FP_Section1_Group2_Phase4_Notebook

August 4, 2022

1 W261 Summer 2022 - Section 1 Group 2

1.1 Phase 4 and Final Report

1.1.1 Prepared by Brian Moon, John Stilb, Jonas Degnan, and Shuhan Yu

2 Idler

[ ]:
```
import time
while True:

    print(1)
    time.sleep(300)
```

3 Phase 4 Workspace

3.1 Abstract

Our team focused on key shortcomings and discoveries made during the prior phase based on our gap analysis, observations, and instructor feedback. During this phase our team: Improved best model F1 by 168%. Addressed unbalanced training data. Engineered 6 more prognostic features. Improved data and modeling pipeline design and workflows.

3.2 Notebook Setup

3.2.1 Load Libraries

[ ]:
```
# General use Python
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
import random
from functools import reduce
import networkx as nx
from itertools import chain
import time

# PySpark
```

```python
from pyspark.sql import functions as ps
from pyspark.sql import Window, Row, DataFrame
from pyspark.sql.types import IntegerType

from pyspark.ml import Pipeline
from pyspark.ml.param import TypeConverters
from pyspark.ml.stat import Correlation, Summarizer
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder,␣
↪BucketedRandomProjectionLSH, VectorSlicer, StandardScaler, Imputer,␣
↪Binarizer, MinMaxScaler, VarianceThresholdSelector
from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.classification import LogisticRegression, LinearSVC,␣
↪RandomForestClassifier, MultilayerPerceptronClassifier, GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator,␣
↪MulticlassClassificationEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.mllib.evaluation import MulticlassMetrics

from pyspark.ml.stat import Correlation
from pyspark.ml.feature import VectorAssembler

# Sklearn
from sklearn import neighbors
from sklearn.ensemble import VotingClassifier

# GraphFrame
import graphframes as gf
```

3.2.2 Environmental Variables

[ ]:
```python
SEED = 2022
WRITE = False
CACHE = True
NOT_IMPLEMENTED = True
RE_TRAIN = False
```

[ ]:
```python
blob_container = "checkpoints" # The name of your container created in https://
↪portal.azure.com
storage_account = "w261s1g2" # The name of your Storage account created in␣
↪https://portal.azure.com
secret_scope = "w261-final-s1g2" # The name of the scope created in your local␣
↪computer using the Databricks CLI
secret_key = "checkpoints" # The name of the secret key created in your local␣
↪computer using the Databricks CLI
blob_url = f"wasbs://{blob_container}@{storage_account}.blob.core.windows.net"
```

```python
mount_path = "/mnt/mids-w261"

spark.conf.set(
  f"fs.azure.sas.{blob_container}.{storage_account}.blob.core.windows.net",
  dbutils.secrets.get(scope = secret_scope, key = secret_key))
```

3.2.3 Utility Functions

Evaluation Helpers

[ ]:
```python
def generate_novel_sample():
    """Generates a random novel flight vector for testing model classification.
 ↪"""
    pass
```

Visualization Functions

[ ]:
```python
# function that creates correlation matrix
def create_corr_matrix(df):
    """Returns a correlation matrix PySpark dataframe.

    Args:
        df (pyspark.sql.DataFrame): _description_

    Returns:
        pyspark.sql.DataFrame: _description_
    """
    # create assembler
    vector_col = "corr_features"
    assembler = VectorAssembler(inputCols=df.columns, outputCol=vector_col)
    df_vector = assembler.transform(df).select(vector_col)

    # create matrix
    matrix = Correlation.corr(df_vector, vector_col).collect()[0][0]
    corrmatrix = matrix.toArray().tolist()
    corrdf = spark.createDataFrame(corrmatrix, df.columns)

    return corrdf

# function that plots correlation heatmap
def plot_corr_heatmap(correlations, attr, fig_no, figsize = (30,20), fontsize =␣
 ↪22):
    """Generate correlation matrix heatmap.

    Args:
        correlations (_type_): _description_
        attr (_type_): _description_
        fig_no (_type_): _description_
```

```
        figsize (tuple, optional): _description_. Defaults to (30,20).
        fontsize (int, optional): _description_. Defaults to 22.
    """
    fig=plt.figure(fig_no)
    ax=fig.add_subplot(111)
    ax.set_title("Correlation Matrix for Specified Attributes")
    ax.set_xticklabels(['']+attr)
    ax.set_yticklabels(['']+attr)
    cax=ax.matshow(correlations,vmax=1,vmin=-1)
    fig.colorbar(cax)
    plt.rcParams["figure.figsize"] = figsize
    plt.rcParams['font.size'] = fontsize
    plt.show()

# function that creates histograms
def plot_hist(df, col_name, bins = 10):
    """Generates dataframe histogram.

    Args:
        df (pyspark.sql.DataFrame): _description_
        col_name (str): _description_
        bins (int, optional): _description_. Defaults to 10.
    """
    out_hist = df.select(col_name).rdd.flatMap(lambda x: x).histogram(bins)

    pd.DataFrame(
        list(zip(*out_hist)),
        columns=['bin', 'frequency']
    ).set_index(
        'bin'
    ).plot(kind='bar')
```

Exception Handling

[ ]:
```
def exception_handler(exception):
    def decorate(func):
        def call_function(*args, **kwargs):
            try:
                func(*args, **kwargs)
            except Exception as e:
                exception(e)
        return call_function
    return decorate

def show_exception(e):
    print(e)
```

```python
# Examples
# @exception_handler(show_exception)
# def foo(a, b):
#     return a + b

# @exception_handler(show_exception)
# def bar(c, d):
#     return c.index(d)

# @exception_handler(show_exception)
# def custom_function(a):
#     if  a % 2 != 0:
#         raise Exception('Buttered side down!')
#     return a**2
```

3.2.4 Blob Storage Reference

[ ]:
```python
display(dbutils.fs.ls(f"{blob_url}/"))
```

3.3 Data Ingestion

3.3.1 Class and Function Space

Classes and functions supporting data processing.

[ ]:
```python
def load_datasets():
    year = ps.udf(lambda x: x, IntegerType())

    df_full = spark.read.parquet(f"{blob_url}/df_full_YEAR=2015")
    # df_full = df_full.withColumn("YEAR", year(2015))
    for x in range(2016, 2020, 1):
        df = spark.read.parquet(f"{blob_url}/df_full_YEAR={x}")
    #     df= df.withColumn("YEAR", year(x))
        df_full = df_full.union(df)

    df_test = spark.read.parquet(f"{blob_url}/df_full_YEAR=2021")
    # df_test= df_test.withColumn("YEAR", year(2021))
    df_hype_param_train = spark.read.parquet(f"{blob_url}/df_full_YEAR=2020")
    # df_hype_param_train= df_hype_param_train.withColumn("YEAR", year(2020))

    hype, ignore = df_full.randomSplit([0.05, 0.95])
    df_full = df_full.union(df_hype_param_train)
    df_hype_param_train, ignore = df_hype_param_train.randomSplit([0.25, 0.75])
    df_hype_param_train = df_hype_param_train.union(hype)

    return [df_full, df_hype_param_train, df_test]
```

[ ]:
```
def cast_data (df):
    '''
    initial casting for the loaded data set.
    '''

    df = df.distinct().dropna(how="any", subset=["ORIGIN", 'DEST'])

    time_cols = ['CRS_DEP_TIME', 'DEP_TIME','CRS_ARR_TIME', 'ARR_TIME']
    for col in time_cols:
        df= df.withColumn(col, ps.lpad(ps.col(col), 4, '0'))

    df = df.withColumn('DELAY_FLAG', ps.coalesce(ps.
↪when(df['ARR_DEL15']==0,None).otherwise(df['ARR_DEL15']),
                                                            ps.
↪when(df['CANCELLED']==0,None).otherwise(df['CANCELLED']),
                                                            ps.
↪when(df['DIVERTED']==0, None).otherwise(df['DIVERTED'])))
    df = df.fillna({'DELAY_FLAG':'0'})

    days = lambda i: i * 86400
    w = (Window()
       .partitionBy([ps.col('OP_UNIQUE_CARRIER'), ps.col('ORIGIN')])\
       .orderBy(ps.col("_utc_arr_ts").cast("long"))\
       .rangeBetween(-days(7), -days(1)))


    df = df.withColumn("DEP_HOUR",ps.coalesce(df['CRS_DEP_TIME'],␣
↪df['DEP_TIME']).substr(0,2).cast("int"))\
    .withColumn("ARR_HOUR",ps.coalesce(df['CRS_ARR_TIME'], df['ARR_TIME']).
↪substr(0,2).cast("int"))\
    .withColumn("YEAR",ps.col('FL_DATE').substr(0,4).cast("int"))\
    .withColumn("origin_weather_Avg_HourlyAltimeterSetting",df.
↪origin_weather_Avg_HourlyAltimeterSetting.cast("float"))\
    .withColumn("dest_weather_Avg_HourlyAltimeterSetting",df.
↪dest_weather_Avg_HourlyAltimeterSetting.cast("float"))\
    .withColumn("origin_weather_Avg_HourlyDewPointTemperature",df.
↪origin_weather_Avg_HourlyDewPointTemperature.cast("int"))\
    .withColumn("dest_weather_Avg_HourlyDewPointTemperature",df.
↪dest_weather_Avg_HourlyDewPointTemperature.cast("int"))\
    .withColumn("origin_weather_Avg_HourlyDryBulbTemperature",df.
↪origin_weather_Avg_HourlyDryBulbTemperature.cast("int"))\
    .withColumn("dest_weather_Avg_HourlyDryBulbTemperature",df.
↪dest_weather_Avg_HourlyDryBulbTemperature.cast("int"))\
    .withColumn("origin_weather_Avg_Precip_Double",df.
↪origin_weather_Avg_Precip_Double.cast("float"))\
```

```
    .withColumn("dest_weather_Avg_Precip_Double",df.
↪dest_weather_Avg_Precip_Double.cast("float"))\
    .withColumn("origin_weather_Avg_HourlyRelativeHumidity",df.
↪origin_weather_Avg_HourlyRelativeHumidity.cast("int"))\
    .withColumn("dest_weather_Avg_HourlyRelativeHumidity",df.
↪dest_weather_Avg_HourlyRelativeHumidity.cast("int"))\
    .withColumn("origin_weather_Avg_HourlyStationPressure",df.
↪origin_weather_Avg_HourlyStationPressure.cast("float"))\
    .withColumn("dest_weather_Avg_HourlyStationPressure",df.
↪dest_weather_Avg_HourlyStationPressure.cast("float"))\
    .withColumn("origin_weather_Avg_HourlyWetBulbTemperature",df.
↪origin_weather_Avg_HourlyWetBulbTemperature.cast("int"))\
    .withColumn("dest_weather_Avg_HourlyWetBulbTemperature",df.
↪dest_weather_Avg_HourlyWetBulbTemperature.cast("int"))\
    .withColumn("origin_weather_Avg_HourlyWindSpeed",df.
↪origin_weather_Avg_HourlyWindSpeed.cast("int"))\
    .withColumn("dest_weather_Avg_HourlyWindSpeed",df.
↪dest_weather_Avg_HourlyWindSpeed.cast("int"))\
    .withColumn("origin_weather_Avg_HourlyWindDirection",df.
↪origin_weather_Avg_HourlyWindDirection.cast("int"))\
    .withColumn("dest_weather_Avg_HourlyWindDirection",df.
↪dest_weather_Avg_HourlyWindDirection.cast("int"))\
    .withColumn("CRS_DEP_TIME",df.CRS_DEP_TIME.cast("int"))\
    .withColumn("CRS_ARR_TIME",df.CRS_ARR_TIME.cast("int"))\
    .withColumn("Avg_delay_past_7_days", ps.mean(ps.col("DELAY_FLAG").
↪cast('int')).over(w))

    return df
```

Persist datasets to memory.

```
[ ]: # preprocess datasets
df_full, df_hype_param_train, df_test = [df.persist() for df in map(cast_data,␣
↪load_datasets())]
```

3.4 Feature Engineering

In Phase 4, our feature engineering has been focused on improving the predicting power of our model and reducing the feature space to speed up runtimes.

3.4.1 Class and Function Space

Classes and functions supporting feature engineering.

```
[ ]: def get_corr(df, var1, var2="DELAY_FLAG"):
    print(f"{var1}-{var2} correlation: {df.stat.corr(var1, var2)}")
```

3.4.2 Engineered Feature Exploration

Small Carrier Bivariate Undercaptialized and understaffed regional carriers are expected to increase the likelihood of delay.

```
[ ]: # create xwalk of smaller carriers using full data
carrier_xwalk = (df_full
                 .groupBy(['OP_UNIQUE_CARRIER'])
                 .count()
                 .withColumnRenamed('count', 'carrier_cnt')
                 .withColumn('carrier_cnt_pcnt', ps.percent_rank().
↪over(Window().partitionBy().orderBy(ps.col('carrier_cnt'))))
                 .withColumn('small_carrier', ps.when(ps.
↪col('carrier_cnt_pcnt')<=0.25, 1).otherwise(0))
                 .drop('carrier_cnt', 'carrier_cnt_pcnt')
                )
```

Airport Congestion PageRank Rank airports by flight volume. Assess departure and inbound aircraft airport PageRank impacts.

PageRank Feature Engineering Unable to complete training on full dataset. Garbage collection overflow stack trace.

```
[ ]: def get_origin_df(df, n: int = 10, verbose: bool = False, store: bool = False):
    """Generates a PageRank dataframe based on origin airport IATA code.

    Args:
        df (pyspark.sql.DataFrame): A full featured dataframe.
        n (int, optional): Number of rows to return when displaying␣
↪intermediate results if verbose == true. Defaults to 10.
        verbose (bool, optional): Displays intermediate results. Defaults to␣
↪false.
        store (bool, optional): Stores dataframe in blob storage. Defaults to␣
↪false.

    Returns:
        pyspark.sql.DataFrame: A PageRank dataframe with three columns -␣
↪indexed IATA id, IATA id, PageRank
    """
    src_dst = df.select('origin_airport_iata', 'dest_airport_iata')\
                .withColumnRenamed('origin_airport_iata', 'iata')

    # build integer indexer for pagerank
    iata_indexer = StringIndexer(inputCol="iata", outputCol="idx",␣
↪handleInvalid = 'keep').fit(src_dst)
    # build numerical graph indexed features for GraphFrames parameters
    iata_src_idx = iata_indexer\
```

```
                .transform(src_dst)\
                .withColumnRenamed('idx','src')\
                .withColumnRenamed('iata','origin_airport_iata')\
                .withColumnRenamed('dest_airport_iata','iata')
    iata_idx = iata_indexer\
            .transform(iata_src_idx)\
            .withColumnRenamed('idx','dst')\
            .withColumnRenamed('iata','dest_airport_iata')\

    # cast indicies as integers
    iata_idx.withColumn('src', iata_idx.src.cast(IntegerType()))\
        .withColumn('dst', iata_idx.dst.cast(IntegerType()))

    # get graph nodes
    nodes = df.select('dest_airport_iso_country','origin_airport_iata',␣
↪'dest_airport_iata')\
            .selectExpr('dest_airport_iso_country',␣
↪'explode(array(origin_airport_iata, dest_airport_iata))')\
            .drop('dest_airport_iso_country')\
            .distinct()\
            .withColumnRenamed('col','iata')
    # generate graphframes vertices
    localVertices = iata_indexer\
                .transform(nodes)\
                .withColumnRenamed('idx','id')\
                .select('id','iata')\
                .rdd.map(tuple)\
                .collect()
    # generate graphframes edges
    localEdges = iata_idx.select('src', 'dst')\
                    .rdd.map(tuple)\
                    .collect()
    # build graphframe and get PR dataframe
    v = spark.createDataFrame(localVertices, ["id", "iata"])
    e = spark.createDataFrame(localEdges, ["src", "dst"])
    iata_pr = gf.GraphFrame(v, e).pageRank(maxIter=10).vertices

    if verbose:
        print(f'Nodes:\n{nodes.show(n)}\n')
        print(f'Vertices:\n{localVertices[:n]}\n')
        print(f'Edges:\n{localEdges[:n]}\n')
        print(f'Origin PR:\n{iata_pr.show(n)}')
    if store:
        iata_pr.write.parquet(f"{blob_url}/origin_iata_pr")
    return iata_pr
```

```python
def build_PR_feature(df_full, df_pr, f: float = 0.000001, verbose: bool =␣
↪False, store: bool = False):
    """Constructs a PageRank dataframe feature with origin city to be joined␣
↪with training dataframe.

    Args:
        df_full (pyspark.sql.DataFrame): A full featured dataframe.
        df_pr (pyspark.sql.DataFrame): A GraphFrames dataframe with id, iata,␣
↪and PageRank columns.
        f (float, optional): Fraction of rows to return when displaying␣
↪intermediate results if verbose == true. Defaults to 0.000001.
        use_stored (bool, optional): Uses a PR dataframe stored in Azure blob␣
↪storage. Defaults to false.
        verbose (bool, optional): Displays intermediate results.. Defaults to␣
↪false.
        store (bool, optional): Stores dataframe in blob storage. Defaults to␣
↪false.

    Returns:
        pyspark.sql.DataFrame: A dataframe
    """
    # convert PR dataframe to dictionary mapper
    iata_pr_mapper = {k:str(v) for (k,v) in [tuple(x) for x in df_pr.drop('id').
↪toPandas().to_dict(orient='split')['data']]}
    # map pagerank IATA dictionary to new column
    origin_iata_pr_feature = df_full\
                                .withColumn("origin_airport_pr", ps.
↪col("origin_airport_iata"))\
                                .replace(to_replace=iata_pr_mapper,␣
↪subset=["origin_airport_pr"])\
                                .withColumn("origin_airport_pr", ps.
↪col("origin_airport_pr").cast("float"))\
                                .select("ORIGIN", "origin_airport_pr")

    if verbose:
        display(origin_iata_pr_feature.sort(origin_iata_pr_feature.
↪origin_airport_pr.desc()).sample(fraction=f))
    if store:
        origin_iata_pr_feature.write.parquet(f"{blob_url}/
↪origin_iata_pr_feature")
    return origin_iata_pr_feature
```

3.4.3 Build Feature Engineered Dataframe

[ ]:
```
holiday_dates = [
    {"month": 12, "days": [i for i in range(20,32,1)]}, # xmas & nye
    {"month": 11, "days": [i for i in range(20,31,1)]}, # thanksgiving
    {"month": 9, "days": [i for i in range(1,10,1)]}, # labor day
    {"month": 7, "days": [i for i in range(1,8,1)]}, # 4th of July
    {"month": 5, "days": [i for i in range(25,32,1)]}, # memorial day
    {"month": 3, "days": [i for i in range(20,32,1)]}, # spring break
    {"month": 2, "days": [i for i in range(9,18,1)]}, # superbowl
    {"month": 1, "days": [i for i in range(1,5,1)]} # new years
]
```

[ ]:
```
# other engineering is done through this function
def engineer_features(in_df):

    mins = lambda i: i * 60

    w = (Window()
     .partitionBy([ps.col('ORIGIN')])
     .orderBy(ps.col("_utc_dept_ts").cast("long"))
     .rangeBetween(-mins(360), -mins(1))
    )
    days = lambda i: i * 86400
    w2 = (Window()
        .partitionBy([ps.col('OP_UNIQUE_CARRIER'), ps.col('ORIGIN')])\
        .orderBy(ps.col("_utc_arr_ts").cast("long"))\
        .rangeBetween(-days(7), -days(1)))
    pr_feature = build_PR_feature(df_full=in_df, df_pr=spark.read.
 ↪parquet(f"{blob_url}/origin_iata_pr")).persist()

    out_df = (
        in_df
        .withColumn('extreme_weather',
                    ps.greatest(
                        ps.col('origin_weather_Present_Weather_IceCrystals'),
                        ps.col('origin_weather_Present_Weather_Snow'),
                        ps.col('origin_weather_Present_Weather_Hail'),
                        ps.col('origin_weather_Present_Weather_Fog'),
                        ps.col('origin_weather_Present_Weather_Smoke'),
                        ps.col('origin_weather_Present_Weather_Storm'),
                        ps.col('origin_weather_Present_Weather_Haze'),
                        ps.col('dest_weather_Present_Weather_IceCrystals'),
                        ps.col('dest_weather_Present_Weather_Snow'),
                        ps.col('dest_weather_Present_Weather_Hail'),
                        ps.col('dest_weather_Present_Weather_Fog'),
                        ps.col('dest_weather_Present_Weather_Smoke'),
```

```
            ps.col('dest_weather_Present_Weather_Storm'),
            ps.col('dest_weather_Present_Weather_Haze')
          ))
    .withColumn('icy_weather',
          ps.when((ps.col('origin_weather_Avg_Precip_Double')>0)&(ps.col('origin_weather_Avg_HourlyDryBulbTemperature')<= 32), 1)
          .when((ps.col('dest_weather_Avg_Precip_Double')>0)&(ps.col('dest_weather_Avg_HourlyDryBulbTemperature')<= 32), 1)
          .otherwise(0)
         )
    .withColumn('holiday',
          ps.when((ps.col('MONTH')==12)&(ps.col('DAY_OF_MONTH').isin([i for i in range(20,32,1)])), 1)
          .when((ps.col('MONTH')==11)&(ps.col('DAY_OF_MONTH').isin([i for i in range(20,31,1)])), 1)
          .when((ps.col('MONTH')==9)&(ps.col('DAY_OF_MONTH').isin([i for i in range(1,10,1)])), 1)
          .when((ps.col('MONTH')==7)&(ps.col('DAY_OF_MONTH').isin([i for i in range(1,8,1)])), 1)
          .when((ps.col('MONTH')==5)&(ps.col('DAY_OF_MONTH').isin([i for i in range(25,32,1)])), 1)
          .when((ps.col('MONTH')==3)&(ps.col('DAY_OF_MONTH').isin([i for i in range(20,32,1)])), 1)
          .when((ps.col('MONTH')==2)&(ps.col('DAY_OF_MONTH').isin([i for i in range(9,18,1)])), 1)
          .when((ps.col('MONTH')==1)&(ps.col('DAY_OF_MONTH').isin([i for i in range(1,5,1)])), 1)
          .otherwise(0)
         )
    .withColumn("icy_runway", ps.max(ps.col("icy_weather").cast('int')).over(w))\
    .withColumn('weekend',
          ps.when((ps.col('DAY_OF_WEEK') > 4), 1)
          .otherwise(0)
         )
    .withColumn('holiday_month',
          ps.when((ps.col('MONTH') > 10), 1)
          .otherwise(0)
         )
    .withColumn("avg_flights_past_7_days", (ps.count(ps.col("_utc_arr_ts")).over(w2)/7))
    .join(carrier_xwalk, ['OP_UNIQUE_CARRIER'], 'left')
#     .join(pr_feature, ['ORIGIN'], 'left')\
#     .drop_duplicates()
  )
```

```
    return out_df
```

Persist full dataframes with novel features to memory.

[ ]:
```
# if RE_TRAIN:
df_full = engineer_features(df_full).persist()
df_hype_param_train = engineer_features(df_hype_param_train).persist()
df_test = engineer_features(df_test).persist()
```

3.5 Feature Selection

Features that have been removed from Phase 3 have been noted by ~~strikethroughs~~, and features that have been added have been ***bolded and italicized***.

Raw	Derived Features	Created Features
~~MONTH~~	Avg_delay_past_7_days	DELAY_FLAG
~~DAY_OF_WEEK~~	DEP_HOUR	***extreme_weather***
OP_UNIQUE_CARRIER	ARR_HOUR	***icy_runway***
AIR_TIME	***small_carrier***	***origin_airport_pr***
DISTANCE	***holiday***	
~~CRS_DEP_HOUR~~	***weekend***	
~~CRS_ARR_HOUR~~		
origin_weather_HourlyWindSpeed		
dest_weather_HourlyWindSpeed		
origin_weather_Avg_HourlyVisibility		
dest_weather_Avg_HourlyVisibility		

Key Features

[ ]:
```
## Set Response & Predictor Variables
myY = ['DELAY_FLAG']

# Feature Family: Limited
cat1 = [
    'holiday',
    'weekend',
    'extreme_weather',
    'icy_runway',
    'small_carrier'
 ]

num1 = [
 'AIR_TIME',
 'DEP_HOUR',
 'ARR_HOUR',
 'Avg_delay_past_7_days',
```

```
 'origin_weather_Avg_HourlyVisibility',
 'origin_weather_Avg_HourlyWindSpeed',
 'dest_weather_Avg_HourlyVisibility',
 'dest_weather_Avg_HourlyWindSpeed']
#  'origin_airport_pr']

X1 = cat1 + num1

features1 = {"categoricals": cat1, "numerics": num1, "myX": X1}

# Feature Family: Extended
cat2 = [
    'MONTH',
    'holiday',
    'weekend',
    'extreme_weather',
    'icy_runway',
    'small_carrier'
 ]

num2 = [
 'AIR_TIME',
 'DISTANCE',
 'DEP_HOUR',
 'ARR_HOUR',
 'Avg_delay_past_7_days',
 'origin_weather_Avg_HourlyStationPressure',
 'origin_weather_Avg_HourlyVisibility',
 'origin_weather_Avg_HourlyWindSpeed',
 'dest_weather_Avg_HourlyStationPressure',
 'dest_weather_Avg_HourlyVisibility',
 'dest_weather_Avg_HourlyWindSpeed']
#  'origin_airport_pr']

X2 = cat2 + num2

features2 = {"categoricals": cat2, "numerics": num1, "myX": X2}
```

[ ]:
```
def select_features(df_train, df_test, df_hype, feat_fam: dict):
    """Returns training dataframes based of feature families.

    Args:
        df_train (pyspark.sql.DataFrame): Unblanced training dataframe
        df_test (pyspark.sql.DataFrame): Unblanced test dataframe
        df_hype (pyspark.sql.DataFrame): Unblanced hyperprameter-tuned training␣
 ↪dataframe
        feat_fam (dict): _description_
```

```
    Returns:
        pyspark.sql.DataFrames: train, test, hyperparameter-tuned, list:␣
↪numerical feature names, list: all feature names
    """
    train = df_train.select(feat_fam["myX"] + myY + ['YEAR']).
↪repartition('ORIGIN').persist()
    test = df_test.select(feat_fam["myX"] + myY + ['YEAR']).
↪repartition('ORIGIN').persist()
    hype_train = df_hype.select(feat_fam["myX"] + myY + ['YEAR']).
↪repartition('ORIGIN').persist()
    categoricals = feat_fam["categoricals"]
    numerics = feat_fam["numerics"]
    myX = feat_fam["myX"]

    return train, test, hype_train, categoricals, numerics, myX
```

```
[ ]: # if RE_TRAIN:
         # choose feature family to test
     train, test, hype_train, categoricals, numerics, myX = select_features(df_full,␣
     ↪df_test, df_hype_param_train, features1)
```

3.6 Data Pipeline

After the feature engineering and selection, we have constructed the data pipeline to transform the training and testing data into a format that the modelling pipeline can consume. The categorical variables has been encoded through OneHotEncoders to transform them into binary vectors to support model training. Then all variables are normalized via MinMaxScaler. The data are normalized to prevent models leaning bias to one specific variables as opposed to others due to the scale. Specifically a MinMaxScaler has been chosen to scale the features to a range of [0,1] as opposed StandardScalers has the features has shown to have none Gaussian distribution during EDA phase, and we would like to preserve the shaping on the features during the training.

Further to data transformation pipeline, we also addressed the imbalance in the dataset. As seen in the EDA, the delayed vs non-delays shows an 8:2 ratio across the data set. The machine learning models may adversely in favor of non-delayed class while it is important to have accurate prediction of delayed flag in this scenario. Thus, training dataset needs to be balanced. We first examine the recent popular Synthetic Minority Oversampling Technique (SMOTE), which oversamples the minor class by synthesizing new examples from the minor class. A fully functioning SMOTE pipeline was built with reference to [6]. However, upon examination, SMOTE methods shows significant performance degradation with larger dataset as it relies on identifying the nearest neighbors of the each sample. With increase in the number of samples and data dimension, SMOTE takes considerately long time to perform.

Upon further literature review, a number of paper shows that SMOTE does not perform so well on high-dimension big data problems, for example, [7], [8], [9]. Specifically, [8] has suggested that random undersampling has outperformed other methods when addressing class imbalance issue. Therefore, we have implemented random undersampling in the training process.

3.6.1 Data Balancing, Feature Transformation, and Scaling

```
[ ]: def prep_data_pipeline(df, y, categoricals, numerics):
         """_summary_

         Args:
             df (pyspark.sql.DataFrme): Unbalanced training dataframe.
             y (list): Prediciton variable.
             categoricals (list): List of categorical features.
             numerics (list): List of numerical features.

         Returns:
             pyspark.Pipeline: Training pipeline with selected features.
         """
         myX = categoricals + numerics

         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+['YEAR'], outputCols =␣
      ↪numerics+['YEAR'])


         # Establish features columns

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

         # Build the stage for the ML pipeline
         model_matrix_stages = list(indexers) + list(ohes) + [imputers] +   \
                               [VectorAssembler(inputCols=featureCols,␣
      ↪outputCol="features"), StringIndexer(inputCol='DELAY_FLAG',␣
      ↪outputCol="label")]

         # Apply MinMaxScaler to create scaledFeatures
         scaler = MinMaxScaler(inputCol="features",
                                 outputCol="scaledFeatures")

         pipeline = Pipeline(stages=model_matrix_stages+[scaler])

         return pipeline

     def transform_data(pos_vectorized, drop_cols):

         keep_cols = [a for a in pos_vectorized.columns if a not in drop_cols]
```

```
    vectorized = pos_vectorized.select(*keep_cols).
 ↪withColumn('label',pos_vectorized['DELAY_FLAG']).drop('DELAY_FLAG')
    return vectorized
```

Prepare Pipeline And Unbalanced Training Datasets

[ ]:
```
RE_TRAIN
```

[ ]:
```
train
```

[ ]:
```
# if RE_TRAIN:
    # vectorising the dataset for features
data_pipeline = prep_data_pipeline(train, myY, categoricals, numerics).
 ↪fit(train).persist()
train = transform_data(data_pipeline.transform(train), myX).persist()
hype_train =  transform_data(data_pipeline.transform(hype_train), myX).persist()
test =  transform_data(data_pipeline.transform(test), myX).persist()
```

Write unbalanced training data to storage.

[ ]:
```
WRITE = True
if WRITE:
    train.write.format("parquet")\
                .mode("overwrite")\
                .save(f"{blob_url}/transformed_train")
    test.write.format("parquet")\
                .mode("overwrite")\
                .save(f"{blob_url}/test")
```

SMOTE: Small Dataset Test

[ ]:
```
def smote(vectorized_sdf, maj_label=0, min_label=1, seed=SEED,␣
 ↪bucketLength=None, threshold=10, input_col='features', multiplier=4, k=2, ):
    '''
    contains logic to perform smote oversampling, given a spark df with 2␣
 ↪classes
    inputs:
    * vectorized_sdf: cat cols are already stringindexed, num cols are␣
 ↪assembled into 'features' vector
      df target col should be 'label'
    * smote_config: config obj containing smote parameters
    output:
    * oversampled_df: spark df after smote oversampling
    '''
    dataInput_min = vectorized_sdf[vectorized_sdf['label'] == min_label]
    dataInput_maj = vectorized_sdf[vectorized_sdf['label'] == maj_label]
```

```
    # LSH, bucketed random projection
    brp = BucketedRandomProjectionLSH(inputCol=input_col,␣
↪outputCol="hashes",seed=int(seed),\
                                      bucketLength=bucketLength)
    # smote only applies on existing minority instances
    model = brp.fit(dataInput_min)
    model.transform(dataInput_min)

    # here distance is calculated from brp's param inputCol
    self_join_w_distance = model.approxSimilarityJoin(dataInput_min,␣
↪dataInput_min, float(threshold), distCol="EuclideanDistance")

    # remove self-comparison (distance 0)
    self_join_w_distance = self_join_w_distance.filter(self_join_w_distance.
↪EuclideanDistance > 0)

    over_original_rows = Window.partitionBy("datasetA").
↪orderBy("EuclideanDistance")

    self_similarity_df = self_join_w_distance.withColumn("r_num", ps.
↪row_number().over(over_original_rows))

    self_similarity_df_selected = self_similarity_df.filter(self_similarity_df.
↪r_num <= int(k))

    over_original_rows_no_order = Window.partitionBy('datasetA')

    # list to store batches of synthetic data
    res = []

    # two udf for vector add and subtract, subtraction include a random factor␣
↪[0,1]
    subtract_vector_udf = ps.udf(lambda arr: random.uniform(0,␣
↪1)*(arr[0]-arr[1]), VectorUDT())
    add_vector_udf = ps.udf(lambda arr: arr[0]+arr[1], VectorUDT())

    # retain original columns
    original_cols = dataInput_min.columns

    for i in range(int(multiplier)):
        print("generating batch %s of synthetic instances"%i)
        # logic to randomly select neighbour: pick the largest random number␣
↪generated row as the neighbour
        df_random_sel = self_similarity_df_selected\
                            .withColumn("rand", ps.rand())\
```

```
                            .withColumn('max_rand', ps.max('rand').
↪over(over_original_rows_no_order))\
                            .where(ps.col('rand') == ps.col('max_rand')).
↪drop(*['max_rand','rand','r_num'])
       # create synthetic feature numerical part
       df_vec_diff = df_random_sel\
           .select('*', subtract_vector_udf(ps.array(f'datasetA.{input_col}',␣
↪f'datasetB.{input_col}')).alias('vec_diff'))
       df_vec_modified = df_vec_diff\
           .select('*', add_vector_udf(ps.array(f'datasetB.{input_col}',␣
↪'vec_diff')).alias(input_col))

       # for categorical cols, either pick original or the neighbour's cat␣
↪values
       for c in original_cols:
           # randomly select neighbour or original data
           col_sub = random.choice(['datasetA','datasetB'])
           val = "{0}.{1}".format(col_sub,c)
           if c != f'{input_col}':
               # do not unpack original numerical features
               df_vec_modified = df_vec_modified.withColumn(c,ps.col(val))

       # this df_vec_modified is the synthetic minority instances,
       df_vec_modified = df_vec_modified.
↪drop(*['datasetA','datasetB','vec_diff','EuclideanDistance'])

       res.append(df_vec_modified)

   dfunion = reduce(DataFrame.union, res)
   dfunion = dfunion.union(dataInput_min.select(dfunion.columns))\
       .sort(ps.rand(seed=seed))\
       .withColumn('row_number', row_number().over(Window.orderBy(ps.
↪lit('A'))))

   dataInput_maj = dataInput_maj.withColumn('row_number', row_number().
↪over(Window.orderBy(ps.lit('A'))))

   # union synthetic instances with original full (both minority and majority)␣
↪df
   oversampled_df = dfunion.union(dataInput_maj.select(dfunion.columns))

   return oversampled_df.sort('row_number').drop(*['row_number'])
```

[ ]:
```
# if RE_TRAIN:
#     smoted_minitest= smote(minitest, maj_label = 0, min_label = 1, seed=SEED,␣
↪bucketLength=5, input_col = 'scaledFeatures', multiplier = 4, k=2).cache()
```

```
#     display(smoted_minitest)
```

Random undersampling

```
[ ]: def random_undersampling(df, ratio, maj_label = 0, min_label = 1, seed =2022, ␣
      ↪label = 'label'):

         if ratio <= 0 or ratio >1:
             raise ValueError('ratio must be between 0 and 1')

         dataInput_min = df[df[label] == min_label]
         dataInput_maj = df[df[label] == maj_label]
         sampled_majority_df = dataInput_maj.sample(False, ratio, seed = seed)
         combined_df = sampled_majority_df.unionAll(dataInput_min)
         return combined_df
```

Build balanced training data and write to storage.

```
[ ]: if RE_TRAIN:
         # balance the training data set and write to blob storage
         balanced_train = random_undersampling(train, 0.25).cache()
         balanced_hype_train = random_undersampling(hype_train, 0.25).cache()

         if WRITE:
             balanced_train.write.format("parquet")\
                         .mode("overwrite")\
                         .save(f"{blob_url}/balanced_train")
             balanced_hype_train.write.format("parquet")\
                         .mode("overwrite")\
                         .save(f"{blob_url}/balanced_hype_train")
```

3.7 Modeling Pipelines

3.7.1 Algorithm choices and loss functions

When it comes to the the modelling choices, given the dimension of the data set, a key concern is the speed performance of the modelling pipeline, especially on the prediction side that we would like to ensure a prediction can be generated in time. Therefore, we did not consider model such k-NearestNeighbors but the ones scale well with dimension. The list of the models we experimented and associated loss functions are listed below:
1. Logistic regression model with a "Log Loss" loss function also known as cross entropy loss.

$$L(w; x, y) = -[ylog(p) + (1 - y)log(1 - p)$$

2. Linear support vector machine model with the "Hinge Loss" loss function to maximize classification margin.

$$\ell(y) = \max(0, 1 - t \cdot y)$$

3. Random forest model with "Gini Impurity" loss function, which indicates the likelihood of new, random data being misclassified if it were given a random class label according to the class

distribution in the dataset.

$$\sum_{k \neq i} p_k = 1 - p_i$$

4. Multilayer perceptron model (the neural network model) was also examined upon stakeholder request, which also uses the log loss function has shown above.

3.7.2 Regularization

We used an elastic net regularization for both linear models.

$$\hat{\beta} \equiv \underset{\beta}{\operatorname{argmin}}(\|y - X\beta\|^2 + \lambda_2\|\beta\|^2 + \lambda_1\|\beta\|_1)$$

3.7.3 Define Metrics

[ ]:
```
def score(model, data):
    predictionAndLabels = model.transform(data).select("prediction", "label").
 ↪rdd
    return predictionAndLabels


def precision(pred):
    metric = MulticlassMetrics(pred)
    precision = metric.precision(1.0)
    return precision

def recall(pred):
    metric = MulticlassMetrics(pred)
    recall = metric.recall(1.0)
    return recall

def f1(pred):
    metric = MulticlassMetrics(pred)
    f1 = metric.fMeasure(1.0)
    return f1
```

3.7.4 Baseline Model Fitting & Evaluation

To establish a baseline, we looked at the Logistic regression.

[ ]:
```
balanced_hype_train = spark.read.parquet(f"{blob_url}/balanced_hype_train")
balanced_train= spark.read.parquet(f"{blob_url}/balanced_train")
df_test = spark.read.parquet(f"{blob_url}/test")
full_train = spark.read.parquet(f"{blob_url}/transformed_train")
```

[ ]:

```
# lr = LogisticRegression(maxIter=10, featuresCol = "scaledFeatures",␣
↪labelCol="label")
# lr_model = lr.fit(balanced_train)

# lr_scores = score(lr_model, df_test)
# print ("LR Test Precision:" + str(precision(lr_scores)))
# print ("LR Test Recall:" + str(recall(lr_scores)))
# print ("LR Test f1:" + str(f1(lr_scores)))
```

3.7.5 Cross Validation and Hyperparameter Tuning

Each model we have chosen has a number of hyperparameters such as regularization that could impact the model performance. Therefore, cross-validation technique has been adopted so that we can fine-tune the hyperparameters. However, given the embedded time-series nature of the flight data, we implemented a blocking cross-validation as opposed to popular k-fold cross validation. In each iteration of the cross-validation, one year of data is taken as the training set and the subsequent year is used as validation set. Specifically for the last year in the training set, a **8:2** split is used for training and validation since there is no 2021 data in the blind test so we cannot cannot use it for validation. The cross-validation will identify the best set of hyperparameters for each model. In terms of the best model, the cross validation will return the model with best hyperparameters trained with the last set of train data to reflect the shifting landscape through time.

Below is a list of hyperparameters tunned for each model. | Model | Parameters |Value | | ——— | ——— |——— | | Logistic Regression | regParam |0.1, 0.01| | Linear Support Vector Classifier | regParam |0.1, 0.01| | Random Forest | maxDepth |5,10 | | |numTrees|50,70| | Neural Network | layers |[input, 100,50,2], [input,50,50,50,2]|

In addition to the hyperparameter tunning, to avoid overtraining, we also implemented early stopping with maximum number of iterations, as all models are trained through an iterative approach.

Custom Crossvalidator

[ ]:
```
# the py script would only work in root dir not in bulb storage.

import os
import sys
import itertools
from multiprocessing.pool import ThreadPool

import numpy as np

from pyspark import keyword_only, since, SparkContext, inheritable_thread_target
from pyspark.ml import Estimator, Transformer, Model
from pyspark.ml.common import inherit_doc, _py2java, _java2py
from pyspark.ml.evaluation import Evaluator
from pyspark.ml.param import Params, Param, TypeConverters
from pyspark.ml.param.shared import HasCollectSubModels, HasParallelism, HasSeed
from pyspark.ml.util import DefaultParamsReader, DefaultParamsWriter,␣
↪MetaAlgorithmReadWrite, \
```

```python
    MLReadable, MLReader, MLWritable, MLWriter, JavaMLReader, JavaMLWriter
from pyspark.ml.wrapper import JavaParams, JavaEstimator, JavaWrapper
from pyspark.sql.functions import col, lit, rand, UserDefinedFunction
from pyspark.sql.types import BooleanType

__all__ = ['ParamGridBuilder', 'CrossValidator', 'CrossValidatorModel',␣
 ↪'TrainValidationSplit',
           'TrainValidationSplitModel']


def _parallelFitTasks(est, train, eva, validation, epm, collectSubModel=False):
    """
    Creates a list of callables which can be called from different threads to␣
 ↪fit and evaluate
    an estimator in parallel. Each callable returns an `(index, metric)` pair.

    Parameters
    ----------
    est : :py:class:`pyspark.ml.baseEstimator`
        he estimator to be fit.
    train : :py:class:`pyspark.sql.DataFrame`
        DataFrame, training data set, used for fitting.
    eva : :py:class:`pyspark.ml.evaluation.Evaluator`
        used to compute `metric`
    validation : :py:class:`pyspark.sql.DataFrame`
        DataFrame, validation data set, used for evaluation.
    epm : :py:class:`collections.abc.Sequence`
        Sequence of ParamMap, params maps to be used during fitting &␣
 ↪evaluation.
    collectSubModel : bool
        Whether to collect sub model.

    Returns
    -------
    tuple
        (int, float, subModel), an index into `epm` and the associated metric␣
 ↪value.
    """
    modelIter = est.fitMultiple(train, epm)

    def singleTask():
        index, model = next(modelIter)
        # TODO: duplicate evaluator to take extra params from input
        #  Note: Supporting tuning params in evaluator need update method
        #  `MetaAlgorithmReadWrite.getAllNestedStages`, make it return
        #  all nested stages and evaluators
        metric = eva.evaluate(model.transform(validation, epm[index]))
```

```
        return index, metric, model if collectSubModel else None

    return [singleTask] * len(epm)


class ParamGridBuilder(object):
    r"""
    Builder for a param grid used in grid search-based model selection.


    .. versionadded:: 1.4.0

    Examples
    --------
    >>> from pyspark.ml.classification import LogisticRegression
    >>> lr = LogisticRegression()
    >>> output = ParamGridBuilder() \
    ...     .baseOn({lr.labelCol: 'l'}) \
    ...     .baseOn([lr.predictionCol, 'p']) \
    ...     .addGrid(lr.regParam, [1.0, 2.0]) \
    ...     .addGrid(lr.maxIter, [1, 5]) \
    ...     .build()
    >>> expected = [
    ...     {lr.regParam: 1.0, lr.maxIter: 1, lr.labelCol: 'l', lr.
↪predictionCol: 'p'},
    ...     {lr.regParam: 2.0, lr.maxIter: 1, lr.labelCol: 'l', lr.
↪predictionCol: 'p'},
    ...     {lr.regParam: 1.0, lr.maxIter: 5, lr.labelCol: 'l', lr.
↪predictionCol: 'p'},
    ...     {lr.regParam: 2.0, lr.maxIter: 5, lr.labelCol: 'l', lr.
↪predictionCol: 'p'}]
    >>> len(output) == len(expected)
    True
    >>> all([m in expected for m in output])
    True
    """

    def __init__(self):
        self._param_grid = {}

    @since("1.4.0")
    def addGrid(self, param, values):
        """
        Sets the given parameters in this grid to fixed values.

        param must be an instance of Param associated with an instance of Params
        (such as Estimator or Transformer).
```

```
        """
        if isinstance(param, Param):
            self._param_grid[param] = values
        else:
            raise TypeError("param must be an instance of Param")

        return self


    @since("1.4.0")
    def baseOn(self, *args):
        """
        Sets the given parameters in this grid to fixed values.
        Accepts either a parameter dictionary or a list of (parameter, value)␣
↪pairs.
        """
        if isinstance(args[0], dict):
            self.baseOn(*args[0].items())
        else:
            for (param, value) in args:
                self.addGrid(param, [value])

        return self


    @since("1.4.0")
    def build(self):
        """
        Builds and returns all combinations of parameters specified
        by the param grid.
        """
        keys = self._param_grid.keys()
        grid_values = self._param_grid.values()

        def to_key_value_pairs(keys, values):
            return [(key, key.typeConverter(value)) for key, value in zip(keys,␣
↪values)]

        return [dict(to_key_value_pairs(keys, prod)) for prod in itertools.
↪product(*grid_values)]



class _ValidatorParams(HasSeed):
    """
    Common params for TrainValidationSplit and CrossValidator.
    """
```

```
    estimator = Param(Params._dummy(), "estimator", "estimator to be␣
↪cross-validated")
    estimatorParamMaps = Param(Params._dummy(), "estimatorParamMaps",␣
↪"estimator param maps")
    evaluator = Param(
        Params._dummy(), "evaluator",
        "evaluator used to select hyper-parameters that maximize the validator␣
↪metric")

    @since("2.0.0")
    def getEstimator(self):
        """
        Gets the value of estimator or its default value.
        """
        return self.getOrDefault(self.estimator)

    @since("2.0.0")
    def getEstimatorParamMaps(self):
        """
        Gets the value of estimatorParamMaps or its default value.
        """
        return self.getOrDefault(self.estimatorParamMaps)

    @since("2.0.0")
    def getEvaluator(self):
        """
        Gets the value of evaluator or its default value.
        """
        return self.getOrDefault(self.evaluator)

    @classmethod
    def _from_java_impl(cls, java_stage):
        """
        Return Python estimator, estimatorParamMaps, and evaluator from a Java␣
↪ValidatorParams.
        """

        # Load information from java_stage to the instance.
        estimator = JavaParams._from_java(java_stage.getEstimator())
        evaluator = JavaParams._from_java(java_stage.getEvaluator())
        if isinstance(estimator, JavaEstimator):
            epms = [estimator._transfer_param_map_from_java(epm)
                    for epm in java_stage.getEstimatorParamMaps()]
        elif MetaAlgorithmReadWrite.isMetaEstimator(estimator):
            # Meta estimator such as Pipeline, OneVsRest
```

```
            epms = _ValidatorSharedReadWrite.
↪meta_estimator_transfer_param_maps_from_java(
                estimator, java_stage.getEstimatorParamMaps())
        else:
            raise ValueError('Unsupported estimator used in tuning: ' +␣
↪str(estimator))

        return estimator, epms, evaluator

    def _to_java_impl(self):
        """
        Return Java estimator, estimatorParamMaps, and evaluator from this␣
↪Python instance.
        """

        gateway = SparkContext._gateway
        cls = SparkContext._jvm.org.apache.spark.ml.param.ParamMap

        estimator = self.getEstimator()
        if isinstance(estimator, JavaEstimator):
            java_epms = gateway.new_array(cls, len(self.
↪getEstimatorParamMaps()))
            for idx, epm in enumerate(self.getEstimatorParamMaps()):
                java_epms[idx] = self.getEstimator().
↪_transfer_param_map_to_java(epm)
        elif MetaAlgorithmReadWrite.isMetaEstimator(estimator):
            # Meta estimator such as Pipeline, OneVsRest
            java_epms = _ValidatorSharedReadWrite.
↪meta_estimator_transfer_param_maps_to_java(
                estimator, self.getEstimatorParamMaps())
        else:
            raise ValueError('Unsupported estimator used in tuning: ' +␣
↪str(estimator))

        java_estimator = self.getEstimator()._to_java()
        java_evaluator = self.getEvaluator()._to_java()
        return java_estimator, java_epms, java_evaluator


class _ValidatorSharedReadWrite:

    @staticmethod
    def meta_estimator_transfer_param_maps_to_java(pyEstimator, pyParamMaps):
        pyStages = MetaAlgorithmReadWrite.getAllNestedStages(pyEstimator)
        stagePairs = list(map(lambda stage: (stage, stage._to_java()),␣
↪pyStages))
```

```
        sc = SparkContext._active_spark_context

        paramMapCls = SparkContext._jvm.org.apache.spark.ml.param.ParamMap
        javaParamMaps = SparkContext._gateway.new_array(paramMapCls,␣
↪len(pyParamMaps))

        for idx, pyParamMap in enumerate(pyParamMaps):
            javaParamMap = JavaWrapper._new_java_obj("org.apache.spark.ml.param.
↪ParamMap")
            for pyParam, pyValue in pyParamMap.items():
                javaParam = None
                for pyStage, javaStage in stagePairs:
                    if pyStage._testOwnParam(pyParam.parent, pyParam.name):
                        javaParam = javaStage.getParam(pyParam.name)
                        break
                if javaParam is None:
                    raise ValueError('Resolve param in estimatorParamMaps␣
↪failed: ' + str(pyParam))
                if isinstance(pyValue, Params) and hasattr(pyValue, "_to_java"):
                    javaValue = pyValue._to_java()
                else:
                    javaValue = _py2java(sc, pyValue)
                pair = javaParam.w(javaValue)
                javaParamMap.put([pair])
            javaParamMaps[idx] = javaParamMap
        return javaParamMaps

    @staticmethod
    def meta_estimator_transfer_param_maps_from_java(pyEstimator,␣
↪javaParamMaps):
        pyStages = MetaAlgorithmReadWrite.getAllNestedStages(pyEstimator)
        stagePairs = list(map(lambda stage: (stage, stage._to_java()),␣
↪pyStages))
        sc = SparkContext._active_spark_context
        pyParamMaps = []
        for javaParamMap in javaParamMaps:
            pyParamMap = dict()
            for javaPair in javaParamMap.toList():
                javaParam = javaPair.param()
                pyParam = None
                for pyStage, javaStage in stagePairs:
                    if pyStage._testOwnParam(javaParam.parent(), javaParam.
↪name()):
                        pyParam = pyStage.getParam(javaParam.name())
                if pyParam is None:
```

```
                    raise ValueError('Resolve param in estimatorParamMaps␣
↪failed: ' +
                                     javaParam.parent() + '.' + javaParam.
↪name())
                javaValue = javaPair.value()
                if sc._jvm.Class.forName("org.apache.spark.ml.util.
↪DefaultParamsWritable") \
                        .isInstance(javaValue):
                    pyValue = JavaParams._from_java(javaValue)
                else:
                    pyValue = _java2py(sc, javaValue)
                pyParamMap[pyParam] = pyValue
            pyParamMaps.append(pyParamMap)
        return pyParamMaps

    @staticmethod
    def is_java_convertible(instance):
        allNestedStages = MetaAlgorithmReadWrite.getAllNestedStages(instance.
↪getEstimator())
        evaluator_convertible = isinstance(instance.getEvaluator(), JavaParams)
        estimator_convertible = all(map(lambda stage: hasattr(stage,␣
↪'_to_java'), allNestedStages))
        return estimator_convertible and evaluator_convertible

    @staticmethod
    def saveImpl(path, instance, sc, extraMetadata=None):
        numParamsNotJson = 0
        jsonEstimatorParamMaps = []
        for paramMap in instance.getEstimatorParamMaps():
            jsonParamMap = []
            for p, v in paramMap.items():
                jsonParam = {'parent': p.parent, 'name': p.name}
                if (isinstance(v, Estimator) and not MetaAlgorithmReadWrite.
↪isMetaEstimator(v)) \
                        or isinstance(v, Transformer) or isinstance(v,␣
↪Evaluator):
                    relative_path = f'epm_{p.name}{numParamsNotJson}'
                    param_path = os.path.join(path, relative_path)
                    numParamsNotJson += 1
                    v.save(param_path)
                    jsonParam['value'] = relative_path
                    jsonParam['isJson'] = False
                elif isinstance(v, MLWritable):
                    raise RuntimeError(
                        "ValidatorSharedReadWrite.saveImpl does not handle␣
↪parameters of type: "
```

```
                        "MLWritable that are not Estimaor/Evaluator/
↪Transformer, and if parameter "
                        "is estimator, it cannot be meta estimator such as␣
↪Validator or OneVsRest")
                else:
                    jsonParam['value'] = v
                    jsonParam['isJson'] = True
                jsonParamMap.append(jsonParam)
            jsonEstimatorParamMaps.append(jsonParamMap)

        skipParams = ['estimator', 'evaluator', 'estimatorParamMaps']
        jsonParams = DefaultParamsWriter.extractJsonParams(instance, skipParams)
        jsonParams['estimatorParamMaps'] = jsonEstimatorParamMaps

        DefaultParamsWriter.saveMetadata(instance, path, sc, extraMetadata,␣
↪jsonParams)
        evaluatorPath = os.path.join(path, 'evaluator')
        instance.getEvaluator().save(evaluatorPath)
        estimatorPath = os.path.join(path, 'estimator')
        instance.getEstimator().save(estimatorPath)

    @staticmethod
    def load(path, sc, metadata):
        evaluatorPath = os.path.join(path, 'evaluator')
        evaluator = DefaultParamsReader.loadParamsInstance(evaluatorPath, sc)
        estimatorPath = os.path.join(path, 'estimator')
        estimator = DefaultParamsReader.loadParamsInstance(estimatorPath, sc)

        uidToParams = MetaAlgorithmReadWrite.getUidMap(estimator)
        uidToParams[evaluator.uid] = evaluator

        jsonEstimatorParamMaps = metadata['paramMap']['estimatorParamMaps']

        estimatorParamMaps = []
        for jsonParamMap in jsonEstimatorParamMaps:
            paramMap = {}
            for jsonParam in jsonParamMap:
                est = uidToParams[jsonParam['parent']]
                param = getattr(est, jsonParam['name'])
                if 'isJson' not in jsonParam or ('isJson' in jsonParam and␣
↪jsonParam['isJson']):
                    value = jsonParam['value']
                else:
                    relativePath = jsonParam['value']
                    valueSavedPath = os.path.join(path, relativePath)
                    value = DefaultParamsReader.
↪loadParamsInstance(valueSavedPath, sc)
```

```
                paramMap[param] = value
            estimatorParamMaps.append(paramMap)

        return metadata, estimator, evaluator, estimatorParamMaps

    @staticmethod
    def validateParams(instance):
        estiamtor = instance.getEstimator()
        evaluator = instance.getEvaluator()
        uidMap = MetaAlgorithmReadWrite.getUidMap(estiamtor)

        for elem in [evaluator] + list(uidMap.values()):
            if not isinstance(elem, MLWritable):
                raise ValueError(f'Validator write will fail because it␣
↪contains {elem.uid} '
                                 f'which is not writable.')

        estimatorParamMaps = instance.getEstimatorParamMaps()
        paramErr = 'Validator save requires all Params in estimatorParamMaps to␣
↪apply to ' \
                   f'its Estimator, An extraneous Param was found: '
        for paramMap in estimatorParamMaps:
            for param in paramMap:
                if param.parent not in uidMap:
                    raise ValueError(paramErr + repr(param))

    @staticmethod
    def getValidatorModelWriterPersistSubModelsParam(writer):
        if 'persistsubmodels' in writer.optionMap:
            persistSubModelsParam = writer.optionMap['persistsubmodels'].lower()
            if persistSubModelsParam == 'true':
                return True
            elif persistSubModelsParam == 'false':
                return False
            else:
                raise ValueError(
                    f'persistSubModels option value {persistSubModelsParam} is␣
↪invalid, '
                    f"the possible values are True, 'True' or False, 'False'")
        else:
            return writer.instance.subModels is not None


_save_with_persist_submodels_no_submodels_found_err = \
    'When persisting tuning models, you can only set persistSubModels to true␣
↪if the tuning ' \
```

```python
    'was done with collectSubModels set to true. To save the sub-models, try␣
↪rerunning fitting ' \
    'with collectSubModels set to true.'


@inherit_doc
class CrossValidatorReader(MLReader):

    def __init__(self, cls):
        super(CrossValidatorReader, self).__init__()
        self.cls = cls

    def load(self, path):
        metadata = DefaultParamsReader.loadMetadata(path, self.sc)
        if not DefaultParamsReader.isPythonParamsInstance(metadata):
            return JavaMLReader(self.cls).load(path)
        else:
            metadata, estimator, evaluator, estimatorParamMaps = \
                _ValidatorSharedReadWrite.load(path, self.sc, metadata)
            cv = CrossValidator(estimator=estimator,
                                estimatorParamMaps=estimatorParamMaps,
                                evaluator=evaluator)
            cv = cv._resetUid(metadata['uid'])
            DefaultParamsReader.getAndSetParams(cv, metadata,␣
↪skipParams=['estimatorParamMaps'])
            return cv


@inherit_doc
class CrossValidatorWriter(MLWriter):

    def __init__(self, instance):
        super(CrossValidatorWriter, self).__init__()
        self.instance = instance

    def saveImpl(self, path):
        _ValidatorSharedReadWrite.validateParams(self.instance)
        _ValidatorSharedReadWrite.saveImpl(path, self.instance, self.sc)


@inherit_doc
class CrossValidatorModelReader(MLReader):

    def __init__(self, cls):
        super(CrossValidatorModelReader, self).__init__()
        self.cls = cls
```

```python
    def load(self, path):
        metadata = DefaultParamsReader.loadMetadata(path, self.sc)
        if not DefaultParamsReader.isPythonParamsInstance(metadata):
            return JavaMLReader(self.cls).load(path)
        else:
            metadata, estimator, evaluator, estimatorParamMaps = \
                _ValidatorSharedReadWrite.load(path, self.sc, metadata)
            numFolds = metadata['paramMap']['numFolds']
            bestModelPath = os.path.join(path, 'bestModel')
            bestModel = DefaultParamsReader.loadParamsInstance(bestModelPath,␣
↪self.sc)
            avgMetrics = metadata['avgMetrics']
            persistSubModels = ('persistSubModels' in metadata) and␣
↪metadata['persistSubModels']

            if persistSubModels:
                subModels = [[None] * len(estimatorParamMaps)] * numFolds
                for splitIndex in range(numFolds):
                    for paramIndex in range(len(estimatorParamMaps)):
                        modelPath = os.path.join(
                            path, 'subModels', f'fold{splitIndex}',␣
↪f'{paramIndex}')
                        subModels[splitIndex][paramIndex] = \
                            DefaultParamsReader.loadParamsInstance(modelPath,␣
↪self.sc)
            else:
                subModels = None

            cvModel = CrossValidatorModel(bestModel, avgMetrics=avgMetrics,␣
↪subModels=subModels)
            cvModel = cvModel._resetUid(metadata['uid'])
            cvModel.set(cvModel.estimator, estimator)
            cvModel.set(cvModel.estimatorParamMaps, estimatorParamMaps)
            cvModel.set(cvModel.evaluator, evaluator)
            DefaultParamsReader.getAndSetParams(
                cvModel, metadata, skipParams=['estimatorParamMaps'])
            return cvModel


@inherit_doc
class CrossValidatorModelWriter(MLWriter):

    def __init__(self, instance):
        super(CrossValidatorModelWriter, self).__init__()
        self.instance = instance

    def saveImpl(self, path):
```

```
        _ValidatorSharedReadWrite.validateParams(self.instance)
        instance = self.instance
        persistSubModels = _ValidatorSharedReadWrite \
            .getValidatorModelWriterPersistSubModelsParam(self)
        extraMetadata = {'avgMetrics': instance.avgMetrics,
                         'persistSubModels': persistSubModels}
        _ValidatorSharedReadWrite.saveImpl(path, instance, self.sc,␣
↪extraMetadata=extraMetadata)
        bestModelPath = os.path.join(path, 'bestModel')
        instance.bestModel.save(bestModelPath)
        if persistSubModels:
            if instance.subModels is None:
                raise␣
↪ValueError(_save_with_persist_submodels_no_submodels_found_err)
            subModelsPath = os.path.join(path, 'subModels')
            for splitIndex in range(instance.getNumFolds()):
                splitPath = os.path.join(subModelsPath, f'fold{splitIndex}')
                for paramIndex in range(len(instance.getEstimatorParamMaps())):
                    modelPath = os.path.join(splitPath, f'{paramIndex}')
                    instance.subModels[splitIndex][paramIndex].save(modelPath)


class _CrossValidatorParams(_ValidatorParams):
    """
    Params for :py:class:`CrossValidator` and :py:class:`CrossValidatorModel`.

    .. versionadded:: 3.0.0
    """

    numFolds = Param(Params._dummy(), "numFolds", "number of folds for cross␣
↪validation",
                     typeConverter=TypeConverters.toInt)

    foldCol = Param(Params._dummy(), "foldCol", "Param for the column name of␣
↪user " +
                    "specified fold number. Once this is specified, :py:class:
↪`CrossValidator` " +
                    "won't do random k-fold split. Note that this column should␣
↪be integer type " +
                    "with range [0, numFolds) and Spark will throw exception on␣
↪out-of-range " +
                    "fold numbers.", typeConverter=TypeConverters.toString)

    def __init__(self, *args):
        super(_CrossValidatorParams, self).__init__(*args)
        self._setDefault(numFolds=3, foldCol="")
```

```
    @since("1.4.0")
    def getNumFolds(self):
        """
        Gets the value of numFolds or its default value.
        """
        return self.getOrDefault(self.numFolds)

    @since("3.1.0")
    def getFoldCol(self):
        """
        Gets the value of foldCol or its default value.
        """
        return self.getOrDefault(self.foldCol)


class CustomCrossValidator(Estimator, _CrossValidatorParams, HasParallelism,␣
↪HasCollectSubModels,
                         MLReadable, MLWritable):
    """
    Modifies CrossValidator allowing custom train and test dataset to be passed␣
↪into the function
    Bypass generation of train/test via numFolds
    instead train and test set is user define
    """

    splitWord = Param(Params._dummy(), "splitWord", "Tuple to split train and␣
↪test set e.g. ('train', 'test')",
                      typeConverter=TypeConverters.toListString)
    cvCol = Param(Params._dummy(), "cvCol", "Column name to filter train and␣
↪test list",
                      typeConverter=TypeConverters.toString)

    @keyword_only
    def __init__(self, *, estimator=None, estimatorParamMaps=None,␣
↪evaluator=None, seed=None, parallelism=1, collectSubModels=False,
                 splitWord = ('train', 'test'), cvCol = 'cv'):
        """
        __init__(self, \\*, estimator=None, estimatorParamMaps=None,␣
↪evaluator=None, numFolds=3,\
                 seed=None, parallelism=1, collectSubModels=False, foldCol="")
        """
        super(CustomCrossValidator, self).__init__()
        self._setDefault(parallelism=1)
        kwargs = self._input_kwargs
        self._set(**kwargs)
```

```
    @keyword_only
    @since("1.4.0")
    def setParams(self, *, estimator=None, estimatorParamMaps=None,␣
↪evaluator=None, seed=None, parallelism=1, collectSubModels=False,
                  splitWord = ('train', 'test'), cvCol = 'cv'):
        """
        Sets params for cross validator.
        """
        kwargs = self._input_kwargs
        return self._set(**kwargs)


    @since("2.0.0")
    def setEstimator(self, value):
        """
        Sets the value of :py:attr:`estimator`.
        """
        return self._set(estimator=value)


    @since("2.0.0")
    def setEstimatorParamMaps(self, value):
        """
        Sets the value of :py:attr:`estimatorParamMaps`.
        """
        return self._set(estimatorParamMaps=value)


    @since("2.0.0")
    def setEvaluator(self, value):
        """
        Sets the value of :py:attr:`evaluator`.
        """
        return self._set(evaluator=value)


    @since("1.4.0")
    def setNumFolds(self, value):
        """
        Sets the value of :py:attr:`numFolds`.
        """
        return self._set(numFolds=value)


    @since("3.1.0")
    def setFoldCol(self, value):
```

```
        """
        Sets the value of :py:attr:`foldCol`.
        """
        return self._set(foldCol=value)


    def setSeed(self, value):
        """
        Sets the value of :py:attr:`seed`.
        """
        return self._set(seed=value)


    def setParallelism(self, value):
        """
        Sets the value of :py:attr:`parallelism`.
        """
        return self._set(parallelism=value)


    def setCollectSubModels(self, value):
        """
        Sets the value of :py:attr:`collectSubModels`.
        """
        return self._set(collectSubModels=value)


    def _fit(self, dataset):
        est = self.getOrDefault(self.estimator)
        epm = self.getOrDefault(self.estimatorParamMaps)
        numModels = len(epm)
        eva = self.getOrDefault(self.evaluator)
        nFolds = len(dataset)
        seed = self.getOrDefault(self.seed)
        metrics = [0.0] * numModels
        matrix_metrics = [[0 for x in range(nFolds)] for y in range(len(epm))]

        pool = ThreadPool(processes=min(self.getParallelism(), numModels))

        for i in range(nFolds):
            validation = dataset[list(dataset.keys())[i]].filter(col(self.
↪getOrDefault(self.cvCol))==(self.getOrDefault(self.splitWord))[0]).cache()
            train = dataset[list(dataset.keys())[i]].filter(col(self.
↪getOrDefault(self.cvCol))==(self.getOrDefault(self.splitWord))[1]).cache()

            print('fold {} start...'.format(i+1))
            tasks = _parallelFitTasks(est, train, eva, validation, epm)
```

```
            for j, metric, subModel in pool.imap_unordered(lambda f: f(),␣
↪tasks):
                #print(j, metric)
                matrix_metrics[j][i] = metric
                metrics[j] += (metric / nFolds)
            print('fold {} end'.format(i+1))
            #print(metrics)
            validation.unpersist()
            train.unpersist()

        if eva.isLargerBetter():
            bestIndex = np.argmax(metrics)
        else:
            bestIndex = np.argmin(metrics)

#          for i in range(len(metrics)):
#              print(epm[i], 'Detailed Score {}'.format(matrix_metrics[i]), 'Avg␣
↪Score {}'.format(metrics[i]))

#          print('Best Model: ', epm[bestIndex], 'Detailed Score {}'.
↪format(matrix_metrics[bestIndex]),
#                'Avg Score {}'.format(metrics[bestIndex]))

        ### Do not bother to train on full dataset, just the latest train␣
↪supplied
        # bestModel = est.fit(dataset, epm[bestIndex])
        train = dataset[list(dataset.keys())[-1]].filter(col(self.
↪getOrDefault(self.cvCol))==(self.getOrDefault(self.splitWord))[1]).cache()
        bestModel = est.fit(train, epm[bestIndex])
        return self._copyValues(CrossValidatorModel(bestModel, metrics))

    def copy(self, extra=None):
        """
        Creates a copy of this instance with a randomly generated uid
        and some extra params. This copies creates a deep copy of
        the embedded paramMap, and copies the embedded and extra parameters␣
↪over.


        .. versionadded:: 1.4.0

        Parameters
        ----------
        extra : dict, optional
            Extra parameters to copy to the new instance
```

```
        Returns
        -------
        :py:class:`CrossValidator`
            Copy of this instance
        """
        if extra is None:
            extra = dict()
        newCV = Params.copy(self, extra)
        if self.isSet(self.estimator):
            newCV.setEstimator(self.getEstimator().copy(extra))
        # estimatorParamMaps remain the same
        if self.isSet(self.evaluator):
            newCV.setEvaluator(self.getEvaluator().copy(extra))
        return newCV


    @since("2.3.0")
    def write(self):
        """Returns an MLWriter instance for this ML instance."""
        if _ValidatorSharedReadWrite.is_java_convertible(self):
            return JavaMLWriter(self)
        return CrossValidatorWriter(self)


    @classmethod
    @since("2.3.0")
    def read(cls):
        """Returns an MLReader instance for this class."""
        return CrossValidatorReader(cls)


    @classmethod
    def _from_java(cls, java_stage):
        """
        Given a Java CrossValidator, create and return a Python wrapper of it.
        Used for ML persistence.
        """

        estimator, epms, evaluator = super(CrossValidator, cls).
↪_from_java_impl(java_stage)
        numFolds = java_stage.getNumFolds()
        seed = java_stage.getSeed()
        parallelism = java_stage.getParallelism()
        collectSubModels = java_stage.getCollectSubModels()
        foldCol = java_stage.getFoldCol()
        # Create a new instance of this stage.
```

```
        py_stage = cls(estimator=estimator, estimatorParamMaps=epms,␣
↪evaluator=evaluator,
                       numFolds=numFolds, seed=seed, parallelism=parallelism,
                       collectSubModels=collectSubModels, foldCol=foldCol)
        py_stage._resetUid(java_stage.uid())
        return py_stage

    def _to_java(self):
        """
        Transfer this instance to a Java CrossValidator. Used for ML␣
↪persistence.

        Returns
        -------
        py4j.java_gateway.JavaObject
            Java object equivalent to this instance.
        """

        estimator, epms, evaluator = super(CrossValidator, self)._to_java_impl()

        _java_obj = JavaParams._new_java_obj("org.apache.spark.ml.tuning.
↪CrossValidator", self.uid)
        _java_obj.setEstimatorParamMaps(epms)
        _java_obj.setEvaluator(evaluator)
        _java_obj.setEstimator(estimator)
        _java_obj.setSeed(self.getSeed())
        _java_obj.setNumFolds(self.getNumFolds())
        _java_obj.setParallelism(self.getParallelism())
        _java_obj.setCollectSubModels(self.getCollectSubModels())
        _java_obj.setFoldCol(self.getFoldCol())

        return _java_obj



class CrossValidatorModel(Model, _CrossValidatorParams, MLReadable, MLWritable):
    """

    CrossValidatorModel contains the model with the highest average␣
↪cross-validation
    metric across folds and uses this model to transform input data.␣
↪CrossValidatorModel
    also tracks the metrics for each param map evaluated.

    .. versionadded:: 1.4.0
    """
```

```
    def __init__(self, bestModel, avgMetrics=None, subModels=None):
        super(CrossValidatorModel, self).__init__()
        #: best model from cross validation
        self.bestModel = bestModel
        #: Average cross-validation metrics for each paramMap in
        #: CrossValidator.estimatorParamMaps, in the corresponding order.
        self.avgMetrics = avgMetrics or []
        #: sub model list from cross validation
        self.subModels = subModels

    def _transform(self, dataset):
        return self.bestModel.transform(dataset)

    def copy(self, extra=None):
        """
        Creates a copy of this instance with a randomly generated uid
        and some extra params. This copies the underlying bestModel,
        creates a deep copy of the embedded paramMap, and
        copies the embedded and extra parameters over.
        It does not copy the extra Params into the subModels.

        .. versionadded:: 1.4.0

        Parameters
        ----------
        extra : dict, optional
            Extra parameters to copy to the new instance

        Returns
        -------
        :py:class:`CrossValidatorModel`
            Copy of this instance
        """
        if extra is None:
            extra = dict()
        bestModel = self.bestModel.copy(extra)
        avgMetrics = list(self.avgMetrics)
        subModels = [
            [sub_model.copy() for sub_model in fold_sub_models]
            for fold_sub_models in self.subModels
        ]
        return self._copyValues(CrossValidatorModel(bestModel, avgMetrics,␣
↪subModels), extra=extra)


    @since("2.3.0")
    def write(self):
```

```
        """Returns an MLWriter instance for this ML instance."""
        if _ValidatorSharedReadWrite.is_java_convertible(self):
            return JavaMLWriter(self)
        return CrossValidatorModelWriter(self)


    @classmethod
    @since("2.3.0")
    def read(cls):
        """Returns an MLReader instance for this class."""
        return CrossValidatorModelReader(cls)


    @classmethod
    def _from_java(cls, java_stage):
        """
        Given a Java CrossValidatorModel, create and return a Python wrapper of␣
↪it.
        Used for ML persistence.
        """
        sc = SparkContext._active_spark_context
        bestModel = JavaParams._from_java(java_stage.bestModel())
        avgMetrics = _java2py(sc, java_stage.avgMetrics())
        estimator, epms, evaluator = super(CrossValidatorModel, cls).
↪_from_java_impl(java_stage)

        py_stage = cls(bestModel=bestModel, avgMetrics=avgMetrics)
        params = {
            "evaluator": evaluator,
            "estimator": estimator,
            "estimatorParamMaps": epms,
            "numFolds": java_stage.getNumFolds(),
            "foldCol": java_stage.getFoldCol(),
            "seed": java_stage.getSeed(),
        }
        for param_name, param_val in params.items():
            py_stage = py_stage._set(**{param_name: param_val})

        if java_stage.hasSubModels():
            py_stage.subModels = [[JavaParams._from_java(sub_model)
                                   for sub_model in fold_sub_models]
                                  for fold_sub_models in java_stage.subModels()]

        py_stage._resetUid(java_stage.uid())
        return py_stage

    def _to_java(self):
```

```
        """
        Transfer this instance to a Java CrossValidatorModel. Used for ML␣
↪persistence.

        Returns
        -------
        py4j.java_gateway.JavaObject
            Java object equivalent to this instance.
        """

        sc = SparkContext._active_spark_context
        _java_obj = JavaParams._new_java_obj("org.apache.spark.ml.tuning.
↪CrossValidatorModel",
                                             self.uid,
                                             self.bestModel._to_java(),
                                             _py2java(sc, self.avgMetrics))
        estimator, epms, evaluator = super(CrossValidatorModel, self).
↪_to_java_impl()

        params = {
            "evaluator": evaluator,
            "estimator": estimator,
            "estimatorParamMaps": epms,
            "numFolds": self.getNumFolds(),
            "foldCol": self.getFoldCol(),
            "seed": self.getSeed(),
        }
        for param_name, param_val in params.items():
            java_param = _java_obj.getParam(param_name)
            pair = java_param.w(param_val)
            _java_obj.set(pair)

        if self.subModels is not None:
            java_sub_models = [[sub_model._to_java() for sub_model in␣
↪fold_sub_models]
                               for fold_sub_models in self.subModels]
            _java_obj.setSubModels(java_sub_models)
        return _java_obj
```

Load balanced training data.

[ ]:
```
balanced_hype_train = spark.read.parquet(f"{blob_url}/balanced_hype_train")
balanced_train= spark.read.parquet(f"{blob_url}/balanced_train")
```

[ ]:
```
def cross_val_split(df):
    d_df = {}
```

```
    for i in range(2015, 2020, 1):
        d_df[f'split_{i}'] = df.filter( (df.YEAR >= i)  &  (df.YEAR <= i+1) )\
                   .withColumn('cv', ps.when(df.YEAR == i, 'train').
 ↪otherwise('test')).persist()
    d_df[f'split_2020'] = df.filter( (df.YEAR == 2020)  )\
                   .withColumn('cv', ps.when(ps.rand() > 0.8, 'train').
 ↪otherwise('test')).persist()
    return d_df
d_train = cross_val_split(balanced_train)
d_hype_train = cross_val_split(balanced_hype_train)
```

[ ]:
```
class RunCrossVal():

    def __init__(self, model, param_map, train, full, test):
        self.model = model
        self.model.setFeaturesCol("scaledFeatures").setLabelCol("label")
        self.full_model = self.model.copy()
        self.param_map = param_map
        self.best_model = None
        self.best_param = None
        self.train = train
        self.test = test
        self.full = full
        self.train_result = {}
        self.test_result = {}
        self.train_time = None
        self.test_time = None

    @property
    def param_map(self):
        return self._param_map

    @param_map.setter
    def param_map(self, param_dict):
        self._param_map = param_dict
        self.paramGrid = ParamGridBuilder()
        for key, value in self._param_map.items():
            self.paramGrid.addGrid(getattr(self.model, key), value)
        self.paramGrid = self.paramGrid.build()

    def run_crossval(self):
        start_time = time.time()
        crossval = CustomCrossValidator(estimator=self.model,
                      estimatorParamMaps=self.paramGrid,
                      evaluator=BinaryClassificationEvaluator(),
                      splitWord = ('train', 'test'), cvCol = 'cv', parallelism=4)
        cvModel = crossval.fit(self.train)
```

```
        self.best_model = cvModel.bestModel
        self.best_param = cvModel.getEstimatorParamMaps()[np.argmax(cvModel.
↪avgMetrics)]
        self.train_time = time.time()-start_time

    def test_eval(self):
        self.test_result = {}
        start_time = time.time()
        test_pred  = score(self.best_model, self.test).cache()
        self.test_time = time.time()-start_time
        self.test_result['test_precision'] = precision(test_pred)
        self.test_result['test_recall'] = recall(test_pred)
        self.test_result['test_f1'] = f1(test_pred)
        return self.test_result

    def train_eval(self):
        self.train_result = {}
        train_pred  = score(self.best_model, self.full).cache()
        self.train_result['train_precision'] = precision(train_pred)
        self.train_result['train_recall'] = recall(train_pred)
        self.train_result['train_f1'] = f1(train_pred)
        return self.train_result

    def run_all(self):
        self.run_crossval()
        result = self.test_eval()
        result.update(self.train_eval())
        return result
```

[ ]:

```
expriments = {
    'lr': {
        'model': LogisticRegression(maxIter=10),
        'param_map': {
            'regParam': [0.1, 0.01]
        }
    },
    'lsvc': {
        'model': LinearSVC(maxIter=10),
        'param_map': {
            'regParam': [0.1, 0.01]
        }
    },
    'rf':{
        'model': RandomForestClassifier(seed = SEED),
        'param_map': {
            'maxDepth': [5, 10],
            'numTrees': [50, 70]
```

```
        }
    },
    'nn':{
        'model': MultilayerPerceptronClassifier(maxIter=10, blockSize=128,␣
 ↪seed=SEED),
        'param_map': {
            'layers': [[18, 100, 50, 2], [18,50,50,50,2]]
        }
     }
}

def run_expriments(expriments, train, full, test):
    results = {}
    for name, settings in expriments.items():
        results[name] = {}
        results[name]['cv'] = RunCrossVal(settings['model'],␣
 ↪settings['param_map'], train, full, test)
        results[name]['score'] = results[name]['cv'].run_all()
        results[name]['train_time'] = results[name]['cv'].train_time
        results[name]['test_time'] = results[name]['cv'].test_time

    return results

def construct_result_table(expriment_result):
    payloads = []
    for name, cv in expriment_result.items():
        print(cv['cv'].best_param)
        payload = {'model': name}
        payload.update(cv['score'])
        payload['train_time'] = cv['train_time']
        payload['test_time'] = cv['test_time']
        payloads.append(payload)
    return pd.DataFrame(payloads)


result = run_expriments(expriments, d_train, full_train, df_test)
result_df = construct_result_table(result)
result_df
```

3.8 Ensemble Voting Classifier

[ ]:
```
def ensemble_voting():
    logreg_clf = LogisticRegression(maxIter=10, regParam = 0.01)
    rf_clf = RandomForestClassifier(maxDepth=10, numTrees=50)
    #nn_clf = MultilayerPerceptronClassifier(maxIter=10, layers=[19, 100, 50,␣
 ↪2], blockSize=128, seed=1234, featuresCol =  "scaledFeatures")
```

```
    lsvc_clf = LinearSVC(maxIter=10, regParam = 0.01)

    voting_clf = VotingClassifier(estimators=[('LSVC', lsvc_clf),␣
 ↪('RandForrest', rf_clf), ('LogReg', logreg_clf)], voting='hard')
    x = np.array(balanced_train.select("scaledFeatures").collect())
    y = np.array(balanced_train.select("label").collect())

    return voting_clf.fit(x, y)
```

[ ]:
```
if NOT_IMPLEMENTED:
    voting_score = score(ensemble_voting(), df_test)
    print ("Voting Test Precision:" + str(precision(voting_score)))
    print ("Voting Test Recall:" + str(recall(voting_score)))
    print ("Voting Test f1:" + str(f1(voting_score)))
```

3.9 End-to-End Pipeline

[ ]:
```

```