Flight Delays Project Overview

Digital Campus (https://digitalcampus.instructure.com/courses/14487/pages/mids-w261-final-project-dataset-and-cluster?module_item_id=1711799)

- Flight delays create problems in scheduling for airlines and airports, leading to passenger inconvenience and huge economic losses. As a result, there is growing interest in predicting flight delays to optimize operations and improve customer satisfaction. In this project, you will predict flight delays using the provided datasets. You will get to a frame machine learning problem that will benefit your main stakeholder (e.g., an airline, an airport, frequent flyers, government), and corresponding machine learning metrics and domain-specific metrics. For example, one could frame the problem to be tackled in this project as follows:
- Our primary customer is the consumer. As a result, we will focus on **predicting departure delays (no delay), where a delay is defined as a 15-minute delay (or greater)** concerning the planned departure time. This **prediction should be made TWO HOURS before departure (DEP_DELAY_Double < 120)** (thereby giving airlines and airports time to regroup and give passengers a heads-up on a delay). We will report progress in terms of F1-Beta, sensitivity, specificityLinks to an external site., etc. As you can imagine, this problem could be framed in many different ways leading to other products, engineering challenges, and metrics for success. Please list some of these alternatives and their potential benefits and challenges in your project proposals.

OTPW - Exploratory Data Analysis and Preprocessing

```
import time
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.cm import get_cmap
from statsmodels.tsa.seasonal import seasonal_decompose
from datetime import datetime, timedelta
from pyspark.sql import functions as F
from pyspark.sql.window import Window
from pyspark.sql.functions import col, count, when, lit, round, to_date, to_timestamp, corr, isnan, udf, desc, dayofyear, monotonically_increasing_id, sum as _sum, isnan, pandas_udf,
PandasUDFType, avg, expr, sin, cos, radians, collect_list
from pyspark.ml.clustering import KMeans
from pyspark.sql.types import IntegerType, DoubleType
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler, StandardScaler
from pyspark.ml import Pipeline
import warnings
import mlflow

print(mlflow.__version__)
spark.conf.set("spark.databricks.mlflow.trackMLlib.enabled", 'true')

# Suppress only the FutureWarning about pandas DataFrame append deprecation
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
2.5.0
```

```
## Connect to Team Cloud Storage
blob_container  = "w261storage"              # The name of your container created in https://portal.azure.com
storage_account = "w261rtang"                # The name of your Storage account created in https://portal.azure.com
secret_scope    = "team_2_1_scope"           # The name of the scope created in your local computer using the Databricks CLI
secret_key      = "team_2_1_key"             # The name of the secret key created in your local computer using the Databricks CLI
team_blob_url   = f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"  #points to the root of your team storage bucket

# the 261 course blob storage is mounted here.
mids261_mount_path      = "/mnt/mids-w261"

# SAS Token: Grant the team limited access to Azure Storage resources
spark.conf.set(
  f"fs.azure.sas.{blob_container}.{storage_account}.blob.core.windows.net",
  dbutils.secrets.get(scope = secret_scope, key = secret_key)
)
# display(dbutils.fs.ls(f"{team_blob_url}"))
```

Store Raw Data in Parquet and Delta Lake

02-Delta Lake Workshop - Including ML (https://pages.databricks.com/rs/094-YMS-629/images/02-Delta%20Lake%20Workshop%20-%20Including%20ML.html)

Delta Lake is very efficient in updating data transformations. It allows updates to only filtered columns, rows, or individual values, while retaining the pre-processed data that does not require any updates.

```
def checkpoint_df_blob(df, check_pt_name, format="delta"):
  blob_container  = "w261storage"       # The name of your container created in https://portal.azure.com
  storage_account = "w261rtang"  # The name of your Storage account created in https://portal.azure.com
  secret_scope    = "team_2_1_scope"           # The name of the scope created in your local computer using the Databricks CLI
  secret_key      = "team_2_1_key"             # The name of the secret key created in your local computer using the Databricks CLI
  team_blob_url   = f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"  #points to the root of your team storage bucket

  # SAS Token: Grant the team limited access to Azure Storage resources
  spark.conf.set(
    f"fs.azure.sas.{blob_container}.{storage_account}.blob.core.windows.net",
    dbutils.secrets.get(scope = secret_scope, key = secret_key)
  )
  # start to checkpoint
  if (format == "delta"):
    df.write.format('delta').save(f'{team_blob_url}/{check_pt_name}.deltalake')
  else:
    df.write.parquet(f"{team_blob_url}/{check_pt_name}.parquet")


def load_df_blob(check_pt_name, format="delta"):
  blob_container  = "w261storage"              # The name of your container created in https://portal.azure.com
  storage_account = "w261rtang"                # The name of your Storage account created in https://portal.azure.com
  secret_scope    = "team_2_1_scope"           # The name of the scope created in your local computer using the Databricks CLI
  secret_key      = "team_2_1_key"             # The name of the secret key created in your local computer using the Databricks CLI
  team_blob_url   = f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"  #points to the root of your team storage bucket

  # SAS Token: Grant the team limited access to Azure Storage resources
  spark.conf.set(
    f"fs.azure.sas.{blob_container}.{storage_account}.blob.core.windows.net",
    dbutils.secrets.get(scope = secret_scope, key = secret_key)
  )
  # loading dataframe
  if (format == "delta"):
    return spark.read.format("delta").load(f'{team_blob_url}/{check_pt_name}.deltalake').cache()
  else:
    return spark.read.parquet(f'{team_blob_url}/{check_pt_name}.parquet').cache()
```

Show code

Select DF

- delta_otpw_3m_2015
- delta_otpw_1yr
- delta_otpw_3yr
- delta_otpw_5yr

```
# SELECTED_DF = delta_otpw_1yr
# display(df_5y_train.limit(2))
```

Data Split

```
df_5y_train = delta_otpw_5yr
```

EDA

- Null check (Missing Values Count, Missing Values in %)
- Select relevant columns for potential feature selection and creation
- Descriptive statistics
- Check the data types of each column and convert them to be compatible with machine learning models.

Helper Functions

```python
def print_df_shape(df, df_name):
  nrows, ncols = len(df.columns), df.count()
  print(f'{df_name} contains {ncols} rows & {nrows} columns')

def check_data_type(df, col_name):
  data_types = df.dtypes
  date_type = [dtype for col, dtype in data_types if col == col_name][0]
  print(f"The data type of {col_name} column is '{date_type}'")



def extract_distinct_values_singleCol(df, col_name):
  unique_val = df.select(col_name).distinct().collect()
  unique_val_list = [row[col_name] for row in unique_val]
  print(f'Length of unique values = {len(unique_val_list)}\nUnique values of "{col_name}":\n{unique_val_list}')

def extract_distinct_values_multiCol(df, cols, nrows=10):
    # Create an empty DataFrame to store results
    pdf = pd.DataFrame(columns=['col_name', 'sample_unique_val', 'total_num_unique', 'data_type'])

    # Iterate over the columns and perform the operations
    for col in cols:
        df_valid = df.filter(df[col].isNotNull())              # Filter out null values
        df_unique = df_valid.select(col).distinct()
        distinct_count = df_unique.count()
        data_type = df.select(F.col(col)).dtypes[0][1]
        sample_values = df_unique.limit(nrows).toPandas()[col].sort_values(ascending=True).tolist()
        pdf = pdf.append({

            'col_name': col,
            'sample_unique_val': sample_values,
            'total_num_unique': distinct_count,
            'data_type': data_type
        }, ignore_index=True)
    return pdf

def display_limited_df(df, nrows=3):
  return display(df.limit(nrows))

def null_check(df, df_name):
  # Calculate total rows in the DataFrame
  total_rows = df.count()
  print_df_shape(df, df_name)

  # Create a DataFrame with columns, their null counts, and percentage of null values
  null_percents = df.select([(round((count(when(col(c).isNull(), c)) / total_rows * 100), 2).alias(c)) for c in df.columns])

  # Count the null
  null_counts = df.select([(count(when(col(c).isNull(), c))).alias(c) for c in df.columns])

  # Create a Pandas DF for the null check outcome and transpose the dfs
  pdf_null_counts, pdf_null_percents = null_counts.toPandas(), null_percents.toPandas()
  pdf_null_counts, pdf_null_percents = pdf_null_counts.T, pdf_null_percents.T
  pdf_null_counts.rename(columns={0: 'missing_count'}, inplace=True)
  pdf_null_percents.rename(columns={0: 'missing_%'}, inplace=True)
  pdf_null_check = pd.concat([pdf_null_counts, pdf_null_percents], axis=1)

  return pdf_null_check.sort_values(by='missing_%', ascending=False)

def drop_null_cols(df, df_null, threshold=100):
  columns_to_drop = [col for col, null_perc in df_null.to_frame().T.iloc[0].to_dict().items() if null_perc >= threshold]
  print(f'columns_to_drop = {columns_to_drop}')
  df_cleaned = df.drop(*columns_to_drop)               # using the asterisk(*), unpack and drop multiple columns
  print(f'\nOut of {len(df.columns)} columns,\n==> {len(df.columns)-len(df_cleaned.columns)} with {threshold}% or greater missing values were removed, leaving a total of
{len(df_cleaned.columns)} remaining columns.')

  return df_cleaned
```

1. Null Check

- Compute the count and percentage of null values in each column
- Drop columns with all missing values

```python
otpw_missing = null_check(df_5y_train, 'df_5y_train')
otpw_missing
```

```
df_5y_train contains 31673119 rows & 214 columns
```

	missing_count	missing_%
MonthlyStationPressure	31673119	100.0
MonthlyDaysWithGT90Temp	31673119	100.0
MonthlyDaysWithGT010Precip	31673119	100.0
MonthlyDaysWithGT001Precip	31673119	100.0
MonthlyAverageRH	31673119	100.0
...	...	...
YEAR	0	0.0
MONTH	0	0.0
origin_airport_name	0	0.0
origin_station_name	0	0.0
QUARTER	0	0.0

214 rows × 2 columns

2. Remove Columns

- Drop Columns with 100% Missing Values
- Drop Columns That Have the Prefix Substrings **Daily** and **Backup**
- Drop Unnecessary Additional Columns

1-1. Drop Columns with All Missing Values

```
df_cleaned = drop_null_cols(df_5y_train, otpw_missing['missing_%'], threshold=100)
# display_limited_df(df_cleaned, nrows=3)
```

```
columns_to_drop = ['MonthlyStationPressure', 'MonthlyDaysWithGT90Temp', 'MonthlyDaysWithGT010Precip', 'MonthlyDaysWithGT001Precip', 'MonthlyAverageRH', 'MonthlyMeanTemperature', 'MonthlyMinSeaLevelPressureValue', 'MonthlyMinSeaLevelPressureValueDate', 'MonthlyMinSeaLevelPressureValueTime', 'MonthlyMinimumTemperature', 'MonthlySeaLevelPressure', 'MonthlyTotalLiquidPrecipitation', 'MonthlyTotalSnowfall', 'MonthlyWetBulb', 'AWND', 'CDSD', 'CLDD', 'MonthlyDaysWithGT32Temp', 'MonthlyDaysWithLT0Temp', 'HDSD', 'MonthlyDaysWithLT32Temp', 'MonthlyDepartureFromNormalHeatingDegreeDays', 'MonthlyDepartureFromNormalMaximumTemperature', 'MonthlyDepartureFromNormalMinimumTemperature', 'MonthlyDepartureFromNormalPrecipitation', 'MonthlyDewpointTemperature', 'MonthlyGreatestPrecip', 'MonthlyGreatestPrecipDate', 'MonthlyGreatestSnowDepth', 'MonthlyGreatestSnowDepthDate', 'MonthlyGreatestSnowfall', 'MonthlyGreatestSnowfallDate', 'MonthlyMaxSeaLevelPressureValue', 'MonthlyMaxSeaLevelPressureValueDate', 'MonthlyMaxSeaLevelPressureValueTime', 'MonthlyMaximumTemperature', 'MonthlyDepartureFromNormalAverageTemperature', 'DSNW', 'HTDD', 'ShortDurationEndDate150', 'ShortDurationPrecipitationValue180', 'ShortDurationPrecipitationValue150', 'ShortDurationPrecipitationValue120', 'ShortDurationPrecipitationValue100', 'ShortDurationPrecipitationValue080', 'ShortDurationPrecipitationValue060', 'ShortDurationPrecipitationValue045', 'ShortDurationPrecipitationValue030', 'ShortDurationPrecipitationValue020', 'NormalsCoolingDegreeDay', 'ShortDurationPrecipitationValue010', 'ShortDurationPrecipitationValue005', 'ShortDurationEndDate180', 'ShortDurationPrecipitationValue015', 'ShortDurationEndDate120', 'ShortDurationEndDate030', 'NormalsHeatingDegreeDay', 'ShortDurationEndDate010', 'ShortDurationEndDate015', 'ShortDurationEndDate020', 'ShortDurationEndDate005', 'MonthlyDepartureFromNormalCoolingDegreeDays', 'ShortDurationEndDate045', 'ShortDurationEndDate060', 'ShortDurationEndDate080', 'ShortDurationEndDate100']

Out of 214 columns,
==> 66 with 100% or greater missing values were removed, leaving a total of 148 remaining columns.
```

```
# print(f'Remaining Columns for Analysis:\n{df_cleaned.columns}')
```

1-2. Drop Columns That Have the Prefix Substrings `Daily` and `Backup`.

```
# Find all columns that have "Daily" or "Backup" as substring prefixes in their names
cols_w_daily_backup_metrics = [column for column in df_cleaned.columns if column.startswith("Daily") or column.startswith("Backup")]
print(f'Columns with Daily or Backup prefix substrings:\n{cols_w_daily_backup_metrics}')

df_cleaned = df_cleaned.drop(*cols_w_daily_backup_metrics)

# print_df_shape(df_cleaned, 'df_cleaned')
# display_limited_df(df_cleaned, nrows=3)
```

```
Columns with Daily or Backup prefix substrings:
['DailyAverageDewPointTemperature', 'DailyAverageDryBulbTemperature', 'DailyAverageRelativeHumidity', 'DailyAverageSeaLevelPressure', 'DailyAverageStationPressure', 'DailyAverageWetBulbTemperature', 'DailyAverageWindSpeed', 'DailyCoolingDegreeDays', 'DailyDepartureFromNormalAverageTemperature', 'DailyHeatingDegreeDays', 'DailyMaximumDryBulbTemperature', 'DailyMinimumDryBulbTemperature', 'DailyPeakWindDirection', 'DailyPeakWindSpeed', 'DailyPrecipitation', 'DailySnowDepth', 'DailySnowfall', 'DailySustainedWindDirection', 'DailySustainedWindSpeed', 'DailyWeather', 'BackupDirection', 'BackupDistance', 'BackupDistanceUnit', 'BackupElements', 'BackupElevation', 'BackupEquipment', 'BackupLatitude', 'BackupLongitude', 'BackupName']
```

1-3. Drop Unnecessary Additional Columns

```
# Drop Unnecessary Additional Columns
cols_to_drop = [
    'CANCELLATION_CODE',
    'DEST_AIRPORT_ID',
    'DEST_AIRPORT_SEQ_ID',
    'DEST_CITY_MARKET_ID',
    'DEST_STATE_ABR',
    'DEST_STATE_FIPS',
    'DEST_STATE_NM',
    'DEST_WAC',
    'dest_station_name',
    'dest_station_id',
    'dest_icao',
    'dest_region',
    'dest_station_lat',
    'dest_station_lon',
    'FIRST_DEP_TIME',
    'NAME',
    'origin_station_name',
    'origin_station_id',
    'origin_icao',
    'origin_region',
    'origin_station_lat',
    'origin_station_lon',
    'ORIGIN_AIRPORT_SEQ_ID',
    'ORIGIN_STATE_FIPS',
    'ORIGIN_WAC',
    'ORIGIN_CITY_MARKET_ID',
    'ORIGIN_STATE_ABR',
    'ORIGIN_STATE_NM',
    'OP_CARRIER_AIRLINE_ID',
    'REM',
    'WindEquipmentChangeDate',
    'WHEELS_OFF',
    'WHEELS_ON',
    '_row_desc']

print(f'Length of the additonal unnecessary columns to drop = {len(cols_to_drop)}')
print_df_shape(df_cleaned, 'df_cleaned')
```

```
Length of the additonal unnecessary columns to drop = 34
df_cleaned contains 31673119 rows & 119 columns
```

```
# # Find the intersection of the two lists
# common_cols = set(df_cleaned.columns).intersection(cols_to_drop)
# print(common_cols)
```

```
df_cleaned = df_cleaned.drop(*cols_to_drop)
```

Final Keep Columns

- Extract Distinct Values for the Selected Columns

IMPORTANT NOTE:

- Drop "FLIGHTS" column
- Drop MONTH, QUARTER, YEAR, DEP_DELAY columns
- Drop columns that are no longer needed e.g. Distance, day_of_year, etc...

```
final_keep_columns = [
  'MONTH',
  'QUARTER',
  'YEAR',
  'DEP_DEL15',
  'CANCELLED',
  'DAY_OF_MONTH',
  'DISTANCE',
  'DAY_OF_WEEK',
  'DEP_DELAY',
  'ORIGIN_AIRPORT_ID',
  'FL_DATE',
  'ELEVATION',
  'CRS_DEP_TIME',
  'CRS_ELAPSED_TIME',
  'DIVERTED',
  'FLIGHTS',
  'DISTANCE_GROUP',
  'DEST',
  'OP_UNIQUE_CARRIER',
  'ORIGIN',
  'TAIL_NUM',
  'HourlyWindSpeed',
  'HourlyPrecipitation',
  'HourlyRelativeHumidity',
  'HourlyVisibility',
 ]

df_cleaned = df_cleaned.select(final_keep_columns)
print(f'nrows = {df_cleaned.count()}\nncols = {len(df_cleaned.columns)}\n{df_cleaned.columns}')
```

```
nrows = 31673119
ncols = 25
['MONTH', 'QUARTER', 'YEAR', 'DEP_DEL15', 'CANCELLED', 'DAY_OF_MONTH', 'DISTANCE', 'DAY_OF_WEEK', 'DEP_DELAY', 'ORIGIN_AIRPORT_ID', 'FL_DATE', 'ELEVATION', 'CRS_DEP_TIME', 'CRS_ELAPSED_TIM
E', 'DIVERTED', 'FLIGHTS', 'DISTANCE_GROUP', 'DEST', 'OP_UNIQUE_CARRIER', 'ORIGIN', 'TAIL_NUM', 'HourlyWindSpeed', 'HourlyPrecipitation', 'HourlyRelativeHumidity', 'HourlyVisibility']
```

```
# extract_distinct_values_multiCol(df_cleaned, final_keep_columns, nrows=15)
```

Feature Engineering

Helper Functions

```python
from pyspark.sql.functions import sin, cos, radians

# =======================================================
#                     CYCLICAL FEATURES
# =======================================================

# Define a function to convert day of the week to radians
def cyclical_feature(df, col, cyclical_val):
    # There are 2*pi radians in a circle and 7 days in a week
    df = df.withColumn(col + '_radians', radians(df[col] * (360 / cyclical_val)))   # 360 = degrees in a circle
    df = df.withColumn(col + '_sin', sin(df[col + '_radians']))
    df = df.withColumn(col + '_cos', cos(df[col + '_radians']))
    df = df.drop(col + '_radians')
    return df

# =======================================================
#                      HOLIDAY EFFECT
# =======================================================

federal_holidays = [
    "New Year's Day",
    "Martin Luther King Jr. Day",
    "Washington's Birthday",
    "Memorial Day",
    "July 4th",
    "Labor Day",
    "Columbus Day",
    "Veterans Day",
    "Thanksgiving Day",
    "Christmas Day"]

# 1. Calculate the specific dates for these holidays for each year.
def calculate_holidays(year):
  holidays = {
      "New Year's Day": f"{year}-01-01",
      "Martin Luther King Jr. Day": f"{year}-01-{15 + (0 if (datetime(year, 1, 1).weekday() <= 0) else 7 - datetime(year, 1, 1).weekday())}",
      "Washington's Birthday": f"{year}-02-{15 + (0 if (datetime(year, 2, 1).weekday() <= 0) else 7 - datetime(year, 2, 1).weekday())}",
      "Memorial Day": f"{year}-05-{31 - datetime(year, 5, 31).weekday()}",
      "Independence Day": f"{year}-07-04",
      "Labor Day": f"{year}-09-{1 + (7 - datetime(year, 9, 1).weekday())}",
      "Columbus Day": f"{year}-10-{8 + (0 if (datetime(year, 10, 1).weekday() <= 0) else 7 - datetime(year, 10, 1).weekday())}",
      "Veterans Day": f"{year}-11-11",
      "Thanksgiving Day": f"{year}-11-{22 + (3 - datetime(year, 11, 1).weekday() + 7) % 7}",
      "Christmas Day": f"{year}-12-25"
  }
  return [date for holiday, date in holidays.items()]

# 2. Generate days of year for the given holidays in each year: df with flight date and day of year
def get_holidays():
  holiday_dates = []
  for y in range(2015, 2020):
    holiday_dates.extend(calculate_holidays(y))
  df_holiday_dates = spark.createDataFrame([(d,) for d in holiday_dates], ['holiday_date']).cache()
  df_holiday_dates = df_holiday_dates.withColumn('day_of_year', dayofyear(col('holiday_date'))).cache()
  return df_holiday_dates

# 3. Generate a dict with keys = flight date, values = day of year
def get_date_holiday_dict(year):
  #Create a list of tuples from the list of rows (= df_holiday_dates.collect())
  holiday_tuples = [(row['holiday_date'], row['day_of_year']) for row in get_holidays().collect()]
  filtered_tuples = [t for t in holiday_tuples if t[0].startswith(year)]

  # Create a dictionary where keys are tuples from filtered_tuples and values are corresponding federal_holidays
  holiday_dict = {tuple_date: holiday for tuple_date, holiday in zip(filtered_tuples, federal_holidays)}
  return holiday_dict

# 4. Generate a consolidated holiday dict with keys = (flight date, day of year), values = name of holiday
def generate_consolidated_holiday_dict():
  years = [str(year) for year in range(2015, 2020)]
  consolidated_holiday_dict = {}
  for year in years:
    consolidated_holiday_dict.update(get_date_holiday_dict(year))
  return consolidated_holiday_dict
```

Feature 1: `Day of Week - Cyclical Feature with Sin & Cos`

- Capture the cyclical nature of days within a week
- The idea is to map each day to a point on a unit circle, where the angle corresponds to the day of the week.

```python
# print(df_cleaned.columns)

df_cleaned = df_cleaned.withColumn("DAY_OF_WEEK", F.col("DAY_OF_WEEK").cast("int")).cache()
df_cleaned = cyclical_feature(df_cleaned, 'DAY_OF_WEEK', 7)
# display(df_cleaned.limit(3))
```

Feature 2: `Hour of Day - Cyclical Feature with Sin & Cos`

- Using `CRS_DEP_TIME`, Capture the cyclical nature of hours within 0 - 23
- The model captures the continuity between the end of one day and the start of another, which can be very useful for predicting events that are influenced by time, like flight departures.

```
# If CRS_DEP_TIME is a string
df_cleaned = df_cleaned.withColumn('CRS_DEP_HOUR', F.expr("substring(CRS_DEP_TIME, 1, length(CRS_DEP_TIME)-2)"))
df_cleaned = df_cleaned.withColumn('CRS_DEP_HOUR', (F.col('CRS_DEP_TIME') / 100).cast('int'))
df_cleaned = cyclical_feature(df_cleaned, 'CRS_DEP_HOUR', 24)

# Show the resulting DataFrame with the extracted hours
display(df_cleaned.limit(2))
```

Table

	MONTH	QUARTER	YEAR	DEP_DEL15	CANCELLED	DAY_OF_MONTH	DISTANCE	DAY_OF_WEEK	DEP_DELAY	ORIGIN_AIRPORT_ID	FL_DATE	ELEVATION
1	2	1	2019	0.0	0.0	14	83.0	4	-5.0	10397	2019-02-14	307.8
2	9	3	2019	0.0	0.0	29	431.0	7	-6.0	14492	2019-09-29	126.8

2 rows

Feature 3: `Hour of Day + Day of Week`

- Combine the hour of the day with the day of the week to capture patterns like **busy weekday mornings** or **relaxed weekend afternoons**.
- The numeric interaction feature assumes there are 24 hours in each day, so by multiplying the day of the week by 24 and adding the hour of the day, you get a unique number for each hour in the week.
  - - This can be particularly useful if you want to maintain `ordinality` where later hours in the week are always greater than earlier ones.

```
# This creates a unique number for each hour in the week (assuming 24-hour days)
df_cleaned = df_cleaned.withColumn('DAY_HOUR_interaction', (col('DAY_OF_WEEK') * 24) + col('CRS_DEP_HOUR'))

print(4 * 24 + 18)
display(df_cleaned.select('DAY_OF_WEEK','CRS_DEP_HOUR','DAY_HOUR_interaction'))
display(df_cleaned.limit(3))
```

114

Table

	DAY_OF_WEEK	CRS_DEP_HOUR	DAY_HOUR_interaction
1	4	16	112
2	7	12	180
3	4	13	109
4	5	11	131
5	1	14	38
6	1	12	36
7	5	7	127

10,000 rows | Truncated data

Table

	MONTH	QUARTER	YEAR	DEP_DEL15	CANCELLED	DAY_OF_MONTH	DISTANCE	DAY_OF_WEEK	DEP_DELAY	ORIGIN_AIRPORT_ID	FL_DATE	ELEVATION
1	2	1	2019	0.0	0.0	14	83.0	4	-5.0	10397	2019-02-14	307.8
2	9	3	2019	0.0	0.0	29	431.0	7	-6.0	14492	2019-09-29	126.8
3	2	1	2018	0.0	0.0	15	227.0	4	14.0	10397	2018-02-15	307.8

3 rows

```
# extract_distinct_values_singleCol(df_cleaned, 'DAY_OF_WEEK')
```

Feature 4: `Weekday vs. Weenend`

- Since weekends might have different traffic patterns compared to weekdays, we could create a binary feature that indicates whether the day is a weekend (e.g. Saturday or Sunday).

```
df_cleaned = df_cleaned.withColumn('DAY_TYPE', when((df_cleaned['DAY_OF_WEEK'] == 6) | (df_cleaned['DAY_OF_WEEK'] == 7), 'weekend').otherwise('weekday'))
display(df_cleaned.select('DAY_OF_WEEK','DAY_TYPE').limit(3))
```

Table

	DAY_OF_WEEK	DAY_TYPE
1	4	weekday
2	7	weekend
3	4	weekday

3 rows

Feature 5: `Time_Interval_of_Day - KMeans`

- `Cluster 0` : 4.978866 <= **Delay Percentage** <= 6.435471
- `Cluster 1` : 0.000553 <= **Delay Percentage** <= 1.705419
- `Cluster 2` : 3.454303 <= **Delay Percentage** <= 3.796307
- `Cluster 3` : 6.847092 <= **Delay Percentage** <= 8.369999

```
df_cleaned = df_cleaned.withColumn("DEP_DELAY", F.col("DEP_DELAY").cast("double")).cache()
df_cleaned_filtered = df_cleaned.filter(F.col('DEP_DELAY') > 0)
count_dep_delay_by_hour = df_cleaned_filtered.groupBy('CRS_DEP_HOUR').agg(F.count('DEP_DELAY').alias('Count_DEP_DELAY')).cache()

# Calculate the total count of 'Count_DEP_DELAY' over the entire DataFrame
windowSpec = Window.partitionBy()
count_dep_delay_by_hour = count_dep_delay_by_hour.withColumn('Total_Count', F.sum('Count_DEP_DELAY').over(windowSpec))

# count_dep_delay_by_days_sorted = count_dep_delay_by_days_sorted.withColumn('Total_Count', total_count)
count_dep_delay_by_hour = count_dep_delay_by_hour.withColumn('Percentage_Dep_Delay_by_Hourly_Interval', (F.col('Count_DEP_DELAY')/F.col('Total_Count')) * 100)
# display(count_dep_delay_by_hour.collect())

pdf_count_dep_delay_by_hour = count_dep_delay_by_hour.toPandas().sort_values('Percentage_Dep_Delay_by_Hourly_Interval', ascending=False)
# pdf_count_dep_delay_by_hour.head()
```

```
# from pyspark.ml.clustering import KMeans

data = list(pdf_count_dep_delay_by_hour[['CRS_DEP_HOUR', 'Percentage_Dep_Delay_by_Hourly_Interval']].itertuples(index=False, name=None))

columns = ['CRS_DEP_HOUR', 'Delay_Percentage']

df = spark.createDataFrame(data, columns)
vec_assembler = VectorAssembler(inputCols=['Delay_Percentage'], outputCol='features')  # Convert 'Delay_Percentage' into a feature vector
kmeans = KMeans(featuresCol='features', predictionCol='cluster', k=4)  # cluster into 4 clusters
pipeline = Pipeline(stages=[vec_assembler, kmeans])                    # Pipeline stages
model = pipeline.fit(df)   # Fit the pipeline to perform clustering

# Transform the DataFrame to add the cluster predictions
df_clustered = model.transform(df)
# display(df_clustered)
```

```
# display(df_cleaned.limit(2))
```

```
# from pyspark.sql.functions import collect_list

# Group by 'cluster' and collect the other values into a list
clusters_list_df = df_clustered.groupBy('cluster').agg(
    collect_list('CRS_DEP_HOUR').alias('CRS_DEP_HOUR_list'),
    collect_list('Delay_Percentage').alias('Delay_Percentage_list')
)

# To collect the lists into a Python object
clusters_list = clusters_list_df.collect()

# Create a dictionary to hold the lists, with cluster numbers as keys
cluster_dict = {}
for row in clusters_list:
    cluster_dict[row['cluster']] = {
        'CRS_DEP_HOUR_list': row['CRS_DEP_HOUR_list'],
        'Delay_Percentage_list': row['Delay_Percentage_list']
    }
df_cluster = pd.DataFrame.from_dict(cluster_dict, orient='index')

# Reset index to make the cluster numbers into a column
df_cluster.reset_index(inplace=True)
df_cluster.rename(columns={'index': 'cluster'}, inplace=True)
# Define functions to get the min and max from the lists
get_min = lambda x: min(x) if isinstance(x, list) and len(x) > 0 else None
get_max = lambda x: max(x) if isinstance(x, list) and len(x) > 0 else None

# Apply the functions to the 'Delay_Percentage_list' column to create new columns
df_cluster['Min_Delay_Percentage'] = df_cluster['Delay_Percentage_list'].apply(get_min)
df_cluster['Max_Delay_Percentage'] = df_cluster['Delay_Percentage_list'].apply(get_max)
# df_cluster
```

```
# print(df_cleaned.columns)
# display(df_cleaned.limit(2))
```

```
df_cleaned_with_cluster = df_clustered.drop('features')
df_cleaned_with_cluster = df_cleaned_with_cluster.withColumnRenamed('cluster', 'TIME_INTERVAL_OF_DAY')
# display(df_cleaned_with_cluster.limit(2))

df_cleaned = df_cleaned.join(df_cleaned_with_cluster.select('CRS_DEP_HOUR', 'TIME_INTERVAL_OF_DAY'),
                              on='CRS_DEP_HOUR',
                              how='left')
# print(df_cleaned.columns)
# display(df_cleaned.limit(2))
```

```
# df_cleaned = df_cleaned.drop(*['day_of_year', 'N_day_distance_from_holiday'])
```

Feature 6: `Flight Distance Group`

- `DIST_SHORTEST (0)` : Distance < 250 Miles
- `DIST_SHORT (1)` : 250 <= Distance < 750
- `DIST_MEDIUM (2)` : 750 <= Distance < 1250
- `DIST_LONG (3)` : Distance >= 1250

```
  # extract_distinct_values_multiCol(df_cleaned,['DISTANCE_GROUP', 'DISTANCE'], 20)
```

```
df_cleaned = df_cleaned.withColumn('DISTANCE', col('DISTANCE').cast('int'))

# Create a new column 'DISTANCE_GROUP' with the labels according to the distance
df_cleaned = df_cleaned.withColumn(
    'FL_DISTANCE_GROUP',
    when(col('DISTANCE') < 250, 'DIST_SHORTEST')
    .when((col('DISTANCE') >= 250) & (col('DISTANCE') < 750), 'DIST_SHORT')
    .when((col('DISTANCE') >= 750) & (col('DISTANCE') < 1250), 'DIST_MEDIUM')
    .when(col('DISTANCE') >= 1250, 'DIST_LONG')
    .otherwise('UNKNOWN')  # Use 'UNKNOWN' for any distance that doesn't fall into the defined buckets
)
# extract_distinct_values_singleCol(df_cleaned, 'FL_DISTANCE_GROUP')
# display(df_cleaned.select('DISTANCE','FL_DISTANCE_GROUP'))
```

```
# print(df_cleaned.columns)
```

Feature 7: `Airlines Grouping by Tail Number`

- `CARRIER_SMALL (0)` : Size of Airline < 300
- `CARRIER_MEDIUM (1)` : 300 <= Size of Airline < 600
- `CARRIER_LARGE (2)` : Size of Airline >= 600

```
display(df_cleaned.select('OP_UNIQUE_CARRIER','TAIL_NUM'))
```

Table

	OP_UNIQUE_CARRIER	TAIL_NUM
1	9E	N801AY
2	9E	N927XJ
3	9E	N819AY
4	9E	N326PQ
5	9E	N8896A
6	9E	N604LR
7	9E	N909XJ

10,000 rows | Truncated data

```
from pyspark.sql.functions import countDistinct, desc

# Assuming df_5y_train is your DataFrame
result_df = df_cleaned.groupBy("OP_UNIQUE_CARRIER").agg(countDistinct("TAIL_NUM").alias("distinct_tail_nums"))
sorted_df = result_df.orderBy(desc("distinct_tail_nums"))
# display(sorted_df.limit(10))
```

```
df_cleaned = df_cleaned.join(sorted_df, on='OP_UNIQUE_CARRIER', how='left')

df_cleaned = df_cleaned.withColumn(
  'CARRIER_SIZE',
  when(df_cleaned['distinct_tail_nums'] < 300, 'CARRIER_SMALL')
  .when((df_cleaned['distinct_tail_nums'] >= 300) & (sorted_df['distinct_tail_nums'] > 600), 'CARRIER_MEDIUM')
  .when(df_cleaned['distinct_tail_nums'] >= 300, 'CARRIER_LARGE')
  .otherwise('UNKNOWN')
)

df_cleaned = df_cleaned.drop('distinct_tail_nums')
print(df_cleaned.columns)
extract_distinct_values_singleCol(df_cleaned, 'CARRIER_SIZE')
display(df_cleaned.select('OP_UNIQUE_CARRIER','CARRIER_SIZE'))
```

```
['OP_UNIQUE_CARRIER', 'CRS_DEP_HOUR', 'MONTH', 'QUARTER', 'YEAR', 'DEP_DEL15', 'CANCELLED', 'DAY_OF_MONTH', 'DISTANCE', 'DAY_OF_WEEK', 'DEP_DELAY', 'ORIGIN_AIRPORT_ID', 'FL_DATE', 'ELEVATIO
N', 'CRS_DEP_TIME', 'CRS_ELAPSED_TIME', 'DIVERTED', 'FLIGHTS', 'DISTANCE_GROUP', 'DEST', 'ORIGIN', 'TAIL_NUM', 'HourlyWindSpeed', 'HourlyPrecipitation', 'HourlyRelativeHumidity', 'HourlyVis
ibility', 'DAY_OF_WEEK_sin', 'DAY_OF_WEEK_cos', 'CRS_DEP_HOUR_sin', 'CRS_DEP_HOUR_cos', 'DAY_HOUR_interaction', 'DAY_TYPE', 'TIME_INTERVAL_OF_DAY', 'FL_DISTANCE_GROUP', 'CARRIER_SIZE']
Length of unique values = 3
Unique values of "CARRIER_SIZE":
['CARRIER_MEDIUM', 'CARRIER_LARGE', 'CARRIER_SMALL']
```

Table

	OP_UNIQUE_CARRIER	CARRIER_SIZE
1	AA	CARRIER_MEDIUM

2	AA	CARRIER_MEDIUM
3	AA	CARRIER_MEDIUM
4	AA	CARRIER_MEDIUM
5	AA	CARRIER_MEDIUM
6	AA	CARRIER_MEDIUM
7	AA	CARRIER_MEDIUM

10,000 rows | Truncated data

Feature 8: `Degree of Hourly Visibility`

- `LOW_VISIBILITY (0)` : HourlyVisibility < 10 Miles
- `HIGH_VISIBILITY (1)` : HourlyVisibility >= 10
- **The shorter the distance, the better the visibility**
- When visibility is low due to fog, heavy rain, snow, dust storms, or other weather conditions, it can lead to delays in flight departures. Pilots need certain visibility levels to taxi, take off, and land safely. Air traffic control may slow down the pace of takeoffs and landings to ensure safety, leading to delays.

```
df_cleaned = df_cleaned.withColumn('HourlyVisibility', col('HourlyVisibility').cast('double'))
# df_cleaned = df_cleaned.withColumn('DISTANCE', col('DISTANCE').cast('int'))

df_cleaned = df_cleaned.withColumn(
  "DEGREE_VISIBILITY",
  when(df_cleaned['HourlyVisibility'] < 10.0, 'HIGH_VISIBILITY')
  .otherwise('LOW_VISIBILITY')
)
extract_distinct_values_singleCol(df_cleaned, 'DEGREE_VISIBILITY')
display(df_cleaned.select('HourlyVisibility','DEGREE_VISIBILITY'))
```

```
Length of unique values = 2
Unique values of "DEGREE_VISIBILITY":
['LOW_VISIBILITY', 'HIGH_VISIBILITY']
```

Table

	HourlyVisibility	DEGREE_VISIBILITY
1	10	LOW_VISIBILITY
2	10	LOW_VISIBILITY
3	10	LOW_VISIBILITY
4	10	LOW_VISIBILITY
5	10	LOW_VISIBILITY
6	10	LOW_VISIBILITY
7	9.94	HIGH_VISIBILITY

10,000 rows | Truncated data

```
print(df_cleaned.columns)
# df_cleaned = df_cleaned.drop('DEGREE_VISIBILITY')
```

```
['OP_UNIQUE_CARRIER', 'CRS_DEP_HOUR', 'MONTH', 'QUARTER', 'YEAR', 'DEP_DEL15', 'CANCELLED', 'DAY_OF_MONTH', 'DISTANCE', 'DAY_OF_WEEK', 'DEP_DELAY', 'ORIGIN_AIRPORT_ID', 'FL_DATE', 'ELEVATIO
N', 'CRS_DEP_TIME', 'CRS_ELAPSED_TIME', 'DIVERTED', 'FLIGHTS', 'DISTANCE_GROUP', 'DEST', 'ORIGIN', 'TAIL_NUM', 'HourlyWindSpeed', 'HourlyPrecipitation', 'HourlyRelativeHumidity', 'HourlyVis
ibility', 'DAY_OF_WEEK_sin', 'DAY_OF_WEEK_cos', 'CRS_DEP_HOUR_sin', 'CRS_DEP_HOUR_cos', 'DAY_HOUR_interaction', 'DAY_TYPE', 'TIME_INTERVAL_OF_DAY', 'FL_DISTANCE_GROUP', 'CARRIER_SIZE', 'DEG
REE_VISIBILITY']
```

Feature 9: `N-number of days before/after the Holiday`

- 3-day before/after July 4th
- 5-day before/after Christmas
- 5-day before/after New Year's day

```
get_date_holiday_dict(str(2016))
```

```
{('2016-01-01', 1): "New Year's Day",
 ('2016-01-18', 18): 'Martin Luther King Jr. Day',
 ('2016-02-15', 46): "Washington's Birthday",
 ('2016-05-30', 151): 'Memorial Day',
 ('2016-07-04', 186): 'July 4th',
 ('2016-09-5', 249): 'Labor Day',
 ('2016-10-10', 284): 'Columbus Day',
 ('2016-11-11', 316): 'Veterans Day',
 ('2016-11-24', 329): 'Thanksgiving Day',
 ('2016-12-25', 360): 'Christmas Day'}
```

```
generate_consolidated_holiday_dict()
```

```
{('2015-01-01', 1): "New Year's Day",
 ('2015-01-19', 19): 'Martin Luther King Jr. Day',
 ('2015-02-16', 47): "Washington's Birthday",
 ('2015-05-25', 145): 'Memorial Day',
 ('2015-07-04', 185): 'July 4th',
 ('2015-09-7', 250): 'Labor Day',
 ('2015-10-12', 285): 'Columbus Day',
 ('2015-11-11', 315): 'Veterans Day',
 ('2015-11-26', 330): 'Thanksgiving Day',
 ('2015-12-25', 359): 'Christmas Day',
 ('2016-01-01', 1): "New Year's Day",
 ('2016-01-18', 18): 'Martin Luther King Jr. Day',
```

```
('2016-02-15', 46): "Washington's Birthday",
('2016-05-30', 151): 'Memorial Day',
('2016-07-04', 186): 'July 4th',
('2016-09-5', 249): 'Labor Day',
('2016-10-10', 284): 'Columbus Day',
('2016-11-11', 316): 'Veterans Day',
('2016-11-24', 329): 'Thanksgiving Day',
('2016-12-25', 360): 'Christmas Day',
('2017-01-01', 1): "New Year's Day",
```

```
df_cleaned = df_cleaned.withColumn('day_of_year', dayofyear(col('FL_DATE'))).cache()
```

```
# # Define the day of the year for each holiday (2016 leap year)
july_4th = when(col('YEAR') == 2016, 186).otherwise(185)
christmas = when(col('YEAR') == 2016, 360).otherwise(359)
new_year = lit(1)

# holiday_date = day of year
def get_holiday_distance(col_day_of_year_, holiday_day_of_year, window):
    return when(
        (col(col_day_of_year_) >= holiday_day_of_year - window) &
        (col(col_day_of_year_) <= holiday_day_of_year + window),
        (col(col_day_of_year_) - holiday_day_of_year)
    ).otherwise(999999)

# Calculate distance from July 4th
df_cleaned = df_cleaned.withColumn(
    '5_DAYS_DIST_FROM_Independence',
    get_holiday_distance('day_of_year', lit(july_4th), lit(5))
)

# Calculate distance from Christmas
df_cleaned = df_cleaned.withColumn(
    '7_DAYS_DIST_FROM_Christmas',
    get_holiday_distance('day_of_year', lit(christmas), lit(7))
)

# Calculate distance from New Year's Day
df_cleaned = df_cleaned.withColumn(
    '7_DAYS_DIST_FROM_NewYear',
    get_holiday_distance('day_of_year', lit(new_year), lit(7))
)

print(df_cleaned.columns)
display(df_cleaned)
```

```
['OP_UNIQUE_CARRIER', 'CRS_DEP_HOUR', 'MONTH', 'QUARTER', 'YEAR', 'DEP_DEL15', 'CANCELLED', 'DAY_OF_MONTH', 'DISTANCE', 'DAY_OF_WEEK', 'DEP_DELAY', 'ORIGIN_AIRPORT_ID', 'FL_DATE', 'ELEVATION', 'CRS_DEP_TIME', 'CRS_ELAPSED_TIME', 'DIVERTED', 'FLIGHTS', 'DISTANCE_GROUP', 'DEST', 'ORIGIN', 'TAIL_NUM', 'HourlyWindSpeed', 'HourlyPrecipitation', 'HourlyRelativeHumidity', 'HourlyVisibility', 'DAY_OF_WEEK_sin', 'DAY_OF_WEEK_cos', 'CRS_DEP_HOUR_sin', 'CRS_DEP_HOUR_cos', 'DAY_HOUR_interaction', 'DAY_TYPE', 'TIME_INTERVAL_OF_DAY', 'FL_DISTANCE_GROUP', 'CARRIER_SIZE', 'DEGREE_VISIBILITY', 'day_of_year', '5_DAYS_DIST_FROM_Independence', '7_DAYS_DIST_FROM_Christmas', '7_DAYS_DIST_FROM_NewYear']
```

Table

	OP_UNIQUE_CARRIER	CRS_DEP_HOUR	MONTH	QUARTER	YEAR	DEP_DEL15	CANCELLED	DAY_OF_MONTH	DISTANCE	DAY_OF_WEEK	DEP_DELAY	ORIGIN
1	UA	22	8	3	2016	1.0	0.0	2	447	2	32	14771
2	UA	22	1	1	2015	0.0	0.0	30	1846	5	4	14771
3	UA	22	3	1	2016	1.0	0.0	10	1846	4	22	14771
4	UA	22	9	3	2016	0.0	0.0	21	224	3	-3	12264
5	UA	22	3	1	2016	0.0	0.0	21	2419	1	-12	12264
6	UA	22	11	4	2016	1.0	0.0	28	2419	1	83	12264
7	UA	22	11	4	2015	0.0	0.0	15	2253	7	-3	14679

6,467 rows | Truncated data

Feature 10: `Lag Feature` from Stephanie

- **Stephanie's note**: The dep_del15_2hr_before represents whether there was any delay_15 in the past 2 hours, not just the direct previous flight. The code for getting the direct previous flight would be more involved, and also even if we get the direct previous flight, those planes might not be in the same vicinity at the airport. But anyway i think it makes sense that if there's a delay in one part of the airport 2 hours ago, that might affect the delay at the same airport now. b/c if there was something wrong with the runway or something that'd affect all planes

```
#======================================
#=========== Stephanie's Code ===========
#======================================

# Helper Function: Remove rows with null in the target variable
def remove_null_n_target_value(df):
  df_no_target_null = df.where(col('DEP_DEL15').isNotNull())
  return df_no_target_null

# Drop null DEP_DEL15 from full 5 year df
df_cleaned = remove_null_n_target_value(df_cleaned)

# Add lag feature to full 5 year df
bronze = df_cleaned.withColumn('FL_DATE_TIME', F.to_timestamp(F.concat_ws('-', F.col('FL_DATE'), F.substring("CRS_DEP_TIME", -4, 2), F.substring("CRS_DEP_TIME", -2, 2)), 'yyyy-MM-dd-H-m')).cache()
silver = bronze.withColumn('FL_DATE_TIME_LONG', F.col('FL_DATE_TIME').cast('long')).cache()
w = Window.partitionBy('ORIGIN_AIRPORT_ID').orderBy('FL_DATE_TIME_LONG').rangeBetween(-60*60*2,0)
gold = silver.withColumn('num_occurrences_in_2_hr', F.sum('DEP_DEL15').over(w)).cache()
df_cleaned = gold.withColumn('dep_del15_2hr_before', F.when((F.col('num_occurrences_in_2_hr') >= 1), 1).otherwise(0))

# cols =['FL_DATE_TIME', 'FL_DATE_TIME_LONG', 'num_occurrences_in_2_hr', 'dep_del15_2hr_before']

cols_to_delete = ['QUARTER','DEP_DELAY','FLIGHTS', 'DISTANCE_GROUP','CRS_DEP_HOUR','day_of_year',
'FL_DATE_TIME', 'FL_DATE_TIME_LONG', 'num_occurrences_in_2_hr','ORIGIN_AIRPORT_ID']
df_cleaned = df_cleaned.drop(*cols_to_delete)

print(df_cleaned.columns)
display(df_cleaned.limit(10))
# extract_distinct_values_multiCol(df_cleaned, cols, 10)
```

```
['OP_UNIQUE_CARRIER', 'MONTH', 'YEAR', 'DEP_DEL15', 'CANCELLED', 'DAY_OF_MONTH', 'DISTANCE', 'DAY_OF_WEEK', 'FL_DATE', 'ELEVATION', 'CRS_DEP_TIME', 'CRS_ELAPSED_TIME', 'DIVERTED', 'DEST', 'ORIGIN', 'TAIL_NUM', 'HourlyWindSpeed', 'HourlyPrecipitation', 'HourlyRelativeHumidity', 'HourlyVisibility', 'DAY_OF_WEEK_sin', 'DAY_OF_WEEK_cos', 'CRS_DEP_HOUR_sin', 'CRS_DEP_HOUR_cos', 'DAY_HOUR_interaction', 'DAY_TYPE', 'TIME_INTERVAL_OF_DAY', 'FL_DISTANCE_GROUP', 'CARRIER_SIZE', 'DEGREE_VISIBILITY', '5_DAYS_DIST_FROM_Independence', '7_DAYS_DIST_FROM_Christmas', '7_DAYS_DIST_FROM_NewYear', 'dep_del15_2hr_before']
```

Table

	OP_UNIQUE_CARRIER	MONTH	YEAR	DEP_DEL15	CANCELLED	DAY_OF_MONTH	DISTANCE	DAY_OF_WEEK	FL_DATE	ELEVATION	CRS_DEP_TIME	CRS_EI
1	OO	11	2016	1.0	0.0	28	93	1	2016-11-28	1291.4	1618	52.0
2	OO	1	2015	0.0	0.0	1	113	4	2015-01-01	2214.7	615	53.0
3	OO	1	2015	0.0	0.0	1	113	4	2015-01-01	2214.7	1422	53.0
4	OO	1	2015	0.0	0.0	2	113	5	2015-01-02	2214.7	615	53.0
5	OO	1	2015	0.0	0.0	2	113	5	2015-01-02	2214.7	1422	53.0
6	OO	1	2015	0.0	0.0	3	113	6	2015-01-03	2214.7	615	53.0
7	OO	1	2015	1.0	0.0	3	113	6	2015-01-03	2214.7	1430	53.0

10 rows

```
# df_cleaned = df_cleaned.drop(*['3_DAY_DIST_FROM_Independence', '6_DAY_DIST_FROM_Christmas', '6_DAY_DIST_FROM_NewYear', '3_DAYS_DIST_FROM_Independence', '6_DAYS_DIST_FROM_Christmas', '6_DAYS_DIST_FROM_NewYear', 'N_DAY_FROM_HOLIDAY'])
```

Create a Function to Create a DF with all the new features and Store it into Cloud

```
# print(final_keep_columns)
# df_cleaned = df_cleaned.drop(*['DEP_DELAY', 'FLIGHTS', 'DISTANCE_GROUP'])
# print()
# print(df_cleaned.columns)
```

Function to Generate DFs for New Features

```
# def feature_engineering_output(df):
#   # new_features = ['DAY_OF_WEEK_sin', 'DAY_OF_WEEK_cos', 'CRS_DEP_HOUR_sin', 'CRS_DEP_HOUR_cos', 'DAY_HOUR_interaction', 'DAY_TYPE', 'TIME_INTERVAL_OF_DAY', 'FL_DISTANCE_GROUP', 'CARRIER_SIZE', 'DEGREE_VISIBILITY', '3_DAYS_DIST_FROM_Independence', '6_DAYS_DIST_FROM_Christmas', '6_DAYS_DIST_FROM_NewYear']

#   df_cleaned_wo_holiday_feature = df.select(new_features)
#   return df, df_cleaned_wo_holiday_feature

# df_cleaned_both_existing_and_new_features, df_cleaned_only_new_features = feature_engineering_output(df_cleaned)
```

```
# display(df_cleaned_both_existing_and_new_features.limit(2))
```

```
# display(df_cleaned_wo_holiday_feature.limit(2))
```

Store DFs to Blob Storage in Delta Lake

```
# DELTALAKE_OTPW_3M_2015 = f'{team_blob_url}/featEng_both_existing_and_new_features_3mo.deltalake/'
# dbutils.fs.rm(DELTALAKE_OTPW_3M_2015, recurse=True)

checkpoint_df_blob(df_cleaned, 'FeatEng_ALL_5yr_w_Holiday_LagFeat', format="delta")
load_df_blob('FeatEng_ALL_5yr_w_Holiday_LagFeat', format="delta")

# checkpoint_df_blob(df_cleaned_both_existing_and_new_features, 'FeatEng_ALL_5yr', format="delta")
# load_df_blob('FeatEng_ALL_5yr', format="delta")

# checkpoint_df_blob(df_cleaned_only_new_features, 'FeatEng_ONLY_NewFeatures_5yr', format="delta")
# load_df_blob('FeatEng_ONLY_NewFeatures_5yr', format="delta")

display(dbutils.fs.ls(f"{team_blob_url}"))
```

Table

	path	name	size	modificationTime
1	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_1yr.deltalake/	FeatEng_ALL_1yr.deltalake/	0	1701500875000
2	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_1yr_with_cancelled.deltalake/	FeatEng_ALL_1yr_with_cancelled.deltalake/	0	1701565587000
3	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3mo.deltalake/	FeatEng_ALL_3mo.deltalake/	0	1701499549000
4	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3yr.deltalake/	FeatEng_ALL_3yr.deltalake/	0	1701503136000
5	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3yr_with_cancelled.deltalake/	FeatEng_ALL_3yr_with_cancelled.deltalake/	0	1701567760000
6	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_5yr.deltalake/	FeatEng_ALL_5yr.deltalake/	0	1701505517000
7	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_5yr_w_HolidayFeat.deltalake/	FeatEng_ALL_5yr_w_HolidayFeat.deltalake/	0	1702003210000

27 rows

```
DELTALAKE_2HRDELAY = f'{team_blob_url}/FeatEng_ALL_5yr_w_HolidayFeat_UPDATED.deltalake'
dbutils.fs.rm(DELTALAKE_2HRDELAY, recurse=True)
display(dbutils.fs.ls(f"{team_blob_url}"))
```

Table

	path	name	size	modificationTime
1	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_1yr.deltalake/	FeatEng_ALL_1yr.deltalake/	0	1701500875000
2	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_1yr_with_cancelled.deltalake/	FeatEng_ALL_1yr_with_cancelled.deltalake/	0	1701565587000
3	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3mo.deltalake/	FeatEng_ALL_3mo.deltalake/	0	1701499549000
4	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3yr.deltalake/	FeatEng_ALL_3yr.deltalake/	0	1701503136000
5	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3yr_with_cancelled.deltalake/	FeatEng_ALL_3yr_with_cancelled.deltalake/	0	1701567760000
6	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_5yr.deltalake/	FeatEng_ALL_5yr.deltalake/	0	1701505517000
7	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_5yr_w_Holiday_LagFeat.deltalake/	FeatEng_ALL_5yr_w_Holiday_LagFeat.deltalake/	0	1702005880000

25 rows

Table

	path	name	size	modificationTime
1	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_1yr.deltalake/	FeatEng_ALL_1yr.deltalake/	0	1701500875000
2	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_1yr_with_cancelled.deltalake/	FeatEng_ALL_1yr_with_cancelled.deltalake/	0	1701565587000
3	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3mo.deltalake/	FeatEng_ALL_3mo.deltalake/	0	1701499549000
4	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3yr.deltalake/	FeatEng_ALL_3yr.deltalake/	0	1701503136000
5	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_3yr_with_cancelled.deltalake/	FeatEng_ALL_3yr_with_cancelled.deltalake/	0	1701567760000
6	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_5yr.deltalake/	FeatEng_ALL_5yr.deltalake/	0	1701505517000
7	wasbs://w261storage@w261rtang.blob.core.windows.net/FeatEng_ALL_5yr_w_HolidayFeat_UPDATED.deltalake/	FeatEng_ALL_5yr_w_HolidayFeat_UPDATED.deltalake/	0	1701931527000

26 rows