databricks MASTER 3M BASELINE TEAM_4_1

(https://databricks.com)

Import Packages

```
from pyspark.sql.functions import *
from pyspark.sql import Window, DataFrame
from pyspark.sql.types import TimestampType
from datetime import timedelta
from functools import reduce
from graphframes import *
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, OneHotEncoder
from pyspark.ml.feature import StandardScaler, Imputer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.mllib.evaluation import MulticlassMetrics
from xgboost.spark import SparkXGBClassifier
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.classification import MultilayerPerceptronClassifier
```

```
Welcome to the W261 final project! (Team 4-1's Version)
```

Set Checkpoint directory

```
spark.sparkContext.setCheckpointDir("/ml_41/df_otpw_checkpoint")
```

Locate raw data

```
data_BASE_DIR = "dbfs:/mnt/mids-w261/"
display(dbutils.fs.ls(f"{data_BASE_DIR}"))
```

```
shaded.databricks.org.apache.hadoop.fs.azure.AzureException: java.util.NoSuchElementException: An error occurred while enumerating the result, check the original exception for details.
```

Load Raw Data

```
# OTPW
df_otpw = spark.read.format("csv").option("header","true").load(f"dbfs:/mnt/mids-w261/OTPW_3M_2015.csv")
#display(df_otpw)
```

```
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 value of Authorization header is formed correctly including the signature.
```

```
# Select only columns needed
df_otpw = df_otpw.select('DEP_DEL15', 'DEP_DELAY_NEW', 'ORIGIN_AIRPORT_ID', 'DEST_AIRPORT_ID', 'FL_DATE',
'CRS_DEP_TIME', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'OP_UNIQUE_CARRIER', 'DEP_TIME_BLK', 'TAIL_NUM', 'MONTH', 'YEAR',
'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyWetBulbTemperature', 'HourlyRelativeHumidity',
'HourlyWindDirection', 'HourlyWindSpeed')

df_otpw.printSchema()
```

```
NameError: name 'df_otpw' is not defined
```

Drop all observations with null for target variable

```
df_otpw = df_otpw.dropna(subset=['DEP_DEL15'])
```

```
NameError: name 'df_otpw' is not defined
```

Drop defunct airlines

```
df_otpw = df_otpw.filter(~col("ORIGIN_AIRPORT_ID").isin("EV", "MQ", "QX", "US", "VX"))
```

```
# df_otpw.groupBy(['MONTH']).count().show()
```

Downsample

```
# Count occurrences of 0 and 1 in DEP_DEL15
count_0 = df_otpw.filter(col("DEP_DEL15") == 0).count()
count_1 = df_otpw.filter(col("DEP_DEL15") == 1).count()
total_count = df_otpw.count()

# Randomly sample and filter to balance counts
df_0 = df_otpw.filter(col("DEP_DEL15") == 0).sample(False, count_1/count_0, seed=42)
df_1 = df_otpw.filter(col("DEP_DEL15") == 1)

# Now both DataFrames have the same number of rows for DEP_DEL15 being 0 and 1
df_otpw = df_0.union(df_1)

# Check the counts after balancing
# df_otpw.groupBy("DEP_DEL15").count().show()
```

```
NameError: name 'df_otpw' is not defined
```

```
# df_otpw.groupBy(['MONTH']).count().show()
```

Create DateTime var

```
df_otpw = df_otpw.withColumn("DAY_OF_MONTH", lpad(col("DAY_OF_MONTH"), 2, "0"))
df_otpw = df_otpw.withColumn("MONTH", lpad(col("MONTH"), 2, "0"))

date_string = concat(
    col("YEAR").cast("string"),
    lit("-"),
    col("MONTH"),
    lit("-"),
    col("DAY_OF_MONTH")
)
df_otpw=df_otpw.withColumn("DATE_VARIABLE", to_date(date_string, 'yyyy-MM-dd'))
#df_otpw=df_otpw.withColumn("DATE_VARIABLE", date_string)
# df_otpw.printSchema()

# Format CRS_DEP_TIME to HH:MM format
df_otpw = df_otpw.withColumn("CRS_DEP_TIME_INT", col("CRS_DEP_TIME").cast("integer"))
df_otpw = df_otpw.withColumn("CRS_DEP_TIME_STR", format_string("%04d", col("CRS_DEP_TIME_INT")))

# Create the HH:MM Format
df_otpw = df_otpw.withColumn("CRS_DEP_TIME_FMT", concat(col("CRS_DEP_TIME_STR").substr(1, 2), lit(":"), col("CRS_DEP_TIME_S

# Combine 'DATE_VARIABLE' and 'CRS_DEP_TIME_FMT' into a single timestamp
df_otpw = df_otpw.withColumn(
    "DateTime",
    to_timestamp(concat(col("DATE_VARIABLE"), lit(" "), col("CRS_DEP_TIME_FMT")), "yyyy-MM-dd HH:mm")
)

#
# df_otpw.select("CRS_DEP_TIME", "CRS_DEP_TIME_FMT", "DateTime").show(100, False)
```

Filter Data

```
# Cast as numeric variables into numeric format from string
df_otpw = df_otpw.withColumn("DEP_DEL15", col("DEP_DEL15").cast('int')) \
    .withColumn("DEP_DELAY_NEW", regexp_replace("DEP_DELAY_NEW", "s", "").cast('int')) \
    .withColumn("YEAR", regexp_replace("YEAR", "s", "").cast('int')) \
    .withColumn("MONTH", regexp_replace("MONTH", "s", "").cast('int')) \
    .withColumn("DAY_OF_MONTH", regexp_replace("DAY_OF_MONTH", "s", "").cast('int')) \
    .withColumn("DAY_OF_WEEK", regexp_replace("DAY_OF_WEEK", "s", "").cast('int')) \
    .withColumn("HourlyDewPointTemperature", regexp_replace("HourlyDewPointTemperature", "s", "").cast('int')) \
    .withColumn("HourlyDryBulbTemperature", regexp_replace("HourlyDryBulbTemperature", "s", "").cast('int')) \
    .withColumn("HourlyWetBulbTemperature", regexp_replace("HourlyWetBulbTemperature", "s", "").cast('int')) \
    .withColumn("HourlyRelativeHumidity", regexp_replace("HourlyRelativeHumidity", "s", "").cast('int')) \
    .withColumn("HourlyWindDirection", regexp_replace("HourlyWindDirection", "s", "").cast('int')) \
    .withColumn("HourlyWindSpeed", regexp_replace("HourlyWindSpeed", "s", "").cast('int')) \

# # Format CRS_DEP_TIME to HH:MM format
# df_otpw = df_otpw.withColumn("CRS_DEP_TIME_STR", format_string("%04d", col("CRS_DEP_TIME")))
# df_otpw = df_otpw.withColumn("CRS_DEP_TIME_FMT", concat(col("CRS_DEP_TIME_STR").substr(1, 2), lit(":"), col("CRS_DEP_TIM

# # Combine 'FL_DATE' and 'CRS_DEP_TIME' into a single timestamp
# df_otpw = df_otpw.withColumn(
#     "DateTime",
#     to_timestamp(concat(col("FL_DATE"), lit(" "), col("CRS_DEP_TIME_FMT")), "yyyy-MM-dd HH:mm")
# )
df_otpw.printSchema()
```

```
|-- FL_DATE: string (nullable = true)
|-- CRS_DEP_TIME: string (nullable = true)
```

```
|-- MONTH: integer (nullable = true)
|-- YEAR: integer (nullable = true)
|-- HourlyDewPointTemperature: integer (nullable = true)
|-- HourlyDryBulbTemperature: integer (nullable = true)
|-- HourlyWetBulbTemperature: integer (nullable = true)
|-- HourlyRelativeHumidity: integer (nullable = true)
|-- HourlyWindDirection: integer (nullable = true)
|-- HourlyWindSpeed: integer (nullable = true)
|-- DATE_VARIABLE: date (nullable = true)
|-- CRS_DEP_TIME_INT: integer (nullable = true)
|-- CRS_DEP_TIME_STR: string (nullable = false)
|-- CRS_DEP_TIME_FMT: string (nullable = false)
|-- DateTime: timestamp (nullable = true)
```

Convert Data to Delta Lake Format

```
# Configure Path
DELTALAKE_GOLD_PATH = "/ml_41/flights.delta"

# Remove table if it exists
dbutils.fs.rm(DELTALAKE_GOLD_PATH, recurse=True)

# Save table as Delta Lake
df_otpw.write.format("delta").mode("overwrite").save(DELTALAKE_GOLD_PATH)
```

Read in Checkpointed Data in Delta Lake Format

```
# Configure Path
DELTALAKE_GOLD_PATH = "/ml_41/flights.delta"

# Re-read as Delta Lake
df_otpw = spark.read.format("delta").load(DELTALAKE_GOLD_PATH)

# Sample if needed
# df_otpw = df_otpw.sample(False, 1000/df_otpw.count(), seed = 42)

# Review data
#display(df_otpw)
```

EDA: Select features for correlation

```
corr_features = df_otpw.select(['DEP_DELAY_NEW', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature',
'HourlyRelativeHumidity', 'HourlyWetBulbTemperature', 'HourlyWindDirection', 'HourlyWindSpeed'])

corr_features_df = corr_features.toPandas()
corr_features_df.dtypes
```

```
DEP_DELAY_NEW                 int32
HourlyDewPointTemperature    float64
HourlyDryBulbTemperature     float64
HourlyRelativeHumidity       float64
HourlyWetBulbTemperature     float64
HourlyWindDirection          float64
HourlyWindSpeed              float64
dtype: object
```

EDA: Import Packages

```
# Import packages
import pyspark.sql.functions as F
from pyspark.sql.functions import isnan, when, count, col, split, trim, lit, avg, sum, length, regexp_replace
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql.types import DoubleType, FloatType
from pyspark.ml.stat import Correlation
from pyspark.ml.feature import VectorAssembler
```

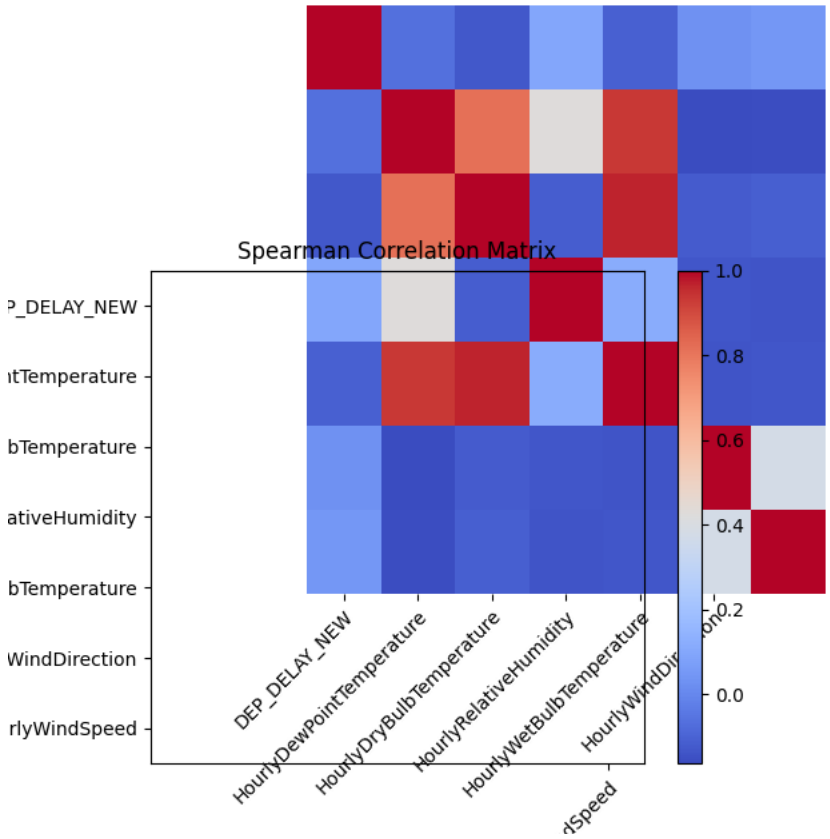
EDA: Correlation matrix

```
# Spearman Correlation Matrix
# Spearman's correlation evaluates the monotonic relationship between two ranked variables. Instead of using raw
data, it uses the rank order of the data.
# Preferred this method since we have non-normal distributions and ordinal variables
NUM_FEATURES = ['DEP_DELAY_NEW', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyRelativeHumidity',
'HourlyWetBulbTemperature', 'HourlyWindDirection', 'HourlyWindSpeed']
df_otpw_features = df_otpw.select(NUM_FEATURES).dropna()

# Assemble the columns into a single feature column
assembler = VectorAssembler(inputCols=df_otpw_features.columns, outputCol="features")
df_assembled = assembler.transform(df_otpw_features)

# Calculate the correlation matrix
corr_matrix = Correlation.corr(df_assembled, "features", method = 'spearman').head()
corr_matrix = corr_matrix[0].toArray()
print(corr_matrix)
```

```
[[ 1.         -0.0674757  -0.12714214  0.09671461 -0.10534385  0.03121306
   0.05009084]
 [-0.0674757   1.          0.81810541  0.42354845  0.93288104 -0.1634755
  -0.15557771]
 [-0.12714214  0.81810541  1.         -0.11360669  0.96669815 -0.12140928
  -0.10988912]
 [ 0.09671461  0.42354845 -0.11360669  1.          0.1156109  -0.13465133
  -0.13867153]
 [-0.10534385  0.93288104  0.96669815  0.1156109   1.         -0.14061448
  -0.13377093]
 [ 0.03121306 -0.1634755  -0.12140928 -0.13465133 -0.14061448  1.
   0.38127378]
 [ 0.05009084 -0.15557771 -0.10988912 -0.13867153 -0.13377093  0.38127378
   1.        ]]
```

EDA: Show heat map

```
# Plot correlation
fig, ax = plt.subplots()
im = ax.imshow(corr_matrix, cmap='coolwarm')

# Add colorbar
cbar = ax.figure.colorbar(im, ax=ax)

# Set ticks and tick labels
ax.set_xticks(np.arange(corr_matrix.shape[1]))
ax.set_yticks(np.arange(corr_matrix.shape[0]))
ax.set_xticklabels(NUM_FEATURES)
ax.set_yticklabels(NUM_FEATURES)

# Rotate the tick labels and set their alignment
plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")

# Set title
ax.set_title("Spearman Correlation Matrix")

# Show the plot
plt.show()
```


Spearman Correlation Matrix
P_DELAY_NEW
ıtTemperature
bTemperature
ativeHumidity
bTemperature
WindDirection
rlyWindSpeed
DEP_DELAY_NEW
HourlyDewPointTemperature
HourlyDryBulbTemperature
HourlyRelativeHumidity
HourlyWetBulbTemperature
HourlyWindDir
dSpeed
1.0
0.8
0.6
0.4
0.2
0.0

Split data into train and test

```
import pyspark.sql.functions as F

## add checkpoint
# spark.sparkContext.setCheckpointDir("/ml_41/df_otpw_checkpoint")

total_records = df_otpw.count()
# df_otpw2.checkpoint()

# Specify the percentage for the training set (e.g., 80%)
training_percentage = 0.8

# Calculate the number of records for the training set
training_records = int(total_records * training_percentage)

# Determine the split_date based on the calculated training_records
window_spec = Window.orderBy("DATE_VARIABLE")
df_otpw_with_rank = df_otpw.withColumn("rank", F.row_number().over(window_spec))
split_date_row = df_otpw_with_rank.filter(col("rank") == training_records).select("DATE_VARIABLE").collect()[0]
split_date = split_date_row["DATE_VARIABLE"]

# drop rank
df_otpw = df_otpw.drop("rank")

# # Filter the DataFrame based on the dynamically determined split_date (Turn on for 5Y)
# train_df = df_otpw.filter(col("DATE_VARIABLE") < ('2018-10-01'))
# test_intermediate_df = df_otpw.filter((col("DATE_VARIABLE") >= '2018-10-01') & (col("DATE_VARIABLE") < '2019-01-
01'))
# test_df = df_otpw.filter(col("DATE_VARIABLE") >= ('2019-01-01'))

# Filter the DataFrame based on the dynamically determined split_date (Turn on for 3M or 1Y)
full_train_df = df_otpw.filter(col("DATE_VARIABLE") < split_date)

# Create intermediate DF
total_train_records = full_train_df.count()
int_training_percentage = 0.8
# Calculate the number of records for the training set
int_training_records = int(total_train_records * int_training_percentage)
# Determine the split_date based on the calculated training_records
window_spec = Window.orderBy("DATE_VARIABLE")
train_df_with_rank = full_train_df.withColumn("rank", F.row_number().over(window_spec))

train_df = train_df_with_rank.filter((train_df_with_rank['rank'] < int_training_records))
test_intermediate_df = train_df_with_rank.filter((train_df_with_rank['rank'] >= int_training_records))
test_df = df_otpw.filter(col("DATE_VARIABLE") >= split_date)

print("df_otpw:", df_otpw.count())
print("full_train_df:", full_train_df.count())
print("train_df:", train_df.count())
print("test_intermediate_df:", test_intermediate_df.count())
print("test_df:", test_df.count())


# print("test")
# test_df.show(1)
# test_df.select("DATE_VARIABLE").distinct().show(365)
# cache dataframes for performance
train_df.cache()
test_df.cache()
```

```
df_otpw: 555428
full_train_df: 438802
train_df: 351040
test_intermediate_df: 87762
test_df: 116626
```

```
DataFrame[DEP_DEL15: int, DEP_DELAY_NEW: int, ORIGIN_AIRPORT_ID: string, DEST_AIRPORT_ID: string, FL_DATE: string, CRS_DE
P_TIME: string, DAY_OF_MONTH: int, DAY_OF_WEEK: int, OP_UNIQUE_CARRIER: string, DEP_TIME_BLK: string, TAIL_NUM: string, M
ONTH: int, YEAR: int, HourlyDewPointTemperature: int, HourlyDryBulbTemperature: int, HourlyWetBulbTemperature: int, Hourl
yRelativeHumidity: int, HourlyWindDirection: int, HourlyWindSpeed: int, DATE_VARIABLE: date, CRS_DEP_TIME_INT: int, CRS_D
EP_TIME_STR: string, CRS_DEP_TIME_FMT: string, DateTime: timestamp]
```

Define function to impute missing values with moving averages

```
def calculate_moving_averages(df: DataFrame, range_low: int, range_high: int) -> DataFrame:
    """
    Applies moving average to specified columns in the DataFrame.

    Args:
    df: The input DataFrame.
    range_low: The lower bound of the window for the moving average.
    range_high: The upper bound of the window for the moving average.

    Returns:
    DataFrame with moving averages applied.
    """
    # Define window specification
    windowSpec = Window.partitionBy("ORIGIN_AIRPORT_ID").orderBy("FL_DATE", "CRS_DEP_TIME").rowsBetween(range_low,
range_high)

    # Columns to calculate moving average
    weather_columns = ['HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyRelativeHumidity',
'HourlyWindDirection', 'HourlyWindSpeed']

    # Calculate moving average and impute missing values
    for column in weather_columns:
        moving_avg_col = f"{column}_MovingAvg"
        df = df.withColumn(moving_avg_col, avg(col(column)).over(windowSpec))

        # Replace null values in the original column with the moving average
        df = df.withColumn(column, when(col(column).isNull(), col(moving_avg_col)).otherwise(col(column)))

        # Optionally, drop the moving average column if it's no longer needed
        df = df.drop(moving_avg_col)

    return df
```

Time-based features: Seasonality, Part of Day, Recency, Past Flight Delays

```
def add_seasonality_feature(df: DataFrame, date_column: str) -> DataFrame:
    """
    Adds a 'Season' column to the DataFrame based on the month of the year from the specified date column.

    Args:
    df: The input DataFrame.
    date_column: The name of the column containing date information.

    Returns:
    DataFrame with an additional 'Season' column.
    """
    # Add a seasonality column
    df_with_season = df.withColumn("Season",
                                   when((month(date_column) >= 3) & (month(date_column) <= 5), lit("Spring"))
                                   .when((month(date_column) >= 6) & (month(date_column) <= 8), lit("Summer"))
                                   .when((month(date_column) >= 9) & (month(date_column) <= 11), lit("Autumn"))
                                   .otherwise(lit("Winter")))
    return df_with_season
```

```
def add_part_of_day_feature(df: DataFrame, timestamp_column: str) -> DataFrame:
    """
    Adds a 'PartOfDay' column to the DataFrame based on the time of day from the specified timestamp column.

    Args:
    df: The input DataFrame.
    timestamp_column: The name of the column containing timestamp information.

    Returns:
    DataFrame with an additional 'PartOfDay' column.
    """
    # Function to categorize part of day
    def part_of_day(hour_col):
        return (when((hour_col >= 6) & (hour_col < 12), lit("Morning"))
                .when((hour_col >= 12) & (hour_col < 18), lit("Afternoon"))
                .when((hour_col >= 18) & (hour_col < 24), lit("Evening"))
                .otherwise(lit("Night")))

    # Add a part of day column
    df_with_part_of_day = df.withColumn("PartOfDay", part_of_day(hour(timestamp_column)))

    return df_with_part_of_day
```

```
def add_recency_feature(df: DataFrame, timestamp_column: str, partition_column: str) -> DataFrame:
    """
    Adds a 'Recency' column to the DataFrame, indicating the time difference in hours from the last delay
    at the same partition (like an airport).

    Args:
    df: The input DataFrame.
    timestamp_column: The name of the column containing timestamp information.
    partition_column: The column name to partition by (e.g., airport ID).

    Returns:
    DataFrame with an additional 'Recency' column.
    """
    # Define the window specification
    windowSpec = Window.partitionBy(partition_column).orderBy(timestamp_column)

    # Calculate the time difference from the last event
    df_with_recency = df.withColumn("LastEventTime", lag(timestamp_column).over(windowSpec))
    df_with_recency = df_with_recency.withColumn("Recency",
                                                 (col(timestamp_column).cast("long") -
col("LastEventTime").cast("long")) / 3600) # in hours

    return df_with_recency
```

```
def add_prev_flight_delay_feature(df, tail_num_col, delay_flag_col, datetime_col):
    """
    Adds a feature to indicate whether the previous flight for the same plane was delayed.

    Args:
    df (DataFrame): Spark DataFrame containing flight data.
    tail_num_col (str): Column name for the tail number identifying each plane.
    delay_flag_col (str): Column name indicating whether a flight is delayed.
    datetime_col (str): Column name for the datetime of each flight.

    Returns:
    DataFrame: Modified DataFrame with an additional 'Prev_Flight_Delayed' column.
    """

    # Define window specification: partition by plane and order by datetime
    windowSpec = Window.partitionBy(tail_num_col).orderBy(datetime_col)

    # Create 'Prev_Flight_Delayed' column using lag function
    df_with_feature = df.withColumn("Prev_Flight_Delayed", lag(delay_flag_col, 1).over(windowSpec))

    # Replace nulls in 'Prev_Flight_Delayed' for first flights of each plane
    df_with_feature = df_with_feature.fillna({"Prev_Flight_Delayed": 0})

    return df_with_feature
```

```
# Impute null values by calling the calculate_moving_averages for train_df
train_df = calculate_moving_averages(train_df, -10, -1)
train_df = calculate_moving_averages(train_df, -16, -1)
train_df = calculate_moving_averages(train_df, -30, -1)
train_df = calculate_moving_averages(train_df, -30, -1)
train_df = calculate_moving_averages(train_df, -50, -1)

# Call the add_seasonality_feature function on train_df
train_df = add_seasonality_feature(train_df, "FL_DATE")

# Call the add_seasonality_feature function on test_df
test_intermediate_df = add_seasonality_feature(test_intermediate_df, "FL_DATE")

# Call the add_part_of_day_feature function on train_df
train_df = add_part_of_day_feature(train_df, "DateTime")

# Call the add_part_of_day_feature function on test_df
test_intermediate_df = add_part_of_day_feature(test_intermediate_df, "DateTime")

# Call the add_recency_feature function on train_df
train_df = add_recency_feature(train_df, "DateTime", "ORIGIN_AIRPORT_ID")

# Call the add_recency_feature function on test_df
test_intermediate_df = add_recency_feature(test_intermediate_df, "DateTime", "ORIGIN_AIRPORT_ID")

# Call the add_prev_flight_delay_feature function on train_df
train_df = add_prev_flight_delay_feature(train_df, "TAIL_NUM", "DEP_DEL15", "DateTime")

# Call the add_prev_flight_delay_feature function on test_df
test_intermediate_df = add_prev_flight_delay_feature(test_intermediate_df, "TAIL_NUM", "DEP_DEL15", "DateTime")
```

Create PageRank Features

```
# Create nodes from unique airport IDs from the DataFrame
vertices = train_df.select("ORIGIN_AIRPORT_ID").distinct()
vertices = vertices.withColumnRenamed("ORIGIN_AIRPORT_ID", "id")
```

```
# Create edges
edges = train_df.groupBy(['ORIGIN_AIRPORT_ID', 'DEST_AIRPORT_ID']).agg({'DEP_DEL15': 'sum'})
edges = edges.withColumnRenamed("ORIGIN_AIRPORT_ID", "src")
edges = edges.withColumnRenamed("DEST_AIRPORT_ID", "dst")
edges = edges.withColumnRenamed("sum(DEP_DEL15)", "sum_delays")
```

```
# Create GraphFrame from nodes and edges
g = GraphFrame(vertices, edges)
```

```
# Run PageRank
results = g.pageRank(resetProbability=0.15, tol=0.01)
```

```
# display(results.vertices)
```

```
# Extract PageRank scores
vertices_df = results.vertices.select("id", "pagerank")

# Join PageRank scores back to training data
vertices_df = vertices_df.withColumnRenamed("id", "ORIGIN_AIRPORT_ID")
train_df = train_df.join(vertices_df, on=['ORIGIN_AIRPORT_ID'], how='left')
```

Set Response and Predictor Variables

```
# print("-------------------------------------------------------------------------------------------------------")
# print("Setting variables to predict flight delays")
# Target variable
myY = "DEP_DEL15"
# Categorical predictor variables
categoricals = ['DAY_OF_MONTH', 'DAY_OF_WEEK', 'OP_UNIQUE_CARRIER', 'DEP_TIME_BLK', 'MONTH']
# Numeric predictor variables
numerics = ['YEAR', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyRelativeHumidity',
'HourlyWindDirection', 'HourlyWindSpeed']
# All predictor variables in a single list
myX = categoricals + numerics

# Create new dataframe with predictor vars, target var, etc.
train_df_bl = train_df.select(myX + [myY, "DATE_VARIABLE"])
```

```
-------------------------------------------------------------------------------------------------------
Setting variables to predict flight delays
```

Baseline Logistic Regression

```
## Current possible ways to handle categoricals in string indexer is 'error', 'keep', and 'skip'
# For each column in categoricals, create "StringIndexer" object to index categorical features by assigning
# them numerical values. Then convert categorical indices created by 'StringIndexer" into a sparse vector.
# Imputer used for imputing missing values and replacing them with mean or median of column.
# indexers = map(lambda c: StringIndexer(inputCol=c, outputCol=c+"_idx", handleInvalid = 'keep'), categoricals)
# ohes = map(lambda c: OneHotEncoder(inputCol=c + "_idx", outputCol=c+"_class"),categoricals)
indexers = map(lambda c: StringIndexer(inputCol=c, outputCol=c+"_idx", handleInvalid='keep'), categoricals)
ohes = map(lambda c: OneHotEncoder(inputCol=c+"_idx", outputCol=c+"_class"), categoricals)
imputers = Imputer(inputCols = numerics, outputCols = numerics)

# Establish features columns
# Categorical variables with "_class" appended for each of the categoricals to use the sparse-encoded vars
# Numerics
featureCols = list(map(lambda c: c+"_class", categoricals)) + numerics

# Build the stage for the ML pipeline
# List contains all transformation stages necessary for preprocessing data:
# -indexing and encoding categorical variables
# -imputing missing values in numerical columns
# -assembling feature columns into a vector
# -indexing the target variable
model_matrix_stages = list(indexers) + list(ohes) + [imputers] + \
                    [VectorAssembler(inputCols=featureCols, outputCol="features"),
StringIndexer(inputCol="DEP_DEL15", outputCol="label",handleInvalid='skip')]

# Apply StandardScaler to create scaledFeatures
# Specifies input column containing the features that need to be scaled, 'features'
# specifies output column where scaled features will be stored 'scaledFeatures'
# withSTD std will be 1
# withMean means the mean will be 0
scaler = StandardScaler(inputCol="features",
                        outputCol="scaledFeatures",
                        withStd=True,
                        withMean=True)
```

Define blocking Time Series Split

```
class BlockingTimeSeriesSplit:
    def __init__(self, n_splits=5):
        self.n_splits = n_splits

    def split(self, X, y=None, groups=None):

        n_samples = X.count()
        # print("n_samples " + str(n_samples))
        k_fold_size = n_samples // self.n_splits
        # print("k_fold_size " + str(k_fold_size))

        margin = 0

        for i in range(self.n_splits):
            start = i * k_fold_size

            # print("start " + str(start))
            stop = start + k_fold_size
            # print("stop " + str(stop))
            mid = int(0.75 * (stop - start)) + start
            # print("mid " + str(mid))
            # print("mid+margin " + str(mid+margin))
            # print("X " + str(X.count()))

            X_with_rank = X.withColumn("rank", F.row_number().over(window_spec))
            train_train_indices = X_with_rank.filter((X_with_rank['rank'] >= start) & (X_with_rank['rank'] < mid))
            train_valid_indices = X_with_rank.filter((X_with_rank['rank'] >= mid + margin) & (X_with_rank['rank'] <
stop))

            # print("train_train_indices " + str(train_train_indices.count()))
            # print("train_valid_indices " + str(train_valid_indices.count()))
            # print("train indicces ")

            # print("train valid indices ")

            yield train_train_indices, train_valid_indices
```

Define getMetrics function

```
def getMetrics(predictions):
    # Calculate precision, recall, and AUC
    binary_evaluator = BinaryClassificationEvaluator()
    multi_evaluator = MulticlassClassificationEvaluator()

    auc = binary_evaluator.evaluate(predictions)
    tp = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: 'truePositiveRateByLabel'})
    fp = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: 'falsePositiveRateByLabel'})

    # Precision: TP/(TP+FP)
    precision = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: 'precisionByLabel',
multi_evaluator.metricLabel: 1.0}) # should be list
    # Recall: TP(TP+FN)
    recall = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: 'recallByLabel',
multi_evaluator.metricLabel: 1.0})
    # F1: Harmonic mean of precision and recall
    f1 = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: 'fMeasureByLabel',
multi_evaluator.metricLabel: 1.0})

    # Display or use the precision, recall, and AUC values as needed
    print(f"AUC: {auc}, Precision: {precision}, Recall: {recall}, F1 Score: {f1}")

    return f1
```

Define mean_encoding function

```
def mean_encoding_df(df,mean_encoding_columns):
    # mean_encoding_columns = [‘DAY_OF_MONTH’, ‘DAY_OF_WEEK’, ‘OP_UNIQUE_CARRIER’, ‘DEP_TIME_BLK’,
‘MONTH’,‘TAIL_NUM’,
    #                 ‘FL_DATE’,‘ORIGIN_AIRPORT_ID’]
    #df = spark.createDataFrame(data, columns)
    # Calculate mean encoding
    for column in mean_encoding_columns:
        # Calculate mean target value for each category in the column
        mean_encoded_subject = (
            df.groupBy(column)
            .agg(F.mean("DEP_DEL15").alias("mean_target"))
            .withColumnRenamed(column, "subject")
        )
        # print("The column encoded is " + column)
        # Join the mean_encoded_subject DataFrame with the original DataFrame
        df = df.join(mean_encoded_subject, df[column] == mean_encoded_subject["subject"], "left_outer")
        #df.show(1)
        # Replace the original column with the mean encoded values
        df = df.withColumn(column, F.col("mean_target")).drop("subject","mean_target")
#.withColumnRenamed("mean_target",column)
        #df.show(1)
    # columns = df.columns
    # # Shift the first column to the last
    # df = df.select(columns[1:] + [columns[0]])
    # Show the result
    #df.show(1)
    return df
```

Define Model Parameter for Baseline

```
# Use logistic regression
# Set number of iterations for optimization algorithm, 10
# Set input column to be used for training
lr = LogisticRegression(maxIter=10, featuresCol = "scaledFeatures")

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

Run K-fold Cross validation on logistic regression (baseline) and evaluate test dat

```
# Define the time series split
btscv = BlockingTimeSeriesSplit(n_splits=5)

# cv_model_set_lr = []
# cv_model_set_lr_f1 = []

# for i, (train_index, valid_index) in enumerate(btscv.split(train_df_bl)):
#     print(f"Fold {i + 1}")

#     train_set = train_df_bl.filter(train_df_bl["index"].isin(train_index))
#     valid_set = train_df_bl.filter(train_df_bl["index"].isin(valid_index))

### Aditya's code ###
cv_model_set_lr = []
cv_model_set_lr_f1 = []

for i, (train_train_index,train_valid_index ) in enumerate(btscv.split(train_df_bl)):
    print(f"Fold {i + 1}")
    print("prepping training...")

    train_train_index.show(1)
    print("train count "+ str(train_train_index.count()))
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    train_valid_index.show(1)
    print("valid count "+str(train_valid_index.count()))
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    train_train_index = train_train_index.checkpoint()
    train_valid_index = train_valid_index.checkpoint()

    col_to_drop = ["index","rank","DATE_VARIABLE"]
    train_train_index = train_train_index.drop(*col_to_drop)
    train_valid_index = train_valid_index.drop(*col_to_drop)


    spark.sparkContext.setCheckpointDir("/ml/5Y_checkpoint")
    ######################################################################
    ### fitting the model ################

    # ## lr pipeline
    # cv_model_lr = pipeline_lr.fit(train_train_index)
    # cv_model_set_lr.append(cv_model_lr)

    ## lr pipeline
    cv_model_lr = pipeline_lr.fit(train_train_index)
    model_name = f"cv_model_set_lr_{i}"  # Creating unique model name
    cv_model_set_lr.append((model_name, cv_model_lr))  # Appending model with name to the list

    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    predictions_lr = cv_model_lr.transform(train_valid_index)
    cv_model_set_lr_f1.append(getMetrics(predictions_lr))

######################################################################
## best model LR EN ################
# print("Running test data on best model for LR pipeline ....")

# best_i_lr = cv_model_set_lr_f1.index(max(cv_model_set_lr_f1))
# print(f"The best model is in fold {best_i_lr+1} ")
# print("Running test data on best model ....")
# cvModel_test = cv_model_set[best_i_lr]
# predictions_test = cvModel_test.transform(test_intermediate_df)
```

```
  # test_f1 = getMetrics(predictions_test)
  # print("Success pipeline for logistic regression finished")
```

```
|          17|          2|               US|   1800-1859|    2|2015|                    65.0|                    73.0|
76.0|             230.0|           8.0|        0|   2015-02-17|693612|
+------------+-----------+-----------------+------------+-----+----+-------------------------+-------------------------+--
--------------------+-------------------+---------------+---------+-------------+------+
only showing top 1 row

train count 130052
+------------+-----------+-----------------+------------+-----+----+-------------------------+-------------------------+--
--------------------+-------------------+---------------+---------+-------------+------+
|DAY_OF_MONTH|DAY_OF_WEEK|OP_UNIQUE_CARRIER|DEP_TIME_BLK|MONTH|YEAR|HourlyDewPointTemperature|HourlyDryBulbTemperature|Ho
urlyRelativeHumidity|HourlyWindDirection|HourlyWindSpeed|DEP_DEL15|DATE_VARIABLE|  rank|
+------------+-----------+-----------------+------------+-----+----+-------------------------+-------------------------+--
--------------------+-------------------+---------------+---------+-------------+------+
|          26|          4|               DL|   1300-1359|    2|2015|                    32.0|                    37.0|
81.0|             300.0|           10.0|        0|   2015-02-26|823664|
+------------+-----------+-----------------+------------+-----+----+-------------------------+-------------------------+--
--------------------+-------------------+---------------+---------+-------------+------+
only showing top 1 row

valid count 43351
AUC: 0.667131362090004, Precision: 0.5589519650655022, Recall: 0.052100293064148484, F1 Score: 0.09531610693275745
```

```
  # Get best performing model
  print(cv_model_set_lr_f1)
```

```
[0.3889204008319153, 0.0, 0.01865747312918272, 0.003481591087126817, 0.09531610693275745]
```

```
  # predictions_test = cv_model_set_lr[0][1].transform(test_intermediate_df)
  # test_f1 = getMetrics(predictions_test)
  # print("Success pipeline for logistic regression finished")
```

```
  # df = spark.createDataFrame([(x,) for x in cv_model_set_lr_f1], ["value"])
  # ##best_i_lr = cv_model_set_lr_f1.index(max(cv_model_set_lr_f1))
  # best_i_lr = int(df.agg(spark_max("value")).collect()[0][0])
  # print(f"The best model is in fold {best_i_lr+1} ")
  # print("Running test data on best model ....")
  # cvModel_test = cv_model_set_lr[best_i_lr]
  # predictions_test = cvModel_test.transform(test_intermediate_df)
  # test_f1 = getMetrics(predictions_test)
  # print("Success pipeline for logistic regression finished")
```

Feature Selection of Feature-Engineered Vars

```
  print("-------------------------------------------------------------------------------------------------")
  print("Setting variables to predict flight delays")
  # Target variable
  myY = "DEP_DEL15"
  # Categorical predictor variables
  categoricals = ['DAY_OF_MONTH', 'DAY_OF_WEEK', 'OP_UNIQUE_CARRIER', 'DEP_TIME_BLK', 'MONTH', 'Season', 'PartOfDay']
  # Numeric predictor variables
  numerics = ['YEAR', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyRelativeHumidity',
  'HourlyWindDirection', 'HourlyWindSpeed', 'Recency', 'Prev_Flight_Delayed', 'pagerank']
  # All predictor variables in a single list
  myX = categoricals + numerics

  # Create new dataframe with predictor vars, target var, etc.
  train_df_adv = train_df.select(myX + [myY, "DATE_VARIABLE"])
```

```
--------------------------------------------------------------------------------------------
Setting variables to predict flight delays
```

Create Model Classifier and Pipelines

```
lr_en = LogisticRegression(maxIter=10, elasticNetParam=0.5, featuresCol = "scaledFeatures")
pipeline_lr_en = Pipeline(stages=model_matrix_stages+[scaler]+[lr_en])

# XGB pipeline
xgb_classifier = SparkXGBClassifier(
    features_col="features",
    label_col="label",
    prediction_Col="prediction",
    num_workers=2,
    device="cuda",

)
pipeline_xgb = Pipeline(stages=model_matrix_stages+[scaler]+[xgb_classifier])

# Define DecisionTreeClassifier
dt_classifier = DecisionTreeClassifier(
    featuresCol="features",
    labelCol="label",
    predictionCol="prediction")

# DT Pipeline
pipeline_dt = Pipeline(stages=model_matrix_stages + [scaler, dt_classifier])

## MLP Pipeline
imputers_mlp = Imputer(inputCols = numerics, outputCols = numerics)
featureCols_mlp = categoricals + numerics
#print(featureCols)
model_matrix_stages_mlp = [imputers_mlp] + \
                    [VectorAssembler(inputCols=featureCols_mlp, outputCol="features"),
StringIndexer(inputCol="DEP_DEL15", outputCol="label",handleInvalid='skip')]

# change input layers depending on the number of features sent to MLP
layers = [len(categoricals) + len(numerics),8,8,2]
mlp = MultilayerPerceptronClassifier(layers=layers, seed=1234, labelCol="label", featuresCol="features", maxIter=100,
blockSize=128)

pipeline_mlp = Pipeline(stages=model_matrix_stages_mlp+[mlp])
```

```
NameError: name 'model_matrix_stages' is not defined
```

Run model pipelines for K-folds and choose best model and evaluate on test data

```
# Define the time series split
btscv = BlockingTimeSeriesSplit(n_splits=5)

cv_model_set_lr_en = []
cv_model_set_lr_en_f1 = []

cv_model_set_xgb = []
cv_model_set_xgb_f1 = []

cv_model_set_dt = []
cv_model_set_dt_f1 = []

# cv_model_set_mlp = []
# cv_model_set_mlp_f1 = []

for i, (train_train_index,train_valid_index ) in enumerate(btscv.split(train_df_adv)):
    print(f"Fold {i + 1}")
    print("prepping training...")

    train_train_index.show(1)
    print("train count "+ str(train_train_index.count()))
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    train_valid_index.show(1)
    print("valid count "+str(train_valid_index.count()))
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    train_train_index = train_train_index.cache()
    train_valid_index = train_valid_index.cache()

    col_to_drop = ["index","rank","DATE_VARIABLE"]
    train_train_index = train_train_index.drop(*col_to_drop)
    train_valid_index = train_valid_index.drop(*col_to_drop)

    ###################################################################
    #### fitting the model ################

    ## lr pipeline
    cv_model_lr_en = pipeline_lr_en.fit(train_train_index)
    model_name = f"cv_model_lr_en_{i}"  # Creating unique model name
    cv_model_set_lr_en.append((model_name, cv_model_lr_en))  # Appending model with name to the list
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    predictions_lr_en = cv_model_lr_en.transform(train_valid_index)
    print(f"LR EN Fold {i + 1} Metrics .............")
    cv_model_set_lr_en_f1.append(getMetrics(predictions_lr_en))

    ## xgb pipeline
    cv_model_xgb = pipeline_xgb.fit(train_train_index)
    model_name = f"cv_model_xgb_{i}"  # Creating unique model name
    cv_model_set_xgb.append((model_name, cv_model_xgb))  # Appending model with name to the list
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    predictions_xgb = cv_model_xgb.transform(train_valid_index)
    print(f"XGB Fold {i + 1} Metrics .............")
    cv_model_set_xgb_f1.append(getMetrics(predictions_xgb))

    ## dt pipeline
    cv_model_dt = pipeline_dt.fit(train_train_index)
    model_name = f"cv_model_dt_{i}"  # Creating unique model name
    cv_model_set_dt.append((model_name, cv_model_dt))  # Appending model with name to the list
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))
```

```
    predictions_dt = cv_model_dt.transform(train_valid_index)
    print(f"DT Fold {i + 1} Metrics ............")
    cv_model_set_dt_f1.append(getMetrics(predictions_dt))

    ## mlp pipeline
    train_train_index = mean_encoding_df(train_train_index,categoricals)
    train_valid_index = mean_encoding_df(train_valid_index,categoricals)

    mlp_model = pipeline_mlp.fit(train_train_index)
    predictions = mlp_model.transform(train_valid_index)
    print(f"MLP Fold {i + 1} Metrics ............")
    tqa1 = getMetrics(predictions)
```

```
Internal error. Attach your notebook to a different compute or restart the current compute.
```

```
# Get best performing LR EN model
print(cv_model_set_lr_en_f1)
```

```
[0.3852617345953255, 0.0, 0.016279770877298764, 0.004431052818149592, 0.23397950364242498]
```

```
predictions_test = cv_model_set_lr_en[0][1].transform(test_intermediate_df)
test_f1 = getMetrics(predictions_test)
print("Success pipeline for logistic regression finished")
```

```
AUC: 0.6563250498248829, Precision: 0.40140625, Recall: 0.2088475065563439, F1 Score: 0.2748144681420071
Success pipeline for logistic regression finished
```

```
# Get best performing XGB model
print(cv_model_set_xgb_f1)
```

```
[0.33256433007985803, 0.0259567387687188, 0.23361778752055118, 0.04718018094191257, 0.26841824960056815]
```

```
predictions_test = cv_model_set_xgb[0][1].transform(test_intermediate_df)
test_f1 = getMetrics(predictions_test)
print("Success pipeline for XGB finished")
```

```
AUC: 0.6462437239584229, Precision: 0.3832769453842436, Recall: 0.22559723492934838, F1 Score: 0.28399580251337314
Success pipeline for XGB finished
```

```
# Get best performing DT model
print(cv_model_set_dt_f1)
```

```
[0.16890825757006536, 0.0, 0.27898878752002226, 0.0008866034222892101, 0.0009792720744246776]
```

```
predictions_test = cv_model_set_dt[2][1].transform(test_intermediate_df)
test_f1 = getMetrics(predictions_test)
print("Success pipeline for decision tree finished")
```

```
AUC: 0.4681028526548694, Precision: 0.3590094404159256, Recall: 0.05335908120743978, F1 Score: 0.09255326679408227
Success pipeline for logistic regression finished
```

```
for i, (model_name, model) in enumerate(cv_model_set_lr):
    # Example: Checkpointing each model
    checkpoint_dir = f"/ml_41/df_otpw_checkpoint/{model_name}"
    model.write().overwrite().save(checkpoint_dir)
```

```
for i, (model_name, model) in enumerate(cv_model_set_lr_en):
    # Example: Checkpointing each model
    checkpoint_dir = f"/ml_41/df_otpw_checkpoint/{model_name}"
    model.write().overwrite().save(checkpoint_dir)
```

```
for i, (model_name, model) in enumerate(cv_model_set_xgb):
    # Example: Checkpointing each model
    checkpoint_dir = f"/ml_41/df_otpw_checkpoint/{model_name}"
    model.write().overwrite().save(checkpoint_dir)
```