databricks MASTER_V2_5Y_HYPERPARAMETER_RANDOM_TEAM_4_1_mlp

(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, MinMaxScaler
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

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

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

Set Checkpoint directory

```
spark.sparkContext.setCheckpointDir("/ml_41/df_otpw_checkpoint_HH_12_15")
```

```
try:
    test_file = dbutils.fs.ls("/mnt/mids-w261/ml_41/")
    for item in test_file:
        print(f"Name: {item.name}, Type: {'Directory' if item.isDir() else 'File'}, Size: {item.size} bytes")
except Exception as e:
    print("Error accessing specific file or directory:", e)
```

```
        at com.databricks.logging.UsageLogging.recordOperation(UsageLogging.scala:571)
        at com.databricks.logging.UsageLogging.recordOperation$(UsageLogging.scala:540)
        at com.databricks.backend.daemon.dbutils.FSUtils.recordOperation(DBUtilsCore.scala:69)
        at com.databricks.backend.daemon.dbutils.FSUtils.recordDbutilsFsOp(DBUtilsCore.scala:133)
        at com.databricks.backend.daemon.dbutils.FSUtils.ls(DBUtilsCore.scala:211)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
        at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:397)
        at py4j.Gateway.invoke(Gateway.java:306)
        at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
        at py4j.commands.CallCommand.execute(CallCommand.java:79)
        at py4j.ClientServerConnection.waitForCommands(ClientServerConnection.java:195)
        at py4j.ClientServerConnection.run(ClientServerConnection.java:115)
        at java.lang.Thread.run(Thread.java:750)
Caused by: hadoop_azure_shaded.com.microsoft.azure.storage.StorageException: Server failed to authenticate the request. M
ake sure the value of Authorization header is formed correctly including the signature.
        at hadoop_azure_shaded.com.microsoft.azure.storage.StorageException.translateException(StorageException.java:87)
        at hadoop_azure_shaded.com.microsoft.azure.storage.core.StorageRequest.materializeException(StorageRequest.java:3
```

```
mount_points = dbutils.fs.mounts()
for mount_point in mount_points:
    print(mount_point.mountPoint)
```

```
/databricks-datasets
/Volumes
/databricks/mlflow-tracking
/databricks-results
/databricks/mlflow-registry
/mnt/azure
/mnt/models
/mnt/mids-w261
/Volume
/mnt/26122spring23-container-1
/mnt/Azure_Team_Storage
/volumes
/
/volume
```

```
# Specify the directory path
directory_path = "dbfs:/mnt/mids-w261/"

# List the contents of the directory
contents = dbutils.fs.ls(directory_path)

# Display the contents
for item in contents:
    print(f"Name: {item.name}, Type: {'Directory' if item.isDir() else 'File'}, Size: {item.size} bytes")
```

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

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.
```

```
df_otpw.groupBy(['OP_CARRIER']).count().show()
```

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

```
# 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()
```

```
root
 |-- DEP_DEL15: string (nullable = true)
 |-- DEP_DELAY_NEW: string (nullable = true)
 |-- ORIGIN_AIRPORT_ID: string (nullable = true)
 |-- DEST_AIRPORT_ID: string (nullable = true)
 |-- FL_DATE: string (nullable = true)
 |-- CRS_DEP_TIME: string (nullable = true)
 |-- DAY_OF_MONTH: string (nullable = true)
 |-- DAY_OF_WEEK: string (nullable = true)
 |-- OP_UNIQUE_CARRIER: string (nullable = true)
 |-- DEP_TIME_BLK: string (nullable = true)
 |-- TAIL_NUM: string (nullable = true)
 |-- MONTH: string (nullable = true)
 |-- YEAR: string (nullable = true)
 |-- HourlyDewPointTemperature: string (nullable = true)
 |-- HourlyDryBulbTemperature: string (nullable = true)
 |-- HourlyWetBulbTemperature: string (nullable = true)
 |-- HourlyRelativeHumidity: string (nullable = true)
 |-- HourlyWindDirection: string (nullable = true)
 |-- HourlyWindSpeed: string (nullable = true)
```

Drop all observations with null for target variable

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

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()
```

```
+---------+------+
|DEP_DEL15| count|
```

```
+---------+------+
|      0.0|278126|
|      1.0|277302|
+---------+------+
```

```
# 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_STR").substr(3, 2)))

# 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_TIME_STR").substr(3, 2)))

# # 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)
|-- DAY_OF_MONTH: integer (nullable = true)
|-- DAY_OF_WEEK: integer (nullable = true)
|-- OP_UNIQUE_CARRIER: string (nullable = true)
|-- DEP_TIME_BLK: string (nullable = true)
|-- TAIL_NUM: 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_5Y.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)
```

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_STR").substr(3, 2)))

# 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")
)
```

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')) \

#df_otpw.printSchema()
```

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

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
```

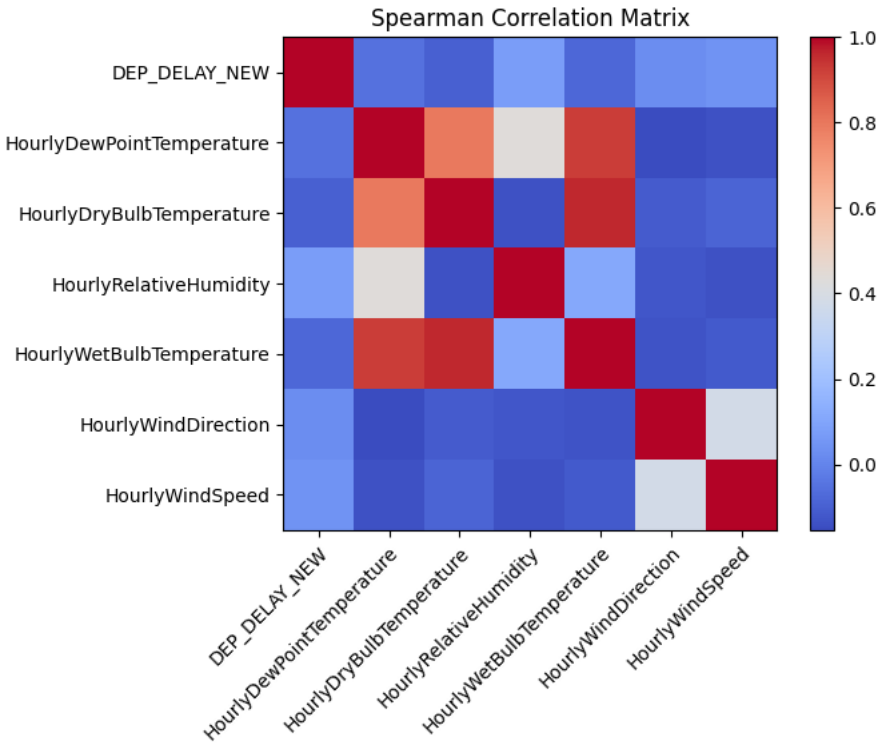
EDA: Correlation matrix

```
# 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.         -0.05136747 -0.09600651  0.07196656 -0.07918758  0.02626487
   0.04394154]
 [-0.05136747  1.          0.7941362   0.43401957  0.92532879 -0.15362637
  -0.13968608]
 [-0.09600651  0.7941362   1.         -0.13888998  0.96098605 -0.11172925
  -0.08915245]
 [ 0.07196656  0.43401957 -0.13888998  1.          0.10983875 -0.1254569
  -0.13639726]
 [-0.07918758  0.92532879  0.96098605  0.10983875  1.         -0.13196971
  -0.11578979]
 [ 0.02626487 -0.15362637 -0.11172925 -0.1254569  -0.13196971  1.
   0.38085768]
 [ 0.04394154 -0.13968608 -0.08915245 -0.13639726 -0.11578979  0.38085768
   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()
```



Skewness and Kurtosis Analysis (NOTE: Make sure to add all the engineered feat

```
import pandas as pd
from scipy.stats import skew, kurtosis


# Convert the Spark DataFrame to a Pandas DataFrame
df_otpw_pandas = train_df.toPandas()

feature_columns = ['YEAR', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyRelativeHumidity',
'HourlyWindDirection', 'HourlyWindSpeed', 'Recency', 'Prev_Flight_Delayed', 'pagerank']

# Initialize lists to store results
skewness_values = []
kurtosis_values = []

for column in feature_columns:
    # Calculate skewness and kurtosis
    skewness = skew(df_otpw_pandas[column])
    kurt = kurtosis(df_otpw_pandas[column])

    # Append to lists
    skewness_values.append((column, skewness))
    kurtosis_values.append((column, kurt))

# Print results
print("Skewness Values:")
for feature, value in skewness_values:
    print(f"{feature}: {value}")

print("\nKurtosis Values:")
for feature, value in kurtosis_values:
    print(f"{feature}: {value}")
```

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

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_intermediate_df.cache()
# test_df.cache()
```

```
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, rank: int]
```

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
```

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')
test_intermediate_df = test_intermediate_df.join(vertices_df, on=['ORIGIN_AIRPORT_ID'], how='left')
test_df = test_df.join(vertices_df, on=['ORIGIN_AIRPORT_ID'], how='left')
```

```
# 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_seasonality_feature function on test_df
test_df = add_seasonality_feature(test_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_part_of_day_feature function on test_df
test_df = add_part_of_day_feature(test_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_recency_feature function on test_df
test_df = add_recency_feature(test_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")

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

# Apply log transformation to train_df
train_df = train_df.withColumn("Prev_Flight_Delayed", log1p("Prev_Flight_Delayed")) \
                        .withColumn("pagerank", log1p("pagerank"))

# Apply log transformation to test_intermediate_df
test_intermediate_df = test_intermediate_df.withColumn("Prev_Flight_Delayed", log1p("Prev_Flight_Delayed")) \
                        .withColumn("pagerank", log1p("pagerank"))

# Apply log transformation to test_df
test_df = test_df.withColumn("Prev_Flight_Delayed", log1p("Prev_Flight_Delayed")) \
                        .withColumn("pagerank", log1p("pagerank"))
```

Set Response and Predictor Variables for Baseline

```
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
```

Creating Stages

```
## 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)
```

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

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")
```

```
Fold 1
  Cancelled
```

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

```
NameError: name 'cv_model_set_lr_f1' is not defined
```

```
# 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
# print("The features count for MLP are :" + str(len(featureCols_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])
```

Hyperparameter tuning with Random Search

```
#  Range of hyperparameters we want to test for each model
# import random

# # Hyperparameter space for Logistic Regression
# param_space_lr = {
#     'regParam': [0.01, 0.1, 0.5],
#     'maxIter': [10, 20, 50]
# }

# # Hyperparameter space for XGBoost
# param_space_xgb = {
#     'learning_rate': [0.01, 0.1, 0.2, 0.3],
#     'max_depth': [3, 5, 7],
#     'min_child_weight': [1, 3, 5]
# }

# # Hyperparameter space for Decision Tree
# param_space_dt = {
#     'maxDepth': [3, 5, 10],
#     'minInstancesPerNode': [1, 2, 4]
# }

# Define the hyperparameter space for MLP
param_space_mlp = {
    'layers': [[len(myX), 5, 4, 2], [len(myX), 10, 5, 2], [len(myX), 15, 10, 2]],
    'blockSize': [64, 128, 256],
    'maxIter': [50, 100, 150]
}
```

(Hyperparameter tuning) Run model pipelines for K-folds and choose best mode

```
from pyspark.ml.tuning import ParamGridBuilder
# 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 = []

# Initialization
# lr_params_metrics = []
# xgb_params_metrics = []
# dt_params_metrics = []
mlp_params_metrics = []

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

    # Randomly sample hyperparameters
    # random_params_lr = {k: random.choice(v) for k, v in param_space_lr.items()}
    # random_params_xgb = {k: random.choice(v) for k, v in param_space_xgb.items()}
    # random_params_dt = {k: random.choice(v) for k, v in param_space_dt.items()}
    random_params_mlp = {k: random.choice(v) for k, v in param_space_mlp.items()}


    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)


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

    # ###################################
    # ## lr pipeline
    # ###################################

    # # Update the model within the pipeline
    # lr_stage = pipeline_lr_en.getStages()[-1]  # Assuming the model is the last stage in the pipeline
    # for param, value in random_params_lr.items():
    #     if lr_stage.hasParam(param):
    #         lr_stage.set(lr_stage.getParam(param), value)
```

```
    # # Fit and evaluate the updated 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))

    # # Store hyperparameters and metrics for Logistic Regression
    # metrics_lr = getMetrics(predictions_lr_en)
    # lr_params_metrics.append((random_params_lr, metrics_lr))

    # ################################
    # ## xgb pipeline
    # ################################
    # xgb_stage = pipeline_xgb.getStages()[-1]
    # for param, value in random_params_xgb.items():
    #     if xgb_stage.hasParam(param):
    #         xgb_stage.set(xgb_stage.getParam(param), value)

    # 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))

    # # Store hyperparameters and metrics for XGBoost
    # metrics_xgb = getMetrics(predictions_xgb)
    # xgb_params_metrics.append((random_params_xgb, metrics_xgb))

    # ################################
    # ## dt pipeline
    # ################################
    # dt_stage = pipeline_dt.getStages()[-1]
    # for param, value in random_params_dt.items():
    #     if dt_stage.hasParam(param):
    #         dt_stage.set(dt_stage.getParam(param), value)

    # 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))

    # # Store hyperparameters and metrics for Decision Tree
    # metrics_dt = getMetrics(predictions_dt)
    # dt_params_metrics.append((random_params_dt, metrics_dt))

    ################################
    ## mlp pipeline
    ################################
    train_train_index = mean_encoding_df(df_otpw2,['DAY_OF_MONTH', 'DAY_OF_WEEK', 'OP_UNIQUE_CARRIER',
'DEP_TIME_BLK', 'MONTH','TAIL_NUM','FL_DATE','ORIGIN_AIRPORT_ID'])

    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)
```

```
######## start of hyperparameter tuning ##############
# Update the MLP model within the pipeline
mlp_stage = pipeline_mlp.getStages()[-1]  # Assuming the MLP model is the last stage in the pipeline
for param, value in random_params_mlp.items():
    if mlp_stage.hasParam(param):
        mlp_stage.set(mlp_stage.getParam(param), value)

# Fit and evaluate the updated pipeline
mlp_model = pipeline_mlp.fit(train_train_index)
predictions_mlp = mlp_model.transform(train_valid_index)
print(f"MLP Fold {i + 1} Metrics ............")
mlp_metrics = getMetrics(predictions_mlp)
mlp_params_metrics.append((random_params_mlp, mlp_metrics))
######### end of mlp hyperparameter tuning #############
```

```
+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------------------+-------------------+-------
-----------+---------+-------------+----+
|           1|          4|               EV|   0800-0859|    1|Winter|  Morning|2015|                      9.0|
23.0|                55.0|              230.0|           18.0|0.03333333333333333|                0.0|2.436247433960859
5|        0|   2015-01-01|   1|
+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------------------+-------------------+-------
-----------+---------+-------------+----+
only showing top 1 row

+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------+-------------------+-------------------+
---------+-------------+-------+
|DAY_OF_MONTH|DAY_OF_WEEK|OP_UNIQUE_CARRIER|DEP_TIME_BLK|MONTH|Season|PartOfDay|YEAR|HourlyDewPointTemperature|HourlyDryB
ulbTemperature|HourlyRelativeHumidity|HourlyWindDirection|HourlyWindSpeed|Recency|Prev_Flight_Delayed|          pagerank|
DEP_DEL15|DATE_VARIABLE|   rank|
+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------+-------------------+-------------------+
---------+-------------+-------+
|           4|          6|               WN|   1100-1159|    7|Summer|  Morning|2015|                     52.0|
```

```
/databricks/python/lib/python3.10/site-packages/xgboost/sklearn.py:782: UserWarning: Loading a native XGBoost model with
Scikit-Learn interface.
  warnings.warn("Loading a native XGBoost model with Scikit-Learn interface.")
```

```
|          24|          4|               NK|   1500-1559|    8|Summer|Afternoon|2017|                     76.0|
78.0|                93.0|              140.0|            5.0|    0.4| 0.6931471805599453|1.4867275195696854|        1|
2017-08-24|5448866|
+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------+-------------------+-------------------+
---------+-------------+-------+
only showing top 1 row

LR EN Fold 4 Metrics ............
AUC: 0.550681984835765, Precision: 0.815641383290847, Recall: 0.08044821010085676, F1 Score: 0.1464516268324805
AUC: 0.550681984835765, Precision: 0.815641383290847, Recall: 0.08044821010085676, F1 Score: 0.1464516268324805
XGB Fold 4 Metrics ............
AUC: 0.6670154229843736, Precision: 0.7069498375465568, Recall: 0.5501722060400202, F1 Score: 0.6187850315200313
AUC: 0.6670154229843736, Precision: 0.7069498375465568, Recall: 0.5501722060400202, F1 Score: 0.6187850315200313
DT Fold 4 Metrics ............
AUC: 0.461194198093253, Precision: 0.6845571153252712, Recall: 0.5577578109433855, F1 Score: 0.6146864535490127
AUC: 0.461194198093253, Precision: 0.6845571153252712, Recall: 0.5577578109433855, F1 Score: 0.6146864535490127
Fold 5
+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------------------+-------------------+--------
----------+---------+-------------+-------+
```

Hyperparameter random search results

```
# Initialize best scores, parameters, and indices
# best_score_lr = float('-inf')
# best_params_lr = None
# best_index_lr = None

# best_score_xgb = float('-inf')
# best_params_xgb = None
# best_index_xgb = None

# best_score_dt = float('-inf')
# best_params_dt = None
# best_index_dt = None

best_score_mlp = float('-inf')
best_params_mlp = None
best_index_mlp = None

# Find best parameters for Logistic Regression
# for idx, (params, score) in enumerate(lr_params_metrics):
#     if score > best_score_lr:
#         best_score_lr = score
#         best_params_lr = params
#         best_index_lr = idx

# # Find best parameters for XGBoost
# for idx, (params, score) in enumerate(xgb_params_metrics):
#     if score > best_score_xgb:
#         best_score_xgb = score
#         best_params_xgb = params
#         best_index_xgb = idx

# # Find best parameters for Decision Tree
# for idx, (params, score) in enumerate(dt_params_metrics):
#     if score > best_score_dt:
#         best_score_dt = score
#         best_params_dt = params
#         best_index_dt = idx

# Find best parameters for MLP
for idx, (params, score) in enumerate(mlp_params_metrics):
    if score > best_score_mlp:
        best_score_mlp = score
        best_params_mlp = params
        best_index_mlp = idx

# Print the results
# print("Best Parameters for Logistic Regression:", best_params_lr, "at index", best_index_lr, "with F1 score",
best_score_lr)
# print("Best Parameters for XGBoost:", best_params_xgb, "at index", best_index_xgb, "with F1 score", best_score_xgb)
# print("Best Parameters for Decision Tree:", best_params_dt, "at index", best_index_dt, "with F1 score",
best_score_dt)
print("Best Parameters for MLP:", best_params_mlp, "at index", best_index_dt, "with F1 score", best_score_dt)
```

```
Best Parameters for Logistic Regression: {'regParam': 0.1, 'maxIter': 50} at index 3 with F1 score 0.1464516268324805
Best Parameters for XGBoost: {'learning_rate': 0.1, 'max_depth': 7, 'min_child_weight': 1} at index 2 with F1 score 0.638
2960555725774
Best Parameters for Decision Tree: {'maxDepth': 10, 'minInstancesPerNode': 2} at index 3 with F1 score 0.6146864535490127
```

Best Parameters for Logistic Regression: {'regParam': 0.1, 'maxIter': 50} at index 3 with F1 score 0.1464516268324805.

```
Best Parameters for XGBoost: {'learning_rate': 0.1, 'max_depth': 7, 'min_child_weight': 1} at index 2 with F1 score 0.6382960555725774.
```

```
Best Parameters for Decision Tree: {'maxDepth': 10, 'minInstancesPerNode': 2} at index 3 with F1 score
```

(Run Best hyperparameters) Run model pipelines for K-folds and choose best mo

```
# 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 = []

# Best hyperparameters identified from previous steps
# best_params_lr = {'regParam': 0.01, 'maxIter': 50}
# best_params_xgb = {'learning_rate': 0.1, 'max_depth': 7, 'min_child_weight': 1}
# best_params_dt = {'maxDepth': 10, 'minInstancesPerNode': 2}
# best_params_mlp = {'maxDepth': 10, 'minInstancesPerNode': 2}

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)


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

    # ## lr pipeline
    # # Update the Logistic Regression model within the pipeline with the best parameters
    # lr_stage = pipeline_lr_en.getStages()[-1]  # Assuming LogisticRegression is the last stage
    # lr_stage.setParams(**best_params_lr)

    # 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
    # # Update the XGBoost model within the pipeline with the best parameters
    # xgb_stage = pipeline_xgb.getStages()[-1]  # Assuming XGBoost is the last stage
    # xgb_stage.setParams(**best_params_xgb)
```

```
# 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
# dt_stage = pipeline_dt.getStages()[-1]  # Assuming DecisionTree is the last stage
# dt_stage.setParams(**best_params_dt)

# 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)
model_name = f"cv_model_mlp_{i}"  # Creating unique model name
cv_model_set_mlp.append((model_name, mlp_model))  # Appending model with name to the list

predictions_mlp = mlp_model.transform(train_valid_index)
print(f"MLP Fold {i + 1} Metrics ............")
cv_model_set_mlp_f1.append(getMetrics(predictions_mlp))

# Clear cache
train_train_index.unpersist()
train_valid_index.unpersist()
```

```
--------+-------------+-------+
|DAY_OF_MONTH|DAY_OF_WEEK|OP_UNIQUE_CARRIER|DEP_TIME_BLK|MONTH|Season|PartOfDay|YEAR|HourlyDewPointTemperature|HourlyDryB
ulbTemperature|HourlyRelativeHumidity|HourlyWindDirection|HourlyWindSpeed|Recency|Prev_Flight_Delayed|         pagerank|D
EP_DEL15|DATE_VARIABLE|   rank|
+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------+-------------------+-----------------+-
--------+-------------+-------+
|          25|          6|               HA|   1900-1959|    2|Winter|  Evening|2017|                     60.0|
68.0|                 76.0|              350.0|            5.0|    0.3| 0.6931471805599453|0.941353117884665|        0|
2017-02-25|4359093|
+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------+-------------------+-----------------+-
--------+-------------+-------+
only showing top 1 row

+------------+-----------+-----------------+------------+-----+------+---------+----+-------------------------+----------
--------------+----------------------+-------------------+---------------+-------+-------------------+-----------------+-------
-----------+---------+-------------+-------+
|DAY_OF_MONTH|DAY_OF_WEEK|OP_UNIQUE_CARRIER|DEP_TIME_BLK|MONTH|Season|PartOfDay|YEAR|HourlyDewPointTemperature|HourlyDryB
ulbTemperature|HourlyRelativeHumidity|HourlyWindDirection|HourlyWindSpeed|           Recency|Prev_Flight_Delayed|
pagerank|DEP_DEL15|DATE_VARIABLE|   rank|
```

```
  # Get best performing LR EN model
  print(cv_model_set_mlp_f1)
```

```
[0.0, 0.0, 0.0, 0.0, 3.15015199483375e-05]
```

```
# columns_to_drop = ["OP_UNIQUE_CARRIER", "DEP_TIME_BLK", "Season", "PartOfDay"]
# test_intermediate_df = test_intermediate_df.drop(*columns_to_drop)
```

```
test_intermediate_df = mean_encoding_df(test_intermediate_df,categoricals)
predictions_test = cv_model_set_mlp[4][1].transform(test_intermediate_df)
test_f1 = getMetrics(predictions_test)
print("Success pipeline for MLP finished")
```

```
AUC: 0.4987690567331066, Precision: 0.5, Recall: 1.0801247328041443e-06, F1 Score: 2.1602447989406163e-06
Success pipeline for MLP finished
```

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

```
[0.6014881838204655, 0.5252106132273504, 0.6396389653933661, 0.6191108551193029, 0.4790645587208539]
```

```
# predictions_test = cv_model_set_xgb[2][1].transform(test_intermediate_df)
# test_f1 = getMetrics(predictions_test)
# print("Success pipeline for XGB finished")
```

```
AUC: 0.6663051402487015, Precision: 0.6241313787061881, Recall: 0.6371807016274239, F1 Score: 0.6305885371152783
Success pipeline for XGB finished
```

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

```
[0.5888514312751361, 0.4072322828568411, 0.5694572818874976, 0.6080477215397387, 0.4969022402297534]
```

```
# predictions_test = cv_model_set_dt[3][1].transform(test_intermediate_df)
# test_f1 = getMetrics(predictions_test)
# print("Success pipeline for decision tree finished")
```

```
AUC: 0.45110372746897304, Precision: 0.6075560474950031, Recall: 0.5110065223836348, F1 Score: 0.555114414417162
Success pipeline for decision tree finished
```

```
# # Get best performing MLP model
print(cv_model_set_mlp_f1)
```

```
predictions_test = cv_model_set_mlp[3][1].transform(test_intermediate_df)
test_f1 = getMetrics(predictions_test)
print("Success pipeline for mlp finished")
```

Use Best Model for Blind Test on 2019 Test Data

```
# Set hyperparameters
best_params_lr = {'regParam': 0.01, 'maxIter': 50}
lr_stage = pipeline_lr_en.getStages()[-1]  # Assuming LogisticRegression is the last stage
lr_stage.setParams(**best_params_lr)

# Fit best model on 2018 data
cv_model_lr_en_best = pipeline_lr_en.fit(test_intermediate_df)

# Make predictions on 2019 data
predictions_test = cv_model_lr_en_best.transform(test_df)

# Get metrics
test_f1 = getMetrics(predictions_test)
```

```
AUC: 0.6592321267176402, Precision: 0.6281916179637506, Recall: 0.5479492157954964, F1 Score: 0.5853331413251511
```

```
# Accessing the Logistic Regression stage
lr_stage = cv_model_lr_en_best.stages[-1]  # Assuming LogisticRegression is the last stage

# Retrieving coefficients
coefficients = lr_stage.coefficients

# Intercept
intercept = lr_stage.intercept

# Print or use the coefficients and intercept as needed
print("Coefficients: ", coefficients)
print("Intercept: ", intercept)
```

```
Coefficients:  (89,[0,1,2,8,9,12,26,27,28,29,31,32,33,36,37,38,39,40,43,44,47,51,52,54,55,56,57,58,59,61,62,64,65,66,67,68,69,70,72,74,75,78,80,81,82,86,87,88],[-0.03422883623974198,-0.02603135943334198,-0.010242850535853857,-0.008265948550499032,0.010022727972964074,-0.0014558480244131519,0.012930908235720262,0.023079398911605764,0.00401818033691441,0.030285584814891367,-0.04816643355232188,-0.029986880989023402,-0.007317494690701334,0.0012823209664944888,0.009012545153167932,-0.05137787572831204,-0.003306739786154914,0.08595723352020157,-0.05651429388778689,0.017297242830388904,0.014651141908265641,7.569346008523492e-05,-0.08592721819466514,0.07630372230564134,-0.035952513651514736,-0.054190957120089985,-0.0058173548170263,-0.018903096722040744,-0.05379285297219848,0.019243697482506296,0.04262300576899214,0.07390698227613333,-0.040449447278198107,0.15598072243618658,0.21224281344616588,0.27654905339301755,0.10670042599811631,-0.018964633425458892,0.1509522197993467,-0.07752430242454837,-0.06394348380919096,0.04654937293183273,0.04594313904019291,-0.04011727254873604,-0.03701301349387 89,-0.14500822383750447,-0.0054952517680821635,-0.05154634403878367])
Intercept:  -0.03489188303629698
```

```
predictions_test.checkpoint()
```

```
DataFrame[ORIGIN_AIRPORT_ID: string, DEP_DEL15: int, DEP_DELAY_NEW: int, DEST_AIRPORT_ID: string, FL_DATE: string, CRS_DEP_TIME: string, DAY_OF_MONTH: int, DAY_OF_WEEK: int, OP_UNIQUE_CARRIER: string, DEP_TIME_BLK: string, TAIL_NUM: string, MONTH: int, YEAR: int, HourlyDewPointTemperature: int, HourlyDryBulbTemperature: int, HourlyWetBulbTemperature: int, HourlyRelativeHumidity: int, HourlyWindDirection: int, HourlyWindSpeed: int, DATE_VARIABLE: date, CRS_DEP_TIME_INT: int, CRS_DEP_TIME_STR: string, CRS_DEP_TIME_FMT: string, DateTime: timestamp, pagerank: double, Season: string, PartOfDay: string, LastEventTime: timestamp, Recency: double, Prev_Flight_Delayed: double, DAY_OF_MONTH_idx: double, DAY_OF_WEEK_idx: double, OP_UNIQUE_CARRIER_idx: double, DEP_TIME_BLK_idx: double, MONTH_idx: double, DAY_OF_MONTH_class: vector, DAY_OF_WEEK_class: vector, OP_UNIQUE_CARRIER_class: vector, DEP_TIME_BLK_class: vector, MONTH_class: vector, features: vector, label: double, scaledFeatures: vector, rawPrediction: vector, probability: vector, prediction: double]
```

```
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)
```

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

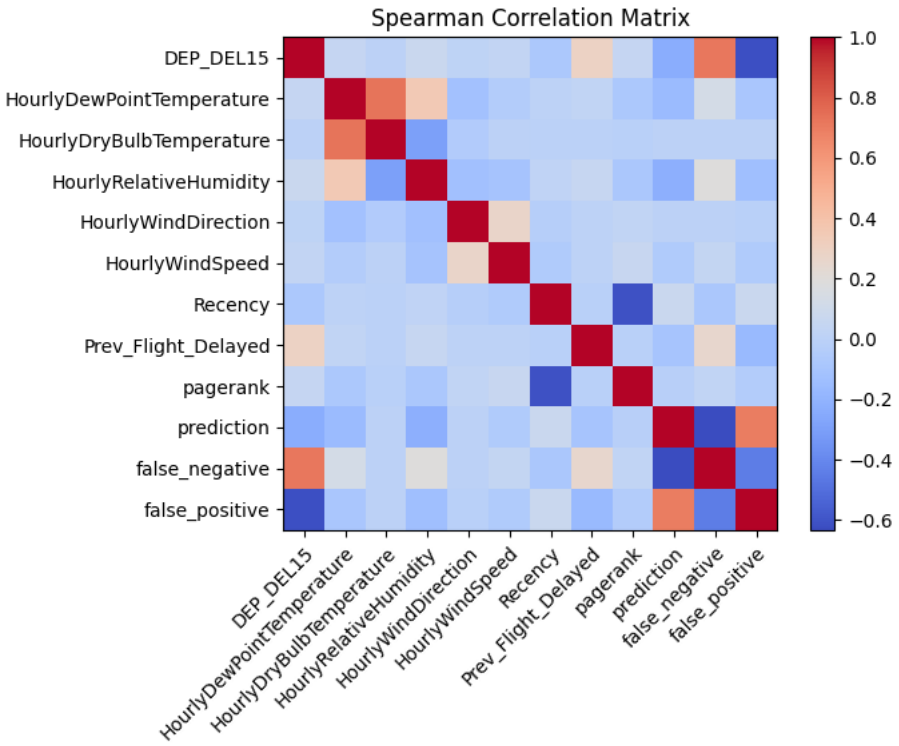
Gap Analysis

```
# Confusion Matrix
pred_test_data = predictions_test.groupBy(
    ['DEP_DEL15','prediction']
    ).agg(
        F.count('*').alias('Count')
    )
pred_test_data_df = pred_test_data.toPandas().pivot_table(
    values='Count',
    index='DEP_DEL15',
    columns='prediction',
    aggfunc='sum',
    fill_value=0
 )
pred_test_data_df.sort_index()
```

prediction	0.0	1.0
DEP_DEL15		
0	508021	615793
1	786852	364470

```
from pyspark.ml.stat import Correlation

# Spearman Correlation Matrix on numeric features
pred_NUM_FEATURES = ['DEP_DEL15', 'HourlyDewPointTemperature', 'HourlyDryBulbTemperature', 'HourlyRelativeHumidity',
'HourlyWindDirection',
                'HourlyWindSpeed', 'Recency', 'Prev_Flight_Delayed', 'pagerank', 'prediction'
]
pred_gap_df_num = predictions_test.select(pred_NUM_FEATURES).dropna()
pred_gap_df_num = pred_gap_df_num.withColumn('false_negative', when((col('DEP_DEL15') == 1) & (col('prediction') ==
0.0), 1).otherwise(0))
pred_gap_df_num = pred_gap_df_num.withColumn('false_positive', when((col('DEP_DEL15') == 0) & (col('prediction') ==
1.0), 1).otherwise(0))

# Assemble the columns into a single feature column
assembler_gap = VectorAssembler(inputCols=pred_gap_df_num.columns, outputCol="features")
gap_assembled = assembler_gap.transform(pred_gap_df_num)

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

```
[ 3.30759602e-02 -4.54398784e-02  2.02924591e-03 -1.03951281e-01
  2.70842790e-01  1.00000000e+00 -5.80608782e-02  6.77467173e-03
  5.96120561e-02 -5.34947552e-02  3.79123453e-02 -5.62560595e-02]
[-7.17802928e-02  6.93507470e-03 -5.84704251e-03  2.11220427e-02
 -2.81841028e-02 -5.80608782e-02  1.00000000e+00 -1.26427253e-02
 -6.12804846e-01  7.39862191e-02 -8.28464193e-02  7.45415887e-02]
[ 2.89582516e-01  2.87919733e-02 -3.63030290e-03  5.24027702e-02
  1.14271400e-02  6.77467173e-03 -1.26427253e-02  1.00000000e+00
 -1.03501507e-02 -9.81177191e-02  2.53513826e-01 -1.63813448e-01]
[ 4.49348199e-02 -7.31336172e-02 -8.79409545e-03 -8.29434188e-02
  2.68185043e-02  5.96120561e-02 -6.12804846e-01 -1.03501507e-02
  1.00000000e+00 -2.01557481e-02  2.94785952e-02 -4.14718114e-02]
[-2.33633046e-01 -1.55944790e-01  2.35901609e-03 -2.23607395e-01
  7.48521918e-04 -5.34947552e-02  7.39862191e-02 -9.81177191e-02
 -2.01557481e-02  1.00000000e+00 -6.32620812e-01  7.00185180e-01]
[ 7.18459566e-01  1.27452286e-01  3.78745039e-03  1.85094251e-01
  3.61244991e-03  3.79123453e-02 -8.28464193e-02  2.53513826e-01
  2.94785952e-02 -6.32620812e-01  1.00000000e+00 -4.42951717e-01]
[-6.16529779e-01 -8.94568675e-02  1.37989760e-03 -1.32375591e-01
 -1.18566433e-02 -5.62560595e-02  7.45415887e-02 -1.63813448e-01
 -4.14718114e-02  7.00185180e-01 -4.42951717e-01  1.00000000e+00]]
```


Spearman Correlation Matrix
DEP_DEL15
HourlyDewPointTemperature
HourlyDryBulbTemperature
HourlyRelativeHumidity
HourlyWindDirection
HourlyWindSpeed
Recency
Prev_Flight_Delayed
pagerank
prediction
false_negative
false_positive
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6