Domestic (US and US territories) Flight Delay Project

w261 Final Project Team 19 Summer 2022

Team Members: Grace Lee, Shivangi Pandey, Sybil Santos-Burgan, Beijing Wu

Abstract

In Q2 of 2022 over 20% of scheduled flights were delayed. In April 2022, consumer complaints increased over 300% from April 2019. Many airlines have staffing issues as a lingering pandemic effect and in recent months “no major carrier [reliably arrived] on time more than 90 percent of the time”, a rarity before COVID. With air travel picking up and staffing issues persisting, delays will continue. These delays cause additional issues at airports needing to choreograph activity within and without.

The goal of this study is to assist airport planning by predicting flight delays. We use domestic flight and airport data from the U.S. Department of Transportation and weather information from the National Oceanic and Atmospheric Administration repository. We will attempt several models to predict delays (defined as departures 15 minutes or more past planned departure times). We will predict using Logistic Regression for Classification and use F2-score, balanced accuracy, Matthews Correlation Coefficient, and AUC to evaluate the best performing model. Predictions will be two hours before planned departure to give airports time to manage operations. We conducted a logistic regression as a baseline, which had poor results and indicated room for improvement via feature engineering and parameter tuning.

Notebook Setup

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
import pyspark.sql.functions as F
from pyspark.ml.feature import StringIndexer
from pyspark.sql import Window

from scipy import stats
from scipy.stats import norm

pd.options.display.float_format = '{:.0f}'.format
from pyspark.sql.functions import col
from pyspark.sql.functions import col, max

from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.model_selection import train_test_split
from sklearn.utils import resample
from pyspark.sql.functions import col,isnan,when,count
```

```
#Set up blob access
blob_container = "261finalproject" # The name of your container created in
https://portal.azure.com
storage_account = "ssburgan" # The name of your Storage account created in
https://portal.azure.com
secret_scope = "261finalprojectscope1" # The name of the scope created in your
local computer using the Databricks CLI
secret_key = "bestteam" # The name of the secret key created in your local
computer using the Databricks CLI
blob_url = f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"
mount_path = "/mnt/mids-w261"

spark.conf.set(
  f"fs.azure.sas.{blob_container}.{storage_account}.blob.core.windows.net",
  dbutils.secrets.get(scope = secret_scope, key = secret_key)
)

# Inspect the Mount's Final Project folder
# Please IGNORE dbutils.fs.cp("/mnt/mids-
w261/datasets_final_project/stations_data/", "/mnt/mids-
w261/datasets_final_project_2022/stations_data/", recurse=True)
data_BASE_DIR = "dbfs:/mnt/mids-w261/datasets_final_project_2022/"
display(dbutils.fs.ls(f"{data_BASE_DIR}"))
```

	path	name
1	dbfs:/mnt/mids-w261/datasets_final_project_2022/parquet_airlines_data/	parqu
2	dbfs:/mnt/mids-w261/datasets_final_project_2022/parquet_airlines_data_1y/	parqu
3	dbfs:/mnt/mids-w261/datasets_final_project_2022/parquet_airlines_data_3m/	parqu
4	dbfs:/mnt/mids-w261/datasets_final_project_2022/parquet_airlines_data_6m/	parqu
5	dbfs:/mnt/mids-w261/datasets_final_project_2022/parquet_weather_data/	parqu
6	dbfs:/mnt/mids-w261/datasets_final_project_2022/parquet_weather_data_1y/	parqu

Showing all 9 rows.

Hypothesis:

We hope to assist airports in their planning by predicted flight delays. We hypothesize that tracking prior flight information will be a major predictor for departure delays.

Null hypothesis: The addition of prior flight information to current flight and weather data has no impact on predicting departure delays

Alternate hypothesis: Adding information about tracking prior flights improves prediction of departure delays.

EDA and Data Join Pipeline

Pipeline Block Diagram

Data Description

We use 2015 - 2021 domestic flight and airport data from the U.S. Department of Transportation and weather data from the National Oceanic and Atmospheric Administration repository. For a basic understanding of the data, we conducted exploratory analysis on 2015 Q1 (3 months) for each dataset. We also conducted further EDA on the joined dataset.

A note on CSV Data

CSV Data can be partitioned and stored in parquet format. Parquet provides compression , storage of file metadata/schema within the file. CSV is row oriented while parquet is column oriented. This format provides advantages such as columnar storage is more efficient than row based as compression algorithms can make use of ‘similarity of adjacent data’ more readily than in row based methods. Also, data retrieval is faster as query can fetch relevant columns only which is beneficial in case of large dataset.

Flights

This dataset contains information about domestic passenger flights on-time performance. The initial dataset contains 107 features, of which the DEP_DEL15 (departure delay 15 minutes or more) field is our target variable. This field is 0 when not delayed and 1 when delayed. From the EDA on the 3 month dataset, around 3%

of this field is missing, which will be removed from the data when modeling. We anticipate that there will be an imbalanced classification problem, since around 20% of the non-missing data is delayed (again from the 3 month dataset).

Each record has a unique identifier for flight, datetime zone is in local time of airport region and military format. Record also contains delay and arrival metrics. Focus will be only on departed flights.

Key features we anticipate including:

- YEAR
- MONTH
- FL_DATE
- DAY_OF_WEEK
- OP_CARRIER
- ORIGIN
- ORIGIN_CITY_MARKET_ID
- DEST
- CRS_DEP_TIME
- ARR_DELAY
- DIVERTED
- TAIL_NUM
- DISTANCE
- CANCELLED
- DEP_DEL15

Load Airline Data 3m

```
# Load 2015 Q1 for Flights
df_airlines = spark.read.parquet(f"{data_BASE_DIR}parquet_airlines_data_3m/")
display(df_airlines)
```

	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE
1	1	2	19	4	2015-02-19
2	1	2	20	5	2015-02-20
3	1	2	21	6	2015-02-21
4	1	2	22	7	2015-02-22
5					

6	1	2	24	2	2015-02-24

Truncated results, showing first 1000 rows.

Weather

This dataset contains weather condition information from 2015-2021 downloaded from the National Oceanic and Atmospheric Administration (https://www.ncei.noaa.gov/access/metadata/landing-page/bin/iso?id=gov.noaa.ncdc:C00679) repository. It contains hourly weather information from stations in UTC datetime format. The initial dataset contains 177 features.

Key features categories we anticipate include:

- STATION
- DATE
- LATITUDE
- LONGITUDE
- NAME
- REPORT_TYPE
- HourlyDewPointTemperature
- HourlySkyConditions
- HourlyVisibility
- HourlyWindSpeed

Load Weather

```
# Load the 2015 Q1 for Weather
df_weather = spark.read.parquet(f"
{data_BASE_DIR}parquet_weather_data_3m/").filter(col('DATE') > "2015-01-
01T00:00:00.000")
display(df_weather)
```

	STATION	DATE	LATITUDE	LONGITUDE	ELEVA
1	52652099999	2015-01-01T02:00:00	39.0833333	100.2833333	1462.0
2	52652099999	2015-01-01T05:00:00	39.0833333	100.2833333	1462.0
3	52652099999	2015-01-01T08:00:00	39.0833333	100.2833333	1462.0
4	52652099999	2015-01-01T11:00:00	39.0833333	100.2833333	1462.0

5	52652099999	2015-01-01T14:00:00	39.0833333	100.2833333	1462.0
6	52652099999	2015-01-01T17:00:00	39.0833333	100.2833333	1462.0

Truncated results, showing first 1000 rows.

Stations

The stations dataset downloaded from the US Department of Transportation and provides metadata about each airport. The dataset contains station data and its neighbor station along with distance between stations. The initial dataset contains 10 features.

Key Features Categories we anticipate include:

- station_id
- neighbor_name
- neighbor_call
- distance_to_neighbor

Load Stations

```
df_stations = spark.read.parquet(f"{data_BASE_DIR}stations_data/*")
display(df_stations)
```

	usaf	wban	station_id	lat	lon	neighb
1	690020	93218	69002093218	36	-121.233	69002
2	690020	93218	69002093218	36	-121.233	69007
3	690020	93218	69002093218	36	-121.233	69014
4	690020	93218	69002093218	36	-121.233	700271
5	690020	93218	69002093218	36	-121.233	70045
6	690020	93218	69002093218	36	-121.233	70063

Truncated results, showing first 1000 rows.

Airline Names

The airline names is available from the Bureau of Transportation Statistics (https://www.bts.gov/topics/airlines-and-airports/airline-codes) and provides the airline name description associated with each code. The original csv was downloaded from here (https://raw.githubusercontent.com/beanumber/airlines/master/data-raw/airlines.csv).

Load Airline Names

```
df_airline_names = spark.read.parquet(f"{blob_url}/airline_names")
df_airline_names = df_airline_names.withColumnRenamed("Code", "Airline_Code")
df_airline_names = df_airline_names.withColumnRenamed("Description",
"Airline_Description")
display(df_airline_names)
```

	Airline_Code	Airline_Description
1	02Q	Titan Airways
2	04Q	Tradewind Aviation
3	05Q	Comlux Aviation, AG
4	06Q	Master Top Linhas Aereas Ltd.
5	07Q	Flair Airlines Ltd.
6	09Q	Swift Air, LLC

Truncated results, showing first 1000 rows.

Airport Codes

The airport codes may refer to either IATA airport code, a three-letter code which is used in passenger reservation, ticketing and baggage-handling systems, or the ICAO airport code which is a four letter code used by ATC systems and for airports that do not have an IATA airport code.

The csv was downloaded from datahub.io (https://datahub.io/core/airport-codes).

Load Airport Codes

```
df_airport_codes = spark.read.parquet(f"{blob_url}/airport_codes")
df_airport_codes.createOrReplaceTempView("airports")
df_airport_codes = spark.sql("SELECT ident, type, name as
airport_name,iata_code FROM airports where iata_code is not null")
display(df_airport_codes)
df_airport_codes.printSchema()
```

	ident	type	airport_name
1	03N	small_airport	Utirik Airport
2	07FA	small_airport	Ocean Reef Club Airport
3	0AK	small_airport	Pilot Station Airport
4	0CO2	small_airport	Crested Butte Airpark
5	0TE7	small_airport	LBJ Ranch Airport
6	13MA	small_airport	Metropolitan Airport

Truncated results, showing first 1000 rows.

```
root
 |-- ident: string (nullable = true)
 |-- type: string (nullable = true)
 |-- airport_name: string (nullable = true)
 |-- iata_code: string (nullable = true)
```

Flights -Join Airline Name

```
#df_airport_names = spark.read.parquet(f"{blob_url}/airline_names")
#df_airport_codes = spark.read.parquet(f"{blob_url}/airport_codes")

df = df_airlines.join(df_airline_names,df_airlines.OP_UNIQUE_CARRIER ==
df_airline_names.Airline_Code,"inner")
df = df.withColumnRenamed("DESCRIPTION", "CARRIER")

display(df)
```

	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE
1	1	2	19	4	2015-02-19
2	1	2	20	5	2015-02-20
3	1	2	21	6	2015-02-21
4	1	2	22	7	2015-02-22
5	1	2	23	1	2015-02-23
6	1	2	24	2	2015-02-24

Truncated results, showing first 1000 rows.

Data Dictionary

The following data dictionaries are subset from the full data. The following featues were decided by domain knowledge and EDA (see below). We plan to use these features for further modeling.

Airlines (Flights) Data:

Feature	Description	data type
DEP_DEL15	Indicator of departure delay 15 minutes or more. (1 = Yes)	float64
YEAR	Year	int32
QUARTER	Quarter	int32
MONTH	Month	int32
FL_DATE	Flight Date yyyy-mm-dd	string
DAY_OF_WEEK	Day of the Week	int32
OP_CARRIER	Airline	string
ORIGIN	Origin Airport	string
DEST	Destination Airport	string
CRS_DEP_TIME	Computer Reservation System scheduled departure time	int32
ARR_DELAY	Difference between scheduled and actual arrival time (in minutes)	float64
DIVERTED	Diverted Flight Indicator (1 = Yes)	float64
TAIL_NUM	Plane tail numbers	string
DISTANCE	Distance between airports (miles)	float64

Feature	Description	data type
CANCELLED	Cancelled Flight Indicator (1 = Yes)	float64

Weather Data:

Feature	Description	data type
STATION	Weather station code	string
DATE	Weather reading date yyyy-mm-dd hh:mm:ss	string
LATITUDE	Latitude of the weather station	float64
LONGITUDE	Longitude of the weather station	float64
NAME	Name of weather station	string
REPORT_TYPE	Code that denotes the type of geophysical surface observation	string
OvercastIndex	Overcast weather indicator (0 = No, 1 = Yes)	float64
WindSpeedAvg	Wind speed (in meters per second) averaged over 2-hour window prior to the time indicated	float64
DewPointTempAvg	Dew point temperature (in Degrees Celsius) averaged over 2-hour window prior to the time indicated	float64
VisibilityAvg	Visibility (in meters) averaged over 2-hour window prior to the time indicated	float64

Stations Data:

Feature	Description	data type
station_id	Weather station code represents base weather station	string

Feature	Description	data type
neighbor_id	Weather station code represents neighboring stations from base station	string
neighbor_name	Name of station	string
neighbor_call	ICAO airport code	string
distance_to_neighbor	Distance from base station. Base station distance is 0.	double

Airport Codes Data:

Feature	Description	data type
ident	ICAO airport code	string
type	Type of location	string
airport_name	Name	string
iata_code	IATA code	string

EDA

The following EDA is done on the 2015 Q1 data (3 month dataset) as an initial view. Below this section will be further EDA on the joined data.

Airline

Summary Information

Flights - Initial Features to Keep

```
#The following features are variables we initially selected as important from
looking at the Department of Transportation's data descriptions

keep_columns = ['YEAR','QUARTER','MONTH', 'DAY_OF_MONTH', 'DAY_OF_WEEK',
'FL_DATE',
'OP_CARRIER', 'ORIGIN_AIRPORT_ID','ORIGIN', 'ORIGIN_CITY_NAME',
'ORIGIN_STATE_ABR',
'ORIGIN_CITY_MARKET_ID','ORIGIN_STATE_FIPS','ORIGIN_STATE_NM','ORIGIN_WAC',
'DEST_AIRPORT_ID', 'DEST', 'DEST_CITY_NAME', 'DEST_STATE_ABR',
'DEST_CITY_MARKET_ID',
'CRS_DEP_TIME', 'DEP_DELAY','ARR_DELAY', 'DEP_DELAY_GROUP',
'DEP_DEL15','DEP_TIME','DEP_TIME_BLK','TAXI_OUT', 'WHEELS_OFF',
'TAXI_IN', 'CRS_ARR_TIME', 'DIVERTED', 'CRS_ELAPSED_TIME',
'ACTUAL_ELAPSED_TIME', 'AIR_TIME',
'FLIGHTS', 'DISTANCE', 'DISTANCE_GROUP', 'FIRST_DEP_TIME', 'TOTAL_ADD_GTIME',
'LONGEST_ADD_GTIME',
'DIV_AIRPORT_LANDINGS', 'DIV_REACHED_DEST', 'DIV_ACTUAL_ELAPSED_TIME',
'DIV_DISTANCE','TAIL_NUM',
'CARRIER_DELAY','WEATHER_DELAY','NAS_DELAY','SECURITY_DELAY','LATE_AIRCRAFT_DEL
AY','CANCELLED'
]

df = df_airlines[keep_columns]
display(df)
```

	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK
1	2015	1	2	19	4
2	2015	1	2	20	5
3	2015	1	2	21	6
4	2015	1	2	22	7
5	2015	1	2	23	1
6	2015	1	2	24	2

Truncated results, showing first 1000 rows.

```
print("Total Count: ",df.count())
df.printSchema()
df.show(1, vertical=True)
```

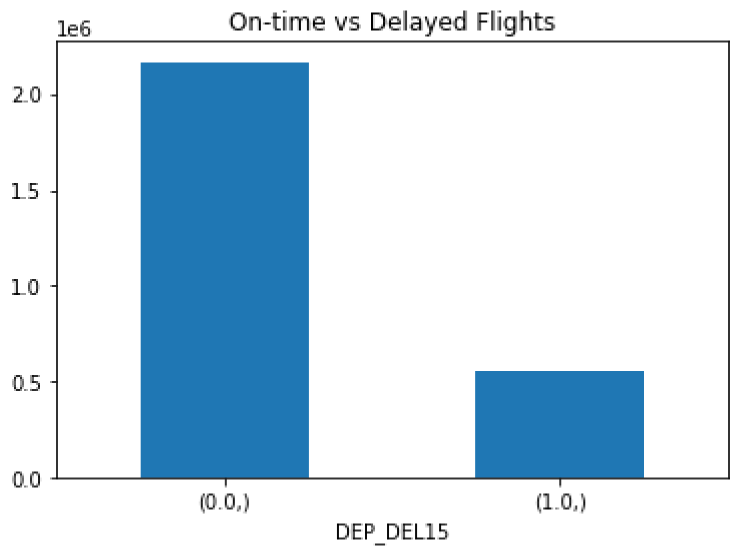
```
Total Count:  2806942
root
 |-- YEAR: integer (nullable = true)
 |-- QUARTER: integer (nullable = true)
 |-- MONTH: integer (nullable = true)
 |-- DAY_OF_MONTH: integer (nullable = true)
 |-- DAY_OF_WEEK: integer (nullable = true)
 |-- FL_DATE: string (nullable = true)
 |-- OP_CARRIER: string (nullable = true)
 |-- ORIGIN_AIRPORT_ID: integer (nullable = true)
 |-- ORIGIN: string (nullable = true)
 |-- ORIGIN_CITY_NAME: string (nullable = true)
 |-- ORIGIN_STATE_ABR: string (nullable = true)
 |-- ORIGIN_CITY_MARKET_ID: integer (nullable = true)
 |-- ORIGIN_STATE_FIPS: integer (nullable = true)
 |-- ORIGIN_STATE_NM: string (nullable = true)
 |-- ORIGIN_WAC: integer (nullable = true)
 |-- DEST_AIRPORT_ID: integer (nullable = true)
 |-- DEST: string (nullable = true)
 |-- DEST_CITY_NAME: string (nullable = true)
 |-- DEST_STATE_ABR: string (nullable = true)
```

On Time vs Delayed EDA

On-time vs Delayed Flights

```
df.select('DEP_DEL15').toPandas().value_counts().plot.bar(title = "On-time vs
Delayed Flights", legend = False, rot = 0)
```

Out[111]:



```
<AxesSubplot:title={'center':'On-time vs Delayed Flights'}, xlabel='DEP_DEL15'>
```

```
# percent of data that is delayed (taking out the missing values)
print('percent late flights', df.filter(F.col('DEP_DEL15') ==
1).select('DEP_DEL15').count()/df.filter((F.col('DEP_DEL15') == 1) |
(F.col('DEP_DEL15')==0)).count())

# count number of delayed instances
df.select('DEP_DEL15').toPandas().value_counts()
```

```
percent late flights 0.20413469535293097
Out[112]: DEP_DEL15
0            2166530
1             555702
dtype: int64
```

Missing Data

```
# Check where departure delays are missing
missing_dep_del15 = df.select('DEP_DEL15').toPandas().isnull().sum()
print('Count of missing', missing_dep_del15)
print('Count of total data: ', df.count())
float(missing_dep_del15)/df.count()
```

```
Count of missing DEP_DEL15    84710
dtype: int64
Count of total data:  2806942
Out[113]: 0.030178749685600914
```

Nulls in Flight Data

```
#View how many nulls are in each column for feature selection

display(df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in
df.columns]))
```

	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK
1	0	0	0	0	0

Showing all 1 rows.

```
#
print('Number of Missing in DEP_DELAY column:',
df.filter(F.col('DEP_DELAY').isNull()).count())
print('Number of Missing in DEP_DELAY when NOT cancelled',
df.filter((F.col('DEP_DELAY').isNull()) & (F.col('CANCELLED') != 1)).count())

print('Number of Missing in ARR_DELAY column:',
df.filter(F.col('ARR_DELAY').isNull()).count())
print('Number of Missing in ARR_DELAY when NOT cancelled',
df.filter((F.col('ARR_DELAY').isNull()) & (F.col('CANCELLED') != 1)).count())
print('Number of Missing in ARR_DELAY when NOT cancelled or diverted',
df.filter((F.col('ARR_DELAY').isNull()) & (F.col('CANCELLED') != 1) &
(F.col('DIVERTED') != 1)).count())

print('Number of Missing in ACTUAL_ELAPSED_TIME column:',
df.filter(F.col('ACTUAL_ELAPSED_TIME').isNull()).count())
print('Number of Missing in ACTUAL_ELAPSED_TIME when NOT cancelled',
df.filter((F.col('ACTUAL_ELAPSED_TIME').isNull()) & (F.col('CANCELLED') !=
1)).count())
print('Number of Missing in ACTUAL_ELAPSED_TIME when NOT diverted',
df.filter((F.col('ACTUAL_ELAPSED_TIME').isNull()) &
(F.col('DIVERTED')!=1)).count())
print('Number of Missing in ACTUAL_ELAPSED_TIME when NOT diverted or
cancelled', df.filter((F.col('ACTUAL_ELAPSED_TIME').isNull()) &
(F.col('CANCELLED')!=1) & (F.col('DIVERTED')!=1)).count())

print('Number of Missing in FIRST_DEP_TIME column:',
df.filter(F.col('TAXI_IN').isNull()).count())
print('Number of Missing in FIRST_DEP_TIME when NOT cancelled',
df.filter((F.col('TAXI_IN').isNull()) & (F.col('CANCELLED') != 1)).count())

print('Number of Missing in TOTAL_ADD_GTIME column:',
df.filter(F.col('TOTAL_ADD_GTIME').isNull()).count())
print('Number of Missing in TOTAL_ADD_GTIME when NOT cancelled',
df.filter((F.col('TOTAL_ADD_GTIME').isNull()) & (F.col('CANCELLED') !=
1)).count())

print('Number of Missing in TAIL_NUM column:',
df.filter(F.col('TAIL_NUM').isNull()).count())
print('Number of Missing in TAIL_NUM when NOT cancelled',
df.filter((F.col('TAIL_NUM').isNull()) & (F.col('CANCELLED') != 1)).count())
```

```
Number of Missing in DEP_DELAY column: 84710
Number of Missing in DEP_DELAY when NOT cancelled 0
Number of Missing in ARR_DELAY column: 93314
Number of Missing in ARR_DELAY when NOT cancelled 6312
Number of Missing in ARR_DELAY when NOT cancelled or diverted 0
Number of Missing in ACTUAL_ELAPSED_TIME column: 93314
Number of Missing in ACTUAL_ELAPSED_TIME when NOT cancelled 6312
Number of Missing in ACTUAL_ELAPSED_TIME when NOT diverted 87002
Number of Missing in ACTUAL_ELAPSED_TIME when NOT diverted or cancelled 0
Number of Missing in FIRST_DEP_TIME column: 88736
Number of Missing in FIRST_DEP_TIME when NOT cancelled 1734
Number of Missing in TOTAL_ADD_GTIME column: 2788958
Number of Missing in TOTAL_ADD_GTIME when NOT cancelled 2703596
Number of Missing in TAIL_NUM column: 16380
Number of Missing in TAIL_NUM when NOT cancelled 0
```

Feature	**Notes about Missing**	**Accounting for Missing**
DEP_DELAY15	All missing values are due to cancelled flights	Consider cancels as extended delay: recode nulls as 1
ARR_DELAY	missing when cancelled or diverted	* We need this column for information about prior flight delay, currently we are undecided on whether to remove missing or fill
AIR_TIME	Null for diverted and cancelled flights	won't use this column because highly correlated with distance
ACTUAL_ELAPSED_TIME	Null for diverted and cancelled flights	won't use this column because highly correlated with distance
FIRST_DEP_TIME	mostly missing	won't use this column
TOTAL_ADD_GTIME	mostly missing	won't use this column
LONGEST_ADD_GTIME	mostly missing	won't use this column
DIV_REACHED_DEST	mostly missing	won't use this column
DIV_ACTUAL_ELAPSED_TIME	mostly missing	won't use this column
DIV_DISTANCE	mostly missing	won't use this column

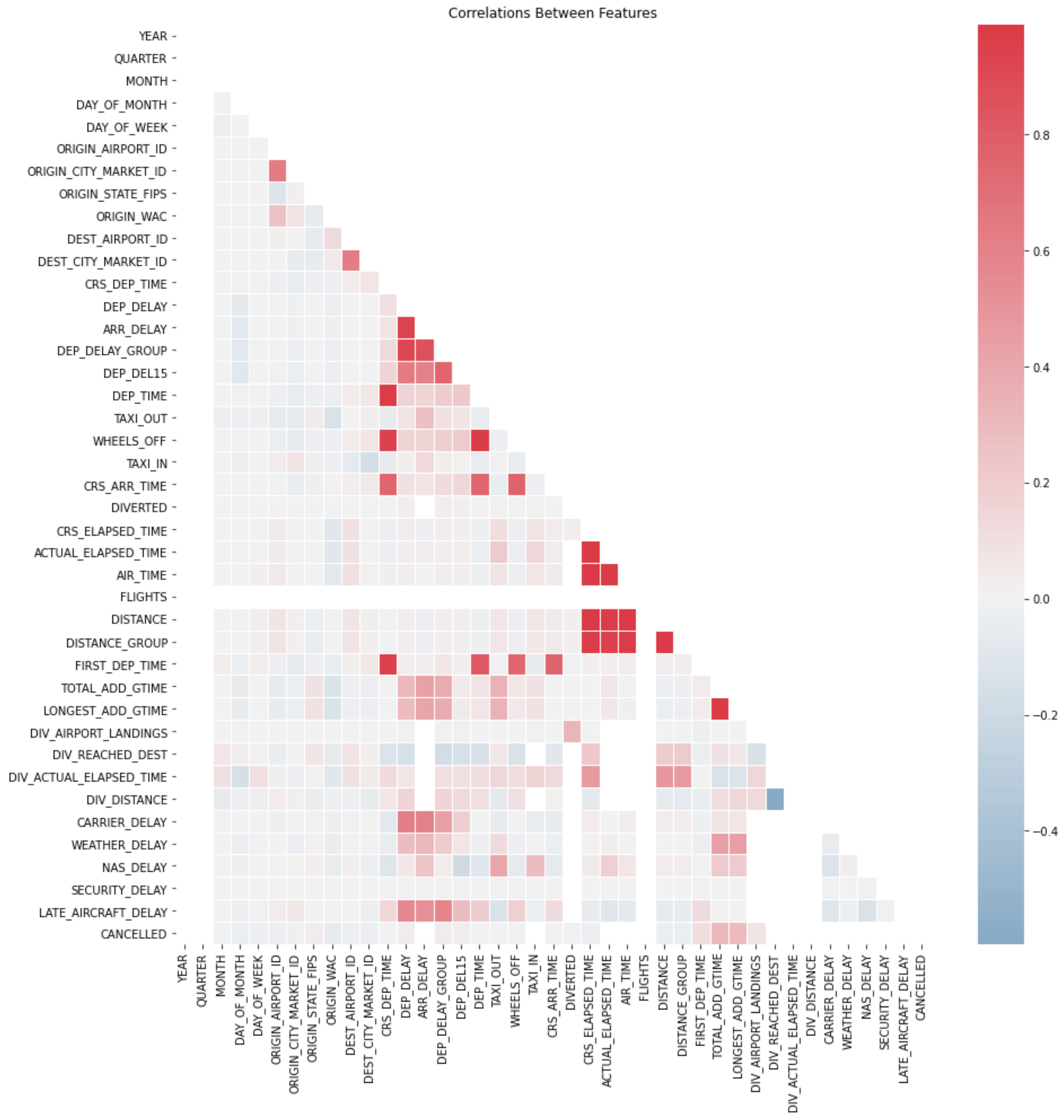
Feature	Notes about Missing	Accounting for Missing
TAIL_NUM	Only missing when flights cancelled	* We need this column for information about prior flight delay, currently we are undecided on whether to remove missing or fill
CARRIER_DELAY	this column, as well as weather, nas, security, late aircraft delays are missing	won't use this column

Correlations Between Features

Correlations Between Features

```
corr = df.toPandas().corr()
fig, ax = plt.subplots(figsize=(15, 15))
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(240, 10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=.5)
plt.title("Correlations Between Features")
plt.show()
```

```
<command-1083953104575997>:3: DeprecationWarning: `np.bool` is a deprecated ali
as for the builtin `bool`. To silence this warning, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the n
umpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devd
ocs/release/1.20.0-notes.html#deprecations
  mask = np.zeros_like(corr, dtype=np.bool)
```


Correlations Between Features
YEAR
QUARTER
MONTH
DAY_OF_MONTH
DAY_OF_WEEK
ORIGIN_AIRPORT_ID
ORIGIN_CITY_MARKET_ID
ORIGIN_STATE_FIPS
ORIGIN_WAC
DEST_AIRPORT_ID
DEST_CITY_MARKET_ID
CRS_DEP_TIME
DEP_DELAY
ARR_DELAY
DEP_DELAY_GROUP
DEP_DEL15
DEP_TIME
TAXI_OUT
WHEELS_OFF
TAXI_IN
CRS_ARR_TIME
DIVERTED
CRS_ELAPSED_TIME
ACTUAL_ELAPSED_TIME
AIR_TIME
FLIGHTS
DISTANCE
DISTANCE_GROUP
FIRST_DEP_TIME
TOTAL_ADD_GTIME
LONGEST_ADD_GTIME
DIV_AIRPORT_LANDINGS
DIV_REACHED_DEST
DIV_ACTUAL_ELAPSED_TIME
DIV_DISTANCE
CARRIER_DELAY
WEATHER_DELAY
NAS_DELAY
SECURITY_DELAY
LATE_AIRCRAFT_DELAY
CANCELLED
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

```
#Narrowing down features to a select group
new_keep_columns =  ['DEP_DEL15', 'YEAR','QUARTER','MONTH', 'DAY_OF_WEEK',
'FL_DATE', 'OP_CARRIER', 'ORIGIN', 'DEST', 'CRS_DEP_TIME', 'ARR_DELAY',
'DIVERTED', 'DISTANCE', 'TAIL_NUM', 'CANCELLED']

df = df.select('DEP_DEL15', 'YEAR', 'MONTH','QUARTER', 'DAY_OF_WEEK',
'FL_DATE', 'OP_CARRIER', 'ORIGIN', 'DEST', 'CRS_DEP_TIME', 'ARR_DELAY',
'DIVERTED', 'DISTANCE', 'TAIL_NUM', 'CANCELLED')

#Summary info
for col in new_keep_columns:
  print(df.select(col).describe().show())
```

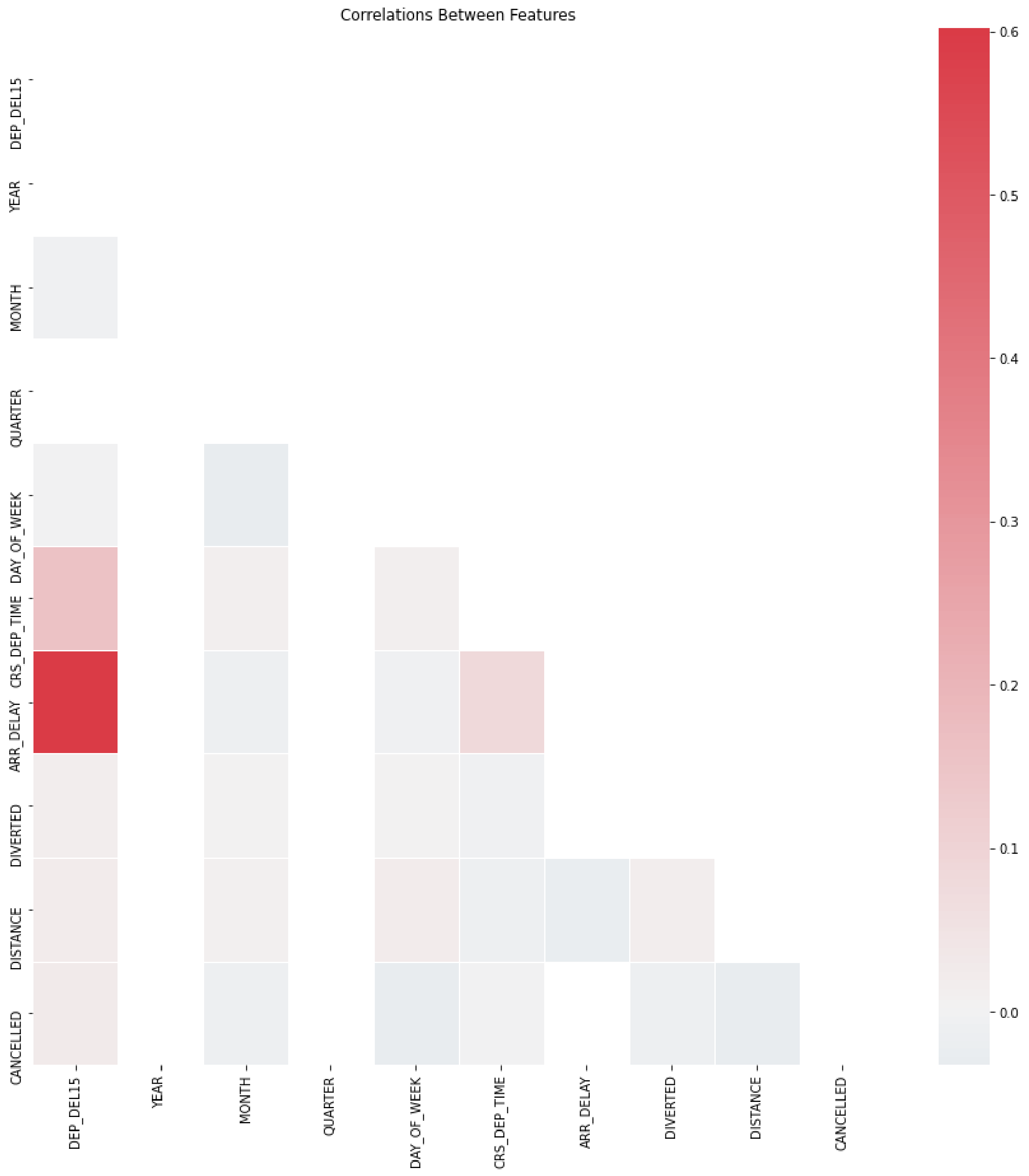
```
+-------+-------------------+
|summary|          DEP_DEL15|
+-------+-------------------+
|  count|            2722232|
|   mean|0.20413469535293097|
| stddev| 0.4030679610021786|
|    min|                0.0|
|    max|                1.0|
+-------+-------------------+

None
+-------+-------+
|summary|   YEAR|
+-------+-------+
|  count|2806942|
|   mean| 2015.0|
| stddev|    0.0|
|    min|   2015|
|    max|   2015|
+-------+-------+
```

Correlations Between Selected Features

Correlations Between Selected Features

```
corr = df.toPandas().corr()
fig, ax = plt.subplots(figsize=(15, 15))
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(240, 10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=.5)
plt.title("Correlations Between Features")
plt.show()
```

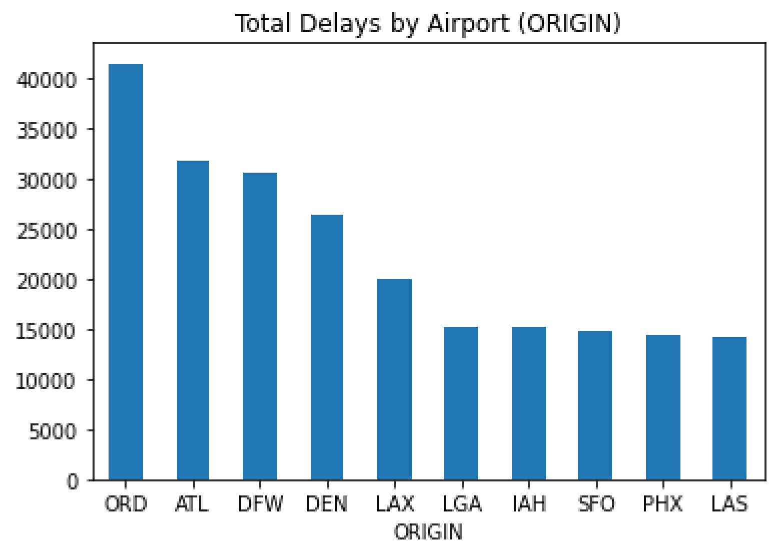
```
<command-4236430293314274>:3: DeprecationWarning: `np.bool` is a deprecated ali
as for the builtin `bool`. To silence this warning, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the n
umpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devd
ocs/release/1.20.0-notes.html#deprecations
  mask = np.zeros_like(corr, dtype=np.bool)
```



Delays by Airport

```
#Delays by Airport Origin
df_airlines.createOrReplaceTempView("airport")
df_airport_delay = spark.sql("select ORIGIN, count(*) as Delays from airport
where DEP_DEL15 = 1 group by ORIGIN order by Delays desc  LIMIT 10")
df_airport_delay.toPandas().plot.bar(title = "Total Delays by Airport
(ORIGIN)", y='Delays',x='ORIGIN', legend = False, rot = 0)
```

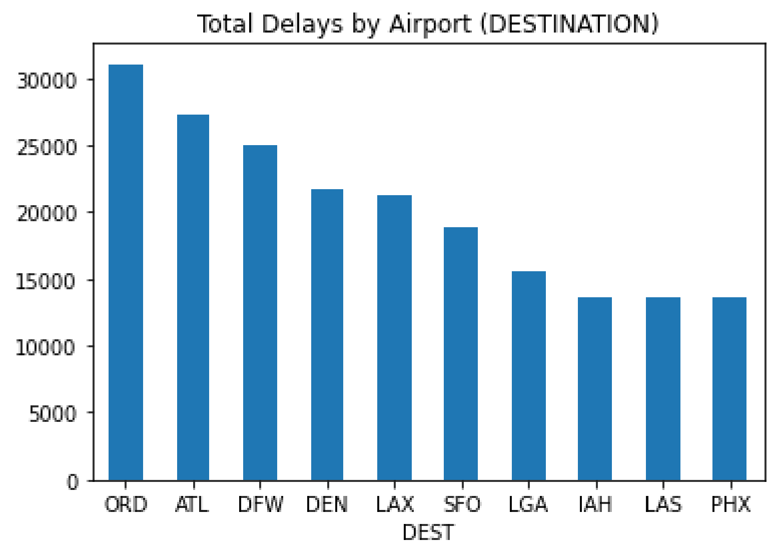
```
Out[119]:
```



```
<AxesSubplot:title={'center':'Total Delays by Airport (ORIGIN)'}, xlabel='ORIGI
N'>
```

```
#Delays by Airport DEST
df_airlines.createOrReplaceTempView("airport")
df_airport_delay = spark.sql("select DEST, count(*) as Delays from airport
where DEP_DEL15 = 1 group by DEST order by Delays desc  LIMIT 10")
df_airport_delay.toPandas().plot.bar(title = "Total Delays by Airport
(DESTINATION)", y='Delays',x='DEST', legend = False, rot = 0)
```

```
Out[120]:
```

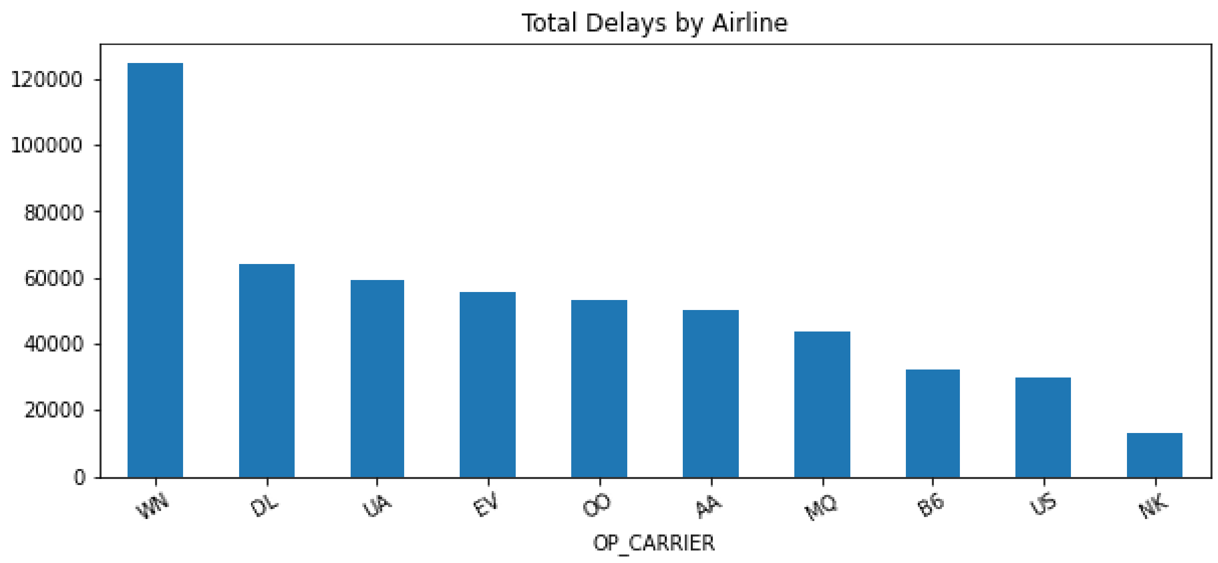


```
<AxesSubplot:title={'center':'Total Delays by Airport (DESTINATION)'}, xlabel
='DEST'>
```

Delays by Airline

```
# Distribution of Delayed by airline
df.createOrReplaceTempView("airline")
df_delay = spark.sql("select OP_CARRIER, count(*) as Delays from airline where
DEP_DEL15 = 1 group by OP_CARRIER order by Delays desc  LIMIT 10")
df_delay.toPandas().plot.bar(title = "Total Delays by Airline",
y='Delays',x='OP_CARRIER', legend = False, figsize = (10,4), rot = 30)
```

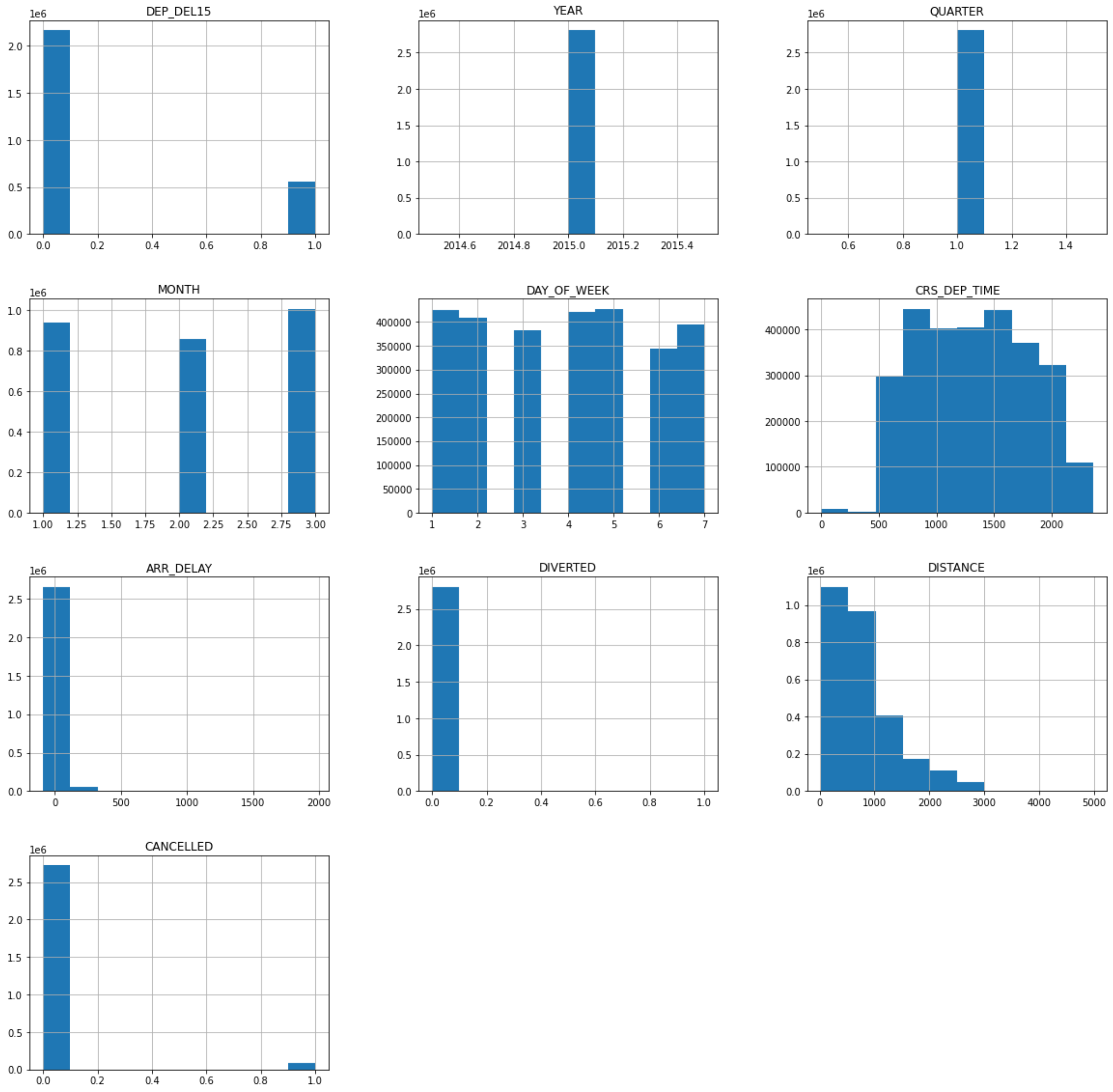
```
Out[121]:
```



```
<AxesSubplot:title={'center':'Total Delays by Airline'}, xlabel='OP_CARRIER'>
```

Flight Data Feature Distribution

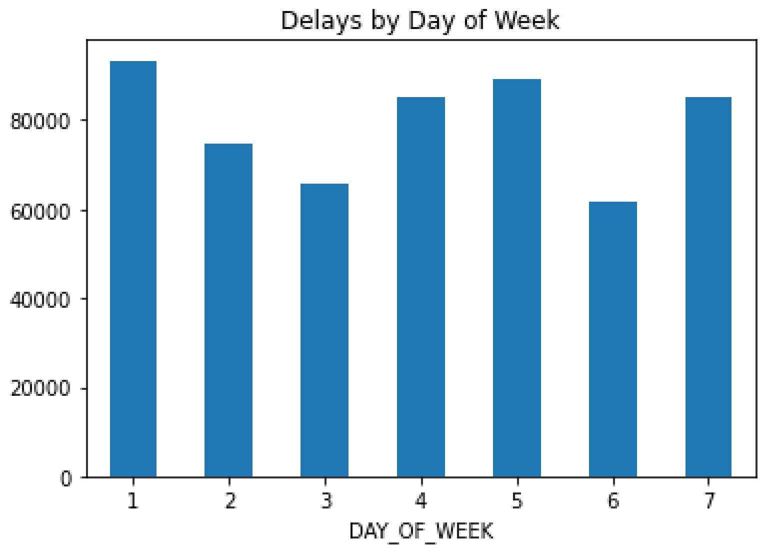
```
df.toPandas()[new_keep_columns].hist(figsize=(20,20), bins=10)
plt.show()
```



Delays by Day of Week

```
#Delays by Day of Week ORD
df.createOrReplaceTempView("airline")
df_day_week = spark.sql("select DAY_OF_WEEK, count(*) as day_count from airline
where DEP_DEL15 == 1 group by DAY_OF_WEEK order by DAY_OF_WEEK")
df_day_week.toPandas().plot(y='day_count',x='DAY_OF_WEEK',kind = 'bar', title =
'Delays by Day of Week', rot = 0, legend = False)
```

Out[123]:



```
<AxesSubplot:title={'center':'Delays by Day of Week'}, xlabel='DAY_OF_WEEK'>
```

```
df.filter((F.col('DIV_REACHED_DEST').isNull()) &
(F.col('DIVERTED')!=1)).toPandas().head(10)
```

Out[124]:

	DEP_DEL15	YEAR	MONTH	QUARTER	DAY_OF_WEEK	FL_DATE	OP_CARRIER	ORIGIN	DE
0	1	2015	2	1	4	2015-02-19	AA	STL	DF
1	0	2015	2	1	5	2015-02-20	AA	STL	DF
2	1	2015	2	1	6	2015-02-21	AA	STL	DF
3	0	2015	2	1	7	2015-02-22	AA	STL	DF
4	NaN	2015	2	1	1	2015-02-23	AA	STL	DF
5	0	2015	2	1	2	2015-02-24	AA	STL	DF
6	0	2015	2	1	3	2015-02-25	AA	STL	DF

50% of the airline data is duplicated, and should be

```
#Duplicates
df_airlines.createOrReplaceTempView("airline")
print(df.count()) #2806942
df_nodup=spark.sql('select distinct * from airline') #1403471
print(df_nodup.count())

print('% duplicates', df_nodup.count() / df.count())
```

```
2806942
1403471
% duplicates 0.5
```

Weather

Summary Information

```
# Load the 2015 Q1 for Weather
df_weather = spark.read.parquet(f"
{data_BASE_DIR}parquet_weather_data_3m/").filter(F.col('DATE') < "2015-04-
01T00:00:00.000")
display(df_weather)
```

1	52652099999	2015-01-01T02:00:00	39.0833333	100.2833333	1462.0
2	52652099999	2015-01-01T05:00:00	39.0833333	100.2833333	1462.0
3	52652099999	2015-01-01T08:00:00	39.0833333	100.2833333	1462.0
4	52652099999	2015-01-01T11:00:00	39.0833333	100.2833333	1462.0
5	52652099999	2015-01-01T14:00:00	39.0833333	100.2833333	1462.0
6	52652099999	2015-01-01T17:00:00	39.0833333	100.2833333	1462.0

Truncated results, showing first 1000 rows.

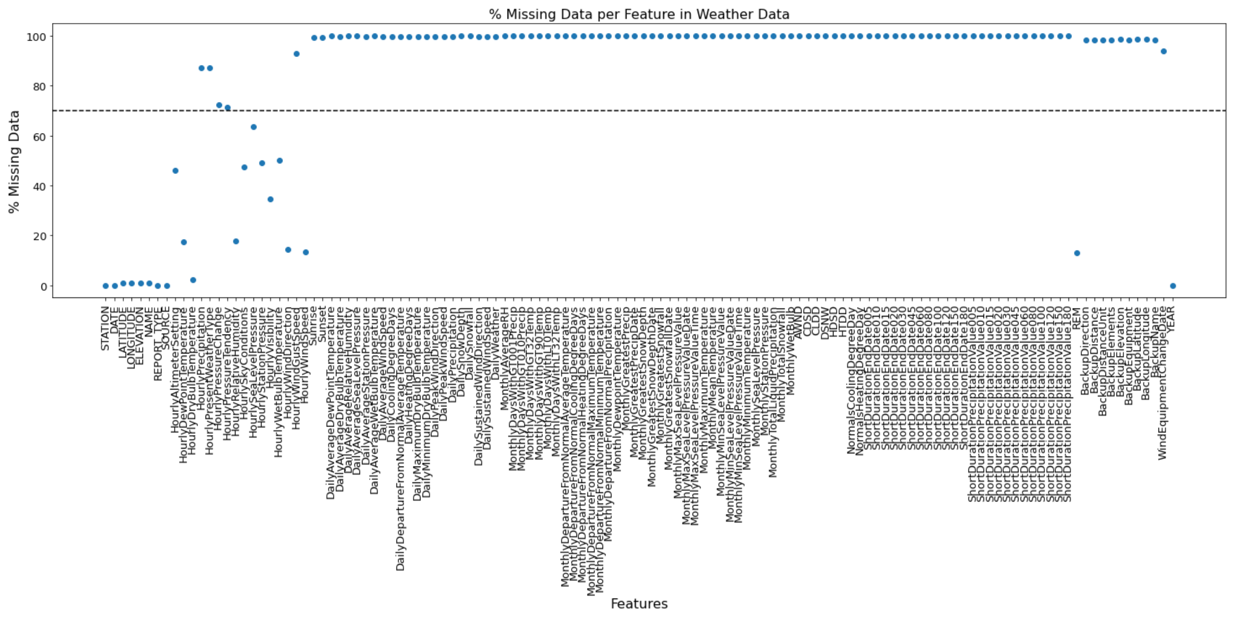
```
pct_missing = df_weather.select([(F.count(F.when(F.col(c).contains("NULL") |
F.col(c).isNull() | (F.col(c) == ""), c)) / F.count(F.lit(1))).alias(c) for c
in df_weather.columns])
display(pct_missing)
```

	STATION	DATE	LATITUDE	LONGITUDE	E
1	0	0	0.007899280812138074	0.007899280812138074	0.

Showing all 1 rows.

Missing Data

```
pct_missing_PD = pct_missing.toPandas().iloc[0]
plt.figure(figsize=(25, 6))
plt.scatter(pct_missing_PD.index, pct_missing_PD.values * 100)
plt.axhline(y = 70, color = "black", linestyle = "--")
plt.title("% Missing Data per Feature in Weather Data", fontsize = 16)
plt.xlabel("Features", fontsize = 16)
plt.ylabel("% Missing Data", fontsize = 16)
plt.xticks(rotation = 90, fontsize = 13)
plt.yticks(fontsize = 13)
plt.show()
```



```
# drop columns that have > 70% missing values
WEATHER_FEATURES = pd.Series(pct_missing_PD).where(lambda x: x <
0.7).dropna().index.tolist()
print(WEATHER_FEATURES)
```

```
['STATION', 'DATE', 'LATITUDE', 'LONGITUDE', 'ELEVATION', 'NAME', 'REPORT_TYP
E', 'SOURCE', 'HourlyAltimeterSetting', 'HourlyDewPointTemperature', 'HourlyDry
BulbTemperature', 'HourlyRelativeHumidity', 'HourlySkyConditions', 'HourlySeaLe
velPressure', 'HourlyStationPressure', 'HourlyVisibility', 'HourlyWetBulbTemper
ature', 'HourlyWindDirection', 'HourlyWindSpeed', 'REM', 'YEAR']
```

- According to the Federal Aviation Administration, inclement weather, including thunderstorms, snowstorms, wind shear, icing and fog, creates potentially hazardous conditions in the nation's airspace system. These conditions are, by far, the largest cause of flight delays. In an average year, inclement weather is the reason for nearly 70 percent of all delays.
- We analyzed the features in Weather data to determine whether they can be used as an indicator for inclement weather.

Feature	**Notes**	**Keep or Drop**
ELEVATION	Elevation of weather station	Drop

Feature	Notes	Keep or Drop
SOURCE	No dictionary to decode	Drop
HourlyAltimeterSetting	Indication of altitude	Drop
HourlyDewPointTemperature	Dew point temp -> amount of moisture in the air -> amount precipitation	Keep
HourlyDryBulbTemperature	Air temp; indication of heat	Drop
HourlyRelativeHumidity	Measure of the water vapor content of air	Drop
HourlySkyConditions	Need to decode to keep only the sky conditions codes	Keep
HourlySeaLevelPressure	For meteorology not flying	Drop
HourlyStationPressure	Pressure at the weather station elevation	Drop
HourlyVisibility	Measure of the horizontal opacity of the atmosphere at the point of observation; low -> fog	Keep
HourlyWetBulbTemperature	Indication of relative humidity	Drop
HourlyWindDirection	Will not consider wind direction to simplify the analysis	Drop
HourlyWindSpeed	Strong wind prevent flight taking off	Keep
REM	Remarks - cannot decode without dictionary	Drop
YEAR	Can extract from DATE if needed	Drop

```
# list of features to be removed after researching their relevance
to_remove = ["ELEVATION",
             "SOURCE",
             "REM",
             "HourlyWindDirection",
             "HourlyAltimeterSetting",
             "HourlyStationPressure",
             "HourlyWindDirection",
             "HourlyDryBulbTemperature",
             "HourlyWetBulbTemperature",
             "HourlyRelativeHumidity",
             "HourlySeaLevelPressure",
             "YEAR"]

WEATHER_FEATURES = [feature for feature in WEATHER_FEATURES if feature not in
to_remove]
```

```
# Subset weather data with only columns has < 70% missing data
df_weather_subset = df_weather.select(WEATHER_FEATURES)

# filter for US only
df_weather_subset_US = df_weather_subset.filter(F.col("NAME").endswith("US"))
display(df_weather_subset_US)
```

	STATION	DATE	LATITUDE	LONGITUDE	NAME
1	99418099999	2015-01-01T00:00:00	48.32	-122.84	SMITH I
2	99418099999	2015-01-01T01:00:00	48.32	-122.84	SMITH I
3	99418099999	2015-01-01T02:00:00	48.32	-122.84	SMITH I
4	99418099999	2015-01-01T03:00:00	48.32	-122.84	SMITH I
5	99418099999	2015-01-01T04:00:00	48.32	-122.84	SMITH I
6	99418099999	2015-01-01T05:00:00	48.32	-122.84	SMITH I

Truncated results, showing first 1000 rows.

```
# Check missing data
pct_missing_US =
df_weather_subset.select([(F.count(F.when(F.col(c).contains("null") |
F.col(c).isNull() | (F.col(c) == ""), c)) / F.count(F.lit(1))).alias(c) for c
in df_weather_subset.columns])
display(pct_missing_US)
```

	STATION	DATE	LATITUDE	LONGITUDE	N

1	0	0	0.007899280812138074	0.007899280812138074	0.

Showing all 1 rows.

Decode HourlySkyConditions to get sky cover code

- OVC = overcast weather causes flight delays
- Parse the string, and keep the last sky cover code
- If OVC, mark OVC; else, mark not OVC
- One-hot encode to 1 = OVC and 0 = not OVC

```
def sky_decode(df):
  sky = F.split(df["HourlySkyConditions"], ":")
  df = (df.withColumn("SkyConditionCode", F.element_at(sky,
-2))).drop("HourlySkyConditions")
  sky_code = F.split(df["SkyConditionCode"], " ")
  df = df.withColumn("SkyConditionCode", sky_code.getItem(2).cast("string"))\
         .withColumn("Overcast", F.when(F.col("SkyConditionCode").like("OVC"),
F.lit("OVC")).otherwise("not OVC"))\
         .drop("SkyConditionCode")
  ovc_indexer = StringIndexer(inputCol = "Overcast", outputCol =
"OvercastIndex")
  df = ovc_indexer.fit(df).transform(df).drop("Overcast")
  return df
```

```
df_weather_subset = sky_decode(df_weather_subset)
display(df_weather_subset)
```

	STATION	DATE	LATITUDE	LONGITUDE	NAME
1	52652099999	2015-01-01T02:00:00	39.0833333	100.2833333	ZHAN
2	52652099999	2015-01-01T05:00:00	39.0833333	100.2833333	ZHAN
3	52652099999	2015-01-01T08:00:00	39.0833333	100.2833333	ZHAN
4	52652099999	2015-01-01T11:00:00	39.0833333	100.2833333	ZHAN
5	52652099999	2015-01-01T14:00:00	39.0833333	100.2833333	ZHAN
6	52652099999	2015-01-01T17:00:00	39.0833333	100.2833333	ZHAN

Truncated results, showing first 1000 rows.

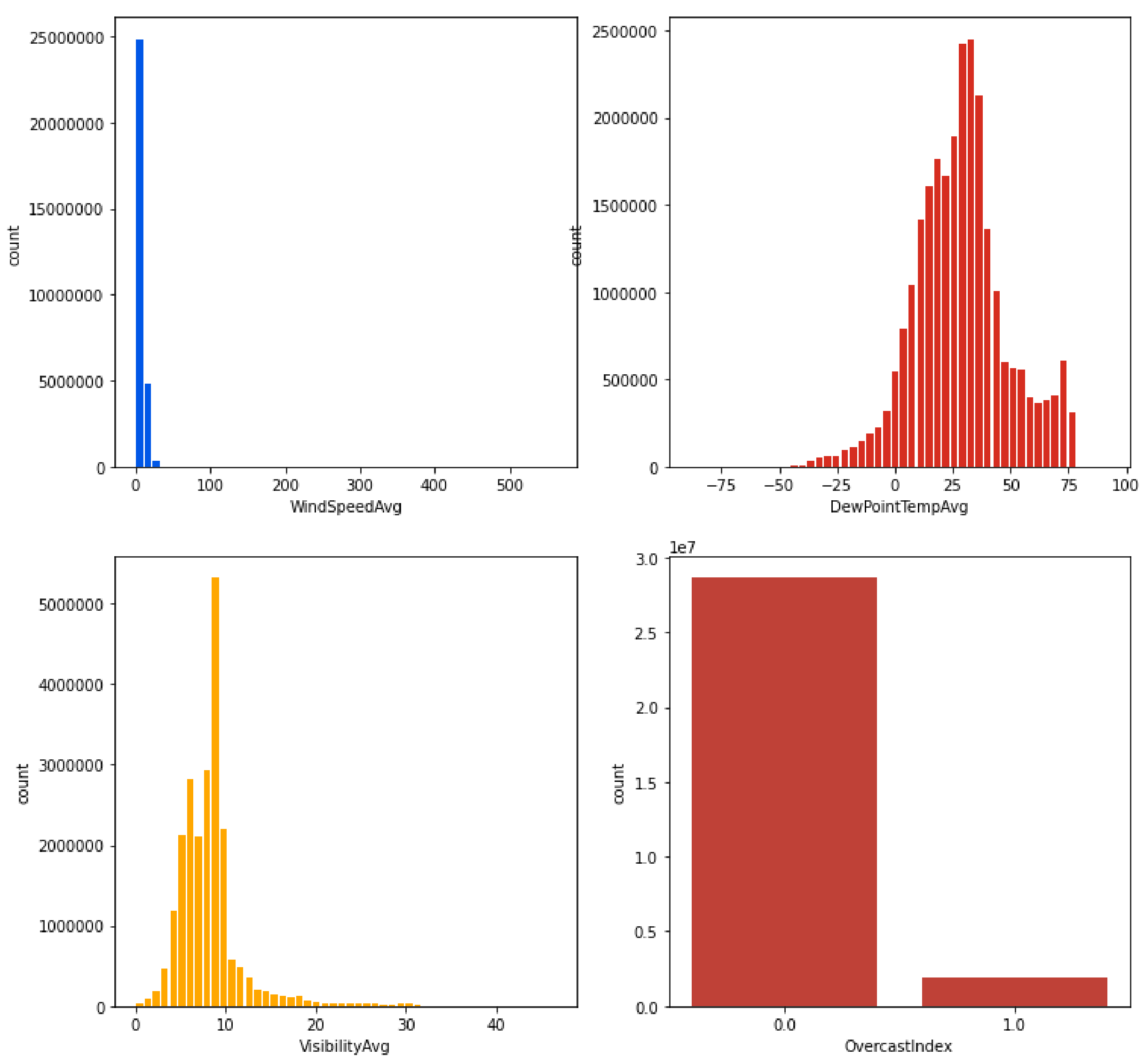
```
# cast record to correct data type

float_columns = ["LATITUDE", "LONGITUDE", "HourlyDewPointTemperature",
"HourlyVisibility", "HourlyWindSpeed"]

for col in float_columns:
  df_weather_subset = df_weather_subset.withColumn(col,
F.col(col).cast("float"))



# average "HourlyDewPointTemperature", "HourlyVisibility", "HourlyWindSpeed" in
a 2hr window prior to indicated timestamp per station
w =
(Window.partitionBy(F.col("STATION")).orderBy(F.col("DATE").cast("long")).rowsB
etween(-2*60*60, 0))

df_weather_agg = df_weather_subset.withColumn("DATE",
F.from_unixtime(F.unix_timestamp("DATE", "yyyy-MM-dd'T'HH:mm:ss")))\
                                  .withColumn("WindSpeedAvg",
F.avg(F.col("HourlyWindSpeed")).over(w))\
                                  .withColumn("DewPointTempAvg",
F.avg(F.col("HourlyDewPointTemperature")).over(w))\
                                  .withColumn("VisibilityAvg",
F.avg(F.col("HourlyVisibility")).over(w))\
                                  .drop("HourlyWindSpeed",
"HourlyDewPointTemperature", "HourlyVisibility")
display(df_weather_agg)
```

	STATION	DATE	LATITUDE	LONGITUDE	NAME
1	01002099999	2015-01-01 04:00:00	80.05	16.25	VERLEG
2	01002099999	2015-01-01 07:00:00	80.05	16.25	VERLEG
3	01002099999	2015-01-01 10:00:00	80.05	16.25	VERLEG
4	01002099999	2015-01-01 13:00:00	80.05	16.25	VERLEG
5	01002099999	2015-01-01 16:00:00	80.05	16.25	VERLEG
6	01002099999	2015-01-01 19:00:00	80.05	16.25	VERLEG

Truncated results, showing first 1000 rows.

```
df_weather_agg_PD = df_weather_agg.toPandas()
```

```
fig, ax = plt.subplots(2, 2, figsize=(12, 12))

ax[0][0].hist(data = df_weather_agg_PD, x = "WindSpeedAvg", color = "#0057e7",
bins = 50, rwidth = 0.8)
ax[0][0].set_xlabel("WindSpeedAvg")
ax[0][0].set_ylabel("count")
ax[0][0].ticklabel_format(style="plain")
ax[0][1].hist(data = df_weather_agg_PD, x = "DewPointTempAvg", color =
"#d62d20", bins = 50, rwidth = 0.8)
ax[0][1].set_xlabel("DewPointTempAvg")
ax[0][1].set_ylabel("count")
ax[0][1].ticklabel_format(style="plain")
ax[1][0].hist(data = df_weather_agg_PD, x = "VisibilityAvg", color = "#ffa700",
bins = 50, rwidth = 0.8)
ax[1][0].set_xlabel("VisibilityAvg")
ax[1][0].set_ylabel("count")
ax[1][0].ticklabel_format(style="plain")
sns.countplot(data = df_weather_agg_PD, x = "OvercastIndex", color = "#d62d20")

plt.show()
```



```
corr = df_weather_agg_PD.iloc[:, [7, 8, 9]].corr()
corr
```

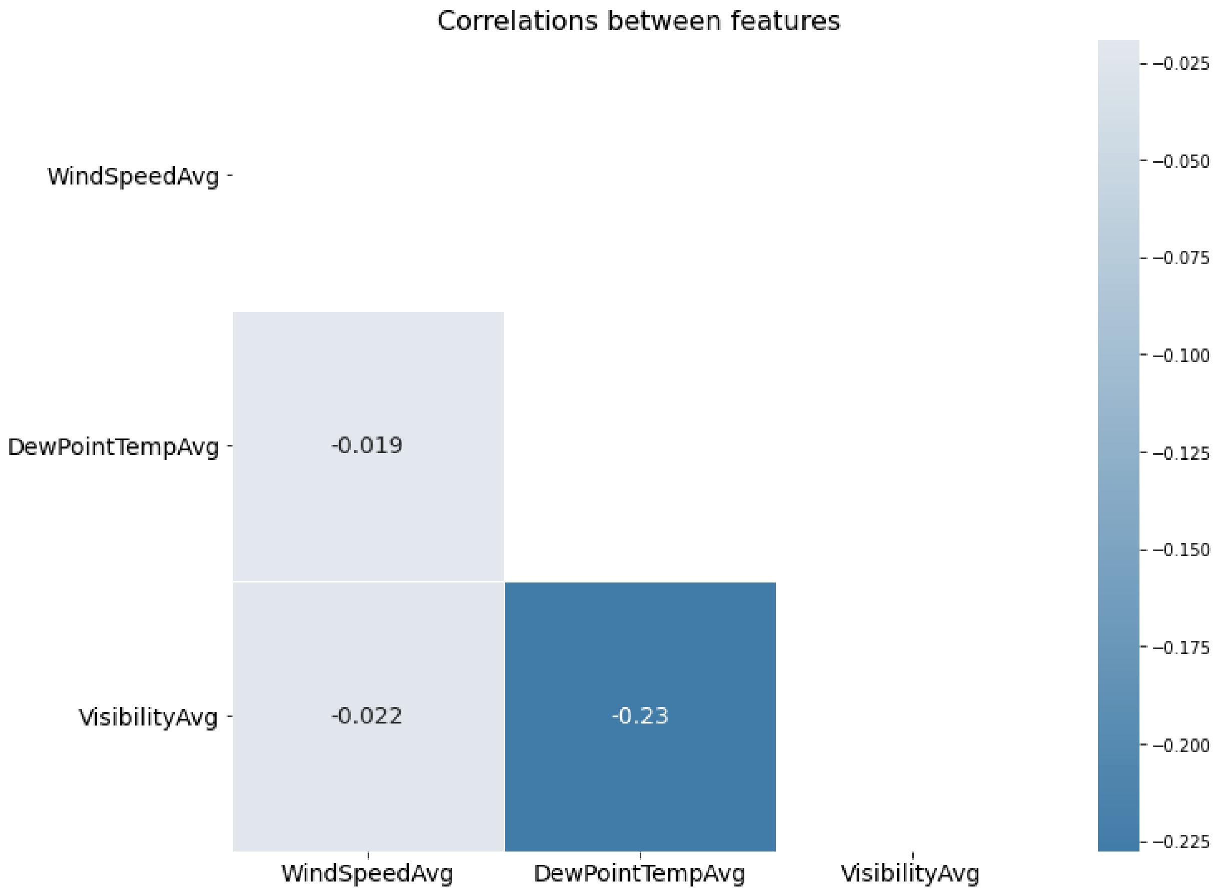
```
Out[139]:
```

	WindSpeedAvg	DewPointTempAvg	VisibilityAvg
WindSpeedAvg	1	-0	-0
DewPointTempAvg	-0	1	-0
VisibilityAvg	-0	-0	1

Correlations Between Features

```
fig, ax = plt.subplots(figsize=(11, 9))
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(240, 10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=.5, annot = True,
annot_kws = {"size": 14})
ax.set_xticklabels(ax.get_xmajorticklabels(), fontsize = 14)
ax.set_yticklabels(ax.get_ymajorticklabels(), fontsize = 14, rotation = 0)
plt.title("Correlations between features", fontsize = 16)
plt.show()
```

```
<command-1083953104576188>:2: DeprecationWarning: `np.bool` is a deprecated ali
as for the builtin `bool`. To silence this warning, use `bool` by itself. Doing
this will not modify any behavior and is safe. If you specifically wanted the n
umpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devd
ocs/release/1.20.0-notes.html#deprecations
  mask = np.zeros_like(corr, dtype=np.bool)
```



Station EDA

Summary Information

1. Stations have self join dataset of station and it neighboring stations along with distance between them
2. Stations with distance_to_neighbor=0 represents the self reported distance of station with itself
3. Stations with distance_to_neighbor=0 are at airport and would be nearest station to airport hence for joining airline with weather we are considering only stations at airport. Evidence that these stations are at airport we can see airport codes.

Stations Observations:

```
print("missing data per column in stations :")
df_sColumns= df_stations
x2=df_sColumns.select([count(when(isnan(c) | F.col(c).isNull(), c)).alias(c)
for c in df_sColumns.columns])
display(x2)
# columns that no null or NAN column
selectList2=list()
for columnName in x2.columns:
  if x2.select(columnName).where(F.col(columnName)==0).count()==1:
    selectList2.append(columnName)
print("List of columns which have non zero column :")
print(selectList2)

print('***********************************************************OBSERVATIONS*
*******************************************************')
print('Repeated neighbors with distance_to_neighbor=0 ::')
df_stations.createOrReplaceTempView("df_stations")
df_stations1=spark.sql('select neighbor_id,neighbor_call,count(*) from
df_stations where int(distance_to_neighbor)=0 group by
neighbor_id,neighbor_call having count(*)>1')
display(df_stations1)

#creating base stations list with min dist=0
df_base_stations=spark.sql('select
station_id,neighbor_call,distance_to_neighbor from df_stations where
int(distance_to_neighbor)=0')
df_base_stations.createOrReplaceTempView("base_stations")

print('*****************WEATHER STATIONS AT AIRPORT WHICH ARE REPEATED IN
STATIONS DATASET ***********************************************')
display(df_stations[(F.col('neighbor_call')=='KLOZ') &
(F.col('distance_to_neighbor')==0) & (F.col('neighbor_id')=='72329003849')])
print('*****************CONCLUSION : Above example illustrates that station_id
72329003849,72424303849 is located at same place*********** ')
print('*****************             Also we dont know how these station_id
72329003849,72424303849 is located at same place hence we are not filtering
them out.\n \
                                    We filter only fixed stations later in
weather dataset based on report-type (FM-12)*********** ')

print('******************proof that all distance 0 neighbors are airport
stations : no output from this query****************************')
display(spark.sql("select * from base_stations bs join airport_codes ac on
neighbor_call=ident where type not like '%airport%'"))
```

```
df_base_stations=spark.sql('select
station_id,neighbor_call,distance_to_neighbor from df_stations where
int(distance_to_neighbor)=0')
display(df_base_stations)
```

```
missing data per column in stations :
```

	usaf	wban	station_id	lat	lon	neighbor_
1	0	0	0	0	0	0

Showing all 1 rows.

```
List of columns which have non zero column :
['usaf', 'wban', 'station_id', 'lat', 'lon', 'neighbor_id', 'neighbor_name', 'n
eighbor_state', 'neighbor_call', 'neighbor_lat', 'neighbor_lon', 'distance_to_n
eighbor']
***********************************************************OBSERVATIONS********
**************************************************
Repeated neighbors with distance_to_neighbor=0 ::
```

	neighbor_id	neighbor_call	count(1)
1	72329003849	KLOZ	2
2	72512114761	KPSB	2
3	70121326638	PIZ	2
4	74694113786	KECG	2
5	A0001163848	K47A	2
6	70121026624	PPIZ	2

Showing all 42 rows.

```
*****************WEATHER STATIONS AT AIRPORT WHICH ARE REPEATED IN STATIONS DAT
ASET ***********************************************
```

	usaf	wban	station_id	lat	lon	neighb
1	723290	03849	72329003849	37.087	-84.077	72329(
2	724243	03849	72424303849	37.087	-84.077	72329(

Showing all 2 rows.

```
*****************CONCLUSION : Above example illustrates that station_id 7232900
3849,72424303849 is located at same place***********
*****************            Also we dont know how these station_id 7232900384
9,72424303849 is located at same place hence we are not filtering them out.
                                     We filter only fixed stations later in wea
ther dataset based on report-type (FM-12)***********
*****************proof that all distance 0 neighbors are airport stations : no
output from this query*****************************
```

	station_id	neighbor_call	distance_to_neighbor	ident	type
1	69014093101	KNZJ	0	KNZJ	closed
2	70231226555	PAFK	0	PAFK	seapla
3	70269596402	PAJV	0	PAJV	closed
4	70381725357	PAEL	0	PAEL	seapla
5	70388625348	PAAP	0	PAAP	seapla
6	72224503882	KPFN	0	KPFN	closed

Showing all 27 rows.

	station_id	neighbor_call	distance_to_neighbor
1	69002093218	KHGT	0
2	69007093217	KOAR	0
3	69014093101	KNZJ	0
4	70027127506	KPBA	0
5	70045027512	LNI	0
6	70063027403	POLI	0

Truncated results, showing first 1000 rows.

Join

Summary on Joining Dataset :

- All datasets are cleaned. Columns are removed if the column has 70% of NaNs, Nulls or invalid categorical information.
- For each quarter in a year
    - Filter by year and quarter
    - Processing the data set quarter by quarter per year
    - Left join airline data with base stations on icao to neighbor call, left join weather on base station and year
    - Filter by Flight date >= weather date
    - Selected max weather row in the 2-5 hour time range before the flight departure date.

Data is partitioned based on year and quarter

Note: Airports have IATA codes. For example, Chicago O'Hare International Airport has IATA code as ORD and ICAO code as KORD. ICAO code is used by weather stations. IATA code might not be the same as ICAO for some airports.

Join Details & Statistics

Join Statistics on all data (table sizes, time to run joins, and cluster sizes) :

Dataset:
a) Airlines dataset : flight data from year 2015-2021 , Record Count=42430592
b) Base Stations dataset: Weather stations at airport , Record Count=16689
c) Weather dataset : All Stations dataset , Record Count=3056388

Output dataset : 33 columns, Record Count = 25043236

Before Join: Filter airline dataset for a year and quarter. Filter weather also for same year.

Join Criteria:
a) Airline left join with base station and left join with weather station
b) ICAO code (custom field) in Airline data = ICAO code in base stations
c) Station id in base station = Station id in weather dataset
d) Year of Weather and Airline is same

Filter Criteria:
a) Difference between Flight UTC timestamp and weather station should range between 2 (inclusive) and 5 hrs. lower bound is 2hrs and upper bound is 5 hrs difference. This is done in accordance to get recent weather data reported from a station just two hours prior to flight. Weather data has aggregated data over two hour window. Hence this bound made sense to avoid the overspilling of joining with unnecessary weather records. (Note : we had no rounding of datetimestamp in weather or airline dataset) b) Flight UTC timestamp will always be greater than weather station date

Output columns in Joined dataset:
airline columns, weather columns and Max of weather datetimestamp

Output File format & details:
Dataset is written in blob storage in parquet format partitioned by year and quarter as join happens iteratively.

Cluster Size during execution of Join query:
Cluster Capacity/Cores : 112 GB, 28 Cores
Worker Nodes: 6 (max based on cluster configuration)

Processing data iteratively per year and quarter
Total Execution Time : ~13.2 hrs (Note: At this time parallel execution of other notebooks were also happening)
Total number of loops (iteration) : 24 (Year (2021-2015)+1 * Quarters (4))
Per Batch (ie per year & quarter) : ~23 mins (best time)

Join Query for reference : Looping this per quarter & year :

```
joined_quarter_dataset=spark.sql("SELECT \
                q.DAY_OF_MONTH, q.DAY_OF_WEEK, q.TAIL_NUM,q.QUARTER,q.DEP_D
                q.CRS_DEP_TIME, q.ARR_DELAY, q.DIVERTED, q.DISTANCE,q.CANCE
                w.STATION, w.LATITUDE,w.LONGITUDE, w.NAME, w.REPORT_TYPE, w
                max(w.date) as max_dt \
                from quarter_dataset q left join base_stations bs on q.icao
                left join weather_dataset w on bs.station_id=w.station and
                where  \
                q.FlightDepartUTCTimestamp>=w.date and \
                 (((bigint(to_timestamp(q.FlightDepartUTCTimestamp))-bigint
                 AND ((bigint(to_timestamp(q.FlightDepartUTCTimestamp))-big
                group by q.DAY_OF_MONTH, q.DAY_OF_WEEK, q.TAIL_NUM,q.QUARTE
                q.CRS_DEP_TIME, q.ARR_DELAY, q.DIVERTED, q.DISTANCE,q.CANCE
                w.STATION, w.LATITUDE,w.LONGITUDE, w.NAME, w.REPORT_TYPE, w


joined_quarter_dataset.write.parquet(f"{blob_url}/final_joined_data/YEAR={y
```

Link to notebook containing join code

Link to w261_final_project_joining_dataset (https://adb-

731998097721284.4.azuredatabricks.net/?o=731998097721284#notebook/148077048169809/command/148077048169881)

```
#To run for full dataset remove /YEAR=2015
df_join = spark.read.parquet(f"{blob_url}/final_joined_data")
display(df_join)
```

	DAY_OF_MONTH	DAY_OF_WEEK	TAIL_NUM	QUARTER	DEP_DEL15
1	1	1	248NV	3	0
2	1	1	255NV	3	null
3	1	1	259NV	3	0
4	1	1	274NV	3	0
5	1	1	276NV	3	0
6	1	1	302NV	3	0

Truncated results, showing first 1000 rows.

```
#Add Hour Column
df_join = df_join.withColumn("HOUR", F.substring('CRS_DEP_TIME', 0, 2))

#Add Airline Description
df_join = df_join.join(df_airline_names, df_join.OP_CARRIER ==
df_airline_names.Airline_Code, "inner")
display(df_join)
```

	DAY_OF_MONTH	DAY_OF_WEEK	TAIL_NUM	QUARTER	DEP_DEL15
1	1	1	248NV	3	0
2	1	1	255NV	3	null
3	1	1	259NV	3	0
4	1	1	274NV	3	0
5	1	1	276NV	3	0
6	1	1	302NV	3	0

Truncated results, showing first 1000 rows.

Join EDA

Dataset size: 33 columns & 25,043,236 rows

```
print("Total Count Joined Set: ",df_join.count())
df_join.printSchema()
df_join.show(1, vertical=True)
```

```
Total Count Joined Set:  25043236
root
 |-- DAY_OF_MONTH: integer (nullable = true)
 |-- DAY_OF_WEEK: integer (nullable = true)
 |-- TAIL_NUM: string (nullable = true)
 |-- QUARTER: integer (nullable = true)
 |-- DEP_DEL15: double (nullable = true)
 |-- YEAR: integer (nullable = true)
 |-- MONTH: integer (nullable = true)
 |-- FL_DATE: date (nullable = true)
 |-- OP_CARRIER: string (nullable = true)
 |-- ORIGIN: string (nullable = true)
 |-- ORIGIN_CITY_MARKET_ID: integer (nullable = true)
 |-- DEST: string (nullable = true)
 |-- CRS_DEP_TIME: string (nullable = true)
 |-- ARR_DELAY: double (nullable = true)
 |-- DIVERTED: double (nullable = true)
 |-- DISTANCE: double (nullable = true)
 |-- CANCELLED: double (nullable = true)
 |-- DELAYED: integer (nullable = true)
 |-- FlightDepartUTCTimestamp: timestamp (nullable = true)
```

On Time vs Delayed Joined

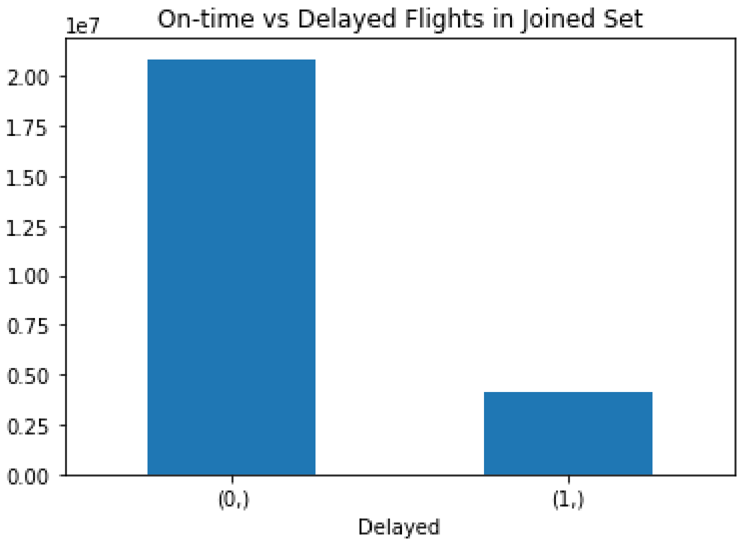
```
df_delayed = df_join.filter(F.col('Delayed')==1)
df_notdelayed = df_join.filter(F.col('Delayed')==0)
print(df_delayed.count())
print(df_notdelayed.count())
```

```
4163553
20879683
```

On-time vs Delayed Flights in Joined Set

```
df_join.select('Delayed').toPandas().value_counts().plot.bar(title = "On-time vs Delayed Flights in Joined Set", legend = False, rot = 0)
```

```
Out[147]:
```



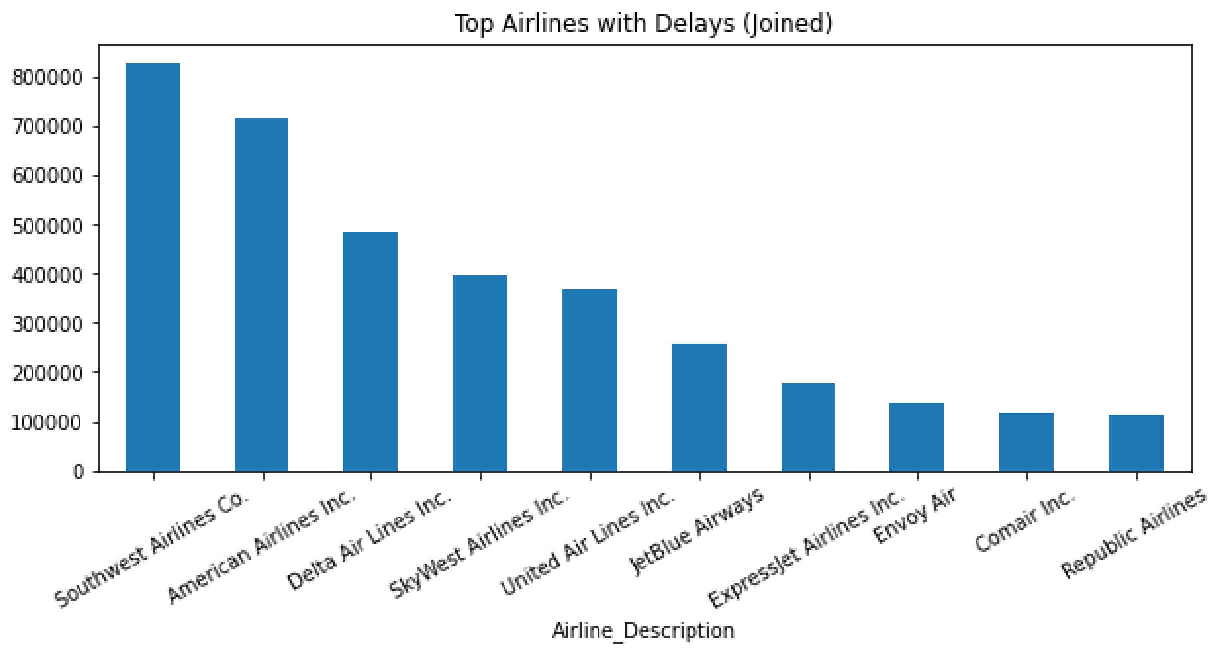
```
<AxesSubplot:title={'center':'On-time vs Delayed Flights in Joined Set'}, xlabe
l='Delayed'>
```

Delays and No Delays by Airline (Joined)

Total Delays by Airline (Joined)

```
# Distribution of Delayed by airline in joined
df_join.createOrReplaceTempView("airline")
df_delay = spark.sql("select Airline_Description, count(*) as Delays from
airline where DELAYED = 1 group by Airline_Description order by Delays desc
LIMIT 10")
df_delay.toPandas().plot.bar(title = "Top Airlines with Delays (Joined)",
y='Delays',x='Airline_Description', legend = False, figsize = (10,4), rot = 30)
```

```
Out[148]:
```

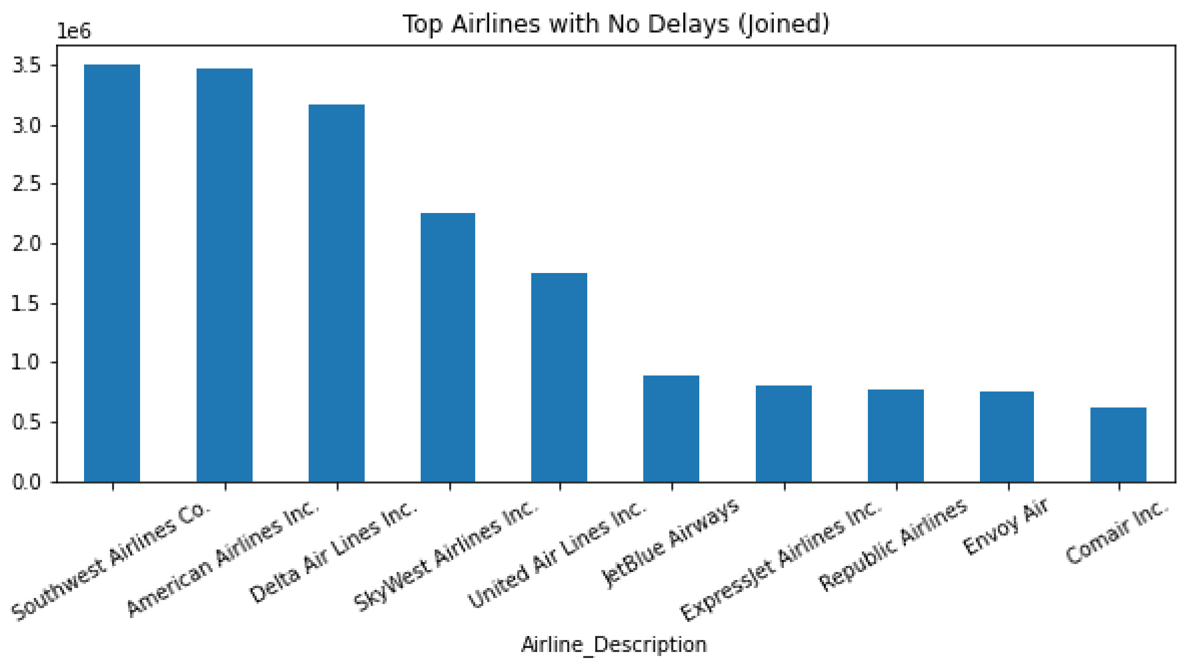


```
<AxesSubplot:title={'center':'Top Airlines with Delays (Joined)'}, xlabel='Airl
ine_Description'>
```

Total No Delays by Airline (Joined)

```
# Distribution of No Delays by Airline (Joined)
df_join.createOrReplaceTempView("airline")
df_delay = spark.sql("select Airline_Description, count(*) as Delays from
airline where DELAYED = 0 group by Airline_Description order by Delays desc
LIMIT 10")
df_delay.toPandas().plot.bar(title = "Top Airlines with No Delays (Joined)",
y='Delays',x='Airline_Description', legend = False, figsize = (10,4), rot = 30)
```

```
Out[149]:
```



```
<AxesSubplot:title={'center':'Top Airlines with No Delays (Joined)'}, xlabel='A
irline_Description'>
```

```
df_join.summary()
```

Correlations Between Features (Joined)

```
corr = df_join.toPandas().corr()
fig, ax = plt.subplots(figsize=(15, 15))
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(240, 10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=.5)
plt.title("Correlations Between Features (Joined)")
plt.show()
```

Joined Data: Missing Data

```
# We plan to follow the plan as shown in the above missing value table

display(df_join.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for
c in df_join.columns]))
```

```
  Cancelled
```

Machine Learning Algorithm

We plan to predict using logistic regression for classification. We will model the relationship of the independent features and the dependent feature, departure Delay or Not Delay.

$$f(y) = \frac{1}{1 + e^{-y}}$$

where

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

The cost function to find the best model is a Maximum Likelihood Estimation Function.

$$\hat{\theta}^{MLE} = \underset{\theta \in \Theta}{argmax} \prod_{i=1}^{n} p_\theta(y_i)$$

The result of the logistic regression prediction is the likelihood that the actual flight departure is 1 (Delayed). The baseline model we will use is when the predicted likelihood greater than or equal to 0.5 is Delayed, and lower than 0.5 is not Delayed.

Because there is imbalanced data, we plan to use undersampling to create the training data used in the model. As we build our model, we plan to use forward chaining cross validation to optimize parameters. In this case, we think it is more important for airports to be prepared for any potential delay, even if the delay prediction is false, rather than miss a delay that might cause a mess in last minute planning at the airports. Therefore, we prefer false positives over false negatives and will adjust the model thresholds accordingly.

Metrics to Measure Success

We will compare model results using the following four metrics to evaluate the model performance:

F2 Score

F2 score is a weighted harmonic mean of precision and recall. This metric gives more weight to recall than to precision, which means False Positives are considered better than False Negatives, which is what we think is important for flight delays.

$$F_2 Score = 5(\frac{Precision * Recall}{4Precision + Recall})$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

MCC (Matthews Correlation Coefficient)

This metric represents the confusion matrix as a single number, and helps with imbalanced data. MCC takes into account the proportion of each class. This metric ranges from -1 to 1, and -1 indicates the model is predicting the opposite class from the actual value, 0 is similar to random guessing, and 1 indicates a perfect classification.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Balanced Accuracy

Since the data is imbalanced, a normal accuracy metric will be high just by predicting the majority class, which is misleading. We instead will use balanced accuracy which takes into account different class sizes.

$$Balanced\ Accuracy = \frac{sensitivity + specificity}{2}$$

$$Sensitivity = \frac{TP}{TP + FN}$$

$$Specificity = \frac{TN}{TN + FP}$$

AUC (Area Under the ROC curve)

This metric evaluates how well the model can distinguish true positives and false positives. Although we would prefer to predict more positives, it would still be good to know how well the model is doing on the positive prediction side. An AUC of 1 indicates a perfect model, while AUC of 0.5 indicates a model that is no better than random guessing

Modeling Pipeline

Pipeline Block Diagram

Baseline Model

Number of input features: Number of experiments: 1 in this proposal

```
print("------------------------------------------------------------------------
------------------------")
print("Choosing variables to predict delays")
myY = "DELAYED"

#airline,
categoricals = [
'OP_CARRIER','HOUR', 'MONTH'
]

numerics = [
'VisibilityAvg'
]

myX = categoricals + numerics

df2 = df_join.select(myX + [myY])

test = df_join.filter(F.col('YEAR') == '2021')
train = df_join.subtract(test)
print('Train ', train.count())
print('Test ', test.count())
```

```
--------------------------------------------------------------------------------
-----------------
```

```
Choosing variables to predict delays
Train  21506557
Test  3536679


#Undersampling
def resample(base_features,class_field,base_class):
    """base_feature: dataframe with the features.
     class_field: name of the column that holds the classes. (dep delay)
     base_class: positive class"""
    pos = base_features.filter(F.col(class_field)==base_class)
    neg = base_features.filter(F.col(class_field)!=base_class)
    total_pos = pos.count()
    total_neg = neg.count()
    fraction=float(total_pos)/float(total_neg)
    sampled = neg.sample(False,fraction)
    return sampled.union(pos)

# separate minority and majority classes
train_resampled = resample(train,"Delayed",1)

train_resampled_delayed = train_resampled.filter(F.col('Delayed')==1)
train_resampled_notdelayed = train_resampled.filter(F.col('Delayed')==0)

print('delayed:', train_resampled_delayed.count())
print('not delayed', train_resampled_notdelayed.count())



delayed: 3571674
not delayed 3569189
```

```python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, OneHotEncoder
from pyspark.ml.feature import StandardScaler, Imputer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.mllib.evaluation import BinaryClassificationMetrics

## Current possible ways to handle categoricals in string indexer is 'error',
'keep', and 'skip'
indexers = map(lambda c: StringIndexer(inputCol=c, outputCol=c+"_idx",
handleInvalid = 'keep'), categoricals)
ohes = map(lambda c: OneHotEncoder(inputCol=c + "_idx",
outputCol=c+"_class"),categoricals)
imputers = Imputer(inputCols = numerics, outputCols = numerics)

# Establish features columns
featureCols = list(map(lambda c: c+"_class", categoricals)) + numerics

# Build the stage for the ML pipeline
model_matrix_stages = list(indexers) + list(ohes) + [imputers] + \
                    [VectorAssembler(inputCols=featureCols,
outputCol="features"), StringIndexer(inputCol="DELAYED", outputCol="label")]
# model_matrix_stages = list(indexers) + list(ohes) + \
#                       [VectorAssembler(inputCols=featureCols,
outputCol="features"), StringIndexer(inputCol="DELAYED", outputCol="label")]
# Apply StandardScaler to create scaledFeatures
scaler = StandardScaler(inputCol="features",
                        outputCol="scaledFeatures",
                        withStd=True,
                        withMean=True)

print("Featured Columns ", featureCols)

# Use logistic regression
lr = LogisticRegression(maxIter=10, elasticNetParam=0.5, featuresCol =
"scaledFeatures")

# Build our ML pipeline
pipeline = Pipeline(stages=model_matrix_stages+[scaler]+[lr])

# Build the parameter grid for model tuning
paramGrid = ParamGridBuilder() \
              .addGrid(lr.regParam, [0.1, 0.01]) \
              .build()
```

```
# Execute CrossValidator for model tuning
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=5)
```

```
Featured Columns  ['OP_CARRIER_class', 'HOUR_class', 'MONTH_class', 'Visibility
Avg']
```

```
# Train the tuned model and establish our best model
cvModel = crossval.fit(train_resampled)
```

```
glm_model = cvModel.bestModel
pred_df = glm_model.transform(test)
```

Train Results

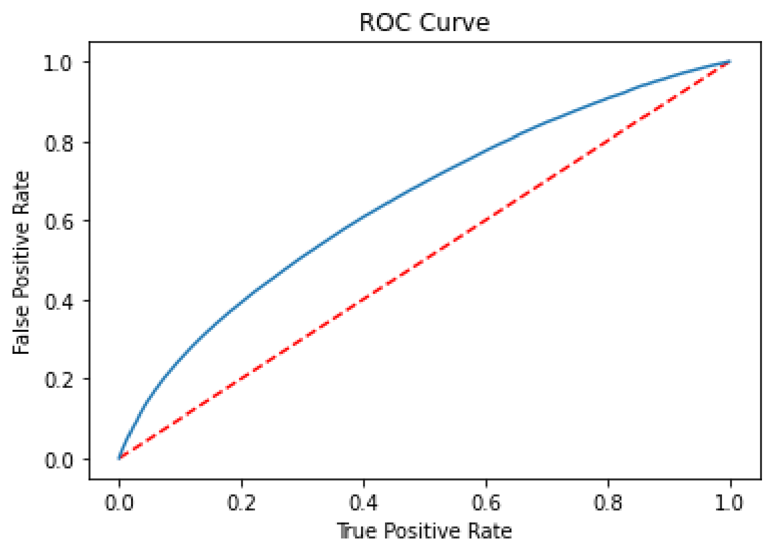
```
#AREA UNDER THE ROC CURVE
#In general, an AUC of 0.5 suggests no discrimination (i.e., ability to
diagnose patients with and without the disease or condition based on the
#test), 0.7 to 0.8 is considered acceptable, 0.8 to 0.9 is considered
excellent, and more than 0.9 is considered outstanding.

# Return ROC
lr_summary = glm_model.stages[len(glm_model.stages)-1].summary
display(lr_summary.roc)

roc = lr_summary.roc.toPandas()
plt.plot([0, 1], [0, 1], 'r--')
plt.plot(roc['FPR'],roc['TPR'])
plt.ylabel('False Positive Rate')
plt.xlabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
print('Training set areaUnderROC: ' + str(lr_summary.areaUnderROC))
```

```
/databricks/spark/python/pyspark/sql/context.py:134: FutureWarning: Deprecated
in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
```

	FPR	TPR
1		

2	0.0000067195382333332605	0.00007620778837993729
3	0.000012599134187498635	0.0001451310087529688
4	0.000023518383816664118	0.0002244207297512124
5	0.00003051790280971892	0.00030623203198261564
6	0.000036957460283329326	0.0003751552523556472

Truncated results, showing first 1000 rows.



```
Training set areaUnderROC: 0.6470152711241384
```

```
def extract(row):
  return (row.Delayed,) + tuple(row.probability.toArray().tolist()) +
(row.label,) + (row.prediction,)

def score(model,data):
  pred = model.transform(data).select("Delayed", "probability", "label",
"prediction")
  pred = pred.rdd.map(extract).toDF(["Delayed", "p0", "p1", "label",
"prediction"])
  return pred

def auc(pred):
  metric = BinaryClassificationMetrics(pred.select("p1", "label").rdd)
  return metric.areaUnderROC

glm_train = score(glm_model, train_resampled)

glm_train.createOrReplaceTempView("glm_train")

print ("GLM Training AUC:" + str(auc(glm_train)))
```

```
/databricks/spark/python/pyspark/sql/context.py:134: FutureWarning: Deprecated
in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
GLM Training AUC:0.6469379138927474
```

```
# Train set performance Metrics
TP = glm_train.filter((F.col('Delayed')==1) & (F.col('prediction') == 1)
).count()
TN = glm_train.filter((F.col('Delayed')==0) & (F.col('prediction') == 0)
).count()
FP = glm_train.filter((F.col('Delayed')==0) & (F.col('prediction') == 1)
).count()
FN = glm_train.filter((F.col('Delayed')==1) & (F.col('prediction') == 0)
).count()

print("True Positives:", TP)
print("True Negatives:", TN)
print("False Positives:", FP)
print("False Negatives:", FN)

import math
precision = TP / (TP + FP)
recall = TP / (TP + FN) # also known as sensitivity
specificity = TN / (TN + FP)

## F2 Score
F2 = 5 * ((precision * recall) / (4*precision + recall))

## MCC
MCC = (TP * TN - FP * FN)/ math.sqrt((TP + FP) * (TP + FN) * (TN + FP) * (TN +
FN))

## Balanced Accuracy
Balanced_Accuracy = (recall + specificity)/2

print('F2 =', F2)
print('MCC =', MCC)
print('Balanced Accuracy =', Balanced_Accuracy)
```

```
True Positives: 1110399
True Negatives: 1718776
False Positives: 1849672
False Negatives: 2461275
F2 = 0.3219151160330513
MCC = -0.2105464850273092
Balanced Accuracy = 0.396274804580701
```

Test results

```
# AUC for Test
glm_test = score(glm_model, test)
glm_test.createOrReplaceTempView("glm_test")
print ("GLM Test AUC :" + str(auc(glm_test)))
```

```
GLM Test AUC :0.648601903743247
```

```
from sklearn.metrics import classification_report, confusion_matrix

y_true = pred_df.select(['Delayed']).collect()
y_pred = pred_df.select(['prediction']).collect()

print(classification_report(y_true, y_pred))
```

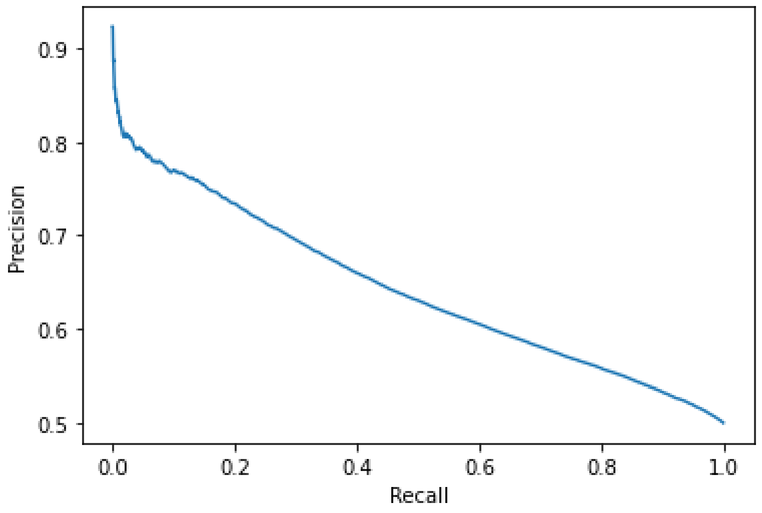
```
              precision    recall  f1-score   support

           0       0.78      0.48      0.59   2944800
           1       0.11      0.32      0.16    591879

    accuracy                           0.45   3536679
   macro avg       0.44      0.40      0.38   3536679
weighted avg       0.67      0.45      0.52   3536679
```

```
# Test set performance metrics
TP = pred_df.filter((F.col('Delayed')==1) & (F.col('prediction') == 1)
).count()
TN = pred_df.filter((F.col('Delayed')==0) & (F.col('prediction') == 0)
).count()
FP = pred_df.filter((F.col('Delayed')==0) & (F.col('prediction') == 1)
).count()
FN = pred_df.filter((F.col('Delayed')==1) & (F.col('prediction') == 0)
).count()

print("True Positives:", TP)
print("True Negatives:", TN)
print("False Positives:", FP)
print("False Negatives:", FN)

precision = TP / (TP + FP)
recall = TP / (TP + FN) # also known as sensitivity
specificity = TN / (TN + FP)

## F2 Score
F2 = 5 * ((precision * recall) / (4*precision + recall))

## MCC
MCC = (TP * TN - FP * FN)/ math.sqrt((TP + FP) * (TP + FN) * (TN + FP) * (TN +
FN))

## Balanced Accuracy
Balanced_Accuracy = (recall + specificity)/2

print('F2 =', F2)
print('MCC =', MCC)
print('Balanced Accuracy =', Balanced_Accuracy)
```

```
True Positives: 191003
True Negatives: 1401318
False Positives: 1543482
False Negatives: 400876
F2 = 0.23281686181939013
MCC = -0.1504134757196193
Balanced Accuracy = 0.3992840097081149
```

```
#pr = trainingSummary.pr.toPandas()
pr = lr_summary.pr.toPandas()
plt.plot(pr['recall'],pr['precision'])
plt.ylabel('Precision')
plt.xlabel('Recall')
plt.show()
```

```
/databricks/spark/python/pyspark/sql/context.py:134: FutureWarning: Deprecated
in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
```



Baseline Model: Experiment Table

	Input features	F2	Balanced Accuracy	MCC	AUC
Train	OP_CARRIER, HOUR, MONTH, VisibilityAvg	0.322	0.396	-0.211	0.647
Test	OP_CARRIER, HOUR, MONTH, VisibilityAvg	0.232	0.4	-0.150	0.649

note: these numbers are slightly different from results in the video due to difference in the random undersampling

Result Discussion

Because we want to predict more positives, we used F2 score. F2 puts more attention on minimizing false negatives rather than false positives. The F2 score is fairly low for both train and test, with the train having slightly higher F2 score of 0.32, indicating that our model is not very good at classifying delays.

Balanced accuracy ranges from 0 to 1, where 1 is the best and 0 is the worst. Both our test and train had similar balanced accuracy around 0.4, which can be considered poor accuracy.

The range of values of MCC lie between -1 to +1, with +1 indicating a perfect classifier. Both our train and test models had negative MCC, indicating that our model is slightly predicting more of the opposite class from the actual value.

The AUC for both train and test was similar around 0.65. This indicates that the model is slightly better than a score of 0.5 (random guessing), but is still a pretty poor performance.

Overall, all four of our metrics for both test and train are indicating that the baseline model is predicting delays quite poorly, and there is room for significant improvement.

Conclusion and Next Steps

The goal of our project is to improve on a baseline model for predicting flight departure delays to help airports plan operating decisions. We believe with features from the flight, weather, station data, and additional feature engineered variables, our modeling pipelines can accurately forecast departure delays 2 hours prior to the scheduled CRS departure. In this phase of the project, we have conducted some exploratory data analysis, joined the data, created a modeling pipeline, and created a baseline model indicating room for significant improvement. The baseline model overall had pretty poor predictions, and we plan to improve on it by adding features, doing feature engineering, and using forward chaining cross validation. As we add features to the model, we also hope to answer our hypothesis that adding prior flight information will be more predictive than having just the current weather and flight information

Team Members and Responsibilities

- **Beijing Wu**: Weather EDA, Model Pipeline
- **Grace Lee**: Airline EDA. Written Sections. Baseline model metrics, Results and Conclusion. Notebook maintenance.
- **Shivangi Pandey**: Station EDA, Join
- **Sybil Santos-Burgan**: Airline & Join EDA. Pipeline block. Notebook maintenance. Blob storage.

Beijing	Shivangi	Grace	Sybil

Open Issues or Problems

a) The data is very large, so we anticipate issues running things, particularly anything that needs cross validation.

References

"Air Travel Consumer Report: Consumer Complaints Against Airlines Rise More than 300 Percent above Pre-Pandemic Levels." Air Travel Consumer Report: Consumer Complaints Against Airlines Rise More Than 300 Percent Above Pre-Pandemic Levels | Bureau of Transportation Statistics, https://www.bts.dot.gov/newsroom/air-travel-consumer-report-consumer-complaints-against-airlines-rise-more-300-percent (https://www.bts.dot.gov/newsroom/air-travel-consumer-report-consumer-complaints-against-airlines-rise-more-300-percent).

US Domestic and International Delays and Cancellations Report, Flight Aware, https://public.tableau.com/app/profile/flightaware/viz/AirlineCancellationDelayUpdate
(https://public.tableau.com/app/profile/flightaware/viz/AirlineCancellationDelayUpdat

https://www.nytimes.com/2022/07/01/travel/summer-travel-flight-delays-cancellations.html (https://www.nytimes.com/2022/07/01/travel/summer-travel-flight-delays-cancellations.html)