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, mean
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

```
shaded.databricks.org.apache.hadoop.fs.azure.AzureException: hadoop_azure_shaded.com.microsoft.azure.storage.StorageException: Server failed to authenticate the request. Make sure the val
ue of Authorization header is formed correctly including the signature.
```

Select DF

- delta_otpw_3m_2015
- delta_otpw_3yr
- delta_otpw_3yr
- delta_otpw_5yr

Data Split

3-Year OTPW Splits for Phase 2:

- Train - Jan 2015 - May 2017
- Validation - June 2017 - Dec 2017

```
df_4y_train = delta_otpw_5yr.filter((F.col('YEAR') < 2019))
```

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

```
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.colu
  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

```
otpw_missing = null_check(df_4y_train, 'df_4y_train')
otpw_missing
```

```
df_4y_train contains 24279321 rows & 214 columns
```

	missing_count	missing_%
MonthlyStationPressure	24279321	100.0
MonthlyDaysWithGT90Temp	24279321	100.0
MonthlyDaysWithGT010Precip	24279321	100.0
MonthlyDaysWithGT001Precip	24279321	100.0
MonthlyAverageRH	24279321	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_4y_train, otpw_missing['missing_%'], threshold=100)
display_limited_df(df_cleaned, nrows=3)
```

```
columns_to_drop = ['MonthlyStationPressure', 'MonthlyDaysWithGT90Temp', 'MonthlyDaysWithGT010Precip', 'MonthlyDaysWithGT001Precip', 'MonthlyAverageRH', 'MonthlyMeanTemperature', 'MonthlyMin
SeaLevelPressureValue', 'MonthlyMinSeaLevelPressureValueDate', 'MonthlyMinSeaLevelPressureValueTime', 'MonthlyMinimumTemperature', 'MonthlySeaLevelPressure', 'MonthlyTotalLiquidPrecipitatio
n', 'MonthlyTotalSnowfall', 'MonthlyWetBulb', 'AWND', 'CDSD', 'CLDD', 'MonthlyDaysWithGT32Temp', 'MonthlyDaysWithLT0Temp', 'HDSD', 'MonthlyDaysWithLT32Temp', 'MonthlyDepartureFromNormalHeat
ingDegreeDays', 'MonthlyDepartureFromNormalMaximumTemperature', 'MonthlyDepartureFromNormalMinimumTemperature', 'MonthlyDepartureFromNormalPrecipitation', 'MonthlyDewpointTemperature', 'Mon
thlyGreatestPrecip', 'MonthlyGreatestPrecipDate', 'MonthlyGreatestSnowDepth', 'MonthlyGreatestSnowDepthDate', 'MonthlyGreatestSnowfall', 'MonthlyGreatestSnowfallDate', 'MonthlyMaxSeaLevelPr
essureValue', 'MonthlyMaxSeaLevelPressureValueDate', 'MonthlyMaxSeaLevelPressureValueTime', 'MonthlyMaximumTemperature', 'MonthlyDepartureFromNormalAverageTemperature', 'DSNW', 'HTDD', 'Sho
rtDurationEndDate150', 'ShortDurationPrecipitationValue180', 'ShortDurationPrecipitationValue150', 'ShortDurationPrecipitationValue120', 'ShortDurationPrecipitationValue100', 'ShortDuration
PrecipitationValue080', 'ShortDurationPrecipitationValue060', 'ShortDurationPrecipitationValue045', 'ShortDurationPrecipitationValue030', 'ShortDurationPrecipitationValue020', 'NormalsCooli
ngDegreeDay', 'ShortDurationPrecipitationValue010', 'ShortDurationPrecipitationValue005', 'ShortDurationEndDate180', 'ShortDurationPrecipitationValue015', 'ShortDurationEndDate120', 'ShortD
urationEndDate030', 'NormalsHeatingDegreeDay', 'ShortDurationEndDate010', 'ShortDurationEndDate015', 'ShortDurationEndDate020', 'ShortDurationEndDate005', 'MonthlyDepartureFromNormalCooling
DegreeDays', 'ShortDurationEndDate045', 'ShortDurationEndDate060', 'ShortDurationEndDate080', 'ShortDurationEndDate100']

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

Table

	QUARTER	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER_AIRLINE_ID	OP_CARRIER	TAIL_NUM	OP_CARRIER_FL_NUM	ORIGIN_AIRPORT_
1	1	15	4	2018-02-15	9E	20363	9E	N819AY	3281	10397
2	2	1	5	2018-06-01	9E	20363	9E	N326PQ	3281	11193
3	4	12	1	2018-11-12	9E	20363	9E	N8896A	3281	12951

3 rows

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', 'DailyAverageWetBulbTemp
erature', 'DailyAverageWindSpeed', 'DailyCoolingDegreeDays', 'DailyDepartureFromNormalAverageTemperature', 'DailyHeatingDegreeDays', 'DailyMaximumDryBulbTemperature', 'DailyMinimumDryBulbTe
mperature', 'DailyPeakWindDirection', 'DailyPeakWindSpeed', 'DailyPrecipitation', 'DailySnowDepth', 'DailySnowfall', 'DailySustainedWindDirection', 'DailySustainedWindSpeed', 'DailyWeathe
r', 'BackupDirection', 'BackupDistance', 'BackupDistanceUnit', 'BackupElements', 'BackupElevation', 'BackupEquipment', 'BackupLatitude', 'BackupLongitude', 'BackupName']
df_cleaned contains 24279321 rows & 119 columns
```

Table

	QUARTER	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER_AIRLINE_ID	OP_CARRIER	TAIL_NUM	OP_CARRIER_FL_NUM	ORIGIN_AIRPORT_
1	1	15	4	2018-02-15	9E	20363	9E	N819AY	3281	10397
2	2	1	5	2018-06-01	9E	20363	9E	N326PQ	3281	11193
3	4	12	1	2018-11-12	9E	20363	9E	N8896A	3281	12951

3 rows

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_ID',
    '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 = 35
df_cleaned contains 24279321 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)

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

```
df_cleaned contains 24279321 rows & 85 columns
```

Table

	QUARTER	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER	TAIL_NUM	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST
1	1	15	4	2018-02-15	9E	9E	N819AY	3281	ATL	Atlanta, GA	TRI
2	2	1	5	2018-06-01	9E	9E	N326PQ	3281	CVG	Cincinnati, OH	DTW
3	4	12	1	2018-11-12	9E	9E	N8896A	3281	LFT	Lafayette, LA	ATL

3 rows

3. Simple Statistics on the Remaining 115 Columns

```
display(df_cleaned.summary().toPandas())
```

Table

	summary	QUARTER	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER	TAIL_NUM	OP_CARRIER_FL_NUM
1	count	24279321	24279321	24279321	24279321	24279321	24279321	24224920	24279321
2	mean	2.513973805115884	15.745889516432523	3.931996615556094	null	null	null	8806.54128440367	2274.3546920443123
3	stddev	1.1048725279949183	8.772778527422744	1.9897185622019127	null	null	null	1.9556760226077607	1784.990480453386
4	min	1	1	1	2015-01-01	9E	9E	215NV	1
5	25%	2.0	8.0	2.0	null	null	null	8805.0	784.0
6	50%	3.0	16.0	4.0	null	null	null	8805.0	1791.0
7	75%	3.0	23.0	6.0	null	null	null	8809.0	3480.0

8 rows

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

```
['QUARTER', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'FL_DATE', 'OP_UNIQUE_CARRIER', 'OP_CARRIER', 'TAIL_NUM', 'OP_CARRIER_FL_NUM', 'ORIGIN', 'ORIGIN_CITY_NAME', 'DEST', 'DEST_CITY_NAME', 'CRS_DEP_TIME', 'DEP_TIME', 'DEP_DELAY', 'DEP_DELAY_NEW', 'DEP_DEL15', 'DEP_DELAY_GROUP', 'DEP_TIME_BLK', 'TAXI_OUT', 'TAXI_IN', 'CRS_ARR_TIME', 'ARR_TIME', 'ARR_DELAY', 'ARR_DELAY_NEW', 'ARR_DEL15', 'ARR_DELAY_GROUP', 'ARR_TIME_BLK', 'CANCELLED', 'DIVERTED', 'CRS_ELAPSED_TIME', 'ACTUAL_ELAPSED_TIME', 'AIR_TIME', 'FLIGHTS', 'DISTANCE', 'DISTANCE_GROUP', 'CARRIER_DELAY', 'WEATHER_DELAY', 'NAS_DELAY', 'SECURITY_DELAY', 'LATE_AIRCRAFT_DELAY', 'TOTAL_ADD_GTIME', 'LONGEST_ADD_GTIME', 'YEAR', 'MONTH', 'origin_airport_name', 'origin_iata_code', 'origin_type', 'origin_airport_lat', 'origin_airport_lon', 'origin_station_dis', 'dest_airport_name', 'dest_iata_code', 'dest_type', 'dest_airport_lat', 'dest_airport_lon', 'dest_station_dis', 'sched_depart_date_time_UTC', 'four_hours_prior_depart_UTC', 'two_hours_prior_depart_UTC', 'STATION', 'DATE', 'LATITUDE', 'LONGITUDE', 'ELEVATION', 'REPORT_TYPE', 'SOURCE', 'HourlyAltimeterSetting', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyPrecipitation', 'HourlyPresentWeatherType', 'HourlyPressureChange', 'HourlyPressureTendency', 'HourlyRelativeHumidity', 'HourlySkyConditions', 'HourlySeaLevelPressure', 'HourlyStationPressure', 'HourlyVisibility', 'HourlyWetBulbTemperature', 'HourlyWindDirection', 'HourlyWindGustSpeed', 'HourlyWindSpeed', 'Sunrise', 'Sunset']
```

```
# Range of DEP_DELAY and DEP_DELAY_Double
'''
DEP_DELAY: -1 to 99
DEP_DELAY_Dboule: -61 to 119
'''
df_cleaned = df_cleaned.withColumn("DEP_DELAY_Double", F.col("DEP_DELAY").cast("double")).cache()
display(df_cleaned.summary().select('summary','DEP_DELAY','DEP_DELAY_Double'))
```

Table

	summary	DEP_DELAY	DEP_DELAY_Double
1	count	23933364	23933364
2	mean	9.5195458523925	9.5195458523925
3	stddev	41.6736729319035	41.6736729319035
4	min	-1.0	-234.0
5	25%	-5.0	-5.0
6	50%	-2.0	-2.0
7	75%	7.0	7.0

8 rows

Final Keep Columns - Compute Summary Statistics

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

2. Store `df_cleaned` to the Cloud Storage in Delta Lake Format

```
# # df_cleaned.write.parquet(f'{team_blob_url}/df_cleaned.parquet')
# # df_cleaned.write.mode('overwrite').parquet(f'{team_blob_url}/df_cleaned.parquet')

# # df_cleaned = spark.read.parquet(f'{team_blob_url}/df_cleaned.parquet/').cache()
# DELTALAKE_2HRDELAY = f'{team_blob_url}/df_cleaned.parquet'
# # dbutils.fs.rm(DELTALAKE_2HRDELAY, recurse=True)
# # df_cleaned.write.format('delta').mode('overwrite').save(DELTALAKE_2HRDELAY)
# delta_in2hrDelay = spark.read.format("delta").load(DELTALAKE_2HRDELAY).cache()

# # Shows "_delta_log", means preserving all the evoluation of your data. Delta lake is parque but the diff is the delta log, which keeps the version control of your data.
# display(dbutils.fs.ls(f"{team_blob_url}/df_cleaned.parquet"))
```

```
# df_cleaned = spark.read.parquet(f'{team_blob_url}/df_cleaned.parquet/').cache()
```

Store `df_cleaned` into a SQL table

```
df_cleaned.createOrReplaceTempView('otpw_cleaned')
```

```
sql_query="""
SELECT
    DEP_DELAY_GROUP,
    COUNT(*) AS delay_group_count,
    CAST(ROUND((COUNT(*) * 100.0 / SUM(COUNT(*)) OVER ()),4) AS DOUBLE) AS delay_group_percentage
FROM otpw_cleaned
WHERE DEP_DELAY_GROUP IS NOT NULL
GROUP BY DEP_DELAY_GROUP
ORDER BY delay_group_count DESC;
"""

delay_group = spark.sql(sql_query)
delay_group.show(truncate=False)
df_delay_group = delay_group.limit(20).toPandas()
```

```
+---------------+-----------------+----------------------+
|DEP_DELAY_GROUP|delay_group_count|delay_group_percentage|
+---------------+-----------------+----------------------+
|-1             |14119672         |58.9958               |
|0              |5377820          |22.47                 |
|1              |1575644          |6.5835                |
|2              |812301           |3.394                 |
```

```
|3             |503158          |2.1023                |
|4             |338011          |1.4123                |
|5             |241329          |1.0083                |
|12            |234008          |0.9777                |
|6             |178276          |0.7449                |
|7             |135834          |0.5676                |
|-2            |116133          |0.4852                |
|8             |104274          |0.4357                |
|9             |81103           |0.3389                |
|10            |64522           |0.2696                |
|11            |51279           |0.2143                |
+--------------+----------------+----------------------+
```

Extract Distinct Values for the Delected Columns

```
cols = ['TAIL_NUM',
          'OP_UNIQUE_CARRIER',
          'DAY_OF_WEEK',
          'DEP_DELAY_GROUP',
          'ARR_DELAY_GROUP',
          'ORIGIN_CITY_NAME',
          'DEST_CITY_NAME',
          'origin_airport_name',
          'dest_airport_name',
          'FL_DATE',
          'DEP_DEL15',
          'DEP_DELAY',
          'DEP_DELAY_Double',
          'DEP_DELAY_NEW']

extract_distinct_values_multiCol(df_cleaned, cols, nrows=15)
```

	col_name	sample_unique_val	total_num_unique	data_type
0	TAIL_NUM	[N303DN, N310DN, N312DN, N313DN, N318DX, N324U...	7540	string
1	OP_UNIQUE_CARRIER	[9E, DL, EV, F9, G4, HA, MQ, NK, OH, OO, UA, V...	19	string
2	DAY_OF_WEEK	[1, 2, 3, 4, 5, 6, 7]	7	string
3	DEP_DELAY_GROUP	[-1, -2, 0, 1, 10, 11, 12, 2, 3, 4, 5, 6, 7, 8...	15	string
4	ARR_DELAY_GROUP	[-1, -2, 0, 1, 10, 11, 12, 2, 3, 4, 5, 6, 7, 8...	15	string
5	ORIGIN_CITY_NAME	[Atlanta, GA, Austin, TX, Bristol/Johnson City...	357	string
6	DEST_CITY_NAME	[Alexandria, LA, Atlanta, GA, Bristol/Johnson ...	357	string
7	origin_airport_name	[Albuquerque International Sunport, Baltimore/...	364	string
8	dest_airport_name	[Birmingham-Shuttlesworth International Airpor...	363	string
9	FL_DATE	[2018-01-01, 2018-01-04, 2018-02-04, 2018-02-1...	1460	string
10	DEP_DEL15	[0.0, 1.0]	2	string
11	DEP_DELAY	[-1.0, -2.0, -3.0, -4.0, -7.0, 0.0, 1.0, 14.0,...	1657	string
12	DEP_DELAY_Double	[-15.0, -12.0, -10.0, -9.0, -6.0, -5.0, -4.0, ...	1657	double
13	DEP_DELAY_NEW	[0.0, 1.0, 110.0, 12.0, 15.0, 2.0, 22.0, 24.0,...	1571	string

```
# extract_distinct_values_singleCol(20, df_cleaned, col='DEP_DELAY_GROUP')
```

```
# extract_distinct_values_singleCol(40, df_cleaned, col='DEP_DELAY_Double')
```

```
# check if they are the same date
display(df_cleaned.select('FL_DATE','DATE').show(10))
```

```
+----------+-------------------+
|   FL_DATE|               DATE|
+----------+-------------------+
|2018-02-15|2018-02-15T14:52:00|
|2018-06-01|2018-06-01T12:52:00|
|2018-11-12|2018-11-12T16:53:00|
|2018-01-01|2018-01-01T13:53:00|
|2018-04-20|2018-04-20T07:00:00|
|2018-06-23|2018-06-23T13:52:00|
|2018-04-20|2018-04-20T15:53:00|
|2018-11-04|2018-11-04T10:54:00|
|2018-12-26|2018-12-26T14:52:00|
|2018-03-02|2018-03-02T18:52:00|
+----------+-------------------+
only showing top 10 rows
```

Delay Metrics:

DEP_DEL15:

- A binary indicator (0 or 1) that denotes whether a flight's departure was delayed by 15 minutes or more

DEP_DELAY:

- (Data Dict) Difference in minutes between scheduled and actual departure time. Early departures show negative numbers.
- Includes both early departures (negative values) and delayed departures (positive values)

DEP_DELAY_NEW:

- Treats early departures as 0 minutes delay, focusing solely on flights that depart later than scheduled

- (= **DepDelayMinutes**) Difference in minutes between scheduled and actual departure time. Early departures set to 0.

DEP_DELAY_GROUP:

- Departure Delay intervals, every (15 minutes from <-15 to >180)

Seasonality Check

: **FL_DATE** - convert date to day of year and use **DEP_DELAY_Double**

- Took avg across 5 years
- 2015 to 2019 (Leap year = 2016)
- Plots for average delays (DepDelay) across date
  - Barplot: hour of day
  - Barplot: day of week (weekdays vs weekends)
  - Lineplot: binary for holidays (0 or 1)
- Create new features using **OneHotEncoder** or **Label Encoding** : e.g. (yes=1, no=0) July 4th, Christmas, etc

TOBY Note: I can't remember exactly but I probably just dropped the February 29th. You don't need to reproduce exactly what I did, it's just an example. The easiest thing to do would be to plot each year on a separate plot with the date on the x axis. Then you could use the list of federal holidays you said you had, and mark them on the plot to see if they correlate with increase or decrease in delays.

Day of Year (https://nsidc.org/data/user-resources/help-center/day-year-doy-calendar)
Day of Year Chart (https://www.scp.byu.edu/docs/doychart.html)

1. Extract `Day of Year` of a Given Flight Date

Create Day of Year in PySpark (https://pandas.pydata.org/docs/reference/api/pandas.Series.dt.dayofyear.html)
Create Day of Year in Python (https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.functions.dayofyear.html)

```
df_cleaned = df_cleaned.withColumn('day_of_year', dayofyear(col('FL_DATE'))).cache()
print_df_shape(df_cleaned, 'df_cleaned')
display_limited_df(df_cleaned, 3)
```

```
df_cleaned contains 24279321 rows & 87 columns
```

Table

	QUARTER	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER	TAIL_NUM	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST
1	1	15	4	2018-02-15	9E	9E	N819AY	3281	ATL	Atlanta, GA	TRI
2	2	1	5	2018-06-01	9E	9E	N326PQ	3281	CVG	Cincinnati, OH	DTW
3	4	12	1	2018-11-12	9E	9E	N8896A	3281	LFT	Lafayette, LA	ATL

3 rows

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

```
['QUARTER', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'FL_DATE', 'OP_UNIQUE_CARRIER', 'OP_CARRIER', 'TAIL_NUM', 'OP_CARRIER_FL_NUM', 'ORIGIN', 'ORIGIN_CITY_NAME', 'DEST', 'DEST_CITY_NAME', 'CRS_DEP_TIME', 'DEP_TIME', 'DEP_DELAY', 'DEP_DELAY_NEW', 'DEP_DEL15', 'DEP_DELAY_GROUP', 'DEP_TIME_BLK', 'TAXI_OUT', 'TAXI_IN', 'CRS_ARR_TIME', 'ARR_TIME', 'ARR_DELAY', 'ARR_DELAY_NEW', 'ARR_DEL15', 'ARR_DELAY_GROUP', 'ARR_TIME_BLK', 'CANCELLED', 'DIVERTED', 'CRS_ELAPSED_TIME', 'ACTUAL_ELAPSED_TIME', 'AIR_TIME', 'FLIGHTS', 'DISTANCE', 'DISTANCE_GROUP', 'CARRIER_DELAY', 'WEATHER_DELAY', 'NAS_DELAY', 'SECURITY_DELAY', 'LATE_AIRCRAFT_DELAY', 'TOTAL_ADD_GTIME', 'LONGEST_ADD_GTIME', 'YEAR', 'MONTH', 'origin_airport_name', 'origin_iata_code', 'origin_type', 'origin_airport_lat', 'origin_airport_lon', 'origin_station_dis', 'dest_airport_name', 'dest_iata_code', 'dest_type', 'dest_airport_lat', 'dest_airport_lon', 'dest_station_dis', 'sched_depart_date_time_UTC', 'four_hours_prior_depart_UTC', 'two_hours_prior_depart_UTC', 'STATION', 'DATE', 'LATITUDE', 'LONGITUDE', 'ELEVATION', 'REPORT_TYPE', 'SOURCE', 'HourlyAltimeterSetting', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyPrecipitation', 'HourlyPresentWeatherType', 'HourlyPressureChange', 'HourlyPressureTendency', 'HourlyRelativeHumidity', 'HourlySkyConditions', 'HourlySeaLevelPressure', 'HourlyStationPressure', 'HourlyVisibility', 'HourlyWetBulbTemperature', 'HourlyWindDirection', 'HourlyWindGustSpeed', 'HourlyWindSpeed', 'Sunrise', 'Sunset', 'DEP_DELAY_Double', 'day_of_year']
```

1-1. `Lineplot` : Average Departure and Arrival Flight Delay by Day of the Year

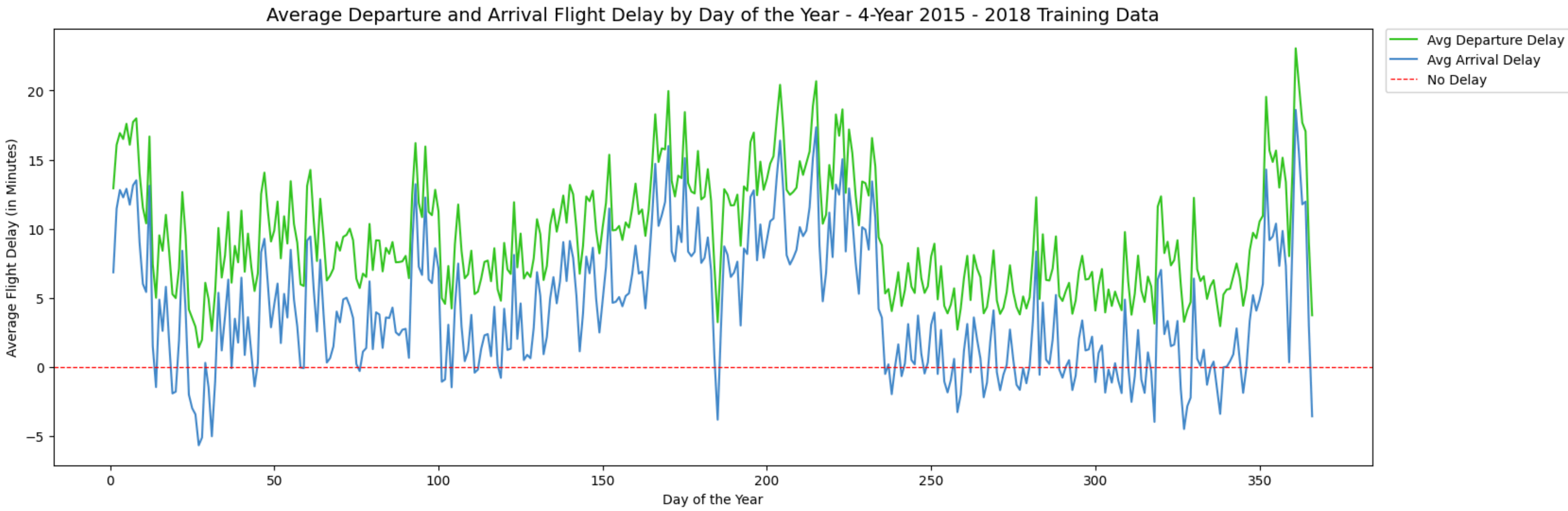
```
# Convert ARR_DELAY and DISTANCE from string to double format
df_cleaned = df_cleaned.withColumn("DEP_DELAY_Double", F.col("DEP_DELAY").cast("double")).cache()
df_cleaned = df_cleaned.withColumn("ARR_DELAY_Double", F.col("ARR_DELAY").cast("double")).cache()

# Calculate the average delays by day of the year
avg_dist_by_days = df_cleaned.groupBy('day_of_year').agg(
  F.avg('DEP_DELAY_Double').alias('Avg_DEP_DELAY'),
  F.avg('ARR_DELAY_Double').alias('Avg_ARR_DELAY')).cache()

display(avg_dist_by_days)

pdf_avg_dist_by_days = avg_dist_by_days.toPandas()

# Plotting
plt.figure(figsize=(18, 6))
sns.lineplot(x='day_of_year', y='Avg_DEP_DELAY', data=pdf_avg_dist_by_days, label='Avg Departure Delay', color='#32c722')
sns.lineplot(x='day_of_year', y='Avg_ARR_DELAY', data=pdf_avg_dist_by_days, label='Avg Arrival Delay', color='#4189cc')
plt.axhline(0, color='red', linestyle='--', linewidth=0.9, label='No Delay')

plt.title('Average Departure and Arrival Flight Delay by Day of the Year - 4-Year 2015 - 2018 Training Data', fontsize=14)
plt.xlabel('Day of the Year')
plt.ylabel('Average Flight Delay (in Minutes)')
plt.legend(loc='upper right', bbox_to_anchor=(1.155,1.015))
plt.show()
```

Table

	day_of_year	Avg_DEP_DELAY	Avg_ARR_DELAY

1	148	9.86772696141846	4.942194763885491
2	243	7.514695205181076	3.1037830745681174
3	31	2.6051167775188393	-5.0184165203008035
4	85	8.191685110445688	3.5420411713540156
5	251	8.920263149387386	3.935009564568946
6	137	10.984684554024655	6.198359487485274
7	65	9.423256825845753	3.8391434262948207

366 rows



1-2. `Lineplot` : Daily and Rolling Averages of Flight Delays With `Holiday Effect`

- Daily Average
- 7-Day Rolling Average
- 28-Day Rolling Average

Helper Functions - Holiday Effect

```python
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",
    "Chiristmas 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

# 5. Extract holidays for the given time series dataset
def day_of_year_by_holiday(df):
  df = df.alias('delay')
  df_holiday_dates = get_holidays().alias('holiday')

  # Join with df_cleaned DataFrame to find matching dates and days of the year
  matched_holidays_df = df.join(
      df_holiday_dates,
      (col('delay.FL_DATE') == col('holiday.holiday_date')) &
      (col('delay.day_of_year') == col('holiday.day_of_year'))
  ).cache()

  # Select the "day_of_year" column and collect as a list of unique holiday
  day_of_year_unique = matched_holidays_df.select("holiday.day_of_year").rdd.flatMap(lambda x: x).distinct().collect()
  return day_of_year_unique

# 6. Generate a dictionary: key = day of year, value = holiday name
def dict_day_holiday(df):
  consolidated_holiday_dict = generate_consolidated_holiday_dict()
  common_days_set = set([(day, holiday_name) for (fl_date, day), holiday_name in consolidated_holiday_dict.items() if day in day_of_year_by_holiday(df)])
  print(f'\ncommon_days_set = {common_days_set}\n')
  return common_days_set
```

```python
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): 'Chiristmas 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): 'Chiristmas Day',
```

```
# ============ CALCULATE AVERAGE DEP_DELAY ============

# First, calculate the daily average of DEP_DELAY
daily_avg_df = df_cleaned.groupBy("day_of_year").agg(F.avg("DEP_DELAY").alias("Daily_Avg_DEP_DELAY")).cache()
display(daily_avg_df.collect())

overall_avg_dep_delay = df_cleaned.agg(
    F.avg("DEP_DELAY").alias("Avg_DEP_DELAY")
).collect()[0]["Avg_DEP_DELAY"]

# Define window specifications for 3-day and 7-day rolling averages
windowSpec7 = Window.orderBy("day_of_year").rowsBetween(-6, 0)   # 7-day window
windowSpec28 = Window.orderBy("day_of_year").rowsBetween(-27, 0)  # 28-day window

# Calculate the 3-day rolling average
daily_avg_df = daily_avg_df.withColumn('7day_Avg_DEP_DELAY', F.avg('Daily_Avg_DEP_DELAY').over(windowSpec7)).cache()   # 7-day rolling average
daily_avg_df = daily_avg_df.withColumn('28day_Avg_DEP_DELAY', F.avg('Daily_Avg_DEP_DELAY').over(windowSpec28)).cache()   # 28-day rolling average

display(daily_avg_df)

# Convert the Spark DataFrame to a Pandas DataFrame for further use
pdf_avg_dist_by_days = daily_avg_df.toPandas()

# Plotting
plt.figure(figsize=(18, 6))
sns.lineplot(x='day_of_year', y='Daily_Avg_DEP_DELAY', data=pdf_avg_dist_by_days, label='Daily Avg Delay', color='#dec90d')
sns.lineplot(x='day_of_year', y='7day_Avg_DEP_DELAY', data=pdf_avg_dist_by_days, label='7-Day Avg Delay', color='#5d9cf5')
sns.lineplot(x='day_of_year', y='28day_Avg_DEP_DELAY', data= pdf_avg_dist_by_days, label='28-Day Avg Delay', color='#7007ba')

plt.axhline(overall_avg_dep_delay, color='red', linestyle='--', linewidth=0.9, label=f'Average = {overall_avg_dep_delay:.2f}')

holidays_features = [1,185,359]
holiday_names = ["New Year's Day", "July 4th", "Christmas"]
# list(zip(holidays_features, holiday_names))
# for day, holiday_name in list(zip(holidays_features, holiday_names)):
#   print()
y_height, text_x_offset, text_y_offset = 15, 15, 15
for day, holiday_name in list(zip(holidays_features, holiday_names)):
  plt.axvline(x=day, color='black', linestyle='--', linewidth=0.6, label=f'{holiday_name}')
  plt.annotate(f'{holiday_name}', xy=(day, y_height),
             xytext=(text_x_offset, text_y_offset), textcoords='offset points',
            #  arrowprops=dict(arrowstyle='->', connectionstyle='angle,angleA=0.2,angleB=45,rad=0'),
             arrowprops=dict(arrowstyle='->', connectionstyle='arc3,rad=.55'),
             bbox=dict(boxstyle="round,pad=0.3", edgecolor='blue', facecolor='lightblue'))

plt.title('Daily and Rolling Averages of Flight Delays With Holiday Effect - 4-Year 2015 - 2018 Training Data', fontsize=14)
plt.xlabel('Day of the Year')
plt.ylabel('Average Delay (in Minutes)')
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(loc='upper right', bbox_to_anchor=(1.15,1.015))
plt.show()
```

Table

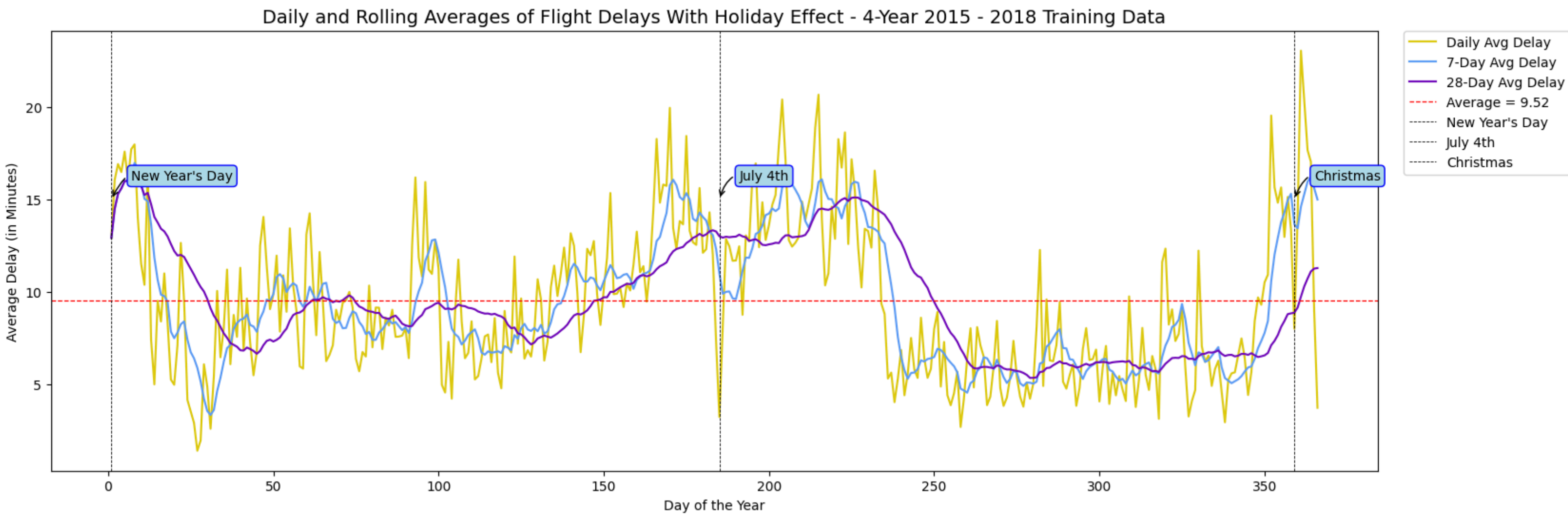
	day_of_year	Daily_Avg_DEP_DELAY
1	148	9.86772696141846
2	243	7.514695205181076
3	31	2.6051167775188393
4	137	10.984684554024655
5	85	8.191685110445688
6	251	8.920263149387386
7	65	9.423256825845753

366 rows

Table

	day_of_year	Daily_Avg_DEP_DELAY	7day_Avg_DEP_DELAY	28day_Avg_DEP_DELAY
1	1	12.930452525462123	12.930452525462123	12.930452525462123
2	2	16.07677352262965	14.503613024045887	14.503613024045887
3	3	16.925858024020012	15.311028024037261	15.311028024037261
4	4	16.495338699407338	15.607105692879781	15.607105692879781
5	5	17.600288440204128	16.00574224234465	16.00574224234465
6	6	16.080966173755442	16.01827956424645	16.01827956424645
7	7	17.729504329952487	16.262740245061597	16.262740245061597

366 rows




```
# display_limited_df(df_cleaned, 3)
```

```
# extract_distinct_values_singleCol(20, df_cleaned, col='DEP_DELAY_Double')
```

`Histogram` : Understand `Data Distribution` and Detecting `Central Tendency`

- Observing variability
- Detecting outliers and gaps
- Assessing normality

```
!pip install pyspark_dist_explore
```

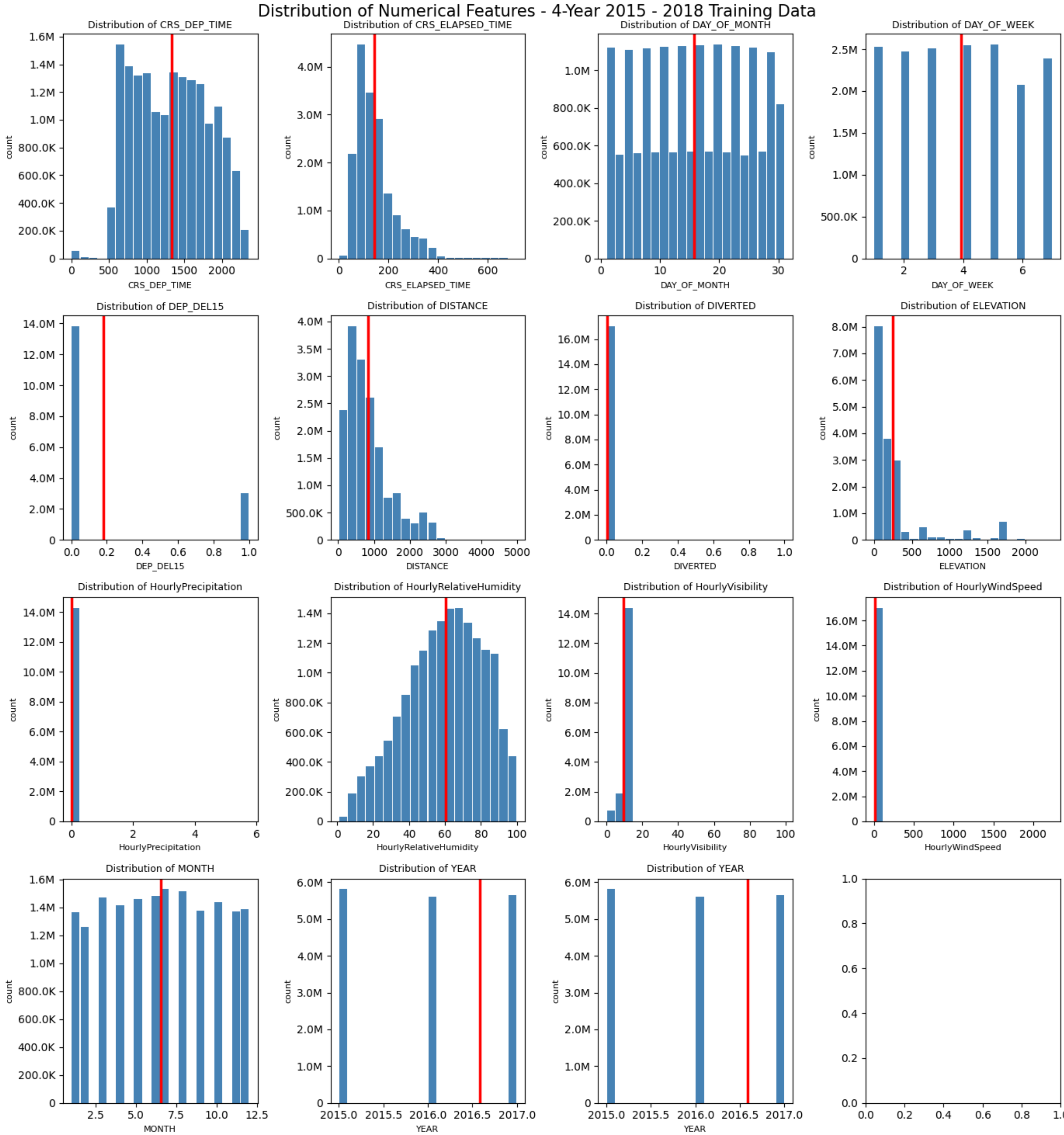
```
Collecting pyspark_dist_explore
  Downloading pyspark_dist_explore-0.1.8-py3-none-any.whl (7.2 kB)
Requirement already satisfied: numpy in /databricks/python3/lib/python3.10/site-packages (from pyspark_dist_explore) (1.21.5)
Requirement already satisfied: matplotlib in /databricks/python3/lib/python3.10/site-packages (from pyspark_dist_explore) (3.5.2)
Requirement already satisfied: pandas in /databricks/python3/lib/python3.10/site-packages (from pyspark_dist_explore) (1.4.4)
Requirement already satisfied: scipy in /databricks/python3/lib/python3.10/site-packages (from pyspark_dist_explore) (1.9.1)
Requirement already satisfied: pillow>=6.2.0 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->pyspark_dist_explore) (9.2.0)
Requirement already satisfied: pyparsing>=2.2.1 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->pyspark_dist_explore) (3.0.9)
Requirement already satisfied: kiwisolver>=1.0.1 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->pyspark_dist_explore) (1.4.2)
Requirement already satisfied: fonttools>=4.22.0 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->pyspark_dist_explore) (4.25.0)
Requirement already satisfied: python-dateutil>=2.7 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->pyspark_dist_explore) (2.8.2)
Requirement already satisfied: cycler>=0.10 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->pyspark_dist_explore) (0.11.0)
Requirement already satisfied: packaging>=20.0 in /databricks/python3/lib/python3.10/site-packages (from matplotlib->pyspark_dist_explore) (21.3)
Requirement already satisfied: pytz>=2020.1 in /databricks/python3/lib/python3.10/site-packages (from pandas->pyspark_dist_explore) (2022.1)
Requirement already satisfied: six>=1.5 in /usr/lib/python3/dist-packages (from python-dateutil>=2.7->matplotlib->pyspark_dist_explore) (1.16.0)
Installing collected packages: pyspark_dist_explore
Successfully installed pyspark_dist_explore-0.1.8

[notice] A new release of pip available: 22.2.2 -> 23.3.1
[notice] To update, run: pip install --upgrade pip
```

```
from pyspark_dist_explore import Histogram, hist, distplot, pandas_histogram

numeric_features = ['CRS_DEP_TIME', 'CRS_ELAPSED_TIME', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'DEP_DEL15', 'DISTANCE', 'DIVERTED', 'ELEVATION', 'HourlyPrecipitation',
'HourlyRelativeHumidity', 'HourlyVisibility' 'HourlyWindSpeed', 'MONTH', 'YEAR']

data_type_map = {
    'HourlyPrecipitation': DoubleType(),
    'DEP_DEL15': DoubleType(),
    'HourlyRelativeHumidity': IntegerType(),
    'HourlyWindSpeed': IntegerType(),
    'HourlyVisibility': DoubleType(),
    'CRS_ELAPSED_TIME': DoubleType(),
    'DAY_OF_MONTH': IntegerType(),
    'MONTH': IntegerType(),
    'CRS_DEP_TIME': IntegerType(),
    'DIVERTED': DoubleType(),
    'DISTANCE': DoubleType(),
    'DISTANCE_GROUP': IntegerType(),
    'DAY_OF_WEEK': IntegerType(),
    'YEAR': IntegerType(),
    'ELEVATION': DoubleType()
   }


df_otpw_3yr_dtype = delta_otpw_3yr\
  .select([col(column_schema[0]).cast(data_type_map.get(column_schema[0], column_schema[1])) for column_schema in delta_otpw_3yr.dtypes])

pd_otpw3yr_summary = df_cleaned.summary().toPandas()

otpw3yr_summary_stats = pd_otpw3yr_summary.T
new_header = otpw3yr_summary_stats.iloc[0]
otpw3yr_summary_stats = otpw3yr_summary_stats[1:]
otpw3yr_summary_stats.columns = new_header
display(otpw3yr_summary_stats)

numeric_features = ['CRS_DEP_TIME', 'CRS_ELAPSED_TIME', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'DEP_DEL15', 'DISTANCE', 'DIVERTED', 'ELEVATION', 'HourlyPrecipitation',
'HourlyRelativeHumidity', 'HourlyVisibility', 'HourlyWindSpeed', 'MONTH', 'YEAR']
feature_index = 0
nrows, ncols = 4, 4
fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(14, 15))
# fig, axes = plt.subplots(nrows=nrows, ncols=ncols)
for r in range(nrows):
    for c in range(ncols):
        if not(r == nrows-1 and c == ncols-1):
            hist(axes[r,c], df_otpw_3yr_dtype.select(numeric_features[feature_index]), bins = 20, color=['#4682B4'], rwidth=0.9)
            mean_value = float(otpw3yr_summary_stats['mean'][numeric_features[feature_index]])
            axes[r,c].axvline(mean_value, c='r', ls='-', lw=2.5, label='mean')
            axes[r,c].set_title(f"Distribution of {numeric_features[feature_index]}", fontsize=9)
            axes[r,c].set_xlabel(numeric_features[feature_index], fontsize=8)
            axes[r,c].set_ylabel("count", fontsize=8)
            # axes[r,c].legend(loc="upper right")
            # axes[r,c].legend(loc="upper right")
            if feature_index != len(numeric_features)-1:
                feature_index += 1
fig.suptitle('Distribution of Numerical Features - 4-Year 2015 - 2018 Training Data', fontsize=15)
plt.tight_layout()
```

Table

	count	mean	stddev	min	25%	50%	75%	max
1	24279321	2.513973805115884	1.1048725279949183	1	2.0	3.0	3.0	4
2	24279321	15.745889516432523	8.772778527422744	1	8.0	16.0	23.0	9
3	24279321	3.931996615556094	1.9897185622019127	1	2.0	4.0	6.0	7
4	24279321	null	null	2015-01-01	null	null	null	2018-12-31
5	24279321	null	null	9E	null	null	null	YX
6	24279321	null	null	9E	null	null	null	YX
7	24224920	8806.54128440367	1.955676022607777	215NV	8805.0	8805.0	8809.0	SS25

88 rows



`Skewness Check`

```
# skewness_results = df_cleaned.select([F.skewness(F.col(c)).alias(c) for c in final_keep_columns]).cache()
# display(skewness_results)
```

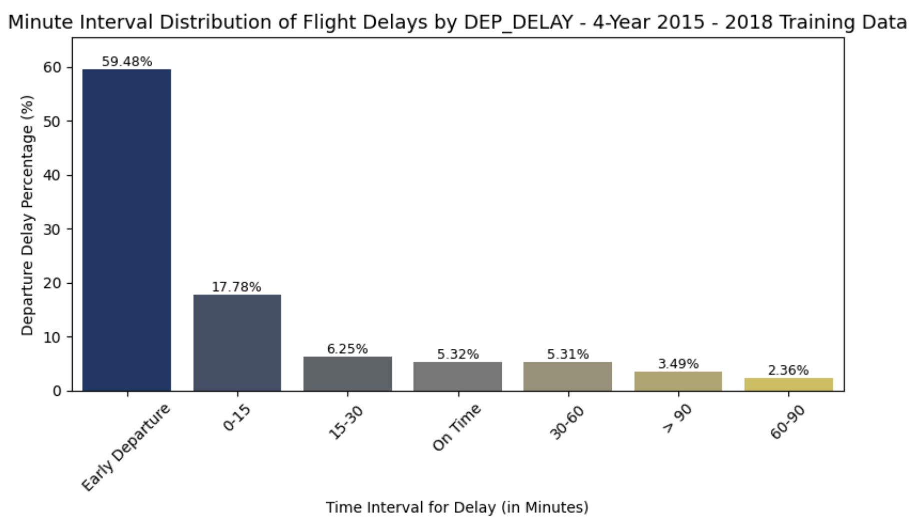
- **MONTH:** Skewness = 0.2455, which suggests a slight right skew. The distribution of months is slightly skewed towards the earlier part of the year.
- **YEAR:** Skewness = 0.4313, indicating a moderate right skew. The years in your dataset may be clustering towards the start of your range.
- **DEP_DEL15:** Skewness = 1.6695, a more pronounced right skew. This suggests that there are more instances with smaller delays, but a few large delays are causing a long right tail.
- **DAY_OF_MONTH:** Skewness = 0.0491, which is very close to 0, indicating a fairly symmetrical distribution of days across the month.
- **DISTANCE:** Skewness = 1.3787, which shows a moderate to high right skew, meaning most flights cover shorter distances, with fewer longer flights.
- **DAY_OF_WEEK:** Skewness = 0.0571, again close to 0, which suggests an almost uniform distribution across days of the week.
- **CANCELLED:** Skewness = 8.3063, which is highly skewed to the right. This indicates that cancellations are rare but when they occur, they can have a wide variation in frequency.

Helper Function: `Barplot`

```
def create_barplot(pdf, x_col, y_col, palette, xlabel, ylabel, title, custom_labels, uom, figsize, dec_places, rotation, dict_lookup, labels_legend, show_legend=False):
  plt.figure(figsize=figsize)
  barplot = sns.barplot(
      x=x_col,
      y=y_col,
      data=pdf,
      palette=palette,
      order=pdf[x_col]
  )
  y_legend = pdf[y_col].map(float)

  # Generate legend
  if show_legend:
      legend = plt.legend(labels=labels_legend, loc='upper right')
      cmap = get_cmap(palette)
      colors = cmap(np.linspace(0,1, len(labels_legend)))
      for bar, color in zip(barplot.patches, colors):
        bar.set_color(color)
      for handle, color in zip(legend.legendHandles, colors):
        handle.set_color(color)

  padding = 0.05
  for p in barplot.patches:
      width = p.get_width()
      plt.text(p.get_x() + width / 2,
              p.get_height() + padding,
              f'{p.get_height():.{dec_places}f}{uom}',
              ha='center',
              va='bottom',
              fontsize=9)

  # Replace the x-ticks with day names using the sorted DAY_OF_WEEK
  plt.xticks(ticks=range(len(pdf)),
            labels=custom_labels,
            rotation=rotation)

  # Set the title and labels
  plt.title(title, fontsize=13)
  plt.xlabel(xlabel)
  plt.ylabel(ylabel)
  max_height = max([p.get_height() for p in barplot.patches])
  plt.ylim(0, max_height + (max_height * 0.1))
  plt.tight_layout()
  plt.show()
```

Flight Delay Distribution by Time Interval (DEP_DELAY)

```
df_cleaned = df_cleaned.withColumn("DEP_DELAY_Double", F.col("DEP_DELAY").cast("double")).cache()

delay_intervals = (F.when((F.col('DEP_DELAY_Double') == 0), 'On Time')
                   .when(F.col('DEP_DELAY_Double') < 0, 'Early Departure')
                   .when((F.col('DEP_DELAY_Double') > 0) & (F.col('DEP_DELAY_Double') <= 15), '0-15')
                   .when((F.col('DEP_DELAY_Double') > 15) & (F.col('DEP_DELAY_Double') <= 30), '15-30')
                   .when((F.col('DEP_DELAY_Double') > 30) & (F.col('DEP_DELAY_Double') <= 60), '30-60')
                   .when((F.col('DEP_DELAY_Double') > 60) & (F.col('DEP_DELAY_Double') <= 90), '60-90')
                   .when(F.col('DEP_DELAY_Double') > 90, '> 90')
                   .otherwise('Others').alias('delay_time_interval'))

# Apply the transformation and perform the aggregation
df_delay_by_time_interval = df_cleaned.filter(F.col('DEP_DELAY_Double').isNotNull()).cache()
df_delay_by_time_interval = df_delay_by_time_interval.select(delay_intervals, 'DEP_DELAY_Double') \
                                                            .groupBy('delay_time_interval') \
                                                            .count() \
                                                            .withColumnRenamed('count', 'count_time_interval_delay')

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

# count_dep_delay_by_days_sorted = count_dep_delay_by_days_sorted.withColumn('Total_Count', total_count)
df_delay_by_time_interval = df_delay_by_time_interval.withColumn('Percentage_Dep_Delay_by_Time_Interval', (F.col('count_time_interval_delay')/F.col('Total_Count')) * 100)
# display(df_delay_by_time_interval.collect())

# Convert to pandas DataFrame if needed
pd_delay_by_time_intervals = df_delay_by_time_interval.toPandas().sort_values('count_time_interval_delay', ascending=False)
# pd_delay_by_time_intervals

# ================== PLOTTING: BARPLOT ==================

labels_sorted = [interval for interval in pd_delay_by_time_intervals['delay_time_interval']]

create_barplot(
    pdf=pd_delay_by_time_intervals,
    x_col='delay_time_interval',
    y_col='Percentage_Dep_Delay_by_Time_Interval',
    palette='cividis',
    xlabel='Time Interval for Delay (in Minutes)',
    ylabel='Departure Delay Percentage (%)',
    title='Minute Interval Distribution of Flight Delays by DEP_DELAY - 4-Year 2015 - 2018 Training Data',
    custom_labels=labels_sorted,
    uom='%',
    figsize=(8, 5),
    dec_places=2,
    rotation=45,
    dict_lookup=None,
    labels_legend=None,
    show_legend=False
  )
```



Flight Delay by DEP_DELAY_GROUP

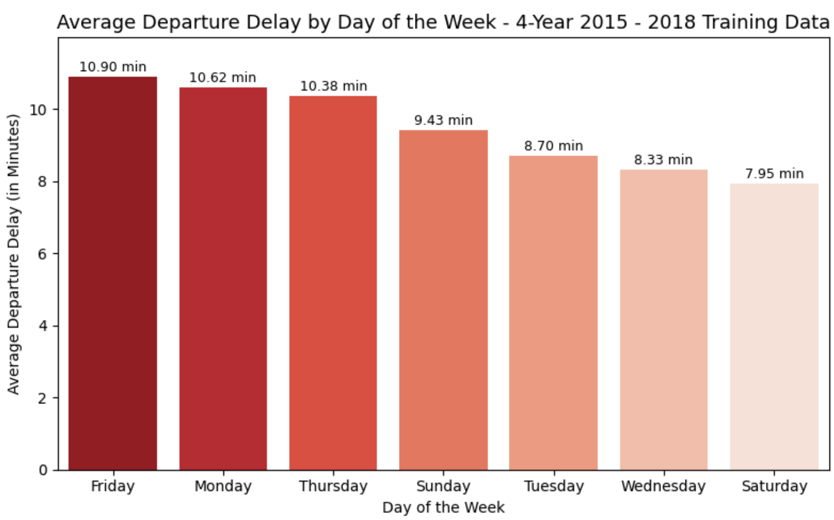
```python
# DEP_DELAY_GROUP: Departure Delay intervals, every (15 minutes from <-15 to >180)
df_cleaned = df_cleaned.withColumn("DEP_DELAY_Double", F.col("DEP_DELAY").cast("double")).cache()
dict_dep_delay_group = {
    '0' : '0 <= Delay < 15',
    '1' : '15 <= Delay < 30',
    '2' : '30 <= Delay < 45',
    '3' : '45 <= Delay < 60',
    '4' : '60 <= Delay < 75',
    '5' : '75 <= Delay < 90',
    '6' : '90 <= Delay < 105',
    '7' : '105 <= Delay < 120',
    '8' : '120 <= Delay < 135',
    '9' : '135 <= Delay < 150',
    '10': '150 <= Delay < 165',
    '11': '165 <= Delay < 180',
    '12': 'Delay >= 180',
    '-1': 'Early Dep (-15 < Delay < -1 )',
    '-2': 'Early Dep (Delay < -15)'}

# Filter data only with the flight delay (DEP_DELAY_Double > 0, as DEP_DELAY_Double = 0 means No Delay
# df_cleaned_filtered = df_cleaned.filter(F.col('DEP_DELAY_Double') > 0)

# Calculate the count of DEP_DELAY_Double group by DEP_TIME_BLK
count_dep_delay_by_minute = df_cleaned.groupBy('DEP_DELAY_GROUP').agg(F.count('DEP_DELAY_Double').alias('Count_DEP_DELAY')).cache()

# Calculate the total count of 'Count_DEP_DELAY' over the entire DataFrame
windowSpec = Window.partitionBy()
count_dep_delay_by_minute = count_dep_delay_by_minute.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_minute = count_dep_delay_by_minute.withColumn('Percentage_Dep_Delay_by_Minute_Interval', (F.col('Count_DEP_DELAY')/F.col('Total_Count')) * 100)

pdf_count_dep_delay_by_minute = count_dep_delay_by_minute.toPandas().sort_values('Percentage_Dep_Delay_by_Minute_Interval', ascending=False)

# ================== PLOTTING: BARPLOT ==================

labels_sorted = [dict_dep_delay_group[label] for label in pdf_count_dep_delay_by_minute['DEP_DELAY_GROUP']]

create_barplot(
    pdf=pdf_count_dep_delay_by_minute,
    x_col='DEP_DELAY_GROUP',
    y_col='Percentage_Dep_Delay_by_Minute_Interval',
    palette='cividis',
    xlabel='Departure Time Block (Minute Delay Intervals)',
    ylabel='Percentage (%)',
    title='Minute Interval Breakdown of Departure Delays in Percentage by DEP_DELAY_GROUP',
    custom_labels=labels_sorted,
    uom='%',
    figsize=(12, 6),
    dec_places=2,
    rotation=45,
    dict_lookup=None,
    labels_legend=None,
    show_legend=False
)
```

```
KeyError: None
```

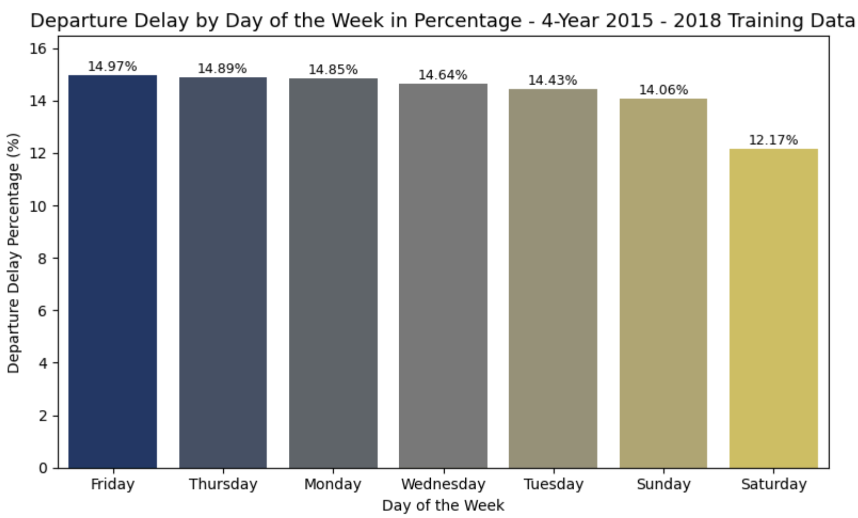
Flight Delay by Day of Week

- Weekadays Vs. Weekends
- DAY_OF_WEEK vs DEL_DELAY
- `Barplot`

```
day_of_week = {
  1:    'Monday',
  2:    'Tuesday',
  3:    'Wednesday',
  4:    'Thursday',
  5:    'Friday',
  6:    'Saturday',
  7:    'Sunday'
  }

# Create a new column, DAY_OF_WEEK_Int, after onverting the column from string to integer format
df_cleaned = df_cleaned.withColumn('DAY_OF_WEEK_Int', F.col('DAY_OF_WEEK').cast("integer")).cache()

# Calculate the average delays by day of the week
avg_dep_delay_by_days = df_cleaned.groupBy('DAY_OF_WEEK_Int').agg(F.avg('DEP_DELAY_Double').alias('Avg_DEP_DELAY')).cache()

# pdf_avg_dep_delay_by_days['DAY_OF_WEEK'] = pdf_avg_dep_delay_by_days['DAY_OF_WEEK'].astype(int)
pdf_avg_dep_delay_by_days = avg_dep_delay_by_days.toPandas()

# Sort the DataFrame by 'Avg_DEP_DELAY' in descending order
pdf_avg_dep_delay_by_days_sorted = pdf_avg_dep_delay_by_days.sort_values('Avg_DEP_DELAY', ascending=False)

# ================== PLOTTING: BARPLOT ==================

labels_sorted = [day_of_week[day] for day in pdf_avg_dep_delay_by_days_sorted['DAY_OF_WEEK_Int']]

create_barplot(
    pdf=pdf_avg_dep_delay_by_days_sorted,
    x_col='DAY_OF_WEEK_Int',
    y_col='Avg_DEP_DELAY',
    palette='Reds_r',
    xlabel='Day of the Week',
    ylabel='Average Departure Delay (in Minutes)',
    title='Average Departure Delay by Day of the Week - 4-Year 2015 - 2018 Training Data',
    custom_labels=labels_sorted,
    uom=' min',
    figsize=(8, 5),
    dec_places=2,
    rotation=0,
    dict_lookup=None,
    labels_legend=None,
    show_legend=False
  )
```



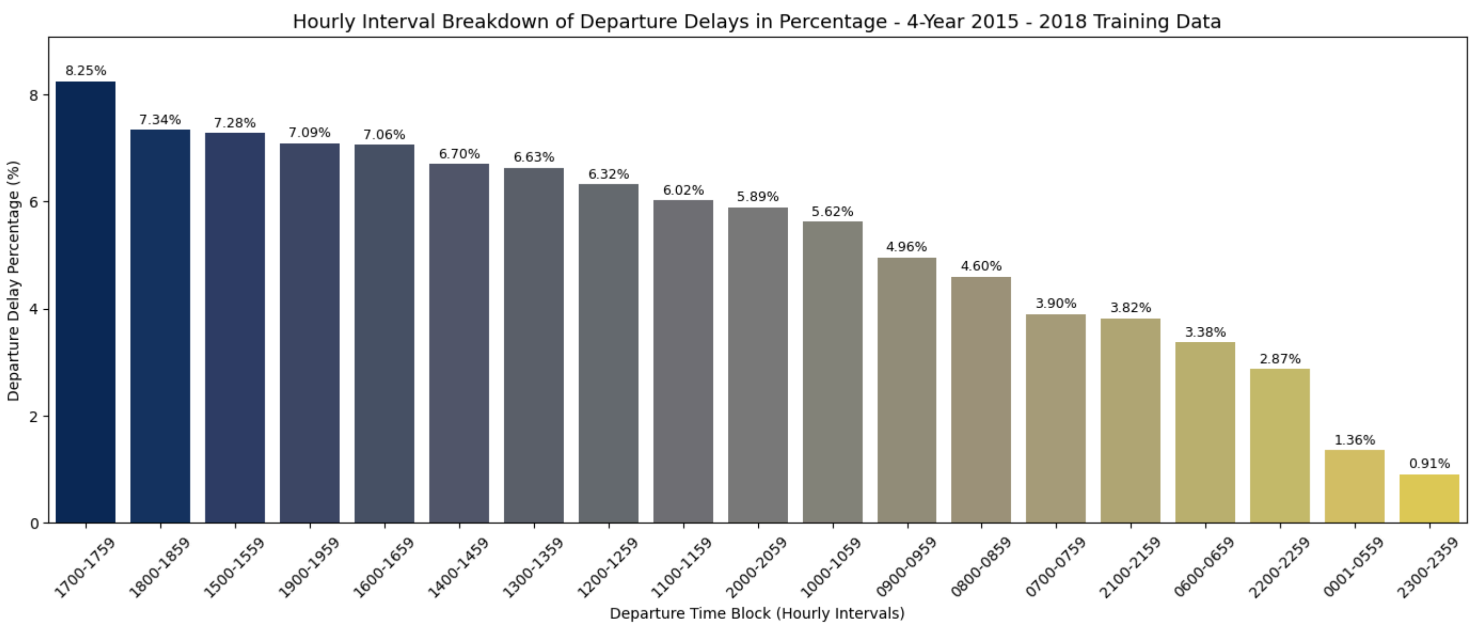
```
count_dep_delay_by_day = df_cleaned.groupBy('DAY_OF_WEEK_Int').agg(F.count(F.when(F.col('DEP_DELAY_Double').isNotNull(), 1)).alias('Count_DEP_DELAY')).cache()

# Calculate the total count of 'Count_DEP_DELAY' over the entire DataFrame
windowSpec = Window.partitionBy()
count_dep_delay_by_day = count_dep_delay_by_day.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_day = count_dep_delay_by_day.withColumn('Percentage_Dep_Delay_by_Minute_Interval', (F.col('Count_DEP_DELAY')/F.col('Total_Count')) * 100)

pdf_count_dep_delay_by_day = count_dep_delay_by_day.toPandas().sort_values('Count_DEP_DELAY', ascending=False)

# ================== PLOTTING: BARPLOT ==================

labels_sorted = [day_of_week[day] for day in pdf_count_dep_delay_by_day['DAY_OF_WEEK_Int']]

create_barplot(
    pdf=pdf_count_dep_delay_by_day,
    x_col='DAY_OF_WEEK_Int',
    y_col='Percentage_Dep_Delay_by_Minute_Interval',
    palette='cividis',
    xlabel='Day of the Week',
    ylabel='Departure Delay Percentage (%)',
    title='Departure Delay by Day of the Week in Percentage - 4-Year 2015 - 2018 Training Data',
    custom_labels=labels_sorted,
    uom='%',
    figsize=(8, 5),
    dec_places=2,
    rotation=0,
    dict_lookup=None,
    labels_legend=None,
    show_legend=False
  )
```



Flight Delay by Hour of Day (0-24 hour)

- DEP_TIME_BLK vs DEL_DELAY
- **DEP_TIME_BLK:** CRS Departure Time Block, Hourly Intervals
- **DEP_TIME:** Actual Departure Time (local time: hhmm)
  Hour of Day (https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hour.html)

```
'''
Filter the departure delay > 0 as:
- 0: No delay
- < 0: Early departure
- > 0: Delayed departure
'''
df_cleaned = df_cleaned.withColumn("DEP_DELAY_Double", F.col("DEP_DELAY").cast("double")).cache()
# Filter data only with the flight delay (DEP_DELAY_Double > 0, as DEP_DELAY_Double = 0 means No Delay
df_cleaned = df_cleaned.filter(F.col('DEP_DELAY_Double') > 0)

# Calculate the count of DEP_DELAY_Double group by DEP_TIME_BLK
count_dep_delay_by_hour = df_cleaned.groupBy('DEP_TIME_BLK').agg(F.count('DEP_DELAY_Double').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)

# ================== PLOTTING: BARPLOT ==================

labels_sorted = [label for label in pdf_count_dep_delay_by_hour['DEP_TIME_BLK']]

create_barplot(
    pdf=pdf_count_dep_delay_by_hour,
    x_col='DEP_TIME_BLK',
    y_col='Percentage_Dep_Delay_by_Hourly_Interval',
    palette='cividis',
    xlabel='Departure Time Block (Hourly Intervals)',
    ylabel='Departure Delay Percentage (%)',
    title='Hourly Interval Breakdown of Departure Delays in Percentage - 4-Year 2015 - 2018 Training Data',
    custom_labels=labels_sorted,
    uom='%',
    figsize=(14, 6),
    dec_places=2,
    rotation=45,
    dict_lookup=None,
    labels_legend=None,
    show_legend=False
  )
```

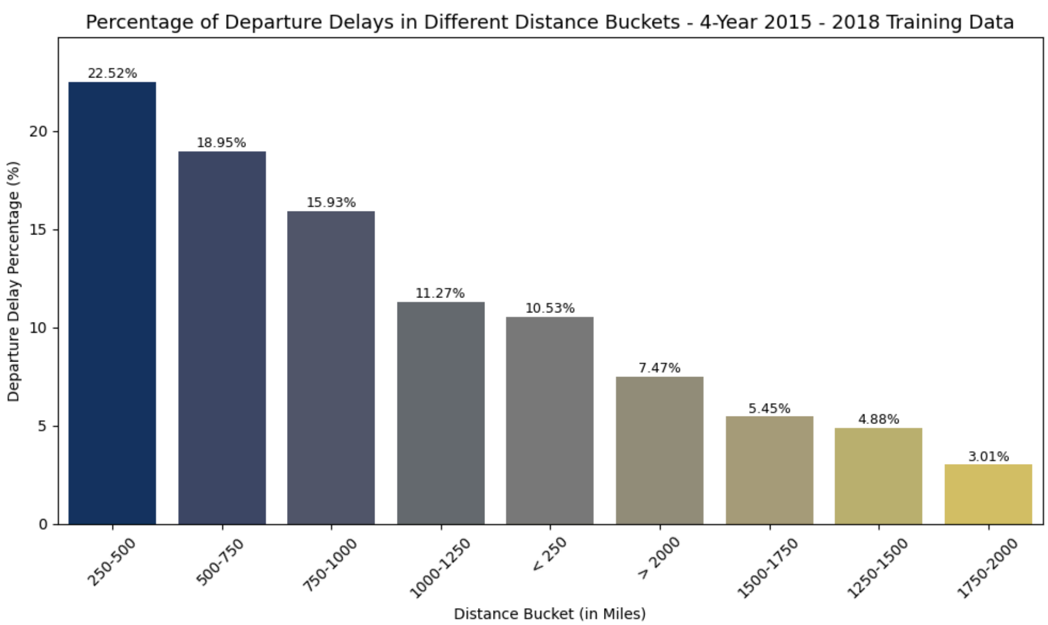



```
# extract_distinct_values_singleCol(20, df_cleaned, col='DISTANCE_Double')
```

Flight Delay by Distance

: could use "DistanceGroup" column to do this

```python
# Create the distance range column
def distance_range_col(distance):
    # Define the distance ranges based on the given distance
    if distance < 250:
        return '< 250'
    elif distance >= 2000:
        return '> 2000'
    else:
        # Define ranges from 250 to 2000 with a step of 250
        for i in range(250, 2000, 250):
            if i <= distance < i + 250:
                return f'{i}-{i+250}'
        return 'Others'  # In case the distance does not fall into defined ranges

df_cleaned = df_cleaned.withColumn("DISTANCE_Double", F.col("DISTANCE").cast("double")).cache()
df_cleaned_ = df_cleaned.withColumn('distance_bucket', F.udf(distance_range_col)(F.col('DISTANCE_Double'))).cache()

# Now group by the 'distance_bucket' column and count the delayed flights
df_delay_by_distance = df_cleaned_.groupBy('distance_bucket').agg(F.count(F.when(F.col('DEP_DELAY_Double').isNotNull(), 1)).alias('count_dep_delayed_dist')).cache()

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

# count_dep_delay_by_days_sorted = count_dep_delay_by_days_sorted.withColumn('Total_Count', total_count)
df_delay_by_distance = df_delay_by_distance.withColumn('Percentage_Dep_Delay_by_Distance', (F.col('count_dep_delayed_dist')/F.col('Total_Count')) * 100)


# If you need to convert it to a Pandas DataFrame
pd_delay_by_distance = df_delay_by_distance.toPandas().sort_values('count_dep_delayed_dist', ascending=False)

# ================== PLOTTING: BARPLOT ==================

labels_sorted = [label for label in pd_delay_by_distance['distance_bucket']]

create_barplot(
    pdf=pd_delay_by_distance,
    x_col='distance_bucket',
    y_col='Percentage_Dep_Delay_by_Distance',
    palette='cividis',
    xlabel='Distance Bucket (in Miles)',
    ylabel='Departure Delay Percentage (%)',
    title='Percentage of Departure Delays in Different Distance Buckets - 4-Year 2015 - 2018 Training Data',
    custom_labels=labels_sorted,
    uom='%',
    figsize=(10, 6),
    dec_places=2,
    rotation=45,
    dict_lookup=None,
    labels_legend=None,
    show_legend=False
  )
```



Flight Delay by OP_UNIQUE_CARRIER

List of airlines of the United States (https://en.wikipedia.org/wiki/List_of_airlines_of_the_United_States)

```
airline_dict = {
    'UA': 'United Airlines',
    'NK': 'Spirit Airlines',
    'AA': 'American Airlines',
    'EV': 'ExpressJet Airlines',
    'B6': 'JetBlue Airways',
    'DL': 'Delta Airlines',
    'OO': 'SkyWest Airlines',
    'F9': 'Frontier Airlines',
    'US': 'US Airways (now American Airlines)',
    'MQ': 'Envoy Air',
    'HA': 'Hawaiian Airlines',
    'AS': 'Alaska Airlines',
    'VX': 'Virgin America (now Alaska Airlines)',
    'WN': 'Southwest Airlines'
}

# Now group by the 'OP_UNIQUE_CARRIER' column and count the delayed flights
count_delay_by_carrier = df_cleaned.groupBy('OP_UNIQUE_CARRIER').agg(F.count(F.when(F.col('DEP_DELAY_Double').isNotNull(), 1)).alias('count_dep_delayed_dist')).cache()
# display(count_delay_by_carrier.collect())

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

count_delay_by_carrier = count_delay_by_carrier.withColumn('Percentage_Dep_Delay_by_Carrier', (F.col('count_dep_delayed_dist')/F.col('Total_Count')) * 100)

# If you need to convert it to a Pandas DataFrame
pd_count_delay_by_carrier = count_delay_by_carrier.toPandas().sort_values('count_dep_delayed_dist', ascending=False)

# ================== PLOTTING: BARPLOT ==================

labels_sorted = [label for label in pd_count_delay_by_carrier['OP_UNIQUE_CARRIER']]
labels_legend = [(sym, airline_dict[sym]) for sym in pd_count_delay_by_carrier['OP_UNIQUE_CARRIER']]

create_barplot(
  pdf=pd_count_delay_by_carrier,
  x_col='OP_UNIQUE_CARRIER',
  y_col='Percentage_Dep_Delay_by_Carrier',
  palette='cividis',
  xlabel='Airlines in the United States',
  ylabel='Departure Delay Percentage (%)',
  title='Percentage of Departure Delays by OP_UNIQUE_CARRIER - 4-Year 2015 - 2018 Training Data',
  custom_labels=labels_sorted,
  uom='%',
  figsize=(12, 6),
  dec_places=2,
  rotation=0,
  dict_lookup=airline_dict,
  labels_legend=labels_legend,
  show_legend=True
)
```

```
KeyError: 'OH'
```

Number of flights per Carrier

```
# Perform the group by and count distinct operation
planes_per_airline = df_cleaned.groupBy('OP_CARRIER') \
    .agg(F.countDistinct(F.when(F.col('TAIL_NUM').isNotNull(), F.col('TAIL_NUM'))).alias('plane_per_airline_count')).cache()

# # Show the DataFrame
planes_per_airline.show(truncate=False)

# To convert the DataFrame to a dictionary where keys are 'OP_CARRIER' and values are 'plane_per_airline_count'
# This function assumes a DataFrame with two columns where the first column values are unique
def dict_from_df_cols(df, key_col, value_col):
    return df.rdd.map(lambda row: (row[key_col], row[value_col])).collectAsMap()

# Call the function to convert the DataFrame to a dictionary
plane_count_dict = dict_from_df_cols(planes_per_airline, 'OP_CARRIER', 'plane_per_airline_count')

# Display the dictionary
print(plane_count_dict)
```

```
+----------+-----------------------+
|OP_CARRIER|plane_per_airline_count|
+----------+-----------------------+
|UA        |805                    |
|NK        |131                    |
|AA        |2055                   |
|EV        |396                    |
|B6        |252                    |
|DL        |979                    |
|OO        |541                    |
|F9        |119                    |
|YV        |145                    |
|US        |351                    |
|MQ        |400                    |
|OH        |137                    |
|HA        |65                     |
|G4        |114                    |
|YX        |191                    |
|AS        |259                    |
|VX        |68                     |
```

Cause of Delay

- **CarrierDelay**: Carrier Delay, in Minutes Analysis - Refers to delays caused by issues within the control of the airline carrier, such as maintenance problems, crew scheduling, or aircraft cleaning.
- **WeatherDelay**: Weather Delay, in Minutes Analysis - Indicates delays attributed to adverse weather conditions that impact safe flight operations, such as storms, fog, et. al.
- **NASDelay**: National Air System Delay, in Minutes Analysis - Represents delays caused by factors within the broader air traffic management system, including congestion, air traffic control limitations, or system malfunctions.
- **SecurityDelay**: Security Delay, in Minutes Analysis - Relates to delays resulting from security-related issues, such as heightened security procedures, breaches, or other security concerns.
- **LateAircraftDelay**: Late Aircraft Delay, in Minutes - Denotes delays caused by the late arrival of the aircraft from a previous flight, affecting the subsequent departure schedule.

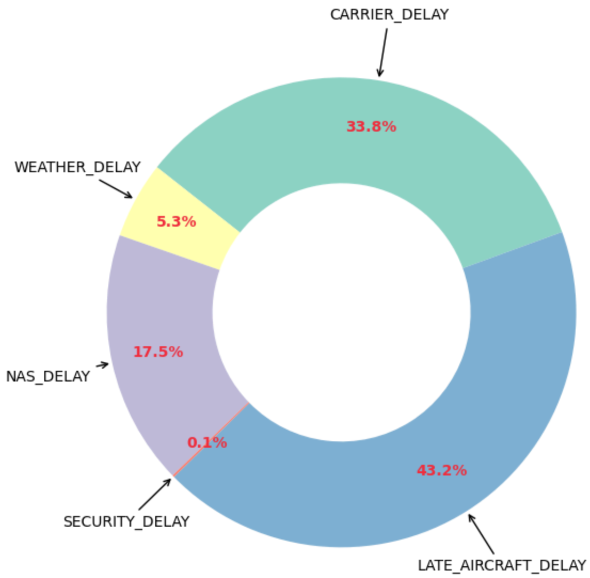
Helper Function: `Donutplot`

: Show the fraction of each delay type that contributes to the departure delay.

```
delay_cause = [
    'CARRIER_DELAY',
    'WEATHER_DELAY',
    'NAS_DELAY',
    'SECURITY_DELAY',
    'LATE_AIRCRAFT_DELAY'
]

def creat_donutplot(df, col_list, col_name, plot_title):

  df = df.withColumn('DEP_DEL15_Int', F.col('DEP_DEL15').cast('int')).cache()
  df = df.filter((df['DEP_DEL15'].isNotNull()) & (df['DEP_DEL15'] == 1)).cache()
  # Aggregating the delay causes
  col_sums = df.agg(*[_sum(col).alias(col) for col in col_list]).cache()
  # print(f'col_sums = {col_sums}')

  pdf = col_sums.toPandas().transpose()
  pdf.columns = ["Total"]
  pdf['Percentage'] = (pdf['Total'] / pdf['Total'].sum()) * 100
  pdf[col_name] = pdf.index

  # Plotting the donut plot
  colors = plt.get_cmap('Set3').colors
  fig, ax = plt.subplots(figsize=(8,6))

  wedges, texts, autotexts = ax.pie(
    pdf['Total'],
    # labels=pdf[col_name],
    autopct='%1.1f%%',             # Add the percentage annotations
    startangle=20,
    colors=colors,
    wedgeprops=dict(width=.45),
    pctdistance=0.8              # Adjust this value to position the percentage labels in the center
  )

  # Set the color of the percentage labels
  for autotext in autotexts:
    autotext.set_color('#f03040')
    autotext.set_fontweight('bold')

  for i, wedge in enumerate(wedges):
    angle = (wedge.theta2 - wedge.theta1) / 2. + wedge.theta1
    # The radius here is set to 1.1 to start from outside the ring
    x = 1 * np.cos(np.deg2rad(angle))
    y = 1 * np.sin(np.deg2rad(angle))
    connectionstyle = f"angle,angleA=0,angleB={angle}"
    ax.annotate(
        pdf[col_name][i],
        xy=(x, y),  # This is the start point of the arrow
        xytext=(1.28 * x, 1.28 * y),  # This is the end point of the arrow with the text
        arrowprops=dict(facecolor='blue', arrowstyle="->", connectionstyle=connectionstyle),
        horizontalalignment='center',
        verticalalignment='center'
    )
  ax.set_title(plot_title, fontdict={'fontsize': 14, 'fontweight': 'bold'}, loc='left', pad=45)
  ax.axis('equal')  # Equal aspect ratio ensures the pie chart is circular.
  plt.tight_layout()
  plt.show()
```

```
creat_donutplot(df_cleaned, delay_cause, 'Cause', "Flight Delay >= 15 Minutes by Delay Type - 4-Year 2015 - 2018 Training Data")
```

Flight Delay >= 15 Minutes by Delay Type - 4-Year 2015 - 2018 Training Data



Helper Function: `Boxplot`

```
from functools import reduce
from pyspark.sql import DataFrame

delay_cause = [
    'CARRIER_DELAY',
    'WEATHER_DELAY',
    'NAS_DELAY',
    'SECURITY_DELAY',
    'LATE_AIRCRAFT_DELAY',
    'DEP_DELAY']

combined_df = None

# Loop through each cause and filter non-null rows, then union the result
for cause in delay_cause:
    # Filter the DataFrame for non-null values in the current cause
    filtered_df = df_cleaned.filter(col(cause).isNotNull())

    # Union the filtered DataFrame with the combined DataFrame
    if combined_df is None:
        combined_df = filtered_df
    else:
        combined_df = combined_df.unionByName(filtered_df)

# Show the combined DataFrame
display(combined_df.limit(10))
```

Table

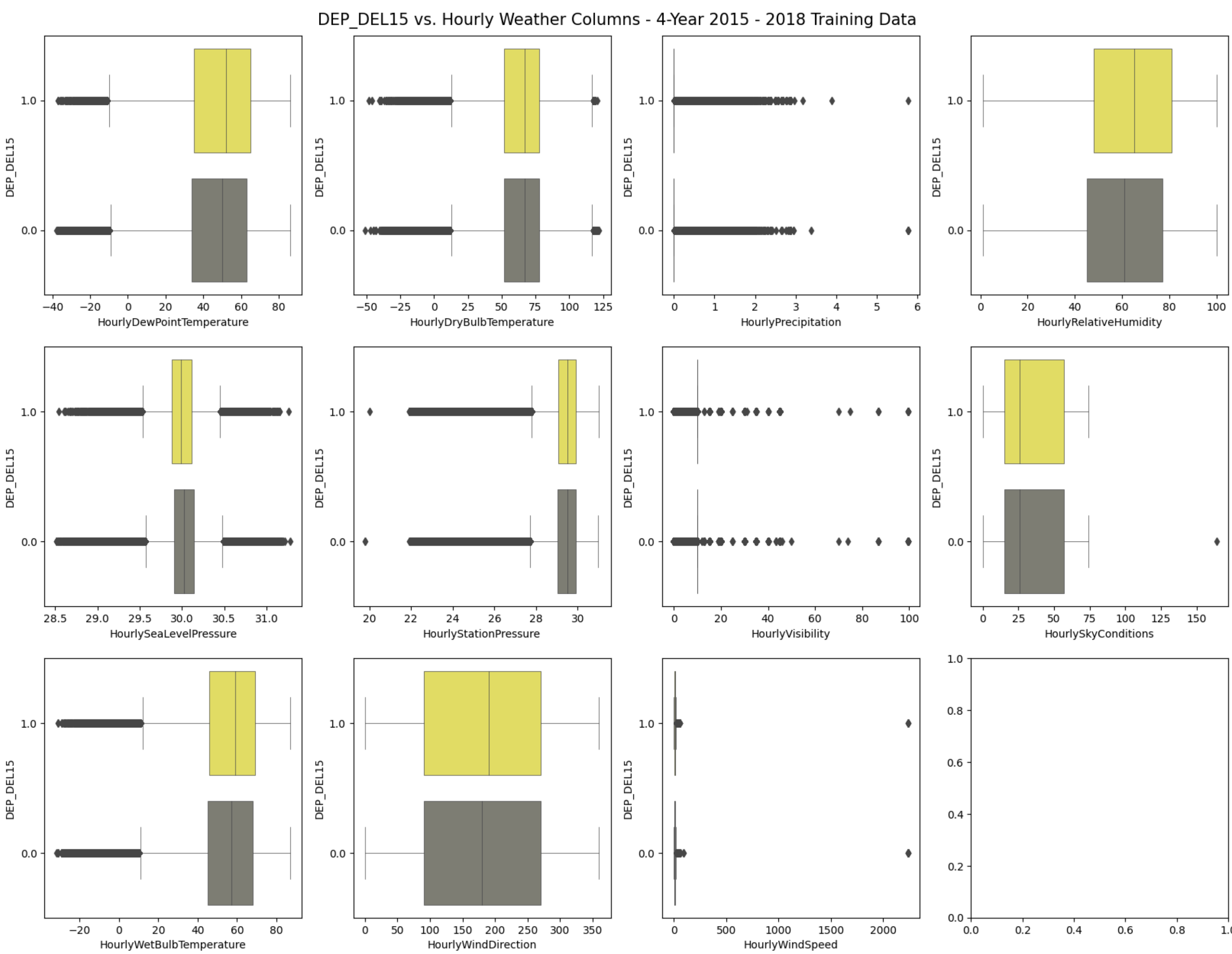
	QUARTER	DAY_OF_MONTH	DAY_OF_WEEK	FL_DATE	OP_UNIQUE_CARRIER	OP_CARRIER	TAIL_NUM	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME
1	4	26	3	2018-12-26	9E	9E	N8908D	3283	ATL	Atlanta, GA
2	2	28	4	2018-06-28	9E	9E	N8869B	3285	TRI	Bristol/Johnson City/Kingsport, TN
3	2	11	5	2018-05-11	9E	9E	N349PQ	3290	CVG	Cincinnati, OH
4	4	28	3	2018-11-28	9E	9E	N131EV	3290	MEM	Memphis, TN
5	1	22	4	2018-03-22	9E	9E	N918XJ	3296	ROC	Rochester, NY
6	2	24	4	2018-05-24	9E	9E	N336PQ	3297	SAV	Savannah, GA
7	3	24	2	2018-07-24	9E	9E	N336PQ	3299	LGA	New York, NY

10 rows

Flight Delay by Hourly Weather Columns

```
# creat_donutplot(df_delayed, selected_weather_cols, 'Weather_cause', "Flight Delay >= 15 Minutes by Hourly Weather Column - 4-Year 2015 - 2018")
```

```
selected_weather_cols = [
  'HourlyDewPointTemperature',
  'HourlyDryBulbTemperature',
  'HourlyPrecipitation',
  'HourlyRelativeHumidity',
  'HourlySeaLevelPressure',
  'HourlyStationPressure',
  'HourlyVisibility',
  'HourlySkyConditions',
  'HourlyWetBulbTemperature',
  'HourlyWindDirection',
  'HourlyWindSpeed'
]

def boxplot_delay(df, cols, title, nrows, ncols, figure_width, figure_height):
  fig, ax_grid = plt.subplots(nrows, ncols, figsize=(figure_width,figure_height))
  for idx, col in enumerate(cols):
      df = df.withColumn(col, F.col(col).cast("double"))
      y_var = df.select('DEP_DEL15').toPandas().squeeze()
      x_var = df.select(col).toPandas().squeeze()
      sns.boxplot(x=x_var, y=y_var, ax=ax_grid[idx//4][idx%4], orient='h', linewidth=.5, palette=['#807e73','#f5f253'])
      ax_grid[idx//4][idx%4].invert_yaxis()

  fig.suptitle(title, fontsize=15, y=0.9)
  plt.show()

df_4y_train.filter(df_4y_train['DEP_DEL15'].isNotNull()).cache()
y_var = df_4y_train.select('DEP_DEL15').toPandas().squeeze()
selected_weather_df = df_cleaned.select(*selected_weather_cols)

boxplot_delay(
  df=df_4y_train,
  cols=selected_weather_cols,
  title="DEP_DEL15 vs. Hourly Weather Columns - 4-Year 2015 - 2018 Training Data",
  nrows=3,
  ncols=4,
  figure_width=20,
  figure_height=15)
```



Interpretation of the Boxplots

- HourlyDewPointTemperature: The boxplot suggests that the median dew point temperature for delayed flights is slightly higher than for non-delayed flights, but the overlap in interquartile ranges suggests that dew point temperature alone may not be a strong predictor of delays.
- HourlyDryBulbTemperature: Similar to the dew point, the median dry bulb temperature for delayed flights is slightly higher. However, the considerable overlap of interquartile ranges indicates a weak distinction between delayed and non-delayed flights based on this variable.
- HourlySeaLevelPressure: This plot shows a very tight interquartile range for both delayed and non-delayed flights, with median values being nearly identical. This suggests that sea level pressure may not be significantly related to flight delays.
- HourlyStationPressure: Similar to sea level pressure, the station pressure shows minimal difference between the medians of delayed and non-delayed flights, suggesting a weak relationship with delays.
- HourlyVisibility: There is a notable difference in visibility between delayed and non-delayed flights. Lower visibility tends to be associated with delays, as seen by the lower median in the boxplot for delayed flights.
- HourlyRelativeHumidity: The boxplot indicates higher median relative humidity for delayed flights compared to non-delayed flights. The spread of values is also wider for delayed flights, suggesting variability in how humidity affects delays.
- HourlyPrecipitation: This plot shows that most flights, whether delayed or not, occur with zero precipitation. However, the presence of precipitation is associated with some delays, as indicated by the points above zero mostly on the side of the delayed flights.
- HourlySkyConditions: The boxplot shows a higher median for delayed flights, suggesting that certain sky conditions (possibly more cloud cover or worse) may be associated with delays.
- HourlyWetBulbTemperature: This plot is similar to the dry bulb and dew point temperature plots, with a slightly higher median for delayed flights but a substantial overlap in interquartile ranges.
- HourlyWindDirection: The boxplot shows a wide range of wind directions for both delayed and non-delayed flights, with no clear pattern indicating a relationship between wind direction and flight delays.
- HourlyWindSpeed: This plot shows that higher wind speeds are associated with some delayed flights, as indicated by the longer upper whisker and higher median for delayed flights compared to non-delayed flights.

Data Processing and Munging for the Selected Columns

- Null Handling
  - High variance => Impute with **median**. Otherwise => Impute with **mean**
  - Categorical variable with decent amount of missing variable ==> Impute with **mode**
  - Use other methods such as **forward fill/backward fill/RandomForestRegressor** if data is continuous
- Convert columns to appropriate **data type**
- Remove bad data
  - e.g. remove 'T' from DailySnowFall, remove '*' from HourlySkyCondition
- Feature Engineering
  - Create new features / Convert Categorial to Numeric:
    - Holiday
    - July 4th
    - Christmas Day
    - New Year's Day
    - Distance breakdown
    -
- Standarize / Scaling / Normalization

Proposed Actions for Managing Missing Values Across Different Percentage Ranges

- **Less than 5% missing:** Drop the missing observations.
- **Between 5% to 20% missing:** Impute the missing values with mean or median.
- **Between 20% to 50% missing:** Either impute or drop the column.
- **More than 50% Missing:** Drop the column or perform a more detailed analysis. *For important columns with a high percentage of missing values, consider replacing nulls with **zeros** or imputing the missing values using the **forward-fill** technique.*

Drop Irrelevant/Problematic Columns

- Data Columns Checklist (https://docs.google.com/spreadsheets/d/1CNbFpyp_-7VgR-AHkjTZkXfxGN7RPl7wUpjXCrVmkjY/edit#gid=602118796): Columns with **data leakage**, where can be applied only after the delay occurs - e.g. DEP_TIME, DEL_DELAY, etc
- Duplicate Columns
- Unnecessary or Irrelavant Columns

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

```
addt_cols_to_drop = [col for col in df_cleaned.columns if col not in final_keep_columns]
print(f'num of cols to drop = {len(addt_cols_to_drop)}\ncols to drop:\n{addt_cols_to_drop}')

df_filtered = df_cleaned.drop(*addt_cols_to_drop)
print_df_shape(df_filtered, 'df_filter')

print(f'\nFinal Selected Columns:\n{df_filtered.columns}')
display_limited_df(df_filtered, nrows=3)
```

```
df_filtered_missing = null_check(df_filtered, 'df_filtered')
df_filtered_missing
```

```
display(df_filtered.summary())
```

Help Functions: `Remove Bad Data on the Hourly Weather Metrics`

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, regexp_replace, when

# Assuming the DataFrame is named df_filtered
def remove_bad_data(df, col_name):
# Step 1: Remove 's' at the end of strings in 'HourlyPrecipitation'
  df = df.withColumn('HourlyPrecipitation', regexp_replace('HourlyPrecipitation', 's$', ''))
  # extract_distinct_values_singleCol(df, 'HourlyPrecipitation')

  # Step 2: Remove 'T' with a negligible number
  df = df.withColumn('HourlyPrecipitation', when(col('HourlyPrecipitation') == 'T', 0.001).otherwise(col('HourlyPrecipitation')))

  # Step 3: Drop rows where 'HourlyPrecipitation' is None
  df = df.na.drop(subset=["HourlyPrecipitation"])

  # Step 4: Convert the string to double data type
  df = df.withColumn('HourlyPrecipitation_Double', col('HourlyPrecipitation').cast('double'))

  # Show the result
  check_data_type(df, 'FL_DATE')
  extract_distinct_values_singleCol(df, 'HourlyPrecipitation_Double')

remove_bad_data(df_filtered, 'HourlyPrecipitation')
```

```
display(df_filtered.limit(3))
```

In this code, the following changes and considerations were made:

The seasonal_decompose function is applied to the non-null values of HourlyPrecipitation to avoid issues with NaN values. The UDF now expects a DataFrame as input, and the seasonal decomposition is applied to the HourlyPrecipitation column of that DataFrame. The schema defines that the UDF will return a DataFrame with the same structure as the input DataFrame. It's assumed that there's a logical grouping for the DataFrame, such as a date or category. You need to group by this column before applying the UDF. After decomposition, missing values are filled using forward-fill (ffill) to handle the NaNs that appear after applying the seasonal decomposition. Make sure to adjust grouping_column to the column that logically divides your time series data into groups that should each be decomposed separately. If your data does not require grouping and each row can be considered independently, you'll need to use a different UDF type or adjust the logic accordingly.

1) Date Columns

- Column names with "date" substring, convert to **date**
- Column names with "date_time" substring, convert to **timestamp**

```
def convert_to_date(df):
    # Step 1: Find columns containing the substring "date" but not "date_time"
    date_columns = [col_name for col_name in df.columns if "date" in col_name.lower() and "date_time" not in col_name.lower()]

    # Convert these columns from string to date
    for date_col in date_columns:
        df = df.withColumn(date_col, to_date(col(date_col)))

    # Step 2: Find columns containing the substring "date_time"
    datetime_columns = [col_name for col_name in df.columns if "date_time" in col_name.lower()]

    # Convert these columns from string to datetime
    for datetime_col in datetime_columns:
        format_string = "yyyy-MM-dd'T'HH:mm:ss"    # (2015-01-10T09:00:00)
        df = df.withColumn(datetime_col, to_timestamp(col(datetime_col), format_string))

    return df

df_final = convert_to_date(df_filtered_selected)
df_final
```

2) Numeric Columns

- Convert to **Integer**
- Convert to **Double**

```
col_int =
['OP_CARRIER_FL_NUM','DEP_DEL15','ORIGIN_WAC','DEST_WAC','CRS_DEP_TIME','DEP_TIME','DEP_DELAY_GROUP','WHEELS_OFF','WHEELS_ON','CRS_ARR_TIME','ARR_TIME','ARR_DELAY_GROUP','DISTANCE_GROU
P','SOURCE','OP_CARRIER_FL_NUM','HourlyDewPointTemperature','HourlyDryBulbTemperature','HourlyWetBulbTemperature','HourlyWindSpeed','HourlyPressureTendency','DAY_OF_MONTH','DAY_OF_WEEK
','DAY_OF_MONTH','YEAR']

col_double =
['DEP_DELAY','DEP_DELAY_NEW','TAXI_OUT','TAXI_IN','ARR_DELAY','ARR_DELAY_NEW','ARR_DEL15','CANCELLED','DIVERTED','CRS_ELAPSED_TIME','ACTUAL_ELAPSED_TIME','AIR_TIME','FLIGHTS','DISTANCE
','origin_station_lat','origin_station_lon','origin_airport_lat','origin_airport_lon','origin_station_dis','dest_station_lat','dest_station_lon','dest_airport_lat','dest_airport_lon','
dest_station_dis','LATITUDE','LONGITUDE','ELEVATION','HourlyAltimeterSetting','HourlyPrecipitation','HourlyRelativeHumidity','HourlySeaLevelPressure','HourlyStationPressure','HourlyVis
ibility','BackupDistance', 'CARRIER_DELAY','WEATHER_DELAY','NAS_DELAY','SECURITY_DELAY','DailySnowfall','LATE_AIRCRAFT_DELAY']
```

```
def convert_to_numeric(df):
  for col_name in keep_columns:
    if col_name in col_int:
      df = df.withColumn(col_name, df[col_name].cast(IntegerType()))

    elif col_name in col_double:
      df = df.withColumn(col_name, df[col_name].cast(DoubleType()))

  return df

df_final = convert_to_numeric(df_final)

print(f'# columns: {len(df_final.columns)}, # rows: {df_final.count()}')
df_final.printSchema()
display(df_final.limit(3))
```

```
# df_non_null_snowfall = df_clean_selected.filter(df_clean_selected.HourlyWindDirection.isNotNull())
# display(df_non_null_snowfall)
```

Simple Statistics

```
# df_final = df_final[keep_columns].select('*')
# display(selected_data.describe())
display(df_final.summary())
```

Imputation Strategies for the Missing Values

- In large time series data, the imputation is crucial for the robust performance of machine learning models. Here are some common strategies:
  1. **Forward Fill or Backward Fill:** This strategy propagates the last known non-null value to the next non-null value (forward fill) or the next known value backward (backward fill).
  2. **Interpolation:** A more sophisticated way that estimates missing values using different interpolation techniques such as **linear or polynomial** based on the values of surrounding data points.
  3. Seasonal Decomposition and Imputation: This involves decomposing the time series into trend, seasonal, and residual components, imputing missing values in these components separately, and then recombining them.
  4. **Model-Based Imputation:** Complex models, such as ARIMA or LSTM networks, can predict missing values based on the patterns learned from the time series data.
  5. Using **Imputer** library, can impute them with **medain, mean, and mode**
  6. Otherwise, we can **simply drop the missing values** it's not a lot (less than 5% missing)
     - **Less than 5% missing:** Drop the missing observations.
     - **Between 5% to 20% missing:** Impute the missing values with mean or median.
     - **Between 20% to 50% missing:** Either impute or drop the column.

- **More than 50% Missing:** Drop the column or perform a more detailed analysis. *For important columns with a high percentage of missing values, consider replacing nulls with **zeros** or imputing the missing values using the **forward-fill** technique.*

```
def null_check_and_categorize(df_null):
    ''' compute the percentage of missing values in each column and categorize them according to given ranges '''

    # Initialize counters for each category
    less_than_5 = 0
    between_5_and_20 = 0
    between_20_and_50 = 0
    more_than_50 = 0

    # Go through each column and increment counters based on the percentage of missing values
    for c in df_null.to_frame().T.columns:
        missing_percent = df_null.to_frame().T.select(c).collect()[0][c]
        missing_percent = df_null[c]
        if missing_percent < 5:
            less_than_5 += 1
        elif 5 <= missing_percent < 20:
            between_5_and_20 += 1
        elif 20 <= missing_percent < 50:
            between_20_and_50 += 1
        elif missing_percent >= 50:
            more_than_50 += 1

    return less_than_5, between_5_and_20, between_20_and_50, more_than_50

# Usage
lt_5, bt_5_20, bt_20_50, mt_50 = null_check_and_categorize(otpw_3m_2015_missing['missing_%'])
print(f"Out of {len(df_cleaned.columns)}, we have:\n============================================")
print(f"Number of columns with < 5% missing values: {lt_5}")
print(f"Number of columns with 5% - 20% missing values: {bt_5_20}")
print(f"Number of columns with 20% - 50% missing values: {bt_20_50}")
print(f"Number of columns with > 50% missing values: {mt_50}")
```

Correlation Study

```
# selected_col = [c for c in df_final.columns if c not in df_final_missing['missing_%']]
col_w_high_nulls = ['DailySnowfall','CARRIER_DELAY','WEATHER_DELAY','NAS_DELAY','SECURITY_DELAY','LATE_AIRCRAFT_DELAY','HourlyPrecipitation']

def correlation_analysis(cols):
  corr_matrix = {c: df_final.stat.corr('DEP_DEL15', c) for c in cols}
  df_corr = pd.DataFrame(list(corr_matrix.items()), columns=['Column', 'Correlation'])

  plt.figure(figsize=(10,8))
  barplot = sns.barplot(x='Correlation', y='Column', data=df_corr)
  for p in barplot.patches:
      width = p.get_width()  # get bar length
      plt.text(width,  # set the text at the end of the bar
              p.get_y() + p.get_height() / 2,  # get Y coordinate + half of bar width
              f'{width:.2f}',  # set variable to display, 2 decimals
              va='center')
  plt.xlabel('Correlation Coefficient')
  plt.ylabel('Column')
  plt.title('Correlation with DEP_DEL15')
  plt.show()

correlation_analysis(col_w_high_nulls)
```

```
display(df_final.limit(3))
```

```
# Extract unique values from a selected column
print(df_final.select('TAIL_NUM').distinct())
# df_final['TAIL_NUM'].distinct()
```

```
df_final.printSchema()
```