Big Data Project – UE18CS322

Airline Delay Prediction

Akash Yadav – PES2201800415

Saiprakash L Shetty – PES2201800730

Naveen Suresh – PES2201800508

Project done under the guidance of Prof. Sudaroli Vijayakumar

OBJECTIVE :

The objective of this project is to perform analysis on the historical flight data to gain valuable insights and build a predictive model to predict whether a flight will be delayed or not for a given set of flight characteristics.

Questions to be answered post analysis:

- Which Airports have the Most Delays?
- Which Routes are typically the most delayed?
- Airport Origin delay per month
- Airport Origin delay per day/hour
- What are the primary causes for flight delays?

The objective of the predictive model(Logistic Regression) is to build a model to predict whether a flight will be delayed or not based on certain characteristics of the flight. Such a model may help both passengers as well as airline companies to predict future delays and minimize them for the future references.

Dataset:

The dataset which consists of around 6 million records has been taken from the **Bureau of Transportation Statistics,** it consists records of all commercial flight operations for the year 2014.
Link to the dataset: [The data. Data expo 09. ASA Statistics Computing and Graphics](), https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID= .

- YEAR : Year Of Flight Departure/Arrival
- MONTH : Month of Flight Departure/Arrival (5-May and 12- December)
- DAY_OF_WEEK : Day of Week (1-7)
- CARRIER : Code assigned by IATA and commonly used to identify a carrier.
- ORIGIN_CITY_NAME: Origin City of flight
- DEST_CITY_NAME: Destination City of flight
- ARR_DELAY_NEW : Difference in minutes between scheduled and actual arrival time. Early arrivals set to 0.
- CRS_DEP_TIME : Flight Departure time in Hours
- ARR_DEL15: Arrival Delay Indicator, More than 15 Minutes (1 = TRUE)
- DISTANCE : Distance between origin and destination in miles.
- CARRIER_DELAY: Carrier Delay, in Minutes
- WEATHER_DELAY: Weather Delay, in Minutes
- NAS_DELAY: National Air System Delay, in Minutes
- SECURITY_DELAY: Security Delay, in Minutes
- LATE_AIRCRAFT_DELAY: Late Aircraft Delay, in Minutes

Project Code and Execution:

For better readability and efficient usage of resources, we have configured pyspark with Jupyter notebook.

```
/*
Import findspark and findspark.init() to make pyspark importable as a regular library
*/
import findspark
findspark.init()
findspark.find()
import pyspark
findspark.find()

/*
A SparkSession can be used to create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files. To create a SparkSession, we use the following builder pattern.
*/

from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
conf = pyspark.SparkConf().setAppName('appName').setMaster('local')
sc = pyspark.SparkContext(conf=conf)
spark = SparkSession(sc)
```

#importing some packages we need

```
from pyspark.sql import SQLContext
from pyspark.sql.types import *
from pyspark.sql import Row
from pyspark.mllib.regression import LabeledPoint
from pyspark.sql.functions import udf
from pyspark.mllib.linalg import Vectors
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.param import Param, Params
from pyspark.mllib.classification import LogisticRegressionWithLBFGS, LogisticRegressionModel
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.stat import Statistics
from pyspark.ml.feature import OneHotEncoder, StringIndexer
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from IPython.display import display
from ipywidgets import interact
import sys
import numpy as np
import pandas as pd
import time
import datetime
import matplotlib.pyplot as plt
import os.path
```

/*

Apache Arrow is a cross-language development platform for in-memory data. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware.
In other words, it facilitates communication between many components, for example, reading a parquet file with Python (pandas) and transforming to a Spark dataframe
****/***

```
import pyarrow as pa
from pyarrow import csv
spark.conf.set("spark.sql.execution.arrow.enabled", "true")
%matplotlib inline
```

/*
Reading csv file
Removing headers of the dataset and renaming it so that further it will be easier for us analyse
****/***

```
textFile = sc.textFile('2014Flightdata.csv')
textFileRDD = textFile.map(lambda x: x.split(','))
header = textFileRDD.first()
textRDD = textFileRDD.filter(lambda r: r != header)
```

```
#Creating the Dataframe from RDD (Resilient Distributed Dataset)

def parse(r):
    try:
        x=Row(Year=int(r[0]),\
          Month=int(r[1]),\
          DayofMonth=int(r[2]),\
          DayOfWeek=int(r[3]),\
          DepTime=int(float(r[8])), \
          CRSDepTime=int(r[7]),\
          ArrTime=int(float(r[11])),\
          CRSArrTime=int(r[10]), \
          UniqueCarrier=r[4],\
          DepDelay=int(float(r[9])),\
          Origin=r[5],\
          Dest=r[6], \
          Distance=int(float(r[12])),\
          CarrierDelay=int(float(r[13])),\
          WeatherDelay=int(float(r[14])),\
          NASDelay= int(float(r[15])),\
          SecurityDelay=int(float(r[16])),\
          LateAircraftDelay=int(float(r[17])))
    except:
        x=None
    return x

rowRDD = textRDD.map(lambda r: parse(r)).filter(lambda r:r != None)
sqlContext = SQLContext(sc)
airline_df = sqlContext.createDataFrame(rowRDD)

/*
DepDelayed is a new column added to dataframe
True for delay > 15 minutes
False for delay <=15 minutes
*/

airline_df = airline_df.withColumn('DepDelayed', airline_df['DepDelay']>15)
def hour_ex(x):
    h = int(str(int(x)).zfill(4)[:2])
    return h

# register as a UDF
sqlContext.udf.register("hour_ex_py",hour_ex, IntegerType())
f_udf = udf(hour_ex, IntegerType())

#CRSDepTime: scheduled departure time (local, hhmm)
airline_df = airline_df.withColumn('hour', f_udf(airline_df.CRSDepTime))
airline_df.registerTempTable("airlineDF")
```

airline_df.head(n=5)

```
[Row(Year=2014, Month=1, DayofMonth=6, DayOfWeek=1, DepTime=1612, CRSDepTime=1510, ArrTime=1710, CRSArrTime=1620, UniqueCarrier='AA', Dep
Delay=62, Origin='DFW', Dest='SAT', Distance=247, CarrierDelay=10, WeatherDelay=0, NASDelay=0, SecurityDelay=0, LateAircraftDelay=40, Dep
Delayed=True, hour=15),
 Row(Year=2014, Month=1, DayofMonth=13, DayOfWeek=1, DepTime=1701, CRSDepTime=1510, ArrTime=1800, CRSArrTime=1620, UniqueCarrier='AA', De
pDelay=111, Origin='DFW', Dest='SAT', Distance=247, CarrierDelay=23, WeatherDelay=0, NASDelay=0, SecurityDelay=0, LateAircraftDelay=77, D
epDelayed=True, hour=15),
 Row(Year=2014, Month=1, DayofMonth=24, DayOfWeek=5, DepTime=1717, CRSDepTime=1510, ArrTime=1831, CRSArrTime=1620, UniqueCarrier='AA', De
pDelay=127, Origin='DFW', Dest='SAT', Distance=247, CarrierDelay=127, WeatherDelay=0, NASDelay=4, SecurityDelay=0, LateAircraftDelay=0, D
epDelayed=True, hour=15),
 Row(Year=2014, Month=1, DayofMonth=31, DayOfWeek=5, DepTime=1545, CRSDepTime=1510, ArrTime=1648, CRSArrTime=1620, UniqueCarrier='AA', De
pDelay=35, Origin='DFW', Dest='SAT', Distance=247, CarrierDelay=28, WeatherDelay=0, NASDelay=0, SecurityDelay=0, LateAircraftDelay=0, Dep
Delayed=True, hour=15),
 Row(Year=2014, Month=1, DayofMonth=6, DayOfWeek=1, DepTime=1741, CRSDepTime=1700, ArrTime=1845, CRSArrTime=1815, UniqueCarrier='AA', Dep
Delay=41, Origin='SAT', Dest='DFW', Distance=247, CarrierDelay=0, WeatherDelay=0, NASDelay=0, SecurityDelay=0, LateAircraftDelay=30, DepD
elayed=True, hour=17)]
```

```
cause_delay = sqlContext.sql("SELECT sum(WeatherDelay) Weather,sum(NASDelay) NAS,sum(SecurityDelay) Security,sum(LateAircraftDelay) lateAircraft,sum(CarrierDelay) Carrier\
                    FROM airlineDF ")
df_cause_delay = cause_delay.toPandas()
df_cause_delay.head()
```

	Weather	NAS	Security	lateAircraft	Carrier
0	2988669	16519549	65350	29483984	21259409

Exploration: Which Airports have the Most Delays?

```
groupedDelay = sqlContext.sql("SELECT Origin, count(*) conFlight,avg(DepDelay) delay \
                       FROM airlineDF \
                       GROUP BY Origin")

df_origin = groupedDelay.toPandas()
df_origin.sort_values('delay',ascending=0).head()
```

	Origin	conFlight	delay
205	LMT	12	133.750000
26	ESC	9	87.777778
209	EGE	318	84.877358
267	ACY	92	84.706522
170	MFR	614	79.980456

```
df = pd.read_csv('airports.dat', index_col=0,\
names = ['name', 'city', 'country','IATA','ICAO','lat','lng','alt','TZone','DST','Tz','air','Ourair'], \
        header=0)
del df['air']
del df['Ourair']
df.head()
```

	name	city	country	IATA	ICAO	lat	lng	alt	TZone	DST	Tz
2	Madang Airport	Madang	Papua New Guinea	MAG	AYMD	-5.207080	145.789001	20	10	U	Pacific/Port_Moresby
3	Mount Hagen Kagamuga Airport	Mount Hagen	Papua New Guinea	HGU	AYMH	-5.826790	144.296005	5388	10	U	Pacific/Port_Moresby
4	Nadzab Airport	Nadzab	Papua New Guinea	LAE	AYNZ	-6.569803	146.725977	239	10	U	Pacific/Port_Moresby
5	Port Moresby Jacksons International Airport	Port Moresby	Papua New Guinea	POM	AYPY	-9.443380	147.220001	146	10	U	Pacific/Port_Moresby
6	Wewak International Airport	Wewak	Papua New Guinea	WWK	AYWK	-3.583830	143.669006	19	10	U	Pacific/Port_Moresby

```
df_airports = pd.merge(df_origin, df, left_on = 'Origin', right_on = 'IATA')
df_airports.head()
```

	Origin	conFlight	delay	name	city	country	IATA	ICAO	lat	lng	alt	TZone	DST	Tz
0	PSE	111	50.495495	Mercedita Airport	Ponce	Puerto Rico	PSE	TJPS	18.008301	-66.563004	29	-4	U	America/Puerto_Rico
1	INL	81	65.308642	Falls International Airport	International Falls	United States	INL	KINL	48.566200	-93.403099	1185	-6	A	America/Chicago
2	DLG	37	36.918919	Dillingham Airport	Dillingham	United States	DLG	PADL	59.044701	-158.505005	81	-9	A	America/Anchorage
3	MSY	7892	54.164470	Louis Armstrong New Orleans International Airport	New Orleans	United States	MSY	KMSY	29.993401	-90.258003	4	-6	A	America/Chicago
4	PPG	36	60.444444	Pago Pago International Airport	Pago Pago	American Samoa	PPG	NSTU	-14.331000	-170.710007	32	-11	U	Pacific/Pago_Pago

```
df_airports.sort_values('delay',ascending=0).head()
```

	Origin	conFlight	delay	name	city	country	IATA	ICAO	lat	lng	alt	TZone	DST	Tz
205	LMT	12	133.750000	Crater Lake-Klamath Regional Airport	Klamath Falls	United States	LMT	KLMT	42.156101	-121.733002	4095	-8	A	America/Los_Angeles
26	ESC	9	87.777778	Delta County Airport	Escanaba	United States	ESC	KESC	45.722698	-87.093697	609	-5	A	America/New_York
209	EGE	318	84.877358	Eagle County Regional Airport	Vail	United States	EGE	KEGE	39.642601	-106.917999	6548	-7	A	America/Denver
267	ACY	92	84.706522	Atlantic City International Airport	Atlantic City	United States	ACY	KACY	39.457600	-74.577202	75	-5	A	America/New_York
170	MFR	614	79.980456	Rogue Valley International Medford Airport	Medford	United States	MFR	KMFR	42.374199	-122.873001	1335	-8	A	America/Los_Angeles

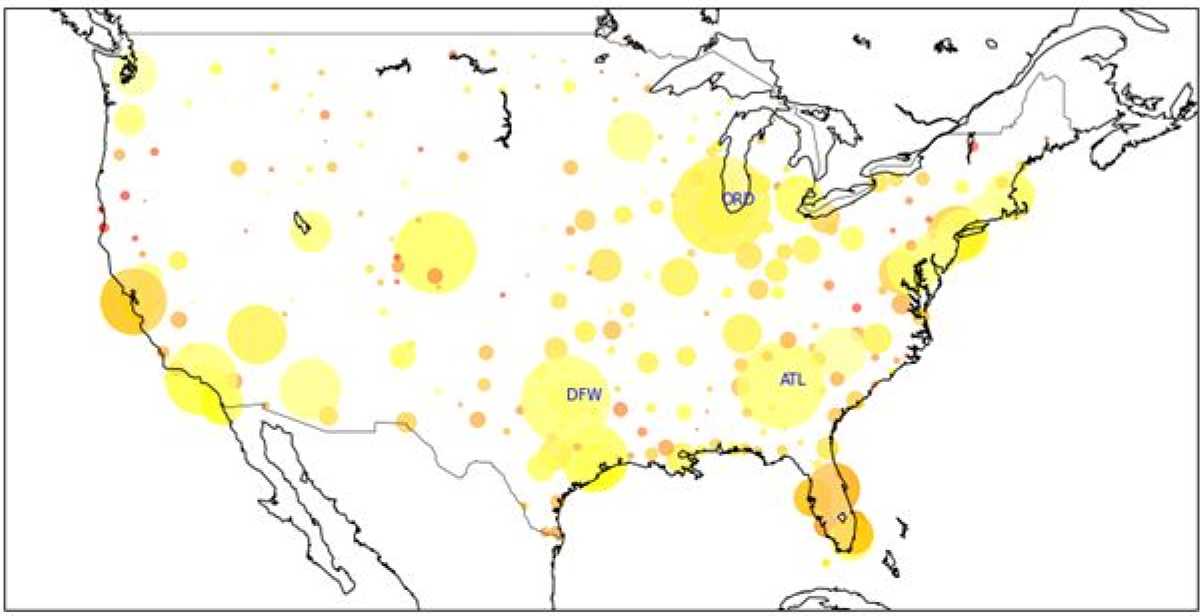
/* ***The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output.Since probability of anything exists only between the range of 0 and 1.***
A Z-score is a numerical measurement that describes a value's relationship to the mean of a group of values. Z-score is measured in terms of standard deviations from the mean. If a Z-score is 0, it indicates that the data point's score is identical to the mean score.
*/

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def zscore(x):
    return (x-np.average(x))/np.std(x)

import os
os.environ['PROJ_LIB'] = r'C:\Users\spl2s\anaconda3\pkgs\proj4-5.2.0-\
ha925a31_1\Library\share'

from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
```

Plotting Map

```
rcParams['figure.figsize'] = (14,10)
my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=-130, llcrnrlat=22, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=-60, urcrnrlat=50) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To create a color map
colors = plt.get_cmap('hot')(np.linspace(0.0, 1.0, 30))
colors=np.flipud(colors)

#----- Scatter -------
countrange=max(df_airports['conFlight'])-min(df_airports['conFlight'])
al=np.array([sigmoid(x) for x in zscore(df_airports['delay'])])
xs,ys = my_map(np.asarray(df_airports['lng']), np.asarray(df_airports['lat']))
val=df_airports['conFlight']*4000.0/countrange

my_map.scatter(xs, ys,  marker='o', s= val, alpha = 0.8,color=colors[(al*20).astype(int)])

#----- Text -------
df_text=df_airports[(df_airports['conFlight']>60000) & (df_airports['IATA'] != 'HNL')]
xt,yt = my_map(np.asarray(df_text['lng']), np.asarray(df_text['lat']))
```

```
txt=np.asarray(df_text['IATA'])
zp=zip(xt,yt,txt)
for row in zp:
    #print zp[2]
    plt.text(row[0],row[1],row[2], fontsize=10, color='blue',)

print("Each marker is an airport.")
print("Size of markers: Airport Traffic (larger means higher number of flights in year)")
print("Color of markers: Average Flight Delay (Redder means longer delays)")

plt.show()
```



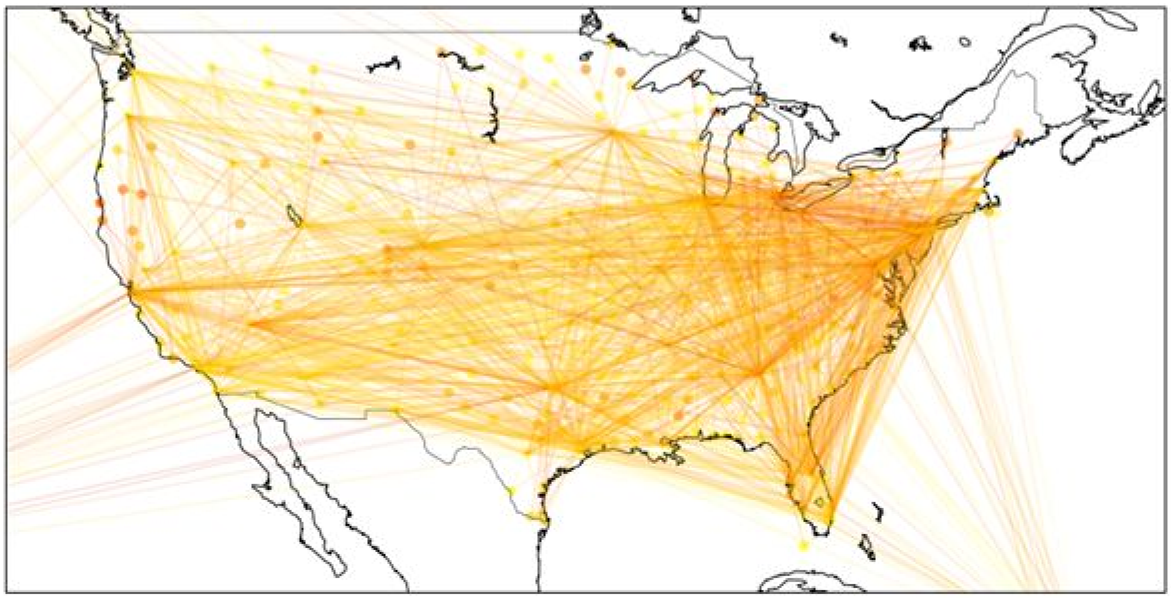

```
grp_rout_Delay = sqlContext.sql("SELECT Origin, Dest, count(*) traffic,avg(Distance) avgDist,\
                                    avg(DepDelay) avgDelay\
                                FROM airlineDF \
                                GROUP BY Origin,Dest")
rout_Delay = grp_rout_Delay.toPandas()

df_airport_rout1 = pd.merge(rout_Delay, df, left_on = 'Origin', right_on = 'IATA')
df_airport_rout2 = pd.merge(df_airport_rout1, df, left_on = 'Dest', right_on = 'IATA')
df_airport_rout = df_airport_rout2[["Origin","lat_x","lng_x","Dest","lat_y","lng_y",\
                                    "avgDelay", "traffic"]]
df_airport_rout.sort_values('avgDelay',ascending=0).head()
```

	Origin	lat_x	lng_x	Dest	lat_y	lng_y	avgDelay	traffic
2679	BNA	36.124500	-86.678200	BHM	33.562901	-86.753502	402.0	1
2242	PIT	40.491501	-80.232903	PBI	26.683201	-80.095596	330.0	1
3776	JFK	40.639801	-73.778900	JAC	43.607300	-110.737999	322.0	1
3983	SYR	43.111198	-76.106300	BTV	44.471901	-73.153297	257.0	1
1936	GTF	47.481998	-111.371002	MSP	44.882000	-93.221802	256.5	2

```
rcParams['figure.figsize'] = (14,10)


my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=-130, llcrnrlat=22, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=-60, urcrnrlat=50) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

delay=np.array([sigmoid(x) for x in zscore(df_airports["delay"])])
colors = plt.get_cmap('hot')(np.linspace(0.0, 1.0, 40))
colors=np.flipud(colors)
xs,ys = my_map(np.asarray(df_airports['lng']), np.asarray(df_airports['lat']))
xo,yo = my_map(np.asarray(df_airport_rout['lng_x']), np.asarray(df_airport_rout['lat_x']))
xd,yd = my_map(np.asarray(df_airport_rout['lng_y']), np.asarray(df_airport_rout['lat_y']))

my_map.scatter(xs, ys,  marker='o',  alpha = 0.8,color=colors[(delay*20).astype(int)])

al=np.array([sigmoid(x) for x in zscore(df_airport_rout["avgDelay"])])
f=zip(xo,yo,xd,yd,df_airport_rout['avgDelay'],al)
for row in f:
    plt.plot([row[0],row[2]], [row[1],row[3]],'-',alpha=0.07, \
             color=colors[(row[5]*30).astype(int)] )


for row in zp:
    plt.text(row[0],row[1],row[2], fontsize=10, color='blue',)

print("Each line represents a route from the Origin to Destination airport.")
```

```
print("The redder line, the higher probablity of delay.")
plt.show()
```

Each line represents a route from the Origin to Destination airport.
The redder line, the higher probability of delay.



```
Origin_Airport="SJC"

df_ORG = sqlContext.sql("SELECT * from airlineDF WHERE origin='"+ Origin_Airport+"'")
df_ORG.registerTempTable("df_ORG")
df_ORG.select('ArrTime','CRSArrTime','CRSDepTime',\
          'DayOfWeek','DayofMonth','DepDelay','DepTime','Dest').show(2)
```

```
+-------+----------+----------+---------+----------+--------+-------+----+
|ArrTime|CRSArrTime|CRSDepTime|DayOfWeek|DayofMonth|DepDelay|DepTime|Dest|
+-------+----------+----------+---------+----------+--------+-------+----+
|   1529|      1340|       725|        4|         2|      94|    859| ORD|
|   1402|      1340|       725|        7|         5|       6|    731| ORD|
+-------+----------+----------+---------+----------+--------+-------+----+
only showing top 2 rows
```

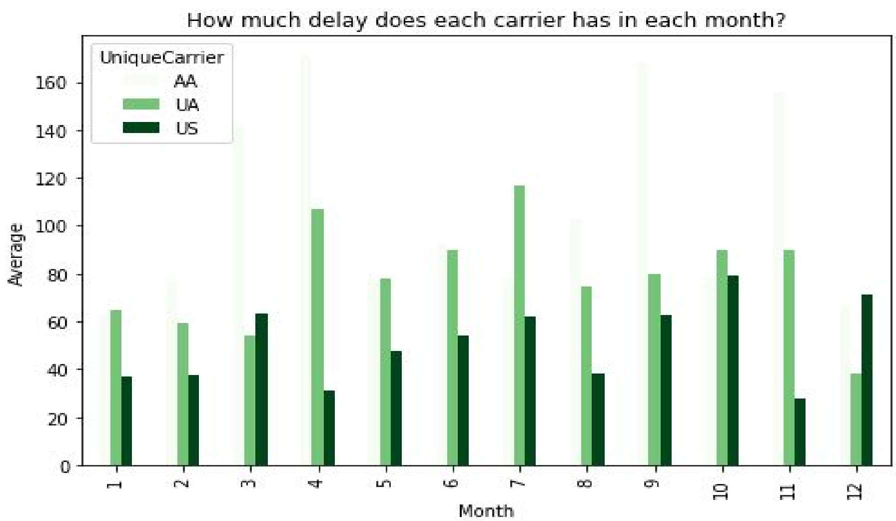
```
print("total flights from this airport: " + str(df_ORG.count()))
```

total flights from this airport: 7830

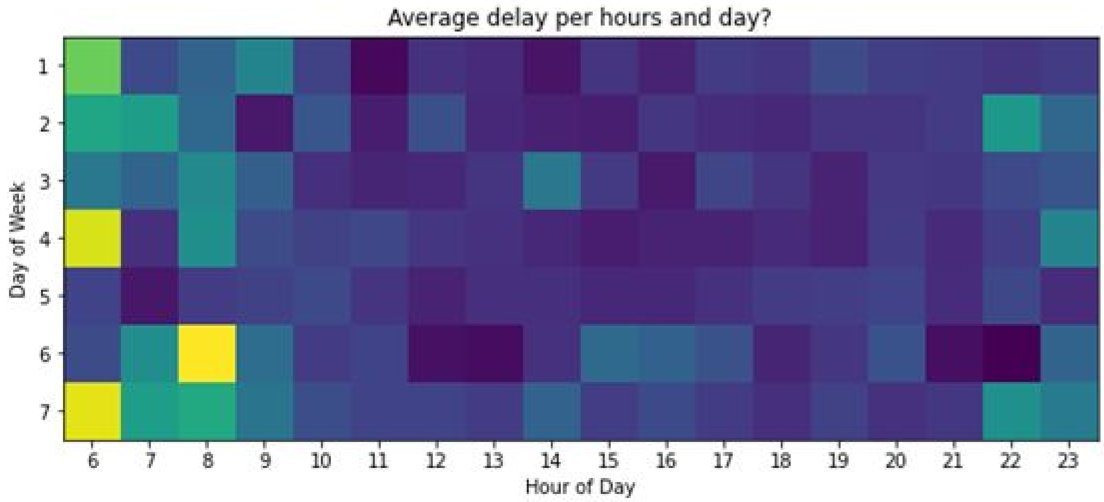
```
grp_carr = sqlContext.sql("SELECT  UniqueCarrier,month, avg(DepDelay) avgDelay from df_ORG \
                    WHERE DepDelayed=True \
                    GROUP BY UniqueCarrier,month")
s = grp_carr.toPandas()
ps = s.pivot(index='month', columns='UniqueCarrier', values='avgDelay')[['AA','UA','US']]
```

```
rcParams['figure.figsize'] = (8,5)
ps.plot(kind='bar', colormap='Greens');
plt.xlabel('Month')
plt.ylabel('Average')
plt.title('How much delay does each carrier has in each month?')
```

```
Text(0.5, 1.0, 'How much delay does each carrier has in each month?')
```




```
hour_grouped =
df_ORG.filter(df_ORG['DepDelayed']).select('DayOfWeek','hour','DepDelay').groupby('DayOfWeek','hour').mean('DepDelay')

rcParams['figure.figsize'] = (10,5)
dh = hour_grouped.toPandas()
c = dh.pivot('DayOfWeek','hour')
X = c.columns.levels[1].values
Y = c.index.values
Z = c.values
plt.xticks(range(0,24), X)
plt.yticks(range(0,7), Y)
plt.xlabel('Hour of Day')
plt.ylabel('Day of Week')
plt.title('Average delay per hours and day?')
plt.imshow(Z)
```




```
Origin_Airport="SJC"

df_ORG=sqlContext.sql("SELECT * from airlineDF WHERE Origin = '"+Origin_Airport+"'")
df_ORG.registerTempTable("df_ORG")


#feature selection
df_model=df_ORG
#stringIndexer encodes a string column of labels to a column of label indices. StringIndexer can encode multiple columns.
stringIndexer1 = StringIndexer(inputCol="Origin", outputCol="originIndex")
model_stringIndexer = stringIndexer1.fit(df_model)
indexedOrigin = model_stringIndexer.transform(df_model)
#One-hot encoding maps a categorical feature, represented as a label index, to a binary vector with at most a single one-value indicating the presence of a specific feature value from among the set of all feature values.
encoder1 = OneHotEncoder(dropLast=False,inputCol="originIndex", outputCol="originVec")
#fit function fits the model into the input dataset
ohe = encoder1.fit(indexedOrigin)
#The transform function applies the values of the parameters on the actual data and gives the normalized value.
df_model = ohe.transform(indexedOrigin)

#VectorAssembler is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector
assembler = VectorAssembler(
    inputCols = ['Year','Month','DayofMonth','DayOfWeek','hour','Distance','originVec'],
    outputCol = "features")
output = assembler.transform(df_model)

#"features" from outputCol is a dense vector and spark 2.0 doesnt recognize this particular dilemma, so we wrote a specified function to resolve to later map the contents to RDD "airlineRDD"
from pyspark.mllib import linalg as mllib_linalg
from pyspark.ml import linalg as ml_linalg
from pyspark.sql.functions import col
def as_old(v):
    if isinstance(v, ml_linalg.SparseVector):
        return mllib_linalg.SparseVector(v.size, v.indices, v.values)
    if isinstance(v, ml_linalg.DenseVector):
        return mllib_linalg.DenseVector(v.values)
```

```
    raise ValueError("Unsupported type {0}".format(type(v)))
airlineRDD=output.select(col("DepDelayed").alias("label"), col("features"))\
           .rdd\
           .map(lambda row: LabeledPoint(row.label, as_old(row.features)))
```

#A labeled point is a local vector, either dense or sparse, associated with a label/response. Local vector has integer-typed and 0-based indices and double-typed values, stored on a single machine.

```
airlineRDD.take(2)
```

```
[LabeledPoint(1.0, [2014.0,1.0,2.0,4.0,7.0,1829.0,1.0]),
 LabeledPoint(0.0, [2014.0,1.0,5.0,7.0,7.0,1829.0,1.0])]
```

```
#  Splitting dataset into train and test datasets to 70% and 30% respectively
trainRDD,testRDD=airlineRDD.randomSplit([0.7,0.3])
```

```
# Build the model
model = LogisticRegressionWithLBFGS.train(trainRDD)
```

```
# Evaluating the model on testing data
labelsAndPreds = testRDD.map(lambda p: (p.label, model.predict(p.features)))
labelsAndPreds.take(2)
```

#Labels are the known values for old data.
#Prediction is your predicted value for new data, where you do not have a label (or pretend that you do not have a #label - in evaluation).
#During training, you try to make your predictions match the labels.

#Training Error: We get the by calculating the classification error of the model on the same data the model was trained on

```
trainErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1]).count() / float(testRDD.count())
print("Training Error = " + str(trainErr))
```

```
Training Error = 0.1634980988593156
```

```
testRDD.take(10)
```

```
[LabeledPoint(1.0, [2014.0,1.0,2.0,4.0,7.0,1829.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,3.0,5.0,13.0,1438.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,20.0,1.0,13.0,1438.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,21.0,2.0,13.0,1438.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,4.0,6.0,6.0,1576.0,1.0]),
 LabeledPoint(0.0, [2014.0,1.0,3.0,5.0,22.0,2116.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,5.0,7.0,6.0,308.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,7.0,2.0,6.0,308.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,4.0,6.0,19.0,308.0,1.0]),
 LabeledPoint(1.0, [2014.0,1.0,12.0,7.0,17.0,948.0,1.0])]
```

```
def conf(r):
    if r[0] == r[1] ==1: x= 'TP'
    if r[0] == r[1] ==0: x= 'TN'
    if r[0] == 1 and  r[1] ==0: x= 'FN'
    if r[0] == 0 and  r[1] ==1: x= 'FP'
    return (x)
acc1 = labelsAndPreds.map(lambda vp: ((vp[1], vp[0]), 1)).reduceByKey(lambda a, b: a + b).take(5)
acc = [(conf(x[0]),x[1]) for x in acc1]

for x in range(len(acc)):
    print (acc[x])
```

```
('FN', 364)
('FP', 23)
('TN', 31)
('TP', 1949)
```

```
TP=TN=FP=FN=0.0
for x in acc:
    if x[0]=='TP': TP= x[1]
    if x[0]=='TN': TN= x[1]
    if x[0]=='FP': FP= x[1]
    if x[0]=='FN': FN= x[1]
```

#The importance of the machine epsilon is that it measures the effects of rounding errors made when adding, subtracting, multiplying, or dividing two numbers.

```
eps = sys.float_info.epsilon
Accuracy = (TP+TN) / (TP + TN+ FP+FN+eps)
print("Model Accuracy for SJC:")
print(float(Accuracy*100))
```

We found the accuracy to be around 83.65% using Logistic Regression further if required other modelling methods can be opted too !!!!!!

OUTPUT

```
Model Accuracy for SJC:
83.65019011406845
```

We found the accuracy to be around 83.65% using Logistic Regression further if required other modelling methods can be opted too !!!!!!

Github : https://github.com/SaiprakashShetty/Big-Data-Airline-Delay-Prediction

Knowledge about Apache Arrow :
https://towardsdatascience.com/a-gentle-introduction-to-apache-arrow-with-apache-spark-and-pandas-bb19ffe0ddae