Predicting Flight Departure Delay using a Scalable Data Mining Approach

Notebook organization:

- Executive summary / Abstract
- Introduction, motivation and problem statement
- Parameter interpretation and its descriptive statistics -- Original parameters
- Preparing data for model taining by joining various data tables
- Parameter interpretation and its descriptive statistics -- Derived parameters
- Data imputation
- Training and testing dataset split
- Exploratory study on the training dataset
- Model training
- Analyzing the results and model comparisons
- Conclusion

Authors: Nathan Nusaputra, Ryan Sawasaki, James Gao, Ankitkumar Patel

Team Number: 29

Abstract

Flight delays are frequent and growing over time due to the increasing air traffic and uncertainity in the global environment, which cost tens of billions of dollars and introduce inconvenience to passangers. According to the U.S. department of transportation, 15% of the flights are delayed beyond their scheduled departure time. Thus, flight delay prediction is becoming a primary issue for airlines and passengers. The goal of the project is to predict flight departure delay using the historial flight schedules and weather information at the original and destination airports in a scalable manner. We design a data processing and transformation pipeline to analyze and mine 31 million historical flight scheudules and 630 million weather data records using the parallel processing map-reduce algorithm. We extract key flight and weather-related features using extensive exploratory analysis which are used to model flight delay and explain its variability. The decision tree algorithm is used to train a model, which results an accuracy of 84% and recall of 35% on the test dataset. The experimental results demonstrate the accuracy an scalability of the flight prediction task that can be achieved by using distributed computing algorithms like map-reduce and frameworks like PySpark.

Question Formulation

Introduction

For this project we will be predicting flight delays using the 2015-2019 flights data from the Bureau of Transportation Statistics containing over 31 million records on U.S. domestic flights with information on schedules, delays, distance, airports, and airline carriers. In addition, we have the weather statistics from the National Oceanic and Atmospheric Administration that may be able to provide relationships between weather and flight delays. Additional datasets such as a global airport and airport timezone database are also used to assist in joining the airline and weather datasets.

Motivation

Flight delays have a significant impact for both airlines and passengers. For passengers, delays can result in missed connections and the uncertainty in flights leads to schedule buffers. For airlines, delays result in additional operational costs in addition to the loss of business. In a study conducted by the Federal Aviation Administration, it was concluded the flight delays result in costs to airlines up to $8 billion per year. The study also finds that the costs to passengers is over double that of airlines at over $18 billion per year.
(https://www.faa.gov/data_research/aviation_data_statistics/media/cost_delay_estimates.pdf
(https://www.faa.gov/data_research/aviation_data_statistics/media/cost_delay_estimates.pdf)). Given the sheer volume of domestic flights per year, the ability to correctly predict even a small percentage of delays could result in cost savings in the hundreds of millions.

Objective

To predict flight delay status two hours ahead of the planned departure time. We define flight delay as the difference between the scheduled and actual departure time. A flight status is marked as delayed if the flight delay is more than 15 minutes.

```
airlines = spark.read.option("header", "true").parquet(f"dbfs:/mnt/mids-w261/datasets_final_project/parquet_airlines_data/201*.parquet")
display(airlines.sample(False, 0.00001))
```

	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER_AIRLINE
1	2019	2	6	2	7	2019-06-02	DL	19790
2	2019	2	6	8	6	2019-06-08	UA	19977
3	2019	2	6	2	7	2019-06-02	OH	20397
4	2019	2	6	5	3	2019-06-05	AS	19930
5	2016	3	8	17	3	2016-08-17	NK	20416
6	2016	3	8	2	2	2016-08-02	DL	19790
7	2016	3	8	2	2	2016-08-02	OO	20304
8	2016	3	8	26	5	2016-08-26	WN	19393

Showing all 298 rows.

```
display(airlines.describe())
```

	summary	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUI
1	count	31746841	31746841	31746841	31746841	31746841	31746841	31746841
2	mean	2017.1512498204152	2.5174877084620797	6.552106365480585	15.749554640727876	3.9346285509162944	null	null
3	stddev	1.4316532810209637	1.1053295681781923	3.3994302561415313	8.774238088354524	1.9917635387471753	null	null
4	min	2015	1	1	1	1	2015-01-01	9E
5	max	2019	4	12	31	7	2019-12-31	YX

Showing all 5 rows.

```
weather = spark.read.option("header", "true")\
                    .parquet(f"dbfs:/mnt/mids-w261/datasets_final_project/weather_data/*.parquet")

f'{weather.count():,}'

Out[25]: '630,904,436'

display(weather.sample(False, 0.000001).describe())
```

	summary	STATION	SOURCE	LATITUDE	LONGITUDE	ELEVATION	NAME
1	count	624	552	627	627	627	627
2	mean	6.03317036968871E10	4.911231884057971	36.144261670015936	-37.49694685773525	371.68178628389165	null
3	stddev	3.1776372196684196E10	1.3476135809478413	23.439222046048272	81.06893537587904	527.9814328804116	null
4	min	1001099999	4	-71.8833333	-166.14667	0.9	
5	max	A0688400416	7	78.65	171.2	3407.4	ZHULIANY INTERNATION

Showing all 5 rows.

Feature Engineering and Join

To join the weather data and the airlines data we will be using the departure time - 2 hours and the weather station code as the primary key to join with the weather time stamp and the weather station. The first step is to assign each airport to its closest weather code. In addition, the departure time will need to be converted from the timezone time to the weather's utc time.

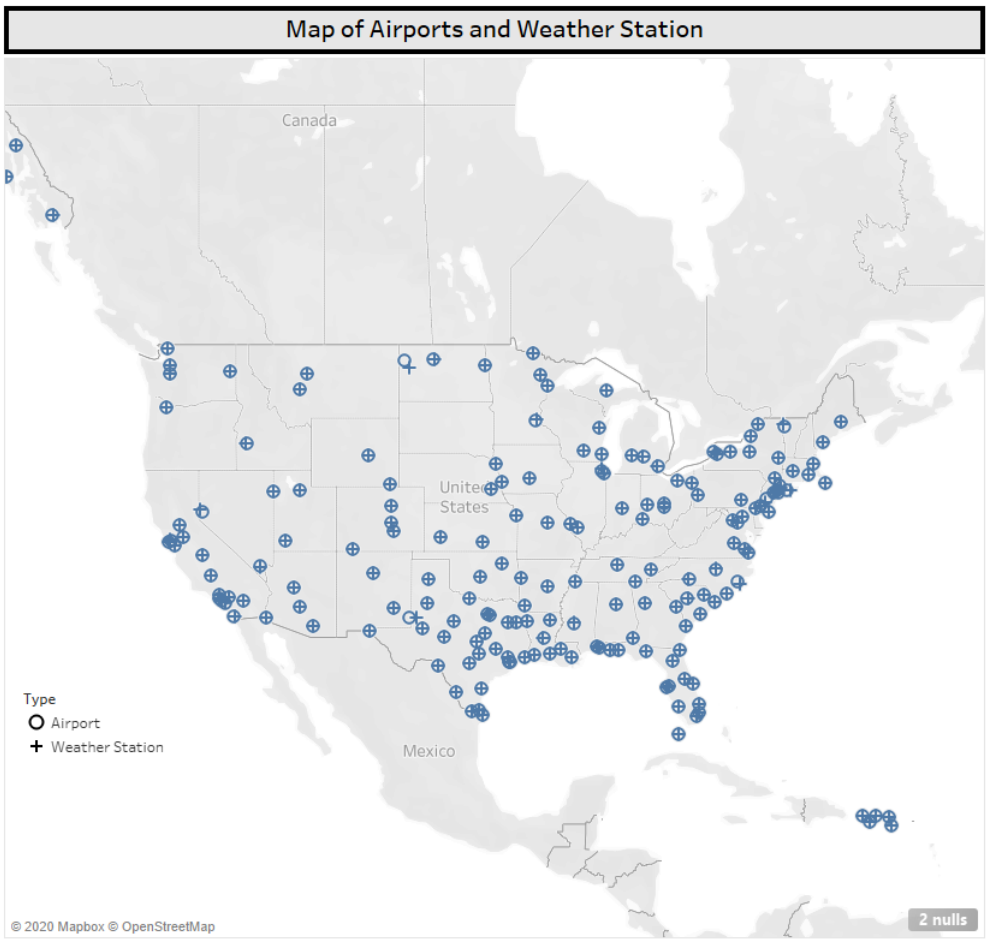
The first challenge we faced when trying to join the datasets is that the weather data is located by weather stations while the key for airlines data is the airport code. We determined that the longtitude and latitude is the best way to join these two data sets. To assist in joining the datasets an external database with longtitude and latitudes for each of the airports was used (https://www.partow.net/miscellaneous/airportdatabase/ (https://www.partow.net/miscellaneous/airportdatabase/)). The weather data already had the longtitiude and latitude within the data set so no additional steps were performed. The next step was calculating the haversine distance which is the angular distance between two points would help us determine which weather station was the closest to each airport. We performed a cross join and ranked each row by ascending order based on distance and finally filtered for only rank "1". Below we show the final output and the Haversine calculation.

```
# # formula for haversine distance calculation found here:
# # https://stackoverflow.com/questions/60086180/pyspark-how-to-apply-a-python-udf-to-pyspark-dataframe-columns

# from pyspark.sql.functions import col, radians, asin, sin, sqrt, cos

# airports_and_stations_exploded_with_distance = airports_and_stations_exploded.withColumn("dlon",
radians(col("longitude_decimal_degrees")) - radians(col("LONGITUDE"))) \
#                                                                         .withColumn("dlat",
radians(col("latitude_decimal_degrees")) - radians(col("LATITUDE"))) \
#                                                                         .withColumn("haversine_dist", asin(sqrt(
#                                                                                                             sin(col("dlat") /
2) ** 2 + cos(radians(col("LATITUDE")))
#                                                                                                             *
cos(radians(col("latitude_decimal_degrees"))) * sin(col("dlon") / 2) ** 2
#                                                                                                             )
#                                                                                                         ) * 2 * 3963 * 5280) \
#                                                                         .drop("dlon", "dlat")

# airports_and_stations_exploded_with_distance.createOrReplaceTempView('airports_and_stations_exploded_with_distance')


# Table showing the final output
airport_weather_station_key = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airport_weather_station_key")
airport_weather_station_key.createOrReplaceTempView('airport_weather_station_key')
test = 'SELECT station_id, station_latitude, station_longitude, airport_code, airport_latitude, airport_longitude, haversine_dist_feet
FROM airport_weather_station_key'
display(sqlContext.sql(test))
```

	station_id	station_latitude	station_longitude	airport_code	airport_latitude	airport_longitude	haversine_dist_feet
1	72512614736	40.29639	-78.32028	AOO	40.296	-78.32	161.96709662798654
2	84752099999	-16.341072	-71.583083	AQP	-16.341	-71.583	39.26067867327079
3	72219013874	33.6301	-84.4418	ATL	33.64	-84.427	5772.012020316871
4	72228013876	33.56556	-86.745	BHM	33.563	-86.753	2607.411638777335
5	96441099999	3.12385	113.020472	BTU	3.172	113.044	19565.70347813837
6	83208099999	-12.694375	-60.098269	BVH	-12.694	-60.098	167.16911411449553
7	16020099999	46.460194	11.326383	BZO	46.461	11.326	309.19762061213015
8	59287099999	23.392436	113.298786	CAN	23.184	113.266	76912.07784504181

Showing the first 1000 rows.

Map of Airports and Weather Station



After applying data imputation, most of the fields are populated with data and we have only 0.03% of airport codes are missing in the table.

```
airport_weather_station_key.count()
```

```
Out[20]: 2911
```

```
airport_weather_station_key.select([(count(when(isnan(c) | col(c).isNull(), c))/airport_weather_station_key.count()).alias(c) for c in
airport_weather_station_key.columns if c not in [""]]).toPandas().head()
```

```
Out[11]:
```

	station_id	station_name	station_latitude	station_longitude	airport_code	airport_name	airport_city	airport_country	airport_latitude	airport_longitude	haversine_dist_f
0	0.0	0.0	0.0	0.0	0.000344	0.0	0.0	0.0	0.0	0.0	

The second challenge we faced is to pull together the airlines and weather datasets. Given that the airlines data only provides local departure time, we needed to convert the times to UTC for consistency. We first created a UDF to convert the local time to a timestamp. With the help of an outside database (https://openflights.org/data.html (https://openflights.org/data.html)), we were able to assign timezones to airports based on their respective IATA codes. Using the pytz library along with the timezones we were able to convert the airport departure times from local to UTC time.

```
# read in and set airport_timezones dataset to variable airport_tz
airport_tz = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airport_timezones")
airport_tz.createOrReplaceTempView('airport_tz')
display(airport_tz)
```

	id	name	city	country	IATA	ICAO	lat
1	3212	Barter Island LRRS Airport	Barter Island	United States	BTI	PABA	70.1340
2	3213	Wainwright Air Station	Fort Wainwright	United States	\N	PAWT	70.6134
3	3214	Cape Lisburne LRRS Airport	Cape Lisburne	United States	LUR	PALU	68.8750

4	3215	Point Lay LRRS Airport	Point Lay	United States	PIZ	PPIZ	69.7329
5	3216	Hilo International Airport	Hilo	United States	ITO	PHTO	19.7213
6	3217	Orlando Executive Airport	Orlando	United States	ORL	KORL	28.5455
7	3218	Bettles Airport	Bettles	United States	BTT	PABT	66.9139
8	3219	Clear Airport	Clear Mews	United States	\N	PACL	64.3012

Showing the first 1000 rows.

```
# # Convert the date and time in the airlines dataset to timestamps

# # UDF for converting year, month, day to timestamps
# def create_datetime_from_parts(year, month, day, local_time):
#     time = str(local_time)
#     num_str = time.zfill(4)
#     hour = num_str[0:2]
#     min = num_str[2:4]
#     return f'{year}-{month}-{day} {hour}:{min}:0'

# # Apply UDF to create column for local departure time
# create_datetime_udf = f.udf(create_datetime_from_parts, types.StringType())
# airlines_with_localtime = airlines_with_weather_station.withColumn("LOCAL_DEP_TIME", create_datetime_udf('YEAR', 'MONTH',
'DAY_OF_MONTH','CRS_DEP_TIME').cast(types.TimestampType()))

# # join airlines dataset with timezone dataset. join based on departure airport 'ORIGIN' and airport 'IATA'
# airlines_with_timezone = airlines_with_localtime.join(f.broadcast(airport_tz), airport_tz.IATA==airlines_with_localtime.ORIGIN,
'inner')

# # use the spark built-in function to convert the local time to UTC time based on timezone
# airlines_with_utc = airlines_with_timezone.withColumn("UTC_DEP_TIME",to_utc_timestamp(col("LOCAL_DEP_TIME"), col("timezone")))
```

Here is our final airlines with weather station and UTC departure time to be joined with the weather data set.

```
# read in airlines_with_weather_station_utc_dep
airlines_with_weather_station_utc_dep = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_station_utc_dep")
airlines_with_weather_station_utc_dep.createOrReplaceTempView('airlines_with_weather_station_utc_dep')
display(airlines_with_weather_station_utc_dep)
airlines_with_weather_station_utc_dep.count()
```

	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER_AIRLINE
1	2019	2	4	5	5	2019-04-05	NK	20416
2	2019	2	4	6	6	2019-04-06	NK	20416
3	2019	2	4	7	7	2019-04-07	NK	20416
4	2019	2	4	8	1	2019-04-08	NK	20416
5	2019	2	4	9	2	2019-04-09	NK	20416
6	2019	2	4	10	3	2019-04-10	NK	20416
7	2019	2	4	11	4	2019-04-11	NK	20416
8	2019	2	4	12	5	2019-04-12	NK	20416

Showing the first 1000 rows.

```
Out[3]: 31573695
```

Feature Engineering

Research of literature provided background knowledge of the causes of flight delays. One article examined the effects of delay propagation (https://github.com/UCB-w261/main/blob/master/Assignments/Final%20Project/airline-delays-literature/1703.06118.pdf (https://github.com/UCB-w261/main/blob/master/Assignments/Final%20Project/airline-delays-literature/1703.06118.pdf)) Delay propogation suggests that once a flight leg is delayed it could likely result in a chain effect of departure delays. Creating this variable could prove to be useful in determining delayed flights.

The steps to create the new variable goes as follows:

1. Partition by tail number and order by UTC departure time
2. Create column "PREV_UTC_DEP_TIME" that has the previous flight leg UTC departure time

3. Create column "DEP_TIME_DIFF" that calculates the time difference between the previous departure time and the current departure time, in hours. This time difference must be greater than 2 hours because we can only use data that we have access to 2 hours prior to departure.
4. To account for changes in day the DEP_TIME_DIFF must be less than 7.5 hours. First flights in the morning are usually not impacted by the last flight from the night before.
5. Create column "PREV_DEP_DEL15" that denotes if the previous flight leg departure was delayed by 15 minutes or more. When the time difference between departures is between 2 and 7.5 hours, "PREV_DEP_DEL15" is equal to the previous flight leg's "DEP_DEL15". Else, "PREV_DEP_DEL15" is set to 0.

```
# Remove the rows with nulls in the DEP_DEL15 column (should remove 474,582 records)
airlines_with_weather_station_remove_nulls =
airlines_with_weather_station_utc_dep.filter(airlines_with_weather_station_utc_dep.DEP_DEL15.isNotNull())
```

```
# import functions
from pyspark.sql import functions as F
from pyspark.sql.functions import col, lag, hour, when
from pyspark.sql.window import Window

# create window that partitions on tail number and orders by UTC departure time
my_window = Window.partitionBy("TAIL_NUM").orderBy('UTC_DEP_TIME')

# create column PREV_UTC_DEP_TIME for previous flight leg UTC departure time
# create column DEP_TIME_DIFF to calculate difference (in hours) between current departure time and previous departure time
# create column PREV_DEP_DEL15 to denote if the previous flight was delayed 15 minutes (DEP_DEL15 = 1) given that DEP_TIME DIFF is
between 2 and 7.5 hours
airports_and_stations_with_prev_del15 = airlines_with_weather_station_remove_nulls.withColumn("PREV_UTC_DEP_TIME",
F.lag(col('UTC_DEP_TIME')).over(my_window)) \
                                    .withColumn("DEP_TIME_DIFF", ((col("UTC_DEP_TIME").cast("long") -
col("PREV_UTC_DEP_TIME").cast("long"))/3600)) \
                                    .fillna({'DEP_TIME_DIFF': 0 }) \
                                    .withColumn("PREV_DEP_DEL15", when((col("DEP_TIME_DIFF") >= 2) & (col("DEP_TIME_DIFF") <= 7.5),
F.lag(col("DEP_DEL15")).over(my_window))
.otherwise(0))

# display dataframe
display(airports_and_stations_with_prev_del15)
airports_and_stations_with_prev_del15.count()
```

	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER_AIRLINE
1	2015	1	1	1	4	2015-01-01	US	20355
2	2015	1	1	1	4	2015-01-01	US	20355
3	2015	1	1	1	4	2015-01-01	US	20355
4	2015	1	1	1	4	2015-01-01	US	20355
5	2015	1	1	2	5	2015-01-02	US	20355
6	2015	1	1	2	5	2015-01-02	US	20355
7	2015	1	1	2	5	2015-01-02	US	20355
8	2015	1	1	2	5	2015-01-02	US	20355

Showing the first 1000 rows.

```
Out[5]: 31099113
```

Exploratory Data Analysis

In this section, the given airline and weather data are analyzed to identify the features correlated with the target variable 'DEP_DEL15'. 'DEP_DEL15' is a binary variable, where value 1 indicates that a flight was delayed by more than 15 minutes from its scheduled departure time and value 0 indicates that a flight departed at the scheduled time. In the given dataset, 18% of flights were delayed more than 15 minutes while 82% of flights were ontime. Thus, the given data is imbalanced in terms of binary classes. In the given dataset, most of the fields for the airline schedules and airports have no missing data; however, we couldn't find weather station information for the 5.5% of the airports.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pyspark.sql.functions import isnan, when, count, col
```

```
pydf = airports_and_stations_with_prev_del15.sample(False, 0.01)
df = pydf.toPandas()
print(f" The fraction of flights delayed more than 15 minutes is {np.mean(df['DEP_DEL15'])} and the fraction of flights on schedule is
{1-np.mean(df['DEP_DEL15'])}")
```

```
 The fraction of flights delayed more than 15 minutes is 0.18187179676206533 and the fraction of flights on schedule is 0.8181282032379
347
```

```
pydf.select([(count(when(isnan(c) | col(c).isNull(), c))/pydf.count()).alias(c) for c in pydf.columns if c not in ["LOCAL_DEP_TIME",
"UTC_DEP_TIME", "PREV_UTC_DEP_TIME"]]).toPandas().head()
```

Out[8]:

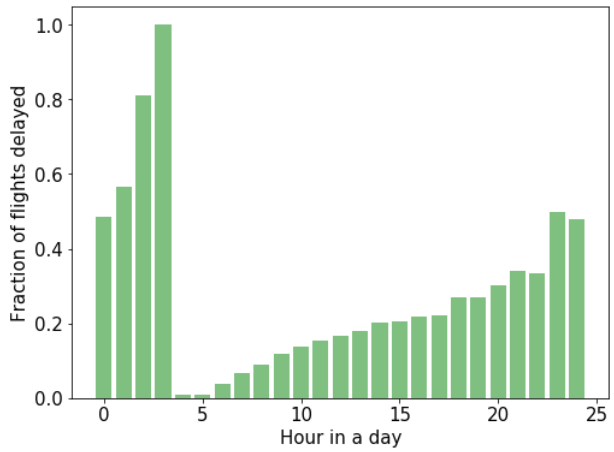
	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER_AIRLINE_ID	OP_CARRIER	TAIL_NUM	OP_CARRIER_FL_
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

1 rows × 126 columns

```
pydf = pydf.fillna(0, subset=['DEP_DELAY_NEW', 'CARRIER_DELAY', 'WEATHER_DELAY', 'NAS_DELAY', 'SECURITY_DELAY', 'LATE_AIRCRAFT_DELAY'])
df = pydf.toPandas()
```

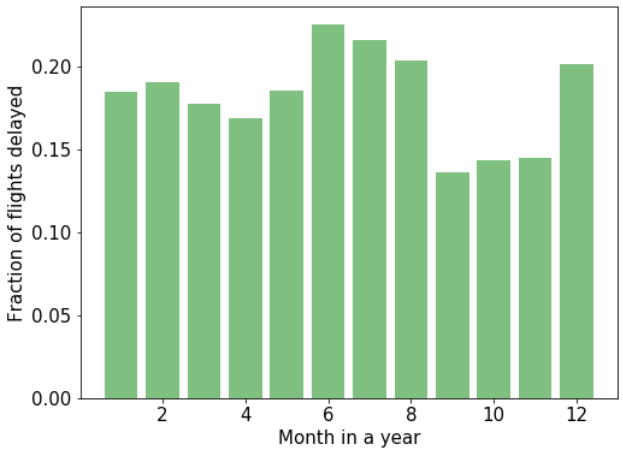
Next, we analyze airline departure delay varies with time. The following plot shows average fraction of flights delayed over the hour of a day. The mean fraction of flights delayed is significantly higher for flights departed later in the evening or early in the morning compared to those that were departed during the day time. Possible reason is the lower atmostpheric visibility and higher precipitation during evening and early morning compared to day. Thus, the hour in a day is highly correlated with the target variable 'DEP_DEL15' and can explain its variability in the delay estimation.

```
df['dep_hour'] = df['DEP_TIME'].apply(lambda x: int(x/100) if not np.isnan(x) else -1)
agg_hour = df.groupby(['dep_hour'], as_index=False).agg({'DEP_DEL15': 'sum', 'TAIL_NUM': 'count'})
agg_hour['frac'] = agg_hour['DEP_DEL15']/agg_hour['TAIL_NUM']
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 15})
plt.bar(agg_hour['dep_hour'], agg_hour['frac'], color='green', alpha=0.5)
plt.xlabel("Hour in a day")
plt.ylabel("Fraction of flights delayed")
plt.show()
```



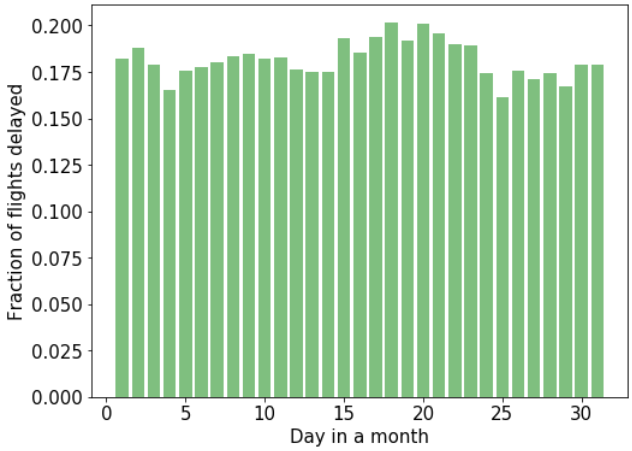
The following chart shows how average fraction of flight delays vary with the month in year. Flights scheduled during summer and winter months tend to delay with higher probabilty compared to the flights scheduled during the spring and fall months. Thus, there appears to be seasonal effects in the flight departure delay, and thus, the month in year can be used to estimate the flight departure delay.

```
agg_month1 = df.groupby(['MONTH'], as_index=False).agg({'DEP_DEL15': 'sum', 'TAIL_NUM': 'count'})
agg_month1['frac'] = agg_month1['DEP_DEL15']/agg_month1['TAIL_NUM']
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 15})
plt.bar(agg_month1['MONTH'], agg_month1['frac'], color='green', alpha=0.5)
plt.xlabel("Month in a year")
plt.ylabel("Fraction of flights delayed")
plt.show()
```



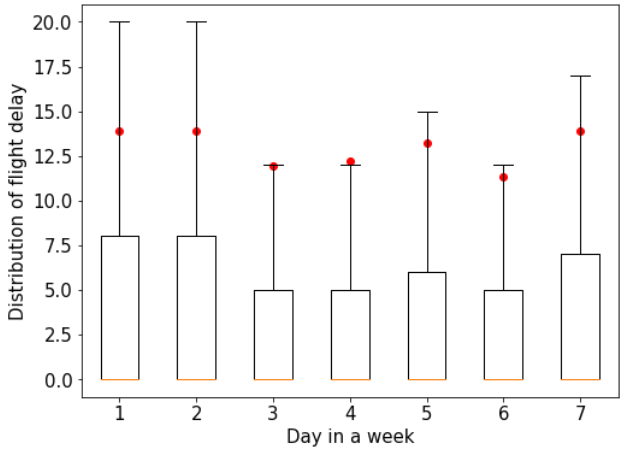
On the other hand, the average fraction of flight delays have minor variabiilty with the day of month, as shown in the following figure. Thus, the day of month may not provide enough information to explain the variability in the flight delays. The day of month can be dropped from the explanatory variables of the model.

```
agg_month_day = df.groupby(['DAY_OF_MONTH'], as_index=False).agg({'DEP_DEL15': 'sum', 'TAIL_NUM': 'count'})
agg_month_day['frac'] = agg_month_day['DEP_DEL15']/agg_month_day['TAIL_NUM']
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 15})
plt.bar(agg_month_day['DAY_OF_MONTH'], agg_month_day['frac'], color='green', alpha=0.5)
plt.xlabel("Day in a month")
plt.ylabel("Fraction of flights delayed")
plt.show()
```



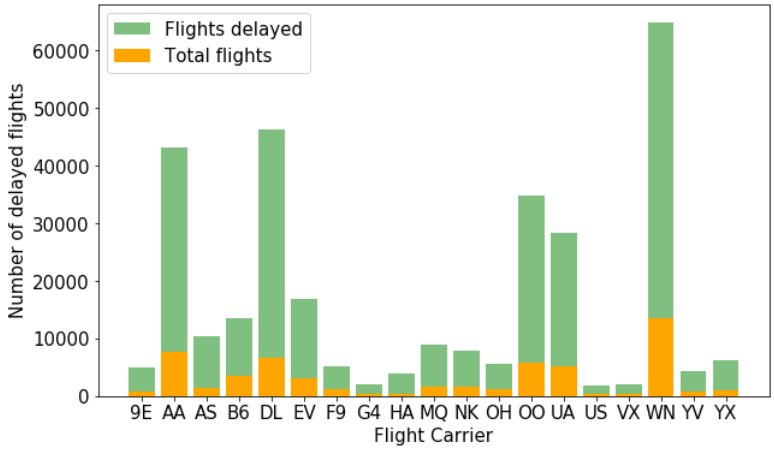
The following chart shows how flight departure delay varies with the day of week. In the box plot, the horizontal orange line represents median value and red dot shows the mean value. To get clear visual of various quartile values, the outliers are dropped from the box plot. We do not find substantial variability in the mean and median values over the day of week, and thus, the flight departure delay is not strongly correlated with the day of week.

```
data_to_plot = []
data_avg_delay = []
labels = []

airline_data = df.dropna(subset=['DEP_DELAY_NEW'])
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 15})

for day in airline_data['DAY_OF_WEEK'].unique():
  data_to_plot.append(airline_data[airline_data['DAY_OF_WEEK']==day]['DEP_DELAY_NEW'].values)
  data_avg_delay.append(np.mean(airline_data[airline_data['DAY_OF_WEEK']==day]['DEP_DELAY_NEW']))
  labels.append(day)

plt.plot([x+1 for x in range(len(data_avg_delay))], data_avg_delay, 'ro')
plt.boxplot(data_to_plot, showfliers=False)
plt.xlabel("Day in a week")
plt.title('')
plt.ylabel("Distribution of flight delay")
plt.show()
```



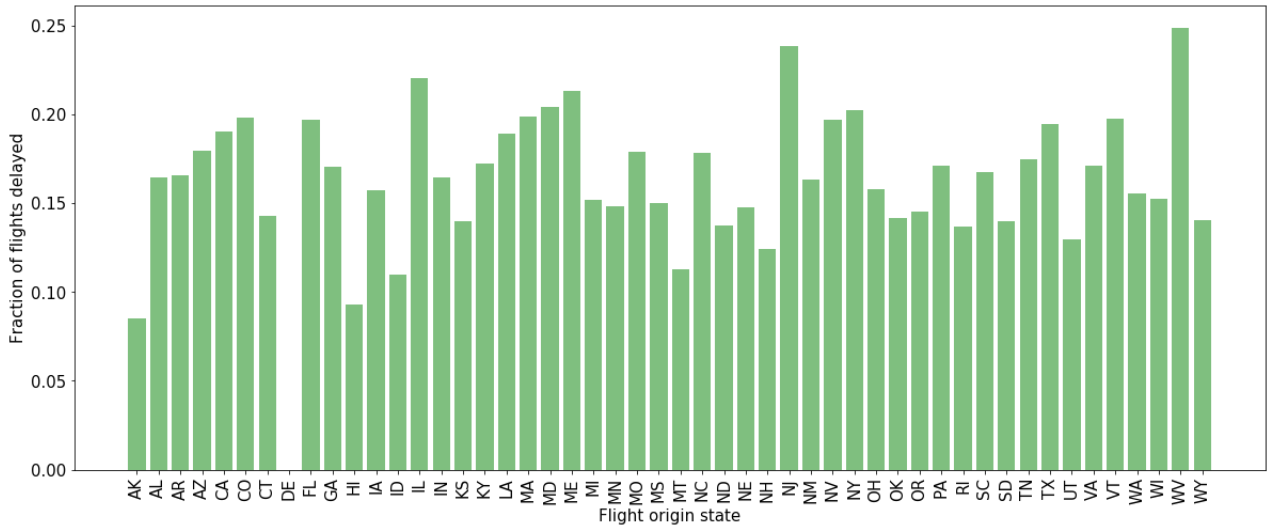
The following chart shows the number of delayed flights and the total number of flights per carrier. A large fraction of flight delays is operated by large carriers compared to small carriers and thus, considering flight carrier in predicting flight delay may improve prediction accuracy of the model.

```
agg_carrier = df.groupby(['OP_CARRIER'], as_index=False).agg({'DEP_DEL15': 'sum', 'TAIL_NUM': 'count'})
agg_carrier['frac'] = agg_carrier['DEP_DEL15']/agg_carrier['TAIL_NUM']
plt.figure(figsize=(10, 6))
plt.rcParams.update({'font.size': 15})
plt.bar(agg_carrier['OP_CARRIER'], agg_carrier['TAIL_NUM'], label="Flights delayed", color='green', alpha=0.5)
plt.bar(agg_carrier['OP_CARRIER'], agg_carrier['DEP_DEL15'], label="Total flights", color='orange')
plt.xlabel("Flight Carrier")
plt.ylabel("Number of delayed flights")
plt.legend()
plt.show()
```

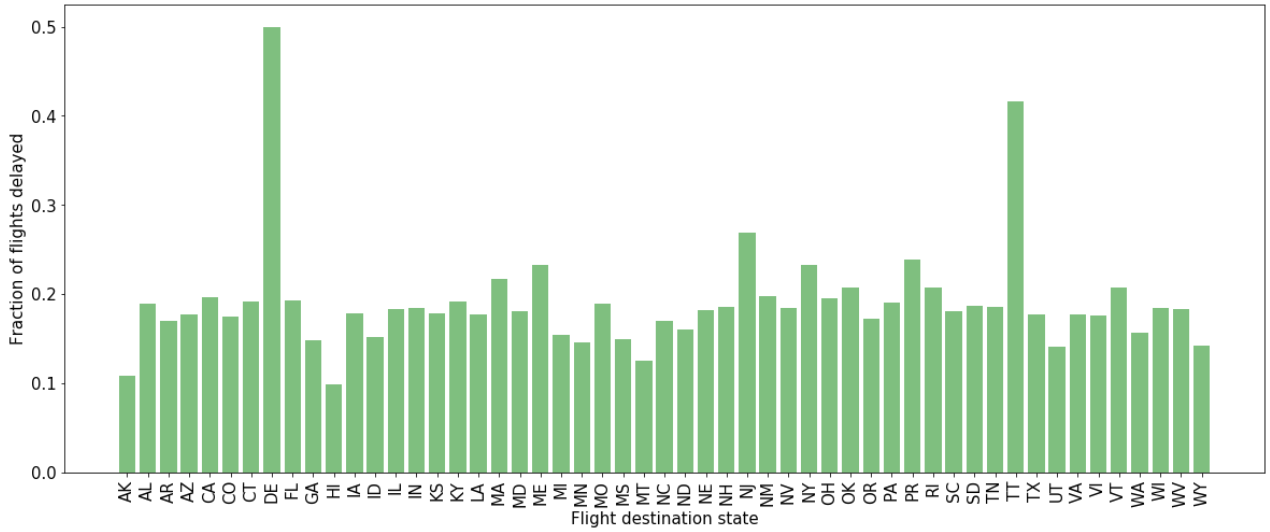


The following two charts show how flight departure delay varies with departure and arrival states. The average fraction of flight delays vary significantly from state-by-state. Potential explanation for such high variations in the departure delay is the variations in air traffic and weather across different states. Thus, the departure and arrival flight states may help the model explain the variabilities the flight departure delay.

```
agg_origin_state = df.groupby(['ORIGIN_STATE_ABR'], as_index=False).agg({'DEP_DEL15': 'sum', 'TAIL_NUM': 'count'})
agg_origin_state['frac'] = agg_origin_state['DEP_DEL15']/agg_origin_state['TAIL_NUM']
plt.figure(figsize=(20, 8))
plt.rcParams.update({'font.size': 15})
plt.bar(agg_origin_state['ORIGIN_STATE_ABR'], agg_origin_state['frac'], color='green', alpha=0.5)
plt.xlabel("Flight origin state")
plt.ylabel("Fraction of flights delayed")
plt.xticks(rotation=90)
plt.show()
```

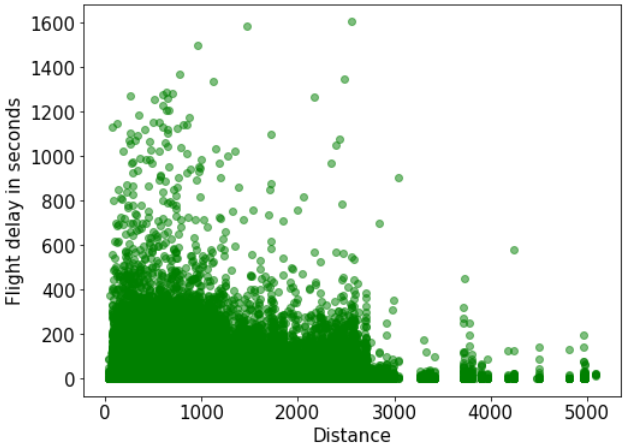



```
agg_dest_state = df.groupby(['DEST_STATE_ABR'], as_index=False).agg({'DEP_DEL15': 'sum', 'TAIL_NUM': 'count'})
agg_dest_state['frac'] = agg_dest_state['DEP_DEL15']/agg_dest_state['TAIL_NUM']
plt.figure(figsize=(20, 8))
plt.rcParams.update({'font.size': 15})
plt.bar(agg_dest_state['DEST_STATE_ABR'], agg_dest_state['frac'], color='green', alpha=0.5)
plt.xlabel("Flight destination state")
plt.ylabel("Fraction of flights delayed")
plt.xticks(rotation=90)
plt.show()
```



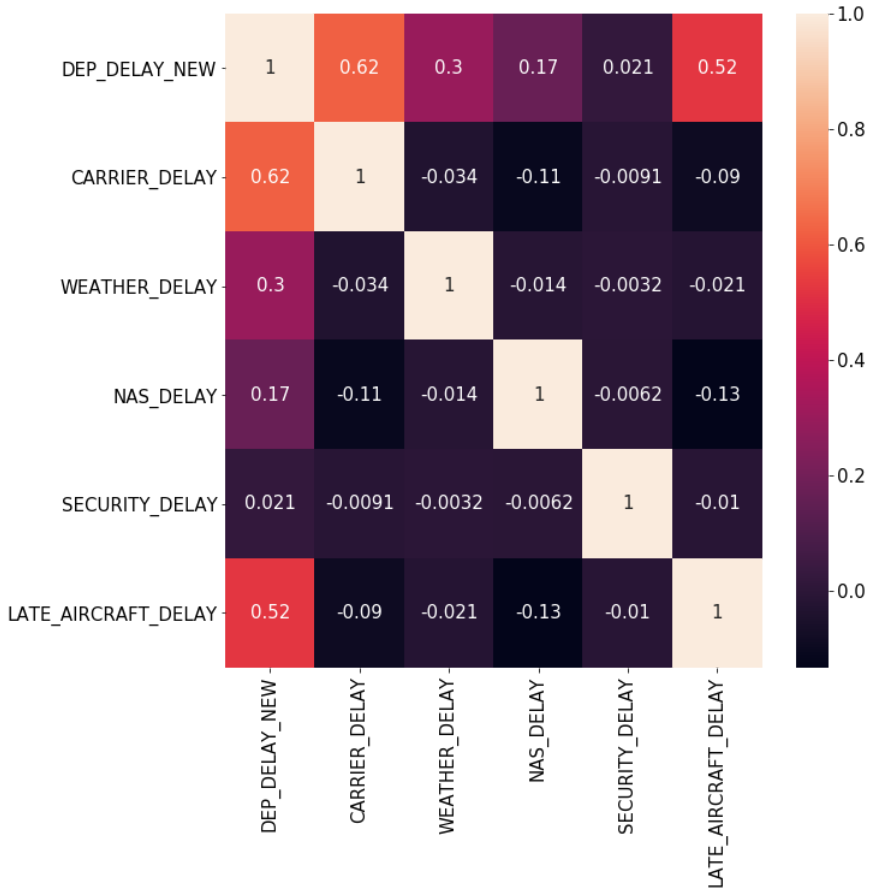
The following scatter plot shows how flight delay varies with the flight distance. As the distance increases, flight delay decreases. One potential reason is that the further a flight needs to travel, the more opportunity the flight has to compensate for delay by adjusting flight speed. Thus, shorter distance flights appear to suffer more than 15 minutes of delay. Since flight departure delay is negatively correlated with the travelled distance, distance can be used to predict the departure delay in the model.

```
plt.figure(figsize=(8, 6))
plt.plot(df['DISTANCE'], df['DEP_DELAY_NEW'], 'go', alpha=0.5)
plt.xlabel("Distance in miles")
plt.ylabel('Flight delay in minutes')
plt.show()
```



The following matrix shows how flight departure delay is correlated with various other type of delays, such as carrier delay, NAS delay, weather delay, security delay, and late aircraft delay. As per the correlation coefficient, the flight departure delay is positively correlated with all delay types. Among various delay categories, the flight departure delay is strongly correlated with carrier delay and late aircraft delay. Late aircraft delay represents delay caused by the propogation of delay from earlier delayed schedules. Since flight delay is highly correlated to the delay propagation, delay propagation may provide strong prediction signal for flight delays.

```
import pandas as pd
import seaborn as sn
corrMatrix = df[['DEP_DELAY_NEW', 'CARRIER_DELAY', 'WEATHER_DELAY', 'NAS_DELAY', 'SECURITY_DELAY', 'LATE_AIRCRAFT_DELAY']].corr()
plt.figure(figsize=(10, 10))
plt.rcParams.update({'font.size': 15})
sn.heatmap(corrMatrix, annot=True)
plt.show()
```



The following table shows the effects of delay propogation on flight delay. The probabilty a flight is delayed given that the previous flight leg was delayed is 0.064 and the probability of a flight is on time given that the flight was on time earlier in a day is 0.7804. Thus, the previous flight status indicates the current flight status for 84% of times. On the other hand, the probability a flight is delayed given that the previous flight leg was not delayed is 0.1178 and the probability a flight is on time given the previous flight was delayed is 0.037. Thus, 16% of times the current flight status is different from the previous flight status in a day. These results suggest that delay propogation can be a strong predictor of flight delays.

```
df2 = df.groupby(['PREV_DEP_DEL15', 'DEP_DEL15'], as_index=False)['TAIL_NUM'].count()
df2['fraction'] = df2['TAIL_NUM']/df.shape[0]
df2.columns = ['PREV_DEP_DEL15', 'DEP_DEL15', 'COUNTS', 'FRACTION']
df2
```

Out[35]:

	PREV_DEP_DEL15	DEP_DEL15	COUNTS	FRACTION
0	0.0	0.0	243503	0.780486
1	0.0	1.0	36764	0.117837
2	1.0	0.0	11744	0.037642
3	1.0	1.0	19978	0.064034

Join

Weather Data Set

We are going to split the weather data set into three tables. One is going to be for the visibility (horzn_dist), wind speed, and temperature. Another is for whether there was thunder or not. The last table is for snow and precipitation. There are different filters that need to be applied for each of these tables like missing data and null values. Below is an example of each data pull. In addition, we will also be imputing values for the precipitation and snow data. If the precipitation or snow data is null, we take the daily average of precipitation/snow.

```
weather.createOrReplaceTempView('weather')


weather_smpl = weather.sample(False, 0.000001)
weather_smpl.select([(count(when(isnan(c) | col(c).isNull(), c))/weather.count()).alias(c) for c in weather.columns if c not in
["DATE"]]).toPandas().head()
```

Out[9]:

	STATION	SOURCE	LATITUDE	LONGITUDE	ELEVATION	NAME	REPORT_TYPE	CALL_SIGN	QUALITY_CONTROL	WND	CIG	VIS	TMP	DEW	SLP	AW1	GA1
0	9.510157e-09	1.172919e-07	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

1 rows × 176 columns

```
# Thunder
thunder_d = spark.sql('''select station,
                                to_date(date_trunc('DD',if(minute(date) <30, to_timestamp(concat(string(to_date(date)), ' ' ,
string(hour(date)) , ':00:00')),to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date)) , ':00:00')) + interval 1
hour)),'YYYY-MM-DD') as updated_date_thunder,
                                                        '1' as thunder
                                                from weather
                                                where int(left(AW1,2)) =26 or int(left(AW1,2)) between 90 and 96''')

thunder_d.createOrReplaceTempView("thunder_d")
display(thunder_d)
```

	station	updated_date_thunder	thunder
1	7650099999	2016-07-12	1
2	7650099999	2016-09-27	1
3	7650099999	2016-10-14	1
4	7650099999	2016-10-14	1
5	7650099999	2016-10-14	1
6	7650099999	2016-11-24	1
7	7650099999	2016-11-24	1
8	7650099999	2016-11-25	1

Showing the first 1000 rows.

```
# Rain and Snow Hourly Data
rain_snow = spark.sql('''
select if(minute(date) <30, to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date))
,':00:00')),to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date)) , ':00:00')) + interval 1 hour) as updated_date,
station,
                                                max(int(left(AW1,2))) as weather_type,
                                                avg(int(left(AN1,3))) as snow_hours,
                                                avg(int(substring(AN1,5,4))) as snow_depth_in,
                                                avg(int(left(aa1,2))) as precipitation_period_time_min,
                                                avg(int(substring(aa1,4,4))) as precipitation_inches
                                          from weather
                                        where int(substring(aa1,4,4)) != 9999
                                        group by 1,2
''')

rain_snow.createOrReplaceTempView("rain_snow")
display(rain_snow)
```

	updated_date	station	weather_type	snow_hours	snow_depth_in	precipitation_period_time_min	precipitation_inches
1	2017-07-05T18:00:00.000+0000	25356099999	null	null	null	12	60
2	2017-01-25T18:00:00.000+0000	25378099999	null	null	null	12	0
3	2017-03-27T06:00:00.000+0000	25378099999	null	null	null	12	0
4	2017-01-16T18:00:00.000+0000	25399099999	null	null	null	12	0
5	2017-04-07T18:00:00.000+0000	25399099999	null	null	null	12	2
6	2017-03-29T21:00:00.000+0000	25400099999	null	null	null	12	0
7	2017-04-05T06:00:00.000+0000	25469099999	null	null	null	12	0
8	2017-05-01T21:00:00.000+0000	25503099999	null	null	null	12	0

Showing the first 1000 rows.

```
# Final table with hourly weather and the rain_snow table and the thunder indicator table

weather_final = spark.sql('''select a.date,
                                  if(minute(date) <30, to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date)) ,
':00:00')),to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date)) , ':00:00')) + interval 1 hour) as updated_date,
                                  concat(date, a.station) as date_station_id,
                                  a.station,
                                  avg(int(right(substring_index(wnd,',', 4),4))) as wind_speed,
                                  avg(int(substring_index(vis,',', 1))) as horzn_dist,
                                  avg(int(right(substring_index(vis,',', 2),1))) as horzn_dist_qual,
                                  avg(int(right(substring_index(tmp,',', 1),4))) as temp_c,
                                  avg(int(right(substring_index(tmp,',', 2),1))) as temp_c_qual,
                                  avg(int(substring_index(dew,',', 1))) as dew_point_temp,
                                  max(weather_type) as weather_type,
                                  avg(snow_hours) as snow_hours,
                                  avg(snow_depth_in) as snow_depth_in,
                                  avg(precipitation_period_time_min) as precipitation_period_time_min,
                                  avg(precipitation_inches) as precipitation_inches,
                                  ifnull(thunder,0) as thunder_ind
                                  from weather a
                                  left join rain_snow b
                                  on if(minute(date) <30, to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date)) ,
':00:00')),to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date)) , ':00:00')) + interval 1 hour) = b.updated_date and
a.station = b.station

                                  left join thunder_d d
                                 on  a.date = d.updated_date_thunder and a.station = d.station

                                  where report_type not in ('SOD','SOM')
                                  --and substring_index(wnd,',', 1) != 999
                                  and right(substring_index(wnd,',', 4),4) != 999
                                  and substring_index(vis,',', 1) != 9999
                                  and int(substring_index(vis,',', 1)) != 999999
                                  and right(substring_index(vis,',', 2),1) in (1,5)
                                  and right(substring_index(tmp,',', 2),1) in (1,5)
                                  and a.station is not null --= 72528014733
                                  group by 1,2,3,4,16
                                  --having avg(int(right(substring_index(vis,',', 2),1))) is not null''')

weather_final.createOrReplaceTempView("weather_final")
display(weather_final)


weather_final.count()

Out[30]: 308505050
  Cancelled
```

The next step we take is to get daily averages for the rain and snow as these columns had quite a bit of nulls compared to the visbility, temperature , and wind speed table.

```
# Take the weather data and find average per day

weather_day = spark.sql('''select
                                  to_date(date_trunc('DD',if(minute(date) <30, to_timestamp(concat(string(to_date(date)), ' ' ,
string(hour(date)) , ':00:00')),to_timestamp(concat(string(to_date(date)), ' ' , string(hour(date)) , ':00:00')) + interval 1
hour)),'YYYY-MM-DD') as updated_date_d,
                                  a.station,
                                  avg(int(left(AN1,3))) as snow_hours_d,
                                  avg(int(substring(AN1,5,4))) as snow_depth_in_d,
                                  avg(int(left(aa1,2))) as precipitation_period_time_min_d,
                                  avg(int(substring(aa1,4,4))) as precipitation_inches_d
                                  from weather a
                                  where int(substring(aa1,4,4)) != 9999 and int(substring(AN1,5,4)) !=9999
                                  group by 1,2
                                  order by avg(int(substring(AN1,5,4))) desc
                                  --having avg(int(right(substring_index(vis,',', 2),1))) is not null''')
weather_day.createOrReplaceTempView("weather_day")
display(weather_day)

  Cancelled
```

Once we have the daily averages we can create an if statement for the nulls in the weather hourly data set.

```
# Final join to bring in average of snow and precipitation by day and impute the nulls.

weather_final_2 = spark.sql('''select to_date(date_trunc('DD',date),'YYYY-MM-DD') as datee,
                                    updated_date,
                                    date_Station_id,
                                    a.station,
                                    wind_speed,
                                    horzn_dist,
                                    horzn_dist_qual,
                                    temp_c,
                                    temp_c_qual,
                                    dew_point_temp,
                                    weather_type,
                                    snow_hours,
                                    snow_depth_in,
                                    precipitation_period_time_min,
                                    precipitation_inches,
                                    precipitation_period_time_min_d,
                                    precipitation_inches_d,
                                    if(precipitation_inches/precipitation_period_time_min is null,
precipitation_inches_d/precipitation_period_time_min_d,precipitation_inches/precipitation_period_time_min) as
precipitation_inches_min_final,
                                    thunder_ind
                                from weather_final a

                                left join weather_day b
                                on date_trunc('DD',date) = b.updated_date_d and a.station = b.station
                                ''')
weather_final_2.createOrReplaceTempView('weather_final_2')
display(weather_final_2)

weather_final_2.count()
```

  Cancelled

Airline Data Set

Here we bring in the airlines data set with the dep_delay feature that we have created along with subtracting two hours from the original departure time to join with the weather data. We also round down for the departure time so that each new departure time that we are joining with the weather data is between 2-3 hours before the actual departure time.

```
# read in airlines_with_weather_station_utc_dep
airlines_with_weather_station_utc = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_station_utc")
airlines_with_weather_station_utc_dep.createOrReplaceTempView('airlines_with_weather_station_utc')
display(airlines_with_weather_station_utc)
airlines_with_weather_station_utc.count()
```

  Cancelled

```
# Hourly Data
# make a time stamp
airlines_with_updated_weather_hourly = spark.sql('''select fl_date,
concat(utc_dep_time,op_unique_carrier,origin_station_id, op_carrier_fl_num) as id,
                                        op_carrier_airline_id,
                                        op_unique_carrier,
                                        year,
                                        quarter,
                                        month,
                                        day_of_month,
                                        day_of_week,
                                        origin_city_name,
                                        op_carrier_fl_num,
                                        op_carrier,
                                        origin_State_abr,
                                        dest_state_abr,
                                        origin,
                                        dest,
                                        arr_delay_new,
                                        distance,
                                        weather_delay,
                                        nas_delay,
                                        security_delay,
                                        carrier_delay,
                                        late_aircraft_delay,

                    origin_station_id,
                    local_dep_time,
                    timezone,
                    utc_dep_time,
                    to_timestamp(concat(date(utc_dep_time), ' ' , hour(utc_dep_time), ':' ,'00', ':' , '00'))- interval 2 hours as
updated_dep_time,
                    dep_delay_new,
                    dep_del15,
                    dep_time_diff,
                    prev_dep_del15
                    from airlines_with_weather_station_utc
                    where origin_station_id is not null
                    group by 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32
                    order by utc_dep_time''')
# move it to a temp table
airlines_with_updated_weather_hourly.createOrReplaceTempView('airlines_with_updated_weather_hourly')

display(airlines_with_updated_weather_hourly)

airlines_with_updated_weather_hourly.count()
```

  Cancelled

Once we have our weather and airlines data we can finally inner join them to ensure we only include values that are on both tables. We also added in features like a weekend indicator which includes Friday, Saturday, and Sunday as a weekend. Lastly, we checked each column to ensure there was no nulls and if there was to drop the row. The final output is displayed below.

```
# read in the final weather data set and the airlines that was previously saved before running the join
airlines_with_weather_final = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_final")
airlines_with_weather_final .createOrReplaceTempView('airlines_with_weather_final ')
display(airlines_with_weather_final)
airlines_with_weather_final.count()
```

  Cancelled

We have a final dataset of 27,212,468 records. Finally we can split the data 90/10 for train/test and read it back in.

```
airlines_with_weather_train = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_train")
airlines_with_weather_train.createOrReplaceTempView('airlines_with_weather_train')
airlines_with_weather_test = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_test")
airlines_with_weather_test.createOrReplaceTempView('airlines_with_weather_test')


display(airlines_with_weather_train.take(5))
```

1	2015-01-01	2015-01-01109719393Baltimore, MD	19393	WN	2015	1	1	1
2	2015-01-01	2015-01-01109719790Los Angeles, CA	19790	DL	2015	1	1	1
3	2015-01-01	2015-01-01109719977Houston, TX	19977	UA	2015	1	1	1
4	2015-01-01	2015-01-01137619790Atlanta, GA	19790	DL	2015	1	1	1
5	2015-01-01	2015-01-01137620436Washington, DC	20436	F9	2015	1	1	1

Showing all 5 rows.

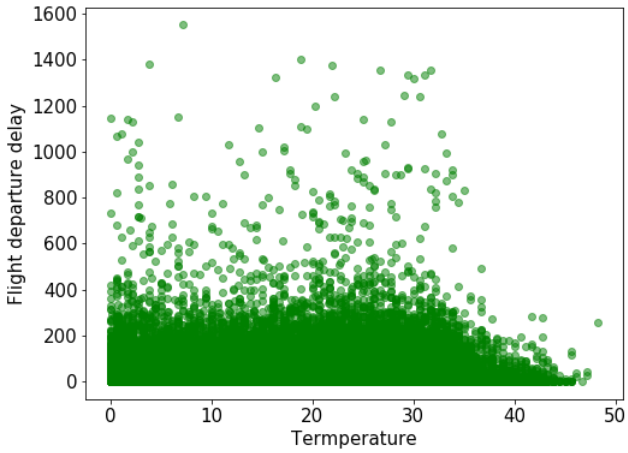
```
from pyspark.sql.functions import isnan, when, count, col
pydf3 = airlines_with_weather_train.sample(False, 0.01)
pydf3.select([(count(when(isnan(c) | col(c).isNull(), c))/pydf3.count()).alias(c) for c in pydf3.columns if c not in ['local_dep_time',
'utc_dep_time']]).toPandas().head()
```

Out[40]:

	fl_date	id	op_carrier_airline_id	op_unique_carrier	year	quarter	month	day_of_month	day_of_week	origin_city_name	op_carrier_fl_num	op_carrier	origin_State_a
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0

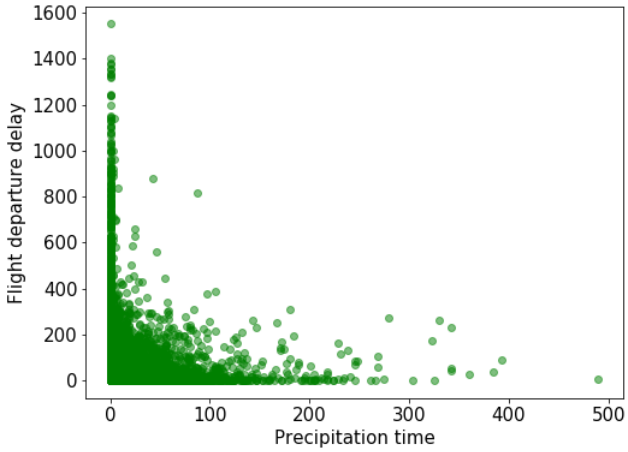
The following plot shows how flight departure delay varies with atmospheric temperature. Up to a certain temperature value, flight dealy is invarient of the temperature; however, after a certain termperature value, flight departure delay decreases. One potential justification is, precipitation, wind, snow, and rain mostly occur in colder termperature, which may leads to higher delay compared to warmer temperature. Thus, flight departure delay is negatively correlated with temperature which can help us explain variability in the flight departure prediction.

```
final_data=pydf3.toPandas()
final_data['temp_c'] = final_data['temp_c']/10
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 15})
#pd.plotting.scatter_matrix(final_data[['dep_delay_new', 'temp_c', 'precipitation_period_time_min']], alpha=0.5, figsize=(20, 20),
diagonal = 'kde')
plt.plot(final_data['temp_c'], final_data['dep_delay_new'], 'go', alpha=0.5)
plt.xlabel("Termperature in celcius")
plt.ylabel("Flight departure delay in minutes")
plt.show()
```



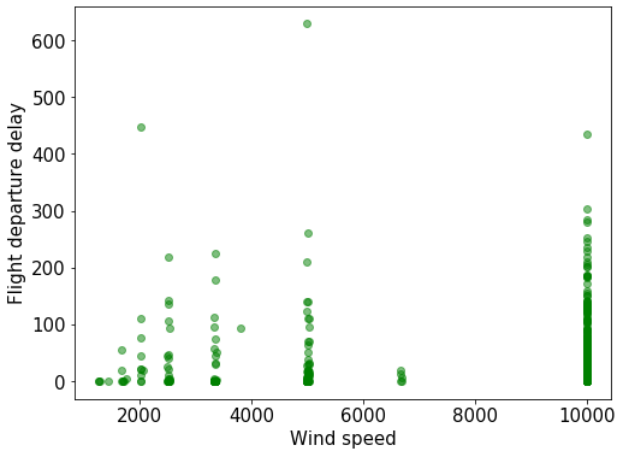
The following chart shows how the flight departure delay varies with the precipitation time. The flight departure delay decreases as the precipitation time increases. One possible explanation is that longer precipitation time allows more predictability in controlling flight schedules compared to the shorter precipitation time. Shorter precipitation time may be caused by sudden weather changes resulting in flight schedules that are difficult to control. Since flight departure delay is negatively correlated with the precipitation time, precipitation time can be used as a feature to explain delay.

```
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 15})
#pd.plotting.scatter_matrix(final_data[['dep_delay_new', 'temp_c', 'precipitation_period_time_min']], alpha=0.5, figsize=(20, 20),
diagonal = 'kde')
plt.plot(final_data['precipitation_period_time_min'], final_data['dep_delay_new'], 'go', alpha=0.5)
plt.xlabel("Precipitation time in minutes")
plt.ylabel("Flight departure delay in minutes")
plt.show()
```



The following graph shows how flight departure delay varies with wind speed. To extract out the correlation between the flight departure delay and wind speed, we remove the data points when wind is lower than 200. In the resultant graph, we can find that the flight departure delay increases with the wind speed. Thus, wind speed can be used to explain the variability in the departure delay.

```
final_data = final_data[final_data['wind_speed'] > 200]
plt.figure(figsize=(8, 6))
plt.rcParams.update({'font.size': 15})
#pd.plotting.scatter_matrix(final_data[['dep_delay_new', 'temp_c', 'precipitation_period_time_min']], alpha=0.5, figsize=(20, 20),
diagonal = 'kde')
plt.plot(final_data['wind_speed'], final_data['dep_delay_new'], 'go', alpha=0.5)
plt.xlabel("Wind speed")
plt.ylabel("Flight departure delay in minutes")
plt.show()
```



The following table shows the number of flights delayed during weekdays and weekends. In the selected data sample, 15% of flights are delayed during weekdays and only 7% of the flights are delayed duging weekend. Thus, flights scheduled over weekend appears to be on time by a factor of 2. Thus, weather the day of a week is either weekday or weekend can be used to predict the flight departure delay in the model.

```
df_delay_daytype = final_data.groupby(['weekend'], as_index=False).agg({'dep_del15': 'sum'})
df_delay_daytype.columns = ['weekend', 'delay_counts']
df_delay_daytype['ratio'] = df_delay_daytype['delay_counts']/final_data.shape[0]
df_delay_daytype
```

Out[46]:

	weekend	delay_counts	ratio
0	0	196.0	0.154818
1	1	99.0	0.078199

Algorithm Exploration

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.tuning import ParamGridBuilder
import numpy as np
from pyspark.ml.tuning import CrossValidator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.feature import StandardScaler, StringIndexer, OneHotEncoder, VectorAssembler
```

Random Forest model with hyper parameter tuning

```
# Reading training and testing dataset
airlines_with_weather_train = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_train")
airlines_with_weather_train.createOrReplaceTempView('airlines_with_weather_train')
airlines_with_weather_test = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_test")
airlines_with_weather_test.createOrReplaceTempView('airlines_with_weather_test')
airlines_with_weather_train = airlines_with_weather_train.withColumn("thunder_ind",
airlines_with_weather_train.thunder_ind.cast("double"))
airlines_with_weather_test = airlines_with_weather_test.withColumn("thunder_ind", airlines_with_weather_test.thunder_ind.cast("double"))

# Selecting the features of interest
df_train = airlines_with_weather_train.select('dep_del15', 'month', 'day_of_month', 'day_of_week', 'op_carrier', 'distance',
'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min', 'weekend')
df_test = airlines_with_weather_test.select('dep_del15', 'month', 'day_of_month', 'day_of_week', 'op_carrier', 'distance',
'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min', 'weekend')

# Sampling the subset of data from the training and testing data set
# NOTE: Due to slowness of the cluster and limitation of resources, We are considering the fraction of data for the model training and
testing purpose.
df_train = df_train.sample(False, 0.0001)
df_test = df_test.sample(False, 0.0001)

# Creating one-hot encoding for the categorial
categoricalColumns = ['month', 'day_of_month', 'day_of_week', 'op_carrier']
numericCols = ['distance', 'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min',
'weekend']
for variable in categoricalColumns:
    #converts string variables to numerical indices
    indexer = StringIndexer(inputCol=variable, outputCol=variable+"index")
    df_train = indexer.fit(df_train).transform(df_train)
    df_test = indexer.fit(df_test).transform(df_test)
    #explodes the now numerical categorical variables into binary variables
    encoder = OneHotEncoder(inputCol=variable+"index", outputCol=variable+"vec")
    df_train = encoder.fit(df_train).transform(df_train)
    df_test = encoder.fit(df_test).transform(df_test)

# Creating a vector of features for model training in ML-lib
assembler = VectorAssembler(
    inputCols=['monthvec',
               'day_of_monthvec',
               'day_of_weekvec',
               'op_carriervec',
               'distance',
               'prev_dep_del15',
               'thunder_ind',
               'temp_c',
               'wind_speed',
               'horzn_dist',
               'precipitation_period_time_min',
               'weekend'
              ],
    outputCol="features")




# Crearing a parameter grid for hyper-parameter tunning in random forest
# Tunnign the number of trees and the maximum depth of each tree

rf = RandomForestClassifier(labelCol="dep_del15", featuresCol="features")
pipeline = Pipeline(stages=[assembler, rf])

paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees, [int(x) for x in np.linspace(start = 1, stop = 100, num = 5)]) \
    .addGrid(rf.maxDepth, [int(x) for x in np.linspace(start = 5, stop = 20, num = 4)]) \
    .build()


# Creating binary classification evaluator object cross validation object for grid search
evaluator = BinaryClassificationEvaluator().setLabelCol("dep_del15")

crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator,
                          numFolds=3)
```

```
df_train_updated = df_train.select('dep_del15', 'distance', 'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist',
'precipitation_period_time_min', 'weekend', 'monthvec', 'day_of_monthvec', 'day_of_weekvec', 'op_carriervec')

df_test_updated = df_test.select('dep_del15', 'distance', 'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist',
'precipitation_period_time_min', 'weekend', 'monthvec', 'day_of_monthvec', 'day_of_weekvec', 'op_carriervec')



# Training random forest model for various hyper parameters
cvModel = crossval.fit(df_train_updated)

MLlib will automatically track trials in MLflow. After your tuning fit() call has completed, view the MLflow UI to see logged runs.

# Predicting flight delays on the test dataset.
predictions = cvModel.transform(df_test_updated)
cm = predictions.select("dep_del15", "prediction")


# Calculating true positive, true negative, false positive, and false negatives
# TP = cm.filter((cm.dep_del15 == cm.prediction) & (cm.prediction == 1)).count()
# TN = cm.filter((cm.dep_del15 == cm.prediction) & (cm.prediction == 0)).count()
# FP = cm.filter((cm.dep_del15 != cm.prediction) & (cm.prediction == 1)).count()
# FN = cm.filter((cm.dep_del15 != cm.prediction) & (cm.prediction == 0)).count()


# Calculating accuract, recall, f1 score, and precision
# accuracy = (TP + TN) / (TP + TN + FP + FN)
# recall = TP/(TP + FN)
# precision = TP/(TP + FP)
# f1_score = 2 * ((precision * recall) / (precision + recall))
# print('Recall - ', recall)
# print('Precision - ', precision)
# print('accuracy - ', accuracy)
# print('f1_score - ', f1_score)


# Finding the best hyper parameters optimizing the model performance
bestPipeline = cvModel.bestModel
bestModel = bestPipeline.stages[1]
print('numTrees - ', bestModel.getNumTrees)
print('maxDepth - ', bestModel.getOrDefault('maxDepth'))

numTrees -  100
maxDepth -  5
```

Testing Logistic Regression, Decision Tree, and Random Forest modeling code

```
airlines_with_weather_train = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_train")
airlines_with_weather_train.createOrReplaceTempView('airlines_with_weather_train')

airlines_with_weather_test = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_test")
airlines_with_weather_test.createOrReplaceTempView('airlines_with_weather_test')

airlines_with_weather_train = airlines_with_weather_train.withColumn("thunder_ind",
airlines_with_weather_train.thunder_ind.cast("double"))
airlines_with_weather_test = airlines_with_weather_test.withColumn("thunder_ind", airlines_with_weather_test.thunder_ind.cast("double"))
```

```
from pyspark.ml.feature import StandardScaler, StringIndexer, OneHotEncoder, VectorAssembler

df_train = airlines_with_weather_train.select('dep_del15', 'month', 'day_of_month', 'day_of_week', 'op_carrier', 'distance',
'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min', 'num_flights_aircraft_per_day',
'weekend')


df_test = airlines_with_weather_test.select('dep_del15', 'month', 'day_of_month', 'day_of_week', 'op_carrier', 'distance',
'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min', 'num_flights_aircraft_per_day',
'weekend')

categoricalColumns = ['month', 'day_of_month', 'day_of_week', 'op_carrier']
numericCols = ['distance', 'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min',
'num_flights_aircraft_per_day', 'weekend']

for variable in categoricalColumns:
    #converts string variables to numerical indices e.g. January to 1, February to 2 etc.
    indexer = StringIndexer(inputCol=variable, outputCol=variable+"index")
    df_train = indexer.fit(df_train).transform(df_train)
    df_test = indexer.fit(df_test).transform(df_test)
    #explodes the now numerical categorical variables into binary variables
    encoder = OneHotEncoder(inputCol=variable+"index", outputCol=variable+"vec")
    df_train = encoder.fit(df_train).transform(df_train)
    df_test = encoder.fit(df_test).transform(df_test)

assembler = VectorAssembler(
    inputCols=['monthvec',
               'day_of_monthvec',
               'day_of_weekvec',
               'op_carriervec',
               'distance',
               'prev_dep_del15',
               'thunder_ind',
               'temp_c',
               'wind_speed',
               'horzn_dist',
               'precipitation_period_time_min',
               'weekend'
              ],
    outputCol="vectorized_features")

df_train = assembler.transform(df_train)
df_test = assembler.transform(df_test)

standardscaler = StandardScaler().setInputCol("vectorized_features").setOutputCol("features")
df_train = standardscaler.fit(df_train).transform(df_train)
df_test = standardscaler.fit(df_test).transform(df_test)


### ALGORITHMS EXPLORED

# LOGISTIC REGRESSION
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(featuresCol = 'features', labelCol = 'dep_del15', maxIter = 3)
lrModel = lr.fit(df_train)
predictions = lrModel.transform(df_test)
predictions.createOrReplaceTempView('predictions')

## Confusion matrix results for Logistic Regression Model
q = 'SELECT dep_del15 as actual, prediction, count(*) FROM predictions GROUP BY 1,2'
display(sqlContext.sql(q))
```

	actual	prediction	count(1)
1	1	1	176572
2	0	1	103913
3	1	0	322483
4	0	0	2118384

Showing all 4 rows.

```
# DECISION TREE
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol = 'vectorized_features', labelCol = 'dep_del15', maxDepth = 3)
dtModel = dt.fit(df_train)
predictions = dtModel.transform(df_test)
predictions.createOrReplaceTempView('predictions')

## Confusion matrix results for Decision Tree Model
q = 'SELECT dep_del15 as actual, prediction, count(*) FROM predictions GROUP BY 1,2'
display(sqlContext.sql(q))
```

	actual	prediction	count(1)
1	1	1	178706
2	0	1	106108
3	1	0	320349
4	0	0	2116189

Showing all 4 rows.

```
# RANDOM FOREST
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol = 'features', labelCol = 'dep_del15', numTrees = 10, maxDepth = 10)
rfModel = rf.fit(df_train)
predictions = rfModel.transform(df_test)
predictions.createOrReplaceTempView('predictions')

## Confusion matrix results for Random Forest Model
q = 'SELECT dep_del15 as actual, prediction, count(*) FROM predictions GROUP BY 1,2'
display(sqlContext.sql(q))
```

	actual	prediction	count(1)
1	1	1	84167
2	0	1	40248
3	1	0	414888
4	0	0	2182049

Showing all 4 rows.

Algorithm Implementation

A simple toy example has been created to demonstrate the underlying details of the decision tree algorithm. This implementation of the decision tree predicts if an applicant receives an admissions offer from MIDS. The features included are the applicant's undergrad school (Cal or Stanford) and GPA (0 to 4.0). A dataframe toy example is created and the decision tree is implemented through Pyspark. We have displayed the decision tree stages and have run a single prediction test case. A detailed hand calculation of the decision tree algorithm is provided and the results match the output from the model fitted in Pyspark.

```
# generate toy data for decision tree classification
import pandas as pd

# features are undergrad college (cal/stanford) and gpa (from 0 to 4.0)
# dependent variable is if the applicant receives a MIDS offer of admission
x = ["cal","cal", "cal","cal","stanford","stanford","stanford"]
y = [ 3.1, 3.2, 3.3, 3.9, 3.5, 3.7, 4.0]
result = ["no", "yes", "yes", "yes", "no", "no", "yes"]

# create dataframe
pd_df = pd.DataFrame([x,y,result]).transpose()
pd_df.columns = ['undergrad', 'gpa', 'offer']
toy_df = spark.createDataFrame(pd_df)
display(toy_df)
```

	undergrad	gpa	offer
1	cal	3.1	no
2			

3	cal	3.3	yes
4	cal	3.9	yes
5	stanford	3.5	no
6	stanford	3.7	no
7	stanford	4	yes

Showing all 7 rows.

```
# prepare data with category indexing, one-hot encoding, and vector assembler
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler

# identify categorical and numeric features
categoricalCols = ['undergrad']
numericCols = ['gpa']

# convert string variables to numeric indices using string indexer
# convert numeric indices into binary variables using one hot encoder
for variable in categoricalCols:
    indexer = StringIndexer(inputCol = variable, outputCol = variable + 'index')
    toy_df = indexer.fit(toy_df).transform(toy_df)
    encoder = OneHotEncoder(inputCols=[indexer.getOutputCol()], outputCols=[variable + "vec"])
    toy_df = encoder.fit(toy_df).transform(toy_df)

# index the dependent variable
label_indexer = StringIndexer(inputCol = 'offer', outputCol = 'label')
toy_df = label_indexer.fit(toy_df).transform(toy_df)

# assemble vectors from categorical and numeric features
assemblerInputs = [c + "vec" for c in categoricalCols] + numericCols
assembler = VectorAssembler(inputCols = assemblerInputs, outputCol = "features")
toy_df = assembler.transform(toy_df)

# show the dataframe
print("Training Dataset Count: " + str(toy_df.count()))
toy_df.select("undergrad", "gpa", "offer", "features", "label").show()
```

```
Training Dataset Count: 7
+---------+---+-----+---------+-----+
|undergrad|gpa|offer| features|label|
+---------+---+-----+---------+-----+
|      cal|3.1|   no|[1.0,3.1]|  1.0|
|      cal|3.2|  yes|[1.0,3.2]|  0.0|
|      cal|3.3|  yes|[1.0,3.3]|  0.0|
|      cal|3.9|  yes|[1.0,3.9]|  0.0|
| stanford|3.5|   no|[0.0,3.5]|  1.0|
| stanford|3.7|   no|[0.0,3.7]|  1.0|
| stanford|4.0|  yes|[0.0,4.0]|  0.0|
+---------+---+-----+---------+-----+
```

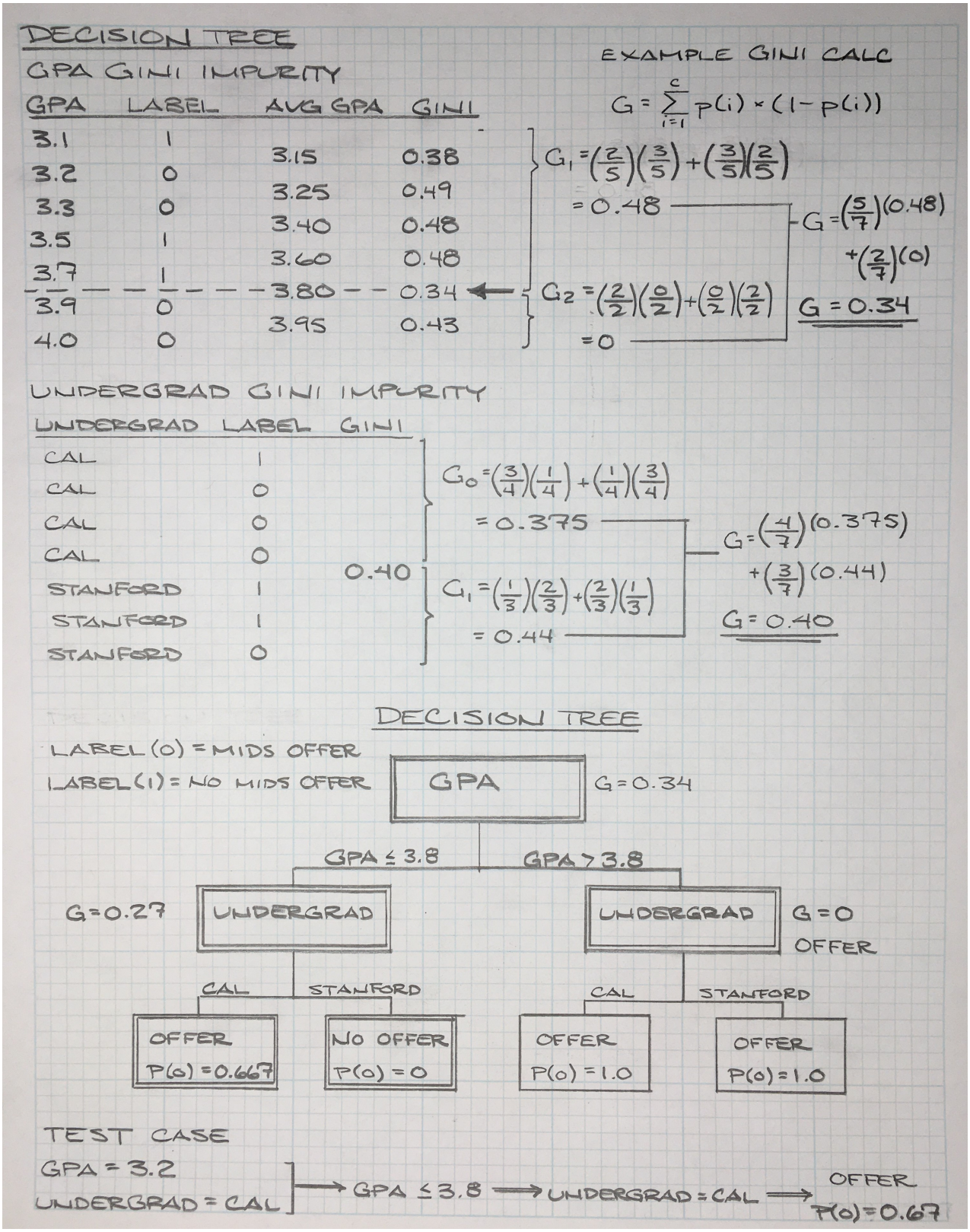
```
# Decision Tree Model
# Fit the training model
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol = 'features', labelCol = 'label', maxDepth = 2)
dtModel = dt.fit(toy_df)


# show the decision tree
dtModel_stages  = dtModel
print(dtModel_stages.toDebugString)
```

```
DecisionTreeClassificationModel: uid=DecisionTreeClassifier_47b25e1ed9c6, depth=2, numNodes=5, numClasses=2, numFeatures=2
  If (feature 1 <= 3.8)
   If (feature 0 in {1.0})
    Predict: 0.0
   Else (feature 0 not in {1.0})
    Predict: 1.0
  Else (feature 1 > 3.8)
   Predict: 0.0
```

```
# run a test case for cal undergrad with 3.2 GPA
import pyspark.sql.functions as f

#create test case
test_df = toy_df.filter(f.col('gpa')==3.2)

# prediction based on test data
predictions_dt = dtModel.transform(test_df)
predictions_dt.select("label", "features", "probability", "prediction").show(10)
```

```
+-----+---------+--------------------+----------+
|label| features|         probability|prediction|
+-----+---------+--------------------+----------+
|  0.0|[1.0,3.2]|[0.66666666666666...|       0.0|
+-----+---------+--------------------+----------+
```

Decision Tree hand calculation



Here is how many algorithm models we ran and their respective accuracy, recall and precision.

B	C	D	E	F	G	H	I	J
Model	**Parameter**	**True Positive**	**False Positive**	**False Negative**	**True Negative**	**Accuracy**	**Recall**	**Precision**
Decision Tree	maxDepth = 3	178,706	106,108	320,349	2,116,189	84.33%	35.81%	62.74%
Decision Tree	maxDepth = 10	172,955	99,344	326,100	2,122,953	84.37%	34.66%	63.52%
Decision Tree	**maxDepth = 10, minInstancesPerNode = 1000**	**178,624**	**106,008**	**320,431**	**2,116,289**	**84.33%**	**35.79%**	**62.76%**
Decision Tree	maxDepth = 5, minInstancesPerNode = 300	178,624	106,008	320,431	2,116,289	84.33%	35.79%	62.76%
Decision Tree	maxDepth = 5, minInstancesPerNode = 1000	178,624	106,008	320,431	2,116,289	84.33%	35.79%	62.76%
Decision Tree	maxDepth = 5, minInstancesPerNode = 10000	178,706	106,108	320,349	2,116,189	84.33%	35.81%	62.74%
Decision Tree	maxDepth = 5, minInstancesPerNode = 25000	178,706	106,108	320,349	2,116,189	84.33%	35.81%	62.74%
Logistic Regression	maxIter = 5	173,527	100,638	325,528	2,121,659	84.34%	34.77%	63.29%
Logistic Regression	maxIter = 10	171,072	98,414	327,983	2,123,883	84.33%	34.28%	63.48%
Random Forest	numTrees = 100, maxDepth = 5	9	0	499,046	2,222,297	81.66%	0.00%	100.00%
Random Forest	numTrees = 10, maxDepth = 5	2,395	879	496,660	2,221,418	81.72%	0.48%	73.15%
Random Forest	numTrees = 1, maxDepth = 5	178,623	106,008	320,432	2,116,289	84.33%	35.79%	62.76%
Random Forest	numTrees = 10, maxDepth = 5, minInstancesPerNode = 1000	6,788	2,925	492,267	2,219,372	81.80%	1.36%	69.89%
Random Forest	numTrees = 20, maxDepth = 5, minInstancesPerNode = 1000, subsamplingRate = 0.8	60	14	498,995	2,222,283	81.66%	0.01%	81.08%
Random Forest	numTrees = 20, maxDepth = 10, minInstancesPerNode = 1000	145,059	78,663	353,996	2,143,634	84.10%	29.07%	64.84%
Random Forest	numTrees = 30, maxDepth = 10	93,692	41,516	405,363	2,180,781	83.58%	18.77%	69.29%
Logistic Regression	maxIter = 5	173,533	100,666	325,522	2,121,631	84.34%	34.77%	63.29%
Decision Tree	maxDepth = 3	178,706	106,108	320,349	2,116,189	84.33%	35.81%	62.74%
ducted on 10% of data]								
Logistic Regression	maxIter = 10	29,168	16,620	60,807	383,526	84.20%	32.42%	63.70%
Decision Tree	maxDepth = 5, minInstancesPerNode = 25000	29,621	15,436	60,354	384,710	84.54%	32.92%	65.74%
Random Forest	numTrees = 1, maxDepth = 5	31,371	16,044	58,604	384,102	84.77%	34.87%	66.16%
Gradient Boosted Trees	maxIter = 10	31,059	15,479	58,916	384,667	84.82%	34.52%	66.74%

Conclusions

Our final model is trained and tested below. The Decision Tree Classifier has taken varying times to complete training (usually depending on what the current load on the cluster is at the time of training). At the quickest, the model has completed training in 2.81 minutes, while the longest training time took 13.43 minutes.

With a focus on the Recall rate (out of all the delayed flights, how many were we able to predict), the performance of the model is:

Accuracy: 84%
Recall: 35%
Precision: 63%

The practical implications resulting from the performance of the model are quite effective. In 2017, United and United Express operated more than 1.6 million flights carrying more than 148 million customers. Assuming a similar delay rate as our dataset (~20%), about 320,000 of these flights experienced a delay. Using our model, we would've correctly predicted 112,000 of these flights to be delayed ahead of time. With proper care taken for a predicted delay (contact customers, make schedule adjustments, etc.), United can potentially be saving $132M.

```
airlines_with_weather_train = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_train")
airlines_with_weather_train.createOrReplaceTempView('airlines_with_weather_train')

airlines_with_weather_test = spark.read.option("header",
"true").parquet(f"dbfs:/user/nathan.nusaputra@ischool.berkeley.edu/FINAL_PROJECT/airlines_with_weather_test")
airlines_with_weather_test.createOrReplaceTempView('airlines_with_weather_test')

airlines_with_weather_train = airlines_with_weather_train.withColumn("thunder_ind",
airlines_with_weather_train.thunder_ind.cast("double"))
airlines_with_weather_test = airlines_with_weather_test.withColumn("thunder_ind", airlines_with_weather_test.thunder_ind.cast("double"))
```

```
from pyspark.ml.feature import StandardScaler, StringIndexer, OneHotEncoder, VectorAssembler

df_train = airlines_with_weather_train.select('dep_del15', 'month', 'day_of_month', 'day_of_week', 'op_carrier', 'distance',
'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min', 'num_flights_aircraft_per_day',
'weekend')


df_test = airlines_with_weather_test.select('dep_del15', 'month', 'day_of_month', 'day_of_week', 'op_carrier', 'distance',
'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min', 'num_flights_aircraft_per_day',
'weekend')

categoricalColumns = ['month', 'day_of_month', 'day_of_week', 'op_carrier']
numericCols = ['distance', 'prev_dep_del15', 'thunder_ind', 'temp_c', 'wind_speed', 'horzn_dist', 'precipitation_period_time_min',
'num_flights_aircraft_per_day', 'weekend']

for variable in categoricalColumns:
    #converts string variables to numerical indices e.g. January to 1, February to 2 etc.
    indexer = StringIndexer(inputCol=variable, outputCol=variable+"index")
    df_train = indexer.fit(df_train).transform(df_train)
    df_test = indexer.fit(df_test).transform(df_test)
    #explodes the now numerical categorical variables into binary variables
    encoder = OneHotEncoder(inputCol=variable+"index", outputCol=variable+"vec")
    df_train = encoder.fit(df_train).transform(df_train)
    df_test = encoder.fit(df_test).transform(df_test)

assembler = VectorAssembler(
    inputCols=['monthvec',
               'day_of_monthvec',
               'day_of_weekvec',
               'op_carriervec',
               'distance',
               'prev_dep_del15',
               'thunder_ind',
               'temp_c',
               'wind_speed',
               'horzn_dist',
               'precipitation_period_time_min',
               'weekend'
              ],
    outputCol="vectorized_features")

df_train = assembler.transform(df_train)
df_test = assembler.transform(df_test)

standardscaler = StandardScaler().setInputCol("vectorized_features").setOutputCol("features")
df_train = standardscaler.fit(df_train).transform(df_train)
df_test = standardscaler.fit(df_test).transform(df_test)


from pyspark.ml.classification import DecisionTreeClassifier

dt = DecisionTreeClassifier(featuresCol = 'vectorized_features', labelCol = 'dep_del15', maxDepth = 10, minInstancesPerNode = 1000)
dtModel = dt.fit(df_train)
predictions = dtModel.transform(df_test)
predictions.createOrReplaceTempView('predictions')


q = 'SELECT dep_del15 as actual, prediction, count(*) FROM predictions GROUP BY 1,2'
display(sqlContext.sql(q))
```

	actual	prediction	count(1)
1	1	1	172810
2	0	1	99028
3	1	0	326245
4	0	0	2123269

Showing all 4 rows.

```
dtModel.featureImportances

Out[37]: SparseVector(73, {2: 0.0001, 6: 0.0, 9: 0.0, 10: 0.0001, 47: 0.0049, 48: 0.0001, 53: 0.0054, 54: 0.0002, 55: 0.0002, 65: 0.003
2, 66: 0.9587, 68: 0.0028, 69: 0.0014, 70: 0.0006, 71: 0.0223})
```

Application of Course Concepts

MapReduce - MapReduce is implemented in our code multiple times in the background whenever we ran a spark query. Examples of this include the .describe() function, joins, printSchema(), etc. In addition, MapReduce is also running in the background when running our models using mllib from setting the parameters of the algorithm, fitting the model to the data, and finally making predictions.

Feature Engineering - A huge portion of the data modeling process is spent on formatting and engineering the dataset in the proper way for the algorithm to train on it accordingly. In this project, we spent a sizable amount of effort and time doing just this. Before passing our dataset into the modeling functions, we had to go through a process of vectorizing and assembling our features into one column. This was a three step process that required us to take into account every categorical variable that we wanted to use and one-hot encode them into its own vector. Once we had a vector for each categorical variable (where the length of the vector is equal to the amount of unique values in the variable and 1 is populated at the spot of the corresponding value), we combined the vectors into one long vector. Additionally, we also appended the numerical values onto this main vector. This process was called Assembling, and resembled the work we did in previous homeworks. Only after creating a new column that contained this large vector were we able to pass it into the modeling function for the algorithm to begin training.

Decision Tree - While many different algorithms were explored, we ultimately settled on Decision Tree as our final model. The decision tree has traditionally been a popular machine learning tool largely because it is easy to implement and convey results. As one of the last concepts covered during this course, the final project presented an opportunity to gain a deeper understanding of decision trees. Of importance, is the process in which decision trees determine the optimal root node and split using the Gini Index calculation, which provides the probability of misclassifying a variable when selected randomly. The Gini calculation can be applied to both numerical and categorical features. The optimal feature is the one with the lowest Gini value. This binary splitting continues until the tree reaches a max depth or the minimum number of inputs on each leaf.

Here us a link to our detailed notebook: https://dbc-c4580dc0-018b.cloud.databricks.com/?o=8229810859276230#notebook/231034620063430/command/2127515060491002 (https://dbc-c4580dc0-018b.cloud.databricks.com/?o=8229810859276230#notebook/231034620063430/command/2127515060491002)