databricks CLEAN 1Y BASELINE TEAM_4_final

(https://databricks.com)

```
from pyspark.sql.functions import isnan, when, count, col, split, trim, lit, avg, sum
print("Welcome to the W261 final project!")
```

```
Welcome to the W261 final project!
```

Know your mount

Here is the mounting for this class, your source for the original data! Remember, you only have Read access, not Write! Also, become familiar with `dbutils` the equivalent of `gcp` in DataProc

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

Table

	path	name	size	modificationTime
1	dbfs:/mnt/mids-w261/HW5/	HW5/	0	0
2	dbfs:/mnt/mids-w261/OTPW_12M/	OTPW_12M/	0	0
3	dbfs:/mnt/mids-w261/OTPW_1D_CSV/	OTPW_1D_CSV/	0	0
4	dbfs:/mnt/mids-w261/OTPW_36M/	OTPW_36M/	0	0
5	dbfs:/mnt/mids-w261/OTPW_3M/	OTPW_3M/	0	0
6	dbfs:/mnt/mids-w261/OTPW_3M_2015.csv	OTPW_3M_2015.csv	1500620247	1679772070000

10 rows

Data for the Project

OTPW Data: This is our joined data (We joined Airlines and Weather). This is the main dataset for your project, the previous 3 are given for reference. You can attempt your own join for Extra Credit. Location `dbfs:/mnt/mids-w261/OTPW_60M/` and more, several samples are given!

```
# OTPW
df_otpw = spark.read.format("csv").option("header","true").load(f"dbfs:/mnt/mids-w261/OTPW_12M/")
#display(df_otpw)
```

```
df_otpw.count()
```

```
11623708
```

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

df_otpw.printSchema()
```

```
root
 |-- DEP_DEL15: string (nullable = true)
 |-- DEP_DELAY_NEW: 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)
 |-- 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)
```

```
# Add date filter
from pyspark.sql.functions import concat, col,lpad, lit, to_date

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()
```

```
root
 |-- DEP_DEL15: string (nullable = true)
 |-- DEP_DELAY_NEW: 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)
 |-- 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)
 |-- DATE_VARIABLE: date (nullable = true)
```

Filter Data

```
from pyspark.sql.functions import *

# Drop all observations with null of target variable
df_otpw = df_otpw.dropna(subset=['DEP_DEL15'])

# Cast as numeric variables into numeric format from string
df_otpw = df_otpw.withColumn("DEP_DELAY_NEW", regexp_replace("DEP_DELAY_NEW", "s", "").cast('int')) \
    .withColumn("Year", regexp_replace("Year", "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()
```

```
root
 |-- DEP_DEL15: string (nullable = true)
 |-- DEP_DELAY_NEW: integer (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)
 |-- MONTH: string (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)
```

Convert Data to Delta Lake Format

```
# Configure Path
DELTALAKE_GOLD_PATH_1Y = "/ml/flights1Y.delta"

# Remove table if it exists
dbutils.fs.rm(DELTALAKE_GOLD_PATH_1Y, recurse=True)

# Save table as Delta Lake
df_otpw.write.format("delta").mode("overwrite").save(DELTALAKE_GOLD_PATH_1Y)
```

```
# Configure Path
DELTALAKE_GOLD_PATH_1Y = "/ml/flights1Y.delta"

# Re-read as Delta Lake (begin execution)
df_otpw = spark.read.format("delta").load(DELTALAKE_GOLD_PATH_1Y)

# Review data
#display(df_otpw)
```

```
# 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
```

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

date_var = ['DATE_VARIABLE']
myX = categoricals + numerics + date_var

# Create new dataframe with predictor vars, target var, etc.
df_otpw2 = df_otpw.select(myX + [myY])
```

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

```
df_otpw2.select("DEP_DEL15").distinct().show()
```

```
+---------+
|DEP_DEL15|
+---------+
|      1.0|
|      0.0|
+---------+
```

Build Grid of GLM Models w/ Standardization+CrossValidation

```
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
import time
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.sql import Row
from pyspark.sql.functions import col
from pyspark.sql import functions as F
from pyspark.sql.window import Window
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.mllib.evaluation import MulticlassMetrics


## add checkpoint
spark.sparkContext.setCheckpointDir("/ml/df_otpw2_checkpoint")

total_records = df_otpw2.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_otpw2_with_rank = df_otpw2.withColumn("rank", F.row_number().over(window_spec))
split_date_row = df_otpw2_with_rank.filter(col("rank") == training_records).select("DATE_VARIABLE").collect()[0]
split_date = split_date_row["DATE_VARIABLE"]

# drop rank
df_otpw2 = df_otpw2.drop("rank")

# Filter the DataFrame based on the dynamically determined split_date
train_df = df_otpw2.filter(col("DATE_VARIABLE") < split_date)
test_df = df_otpw2.filter(col("DATE_VARIABLE") >= split_date)

# cache dataframes for performance
train_df.cache()
test_df.cache()


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)
from pyspark.ml.evaluation import RegressionEvaluator

featureCols = list(map(lambda c: c+"_class", categoricals)) + numerics

model_matrix_stages = list(indexers) + list(ohes) + [imputers] + \
                     [VectorAssembler(inputCols=featureCols, outputCol="features"),
StringIndexer(inputCol="DEP_DEL15", outputCol="label",handleInvalid='skip')]


scaler = StandardScaler(inputCol="features",
                        outputCol="scaledFeatures",
                        withStd=True,
                        withMean=True)
```

```
lr = LogisticRegression(maxIter=10, elasticNetParam=0.5, featuresCol = "scaledFeatures")


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

paramGrid = ParamGridBuilder() \
              .addGrid(lr.regParam, [0.1, 0.01]) \
              .build()

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))
        # Window specification for ordering and getting start and end dates for each fold
        #window_spec = Window.orderBy('DATE_VARIABLE')
        margin = 0

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

            print("start " + str(start))
            print(type(start))
            stop = start + k_fold_size
            print(type(stop))
            print("stop " + str(stop))
            mid = int(0.75 * (stop - start)) + start
            print("mid " + str(mid))
            print(type(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_train_indices.show(10))
            print("train valid indices ")
          #  print(train_valid_indices.show(10))
            yield train_train_indices, train_valid_indices



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


cv_model_set = []

for i, (train_train_index,train_valid_index ) in enumerate(btscv.split(train_df)):
    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)
    cv_model = pipeline.fit(train_train_index)
    cv_model_set.append(cv_model)
    #print(time.strftime('%H:%M%p %Z on %b %d, %Y'))

    predictions = cv_model.transform(train_valid_index)

    binary_evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")
    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 = multi_evaluator.evaluate(predictions, {multi_evaluator.metricName: 'precisionByLabel',
multi_evaluator.metricLabel: 1.0})

    # 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"Fold {i+1} metrics : AUC:{auc} Precision: {precision}, Recall: {recall}, F1 Score: {f1}")
    print(f"Fold {i+1} completed at " + time.strftime('%H:%M%p %Z on %b %d, %Y'))
```

```
n_samples 9151300
k_fold_size 1830260
start 0
<class 'int'>
<class 'int'>
stop 1830260
mid 1372695
<class 'int'>
mid+margin 1372695
X 9151300
train_train_indices 1372694
train_valid_indices 457565
train indicces
train valid indices
Fold 1
prepping training...
+------------+-----------+-----------------+------------+-----+----+-------------------------+------------------------+--
--------------------+-------------------+---------------+-------------+---------+----+
|DAY_OF_MONTH|DAY_OF_WEEK|OP_UNIQUE_CARRIER|DEP_TIME_BLK|MONTH|YEAR|HourlyDewPointTemperature|HourlyDryBulbTemperature|Ho
urlyRelativeHumidity|HourlyWindDirection|HourlyWindSpeed|DATE_VARIABLE|DEP_DEL15|rank|
+------------+-----------+-----------------+------------+-----+----+-------------------------+------------------------+--
```

Run best model

```
Test set : Fold 5 metrics : AUC:0.5845258839060238 Precision: 0.18844351900052056, Recall: 0.007435135121257401, F1 Scor
e: 0.01430582656161237
```