height=600, title="Model evaluation results
fig.show()
```

In [150…

```
# Run the evaluation metrics after prediction. Use the model trained with
model_eval_smoted = []
for (l2_name, l2_model, l1_model)  in [
  ("LR-smoted", model_trained_ensemble_lr_smoted_load, ensemble_model_smot
  ("SVM-smoted", model_trained_ensemble_svm_smoted_load, ensemble_model_sm
  ("RF-smoted", model_trained_ensemble_rf_smoted_load, ensemble_model_smot
] :
  for data_name, data in [("test set", ensemble_test_load), ("validation",
      print("Level 2 model type = {}, running on {}".format(l2_name,data_n
      ensemble_test_prediction = FinalEnsmblePipeline(l2_model, l1_model,
      eval = EvaluateModelPredictions(ensemble_test_prediction, dataName=d

      # Collect the evaluation metrics.
      model_eval_smoted.append({ 'l2_name' : l2_name, 'data_name' : data_n
```

```
Level 2 model type = LR-smoted, running on test set
Level 2 model type = LR-smoted, running on validation
Level 2 model type = SVM-smoted, running on test set
Level 2 model type = SVM-smoted, running on validation
Level 2 model type = RF-smoted, running on test set
Level 2 model type = RF-smoted, running on validation
```

In [151…

```
model_eval = model_eval_smoted
headerColor  = 'lightgrey'
rowEvenColor = 'lightgrey'
rowOddColor  = 'white'

fig = go.Figure(data=[go.Table(
  header=dict(
    values=['<b>Run type</b>','<b>Accuracy</b>','<b>Precision</b>','<b>Rec
            '<b>AUROC</b>', '<b>AUPRC</b>', '<b>TP</b>', '<b>TN</b>', '<b>
    line_color='darkslategray',
    fill_color=headerColor,
    align=['left','center'],
    font=dict(color='black', size=13)
  ),
  cells=dict(
    values=[
      [ev['l2_name'] + '<br>(' + ev['data_name'] + ')' for ev in model_eva
      [ev['result']['Accuracy'] for ev in model_eval],
      [ev['result']['Precision'] for ev in model_eval],
      [ev['result']['Recall'] for ev in model_eval],
      [ev['result']['f-score'] for ev in model_eval],
      [ev['result']['areaUnderROC'] for ev in model_eval],
      [ev['result']['AreaUnderPRC'] for ev in model_eval],
      [ev['result']['metric']['TP'] for ev in model_eval],
      [ev['result']['metric']['TN'] for ev in model_eval],
      [ev['result']['metric']['FP'] for ev in model_eval],
      [ev['result']['metric']['FN'] for ev in model_eval],
    ],
    line_color='darkslategray',
    align = ['left'],
    font = dict(color = 'darkslategray', size = 13)
    ))
])
fig.update_layout(width=1400, height=600, title="Model evaluation results
fig.show()
```

In the tables shown above, we see the performance results for both dataset balancing approaches across the different voting models. One thing that we want to look for is consistency across both the validation and test sets, which is something that we see for most metrics, but there does appear to be a stark difference in most of the runs between the two data balancing methods. Accuracy seems to be consistently higher for majority class splitting and greater than 0.5 but both approaches have fairly decent AUROC. In some cases, recall seems to fair better for sMOTE, which is great if the airlines and airports are using this model to prepare for delays from a resource perspective, but the lower precision means people might be more likely to miss their flights if they rely on this model to predict departure delays. The f-score, which attempts to balance both precision and recall into one metric, seems to hover around the same 0.22 regardless of data balancing approaches or voting models, as does the AUPRC at around 0.12.

One thing that stands out is the low accuracy on SVM and Random Forest voting models with SMOTEing, which appears to be due to the very high number of false positives, which indicates that the models are predicting a lot of the flights as delayed when they are not--this could be because we generated a lot of synthetic data for the minority class, some of which might have actually been similar to non-delayed flights. This is especially exemplified by the high recall of 1 and 0.92/0.98 for SVM and Random Forests respectively, which also have fairly small numbers of true negatives and false negatives. In fact, in the case of the SVM voting model, the model always predicted delay, which is the inverse of the problem we were worried about when it came to data balancing--if anything, SVM appears to have gotten fairly confused between delay and no delay flights. But it's important to note that the Logistic Regression voting model actually does fairly well and doesn't seem to fall into the trap of always predicting delays. It does seem to outperform the other voting models on many of the statistics we've considered, likey due to the fact that in the model's simplest form, it will just predict the average, which is essentially a majority vote among all the individual random forests.

By comparison, the voting models that leveraged majority class splitted trees seem to more consistently perform well across all of the voting models considered. We do see that all three models generally have higher performance compared to the SMOTEd voting models, but Logistic Regression and SVM seems to be approximately tied across all metrics for the majority class splitted models in terms of performance on both

validation and test sets. In general though, across both data balancing techniques, Logistic Regression seems to fare as better voting model; although, to know for sure, we would need to do proper cross validation and hyperparameter experimentation.

At the end of the day, it depends on what your priorities are and who is going to use this model. But, while the metrics might not be perfect, our core question really had two sides to it--we want to have decent performance but also explain what is going on to help us try to find a solution to the problem. With that, we will once again explore the feature importances in a side by side comparison for the two data balancing approaches, which are shown below:

In [153…

```
import numpy as np

fig = make_subplots(rows=rows, cols=columns, subplot_titles=("Majority cla

def normalize_vec(x) :
    vec = np.stack(x).sum(axis=0)  # Normalization step.
    return(vec/np.linalg.norm(vec))

for (row, col), value in [((1, 1), [m.stages[-1].featureImportances.toArra
                          ((1, 2), [m.stages[-1].featureImportances.toArra
    fig.add_trace(go.Bar(
      x = all_ensemble_features,
      y = normalize_vec(value),
      marker_color=list(map(lambda x: px.colors.sequential.thermal[x] if x
                            range(0,len(all_ensemble_features)))),
      name = '',
      showlegend = False,
    ), row=row, col=col)

    fig.update_xaxes(categoryorder='total descending', row=row, col=col)
    fig.update_xaxes(categoryorder='total descending', row=row, col=col)
    fig.update_xaxes(tickangle=-60)

fig.update_layout(height=600, width=1500, title_text="<b>Feature importanc
fig.show()
```

At first glance, these plots appear to tell us that the random forests trained on SMOTEd data and Majority Class Splitted data seemed to have learned something different. In the Majority Class Splitted case, we see that `Dep_Time_Of_Week` has the highest importance and for SMOTEd data, `Origin_Activity` ranks the highest, and the features that follow for both are different. But we need to remember something about decision trees in general--they have the ability to build their own interaction terms. In this case, `Dep_Time_Of_Week` in the majority class splitted models is the cross of `Dep_Time & Day_Of_Week`, which are highly ranked in the SMOTEd Models; same goes for `Day_Of_Year` as the interaction of `Month` and `Day_Of_Month`. Though the plots may look different, this is an artifact of the feature engineering we did. In reality, both kinds of ensembles tells us a similar story. Features relating to the origin airport and the traffic at it, the scheduled departure time (both time and day of week), as well as the time of year seem to be important indicators, whereas the length of the flight (both in terms of elapsed time and distance) and the airline carriers are less so. For airlines and airports, this can give useful information about where they can fundamentally make changes to the infrastructure to try to reduce delays and help address the underlying problems that lead to departure delays.

VI. Conclusions

As we draw to a close in this investigation, we want to revisit the question we posed at the start of this analysis: **Given known information prior to a flight's departure, can we predict departure delays and identify the likely causes of such delays?** Throughout this analysis, we've explored a variety of models and settled on decision trees as the fundamental algorithm to help answer this question. After experiment with decision trees and extending to random forests and stacked ensembles of random forests, we came to develop models that could predict departure delays, given information known 6 hours prior to the scheduled departure time. Depending on the data balancing techniques we used, as well as the voting models we chose, the models were able to perform well across metrics such as accuracy, recall, area under ROC, especially considering the high-degree of imbalance present in the original dataset. But we were

also able to identify the likely causes of delays, which gave us consistent information regardless of the data balancing techniques used. Namely, features relating to the departure time, day of the week, time of the year, as well as the origin airport and traffic at it were very good indicators of departure delays, suggesting that these features may be able to explain the causes of such delays. This gives airlines and airports the ability to act on this information and potentially make fundamental changes to infrastructure to reduce the departure delays.

From an exploration perspective, scalability challenges were present every step of the way, whether it was with SMOTEing our dataset, training our ensembles, or anticipating challenges with our feature engineering approaches. Due to limitations on our computational capacity, we did have to approximate SMOTE with K-means and we were limited in the amount of experimentation we could do with our ensembles. For the future, we'd look to trying out more data balancing approaches and comparing them to what we've already tried, as well as bringing in some more datasets to help introduce more features, such as weather data.

VII. Applications of Course Concepts

Bias-variance tradeoff

Bias variance tradeoff came up throughout the project at different places. During Algorithm Exploration we broadly classified algorithms as those that underfit with high bias and low variance and those that tend to over-fit with low bias and high variance. The Logistic Regression and Naive Bayes belonged to the former category while Decision Tree and Support Vector Machines belonged to the latter. Lastly, during algorithm performance evaluation of decision trees it became clear that this algorithm due to the higher complexity and low bias tended to overfit to the given training set. Because of that there was high variance between training and validation sets. To reduce the over-fitting and high variance, we used random forests and ensembles of random forests to help generalize the models to the scenario. Some limited hyperparameter tuning using random forests helped us to get closer to a solution that balanced both bias and variance.

Breiman's Theorem

We applied Breiman's theorem to all of the unordered categorical features to generate a ranking within each categorical feature. We accomplished this by ordering each category based on the ranking obtained from the calculation of the average outcome. This method helped us convert categorical features to ranked numerical features. In our dataset, we applied Breiman's Theorem to the following features.
`Op_Unique_Carrier` , `Origin` , `Dest` and for the following interacted features `Day_Of_Year` , `Origin_Dest` , `Dep_Time_Of_Week` , `Arr_Time_Of_Week` , `Holiday` . For example, if you consider the feature `Op_Unique_Carrier` it had 19 unique categories. Using Breiman's method the potential 262,143 splits were reduced to

18 splits by ranking them based on the average outcome value. The scalability benefits were even more pronounced for features like `Origin_Dest` and `Day_Of_Year` where the number of categories were much larger.

Data storage on cluster - parquet

The original airlines dataset has roughly 31 million records and 54 features. While analyzing this data, it is crucial to be efficient with use of disk and I/O memory. Parquet files is a column oriented efficient way of storing this data and is very helpful in transporting the data before unpacking it. In our project we used this format when originally ingesting the data. In addition we made use of the convenience of parquet format, in storing the mini_train, train, validation, test data. We also benefitted from parquet-formatted storage when running EDA, where many of the EDA tasks were isolated to just a few columns at a time (thus benefit from the column-wise storage of data). We also used this format extensively during the feature engineering phase where we augmented the dataset by adding new features/columns through interactions, binning, applying Breiman, etc. Another place this format came in handy was while oversampling the imbalanced data using SMOTE. The transformed dataset was then saved in parquet to be accessed during algorithm evaluation by decision tree, random forests and ensembles.

Scalability for Data Sampling & Ensemble Training

Given the high-degree of imbalance in the dataset, we decided to use SMOTE to create a more balanced set. We had scalability challenges in implementing the KNN algorithm required of the original SMOTE algorithm. Namely, trying to create K nearest neighbors for approximately 2 million samples from the minority class didn't scale well. This was due to the fact that we would need to store all of these samples in memory in order to find the K nearest neighbors for each sample. To address this challenge we used only a small random sample of the minority class which fit in memory well. The second approach was to create 1000 clusters of minority samples using the K-Means algorithm and run the KNN algorithm in parallel on these smaller clusters to generate synthetic data. The second approach was much more scalable compared to the first (2.5 hrs Vs 24+hrs) and took much less time. It also yielded a set of synthetic samples closer to the distribution of the original minority dataset.

We also saw scalability concerns with training our ensembles, given that each of the random forests could be trained in parallel. In order to solve this "embarrassingly parallel" problem, we looked to using threadpools to run the training in parallel, which worked at first, but started to give broadcast timeout errors as the clusters became busier.

Broadcasting (for SMOTE, Breiman's Theorem, Holiday feature)

Broadcasted variables allow the programmer to specify to the Spark execution engine that the data stored in a given broadcasted variable is small enough for a pure copy to be shipped to each worker that needs to reference it. This is especially useful in situations where we need to join a larger dataset (like the *Airline Delays* dataset) with a small one (like the Holidays dataset). In this case, if we specify the smaller dataset in a broadcast variable prior to joining, we can trigger a broadcast join, which will allow copies of the dataset to be shipped to each worker that proccesses a subset of the larger dataset and do the join on the workers, rather than having to shuffle partitions of the large and small datasets to be joined down-stream. This was particularly useful when generating the `Holiday` feature, `Origin_Activity` feature, and when joining our Breiman ranks back to the original dataset when applying Breiman's Theorem.

VIII. References

- *Airline Delays* Dataset
  - https://www.transtats.bts.gov/HomeDrillChart.asp
  - Prepared by Luis Villarreal
- References on the Bureau of Transportation Statistics
  - https://www.bts.gov/topics/airlines-and-airports/understanding-reporting-causes-flight-delays-and-cancellations
  - https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236
- Holidays Dataset
  - https://gist.github.com/shivaas/4758439
- SMOTE Algorithm
  - Chawla, Nitesh V., et al. "SMOTE: synthetic minority over-sampling technique." Journal of artificial intelligence research16 (2002): 321–357: https://arxiv.org/pdf/1106.1813.pdf
  - https://www.youtube.com/watch?v=FheTDyCwRdE
- Majority Class Splitting & Stacking Algorithm
  - https://en.wikipedia.org/wiki/Ensemble_learning
  - https://www.mdpi.com/2076-3417/8/5/815/pdf
  - http://marmota.dlsi.uji.es/WebBIB/papers/2003/paa-2.pdf
- Feature Importance for Random Forests
  - https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark
- General Airline Study
  - "Flight delays are costing airlines serious money", by The Associated Press, DEC 10, 2014.: https://mashable.com/2014/12/10/cost-of-delayed-flights/