databricks
(https://databricks.com)

Week11_Pyspark (Python)

Import notebook

1

```
# List only datasets larger than 1 GB
def list_large_datasets(path="/databricks-datasets", max_depth=2, curr
    if current_depth >= max_depth:
        return
    try:
        files = dbutils.fs.ls(path)
        for f in files:
            if f.isDir():
                # Compute folder size
                try:
                    sub_files = dbutils.fs.ls(f.path)
                    total_size = sum(sf.size for sf in sub_files)
                    size_gb = total_size / (1024 ** 3)
                    if size_gb > 1:
                        print(f"{f.path}  -->  {size_gb:.2f} GB")
                    # Recurse deeper
                    list_large_datasets(f.path, max_depth, current_dep
                except Exception:
                    pass
    except Exception as e:
        print(f"Error accessing {path}: {e}")

# Run it
list_large_datasets()
```

```
dbfs:/databricks-datasets/COVID/CORD-19/  -->  7.00 GB
dbfs:/databricks-datasets/airlines/  -->  120.06 GB
dbfs:/databricks-datasets/asa/airlines/  -->  11.20 GB
dbfs:/databricks-datasets/genomics/grch37/  -->  1.65 GB
dbfs:/databricks-datasets/genomics/grch37_merged_vep_96/  -->  13.28 GB
dbfs:/databricks-datasets/genomics/grch37_refseq_vep_96/  -->  10.94 GB
dbfs:/databricks-datasets/genomics/grch37_star/  -->  27.85 GB
dbfs:/databricks-datasets/genomics/grch37_vep/  -->  13.28 GB
dbfs:/databricks-datasets/genomics/grch37_vep_96/  -->  11.50 GB
dbfs:/databricks-datasets/genomics/grch38/  -->  1.57 GB
dbfs:/databricks-datasets/genomics/grch38_merged_vep_96/  -->  13.91 GB
```

```
dbfs:/databricks-datasets/genomics/grch38_refseq_vep_96/  -->  11.17 GB
dbfs:/databricks-datasets/genomics/grch38_star/  -->  33.43 GB
dbfs:/databricks-datasets/genomics/grch38_vep/  -->  14.67 GB
dbfs:/databricks-datasets/genomics/grch38_vep_96/  -->  11.88 GB
dbfs:/databricks-datasets/learning-spark-v2/sf-fire/  -->  1.07 GB
dbfs:/databricks-datasets/med-images/camelyon16/  -->  109.81 GB
dbfs:/databricks-datasets/sai-summit-2019-sf/  -->  1.06 GB
dbfs:/databricks-datasets/timeseries/Fires/  -->  1.76 GB
dbfs:/databricks-datasets/wiki/  -->  4.41 GB
```

;

```
# Simple Databricks cell: get dataset size and shape

path = 'dbfs:/databricks-datasets/timeseries/Fires/'

df = spark.read.option("header", True).csv(path)
rows = df.count()
cols = len(df.columns)

# get total size in GB
size_bytes = sum(f.size for f in dbutils.fs.ls(path))
size_gb = size_bytes / (1024 ** 3)

print(f"Rows: {rows}")
print(f"Columns: {cols}")
print(f"Size: {size_gb:.3f} GB")
```

df: pyspark.sql.connect.dataframe.DataFrame = [Call Number: string, Unit ID: string ... 32 more fields]

```
Rows: 5120231
Columns: 34
Size: 1.763 GB
```

```
df.printSchema()
```

```
 |-- Available DtTm: string (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
```

```
 |-- Zipcode of Incident: string (nullable = true)
 |-- Battalion: string (nullable = true)
 |-- Station Area: string (nullable = true)
 |-- Box: string (nullable = true)
 |-- Original Priority: string (nullable = true)
 |-- Priority: string (nullable = true)
 |-- Final Priority: string (nullable = true)
 |-- ALS Unit: string (nullable = true)
 |-- Call Type Group: string (nullable = true)
 |-- Number of Alarms: string (nullable = true)
 |-- Unit Type: string (nullable = true)
 |-- Unit sequence in call dispatch: string (nullable = true)
 |-- Fire Prevention District: string (nullable = true)
 |-- Supervisor District: string (nullable = true)
 |-- Neighborhooods - Analysis Boundaries: string (nullable = true)
 |-- Location: string (nullable = true)
 |-- RowID: string (nullable = true)
```

```
# Select important columns from Fire Calls dataset
df_spark_select = df.select(
    "Call Type",
    "City",
    "Response DtTm",
    "On Scene DtTm"
)

print("PySpark:")
df_spark_select.show(5)
```

df_spark_select: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 2 more fields]

```
PySpark:
+--------------+-------------+--------------------+--------------------+
|     Call Type|         City|       Response DtTm|       On Scene DtTm|
+--------------+-------------+--------------------+--------------------+
|        Alarms|San Francisco|10/18/2019 12:08:...|10/18/2019 12:11:...|
|        Alarms|San Francisco|                NULL|                NULL|
|        Alarms|San Francisco|                NULL|                NULL|
|        Alarms|San Francisco|10/18/2019 12:09:...|10/18/2019 12:09:...|
```

```
|Structure Fire|San Francisco|10/18/2019 12:13:...|10/18/2019 12:16:...|
+--------------+-------------+--------------------+--------------------+
only showing top 5 rows
```

```
from pyspark.sql.functions import col

# Apply multiple filters on Fire Calls dataset
df_filtered = (
    df_spark_select
    # Filter 1: Keep only rows where Call Type is "Structure Fire"
    .filter(col("Call Type") == "Structure Fire")
    # Filter 2: Keep only rows where City is "San Francisco"
    .filter(col("City") == "San Francisco")
)

print("✅ PySpark – Filtered Fire Calls (Structure Fires in San Francisco):")
df_filtered.show(5, truncate=False)
```

df_filtered: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 2 more fields]

```
✅ PySpark – Filtered Fire Calls (Structure Fires in San Francisco):
+--------------+-------------+----------------------+----------------------+
|Call Type     |City         |Response DtTm         |On Scene DtTm         |
+--------------+-------------+----------------------+----------------------+
|Structure Fire|San Francisco|10/18/2019 12:13:52 AM|10/18/2019 12:16:16 AM|
|Structure Fire|San Francisco|10/18/2019 12:14:28 AM|NULL                  |
|Structure Fire|San Francisco|10/18/2019 02:24:56 AM|10/18/2019 02:28:08 AM|
|Structure Fire|San Francisco|09/17/2017 10:06:29 AM|NULL                  |
|Structure Fire|San Francisco|10/18/2019 07:08:33 AM|NULL                  |
+--------------+-------------+----------------------+----------------------+
```

```
-+
only showing top 5 rows
```

```
# Aggregation on the Fire Calls data
from pyspark.sql.functions import col, to_timestamp, avg, count,
min as min_, max as max_, desc, expr

TS_FMT = "M/d/yyyy h:mm:ss a"

# 1) Add response delay (minutes) from the selected columns
df_with_delay = (
    df_spark_select
    .withColumn("received_ts", to_timestamp(col("Response DtTm"),
TS_FMT))
    .withColumn("on_scene_ts", to_timestamp(col("On Scene DtTm"),
TS_FMT))
    .withColumn("response_delay_min",
(col("on_scene_ts").cast("long") -
col("received_ts").cast("long")) / 60.0)
)

# 2) Aggregation per Call Type, compute count, avg, median, p95,
min, max delays
agg_result = (
    df_with_delay
    .filter(col("response_delay_min").isNotNull() &
(col("response_delay_min") > 0))
    .groupBy("Call Type")
    .agg(
        count("*").alias("count_calls"),
        avg("response_delay_min").alias("avg_delay_min"),
        expr("percentile_approx(response_delay_min,
0.5)").alias("median_delay_min"),
        expr("percentile_approx(response_delay_min,
0.95)").alias("p95_delay_min"),
        min_("response_delay_min").alias("min_delay_min"),
        max_("response_delay_min").alias("max_delay_min"),
    )
    .orderBy(desc("p95_delay_min"))
```

```
        .limit(10)
    )

    agg_result.show(truncate=False)
```

agg_result: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, count_calls: long ... 5 more fields]

df_with_delay: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 5 more fields]

```
+
|Administrative                    |66        |27.54848484848485 |14.03
3333333333333|90.25             |0.05               |93.16666666666667
|
|Mutual Aid / Assist Outside Agency|173       |28.03150289017341 |12.61
6666666666667|77.4              |0.03333333333333333 |265.25
|
|Aircraft Emergency                |507       |19.527843523997372|14.35
|49.81666666666667 |0.05               |119.06666666666666|
|Watercraft in Distress            |457       |9.325054704595184 |5.916
666666666667 |30.25             |0.13333333333333333 |116.51666666666667
|
|Train / Rail Incident             |885       |7.62864406779661  |4.316
666666666666 |26.833333333333332|0.016666666666666666|116.1
|
|Water Rescue                      |12277     |8.685153810648638 |6.0
|24.25             |0.016666666666666666|226.96666666666667|
|Suspicious Package                |220       |7.390757575757576 |4.8
|22.75             |0.016666666666666666|43.15             |
|High Angle Rescue                 |788       |9.081641285956005 |5.966
666666666667 |20.833333333333332|0.03333333333333333 |388.3666666666667
```

```
    from pyspark.sql.functions import col, avg, count, desc,
    to_timestamp

    # GroupBy with aggregation on Fire Calls dataset
    TS_FMT = "M/d/yyyy h:mm:ss a"

    # Add response delay in minutes
    df_with_delay = (
```

```
    df_spark_select
    .withColumn("received_ts", to_timestamp(col("Response DtTm"),
TS_FMT))
    .withColumn("on_scene_ts", to_timestamp(col("On Scene DtTm"),
TS_FMT))
    .withColumn("response_delay_min",
(col("on_scene_ts").cast("long") -
col("received_ts").cast("long")) / 60.0)
)

# Group by Call Type and calculate average delay and count of
calls
df_grouped = (
    df_with_delay
    .filter(col("response_delay_min").isNotNull() &
(col("response_delay_min") > 0))
    .groupBy("Call Type")
    .agg(
        avg("response_delay_min").alias("avg_response_delay_min"),
        count("*").alias("count_calls")
    )
    .orderBy(desc("avg_response_delay_min"))
)

print("✅ PySpark - GroupBy with Aggregation:")
df_grouped.show(10, truncate=False)
```

▸ df_grouped: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, avg_response_delay_min: double ... 1 more field]

▸ df_with_delay: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 5 more fields]

```
✅ PySpark - GroupBy with Aggregation:
+------------------------------------+----------------------+-----------+
|Call Type                           |avg_response_delay_min|count_calls|
+------------------------------------+----------------------+-----------+
|Mutual Aid / Assist Outside Agency  |28.03150289017341     |173        |
|Administrative                      |27.54848484848485     |66         |
|Aircraft Emergency                  |19.527843523997372    |507        |
|Watercraft in Distress              |9.325054704595184     |457        |
|High Angle Rescue                   |9.081641285956005     |788        |
|Water Rescue                        |8.685153810648638     |12277      |
```

```
|Train / Rail Incident                 |7.62864406779661      |885        |
|Suspicious Package                    |7.390757575757576     |220        |
|Marine Fire                           |6.943526785714285     |224        |
|Confined Space / Structure Collapse|6.7111986001749795    |381        |
+------------------------------------+---------------------+-----------+
only showing top 10 rows
```

```
from pyspark.sql.functions import col, to_timestamp, round, upper,
concat_ws

# Column transformations using withColumn on Fire Calls dataset
TS_FMT = "M/d/yyyy h:mm:ss a"

df_transformed = (
    df_spark_select
    # Convert timestamps
    .withColumn("received_ts", to_timestamp(col("Response DtTm"),
TS_FMT))
    .withColumn("on_scene_ts", to_timestamp(col("On Scene DtTm"),
TS_FMT))
    # Compute new column: response delay in minutes
    .withColumn("response_delay_min",
round((col("on_scene_ts").cast("long") -
col("received_ts").cast("long")) / 60.0, 2))
    # Uppercase city name
    .withColumn("City_Upper", upper(col("City")))
    # Combine city and call type into one string
    .withColumn("Call_Info", concat_ws(" - ", col("City"),
col("Call Type")))
)

print("✅ PySpark - Column Transformations using withColumn:")
df_transformed.select("City", "Call Type", "response_delay_min",
"City_Upper", "Call_Info").show(5, truncate=False)
```

_sqldf: pyspark.sql.connect.dataframe.DataFrame = [day_of_week: string, total_calls: long]

Table

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

```
%sql
-- Assumes a temp view named `fire_calls` exists with columns:
-- `Call Type`, City, `Response DtTm`, `On Scene DtTm`

-- Query 1: Average response delay per Call Type & City (parses
timestamps inline)
SELECT
  `Call Type`,
  City,
  AVG(
    (unix_timestamp(`On Scene DtTm`, 'M/d/yyyy h:mm:ss a') -
     unix_timestamp(`Response DtTm`, 'M/d/yyyy h:mm:ss a')) / 60.0
  ) AS avg_delay_min,
  COUNT(*) AS count_calls
FROM fire_calls
GROUP BY `Call Type`, City
ORDER BY avg_delay_min DESC
LIMIT 10;

-- Query 2: Count of "slow" responses (>10 min) per City (again
computed inline)
```

```
SELECT
  City,
  COUNT(*) AS slow_calls
FROM fire_calls
WHERE
  (unix_timestamp(`On Scene DtTm`, 'M/d/yyyy h:mm:ss a') -
   unix_timestamp(`Response DtTm`, 'M/d/yyyy h:mm:ss a')) / 60.0 >
10
GROUP BY City
ORDER BY slow_calls DESC
LIMIT 10;
```

_sqldf: pyspark.sql.connect.dataframe.DataFrame = [City: string, slow_calls: long]

Table

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

```
%sql
-- ✅ OPTIMIZED (works with the columns: uses Response DtTm, not
Received DtTm)
```

```
-- Assumes a temp view `fire_calls` with: `Call Type`, City,
`Response DtTm`, `On Scene DtTm`

-- 0) Stage: precompute delay & apply EARLY filters (narrow
columns)
CREATE OR REPLACE TEMP VIEW fire_calls_stage AS
SELECT
  `Call Type` AS call_type,
  City        AS city,
  (
    unix_timestamp(`On Scene DtTm`, 'M/d/yyyy h:mm:ss a') -
    unix_timestamp(`Response DtTm`,  'M/d/yyyy h:mm:ss a')
  ) / 60.0 AS delay_min
FROM fire_calls
WHERE City = 'San Francisco'         -- early filter
  AND `Call Type` IS NOT NULL;       -- drop null group keys


CREATE OR REPLACE TEMP VIEW fire_calls_stage_part AS
SELECT /*+ REPARTITION(16, call_type) */  -- tune 16 to your
cluster size
  call_type, city, delay_min
FROM fire_calls_stage
WHERE delay_min > 0;                  -- prune before wide ops

-- Query A: Average delay per Call Type (aggregate after pruning)
SELECT
  call_type,
  COUNT(*)                 AS count_calls,
  ROUND(AVG(delay_min),2)  AS avg_delay_min
FROM fire_calls_stage_part
GROUP BY call_type
ORDER BY avg_delay_min DESC
LIMIT 10;

-- Query B: Count of "slow" responses (>10 min) per City
SELECT
  city,
  COUNT(*) AS slow_calls
FROM fire_calls_stage_part
WHERE delay_min > 10
GROUP BY city
```

```
ORDER BY slow_calls DESC
LIMIT 10;
```

_sqldf: pyspark.sql.connect.dataframe.DataFrame = [city: string, slow_calls: long]

Table

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

```
from pyspark.sql.functions import col, avg, count, desc

# (City/Call Type filtered, response_delay_min computed, and > 0)

# 1) Keep only columns needed before wide ops (narrow -> fewer
bytes to shuffle)
df_narrow = df_clean.select("Call Type", "response_delay_min")

# 2) Partition by the group key to reduce shuffle during the
aggregation
df_partitioned = df_narrow.repartition(16, "Call Type")  # adjust
16 to your cluster size

# 3) Single wide op: groupBy + agg
df_agg = (
    df_partitioned
    .groupBy("Call Type")
    .agg(
```

```
            avg("response_delay_min").alias("avg_delay_min"),
            count("*").alias("count_calls")
        )
    )

    # 4) Global sort only after aggregation (on many fewer rows)
    top10 = df_agg.orderBy(desc("avg_delay_min")).limit(10)

    print("✅ PySpark — Optimized (filters early + partition by key + minimal shuffles, no persist):")
    top10.show(truncate=False)
```

▸ df_agg: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, avg_delay_min: double ... 1 more field]

▸ df_narrow: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, response_delay_min: double]

▸ df_partitioned: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, response_delay_min: double]

▸ top10: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, avg_delay_min: double ... 1 more field]

```
✅ PySpark — Optimized (filters early + partition by key + minimal shuffle
s, no persist):
+-----------------------------------+------------------+-----------+
|Call Type                          |avg_delay_min     |count_calls|
+-----------------------------------+------------------+-----------+
|Mutual Aid / Assist Outside Agency|86.18238095238097 |21         |
|High Angle Rescue                  |11.198544061302687|261        |
|Watercraft in Distress             |11.052735849056605|106        |
|Water Rescue                       |8.801310844464568 |4417       |
|Train / Rail Incident              |8.619045801526717 |262        |
|Suspicious Package                 |7.456808510638298 |47         |
|Marine Fire                        |7.300263157894739 |38         |
|Medical Incident                   |6.5862889263119415|921003     |
|Other                              |5.812180554645021 |15253      |
|Train / Rail Fire                  |5.754761904761905 |21         |
+-----------------------------------+------------------+-----------+
```

```
    # --- 1) Discover catalogs/schemas available in this workspace ---
```

```
print("Available catalogs:")
spark.sql("SHOW CATALOGS").show(truncate=False)

current_catalog = spark.sql("SELECT current_catalog()").first()[0]
current_schema  = spark.sql("SELECT current_schema()").first()[0]
print(f"current_catalog={current_catalog}, current_schema=
{current_schema}")
```

```
Available catalogs:
+---------+
|catalog  |
+---------+
|samples  |
|system   |
|workspace|
+---------+

current_catalog=workspace, current_schema=default
```

```
# ✅ Use the workspace catalog and default schema (supported on
serverless)
spark.sql("USE CATALOG workspace")
spark.sql("USE SCHEMA default")

# Create a Volume once (safe to re-run)
spark.sql("CREATE VOLUME IF NOT EXISTS analytics_vol COMMENT
'Analysis outputs'")

# Write your results to the Volume
dest_path =
"/Volumes/workspace/default/analytics_vol/fire_calls_top10_parquet
"

# Save as Parquet
top10.write.mode("overwrite").parquet(dest_path)

print("✅ Successfully wrote results to:", dest_path)
```

```
✅ Successfully wrote results to: /Volumes/workspace/default/analytics_vo
l/fire_calls_top10_parquet
```

```
# Actions vs Transformations
from pyspark.sql.functions import col, to_timestamp, round
import time

TS_FMT = "M/d/yyyy h:mm:ss a"
df = df_spark_select

# ---- Transformations (lazy) ----
t0 = time.time()
t = (
    df
    .filter(col("City") == "San Francisco")
    .filter(col("Call Type").isNotNull())
    .withColumn("received_ts", to_timestamp(col("Response DtTm"),
TS_FMT))
    .withColumn("on_scene_ts", to_timestamp(col("On Scene DtTm"),
TS_FMT))
    .withColumn(
        "response_delay_min",
        round((col("on_scene_ts").cast("long") -
col("received_ts").cast("long")) / 60.0, 2)
    )
    .select("Call Type", "City", "response_delay_min")
)
print(f"Transformations built in {time.time() - t0:.4f}s (no Spark
job yet)")

# ---- Actions (eager) ----
print("\nAction 1: count()")
t1 = time.time()
cnt = t.count()     # triggers a job
print(f"count() = {cnt:,} rows, took {time.time() - t1:.2f}s")

print("\nAction 2: show()")
t2 = time.time()
t.show(5, truncate=False)  # triggers another job
```

```
print(f"show() took {time.time() - t2:.2f}s")

print("\n🔍 Each action re-executes the transformations unless you
materialize it.")

# localCheckpoint() breaks lineage and materializes the result in
executor memory/disk
# without using metastore features that are blocked on serverless.
t_ckpt = t.localCheckpoint(eager=True)

print("\nAction 3 (after local checkpoint): show()")
t3 = time.time()
t_ckpt.show(5, truncate=False)  # should avoid recomputing full
upstream lineage
print(f"show() after local checkpoint took {time.time() -
t3:.2f}s")
```

df: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 2 more fields]

t: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 1 more field]

t_ckpt: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 1 more field]

```
|Alarms        |San Francisco|NULL              |
|Alarms        |San Francisco|0.0               |
|Structure Fire|San Francisco|2.4               |
+--------------+-------------+------------------+
only showing top 5 rows
show() took 0.27s

🔍 Each action re-executes the transformations unless you materialize i
t.

Action 3 (after local checkpoint): show()
+--------------+-------------+------------------+
|Call Type     |City         |response_delay_min|
+--------------+-------------+------------------+
|Alarms        |San Francisco|2.7               |
|Alarms        |San Francisco|NULL              |
|Alarms        |San Francisco|NULL              |
|Alarms        |San Francisco|0.0               |
|Structure Fire|San Francisco|2.4               |
```

```
+--------------+-------------+------------------+
only showing top 5 rows
```

```
# MLlib — Binary Classification on Fire Calls (serverless-safe)
from pyspark.sql.functions import col, when, to_timestamp, hour,
dayofweek
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator,
MulticlassClassificationEvaluator

TS_FMT = "M/d/yyyy h:mm:ss a"

# Use your selected columns DataFrame from earlier
df_src = df_spark_select

# Pick the available "start" timestamp column
start_ts_col = "Received DtTm" if "Received DtTm" in
df_src.columns else "Response DtTm"

# 1) Prepare label & features
df_ml = (
    df_src
    .withColumn("start_ts", to_timestamp(col(start_ts_col),
TS_FMT))
    .withColumn("on_scene_ts", to_timestamp(col("On Scene DtTm"),
TS_FMT))
    .withColumn("response_delay_min",
(col("on_scene_ts").cast("long") -
col("start_ts").cast("long"))/60.0)
    .withColumn("slow_response", when(col("response_delay_min") >=
10, 1).otherwise(0))
    .withColumn("hr", hour(col("start_ts")))
    .withColumn("dow", dayofweek(col("start_ts")))
    .dropna(subset=["slow_response", "hr", "dow", "City", "Call
Type"])
)
```

```
# Categorical -> index -> one-hot
cat_cols = ["Call Type", "City"]
idx_cols = [c + "_idx" for c in cat_cols]
ohe_cols = [c + "_vec" for c in cat_cols]

indexers = [StringIndexer(inputCol=c, outputCol=i,
handleInvalid="keep") for c, i in zip(cat_cols, idx_cols)]
encoder  = OneHotEncoder(inputCols=idx_cols, outputCols=ohe_cols)
assembler = VectorAssembler(inputCols=ohe_cols + ["hr", "dow"],
outputCol="features")

# 2) Model
lr = LogisticRegression(labelCol="slow_response",
featuresCol="features", maxIter=50)

pipeline = Pipeline(stages=indexers + [encoder, assembler, lr])

# 3) Train / Test
train_df, test_df = df_ml.randomSplit([0.8, 0.2], seed=42)
model = pipeline.fit(train_df)
pred  = model.transform(test_df)

# 4) Evaluate
auc = BinaryClassificationEvaluator(labelCol="slow_response",
rawPredictionCol="rawPrediction").evaluate(pred)
f1  = MulticlassClassificationEvaluator(labelCol="slow_response",
predictionCol="prediction", metricName="f1").evaluate(pred)
print(f"✅ Test AUC: {auc:.4f}")
print(f"✅ Test F1 : {f1:.4f}")

# Inspect
pred.select("Call
Type","City","response_delay_min","slow_response","prediction","pr
obability").show(10, truncate=False)
print("\nConfusion matrix (prediction vs label):")
(pred.groupBy("slow_response","prediction").count().orderBy("slow_
response","prediction")).show()
```

df_ml: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 8 more fields]

df_src: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 2

more fields]

pred: pyspark.sql.connect.dataframe.DataFrame

test_df: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 8 more fields]

train_df: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 8 more fields]

```
--+----------------------------------------+
|Aircraft Emergency|SFO       |NULL              |0            |0.0
|[0.5992085342183672,0.4007914657816328]  |
|Aircraft Emergency|SFO       |12.6              |1            |0.0
|[0.5875709822564589,0.4124290177435411]  |
|Alarms            |DC        |NULL              |0            |0.0
|[0.98835964573658,0.011640354263420027]  |
|Alarms            |Fort Mason|5.75              |0            |0.0
|[0.9806818508422595,0.01931814915774055] |
|Alarms            |Fort Mason|4.633333333333334 |0            |0.0
|[0.9807713700576458,0.019228629942354192]|
|Alarms            |Fort Mason|6.35              |0            |0.0
|[0.9807713700576458,0.019228629942354192]|
|Alarms            |Fort Mason|3.9166666666666665|0            |0.0
|[0.9807713700576458,0.019228629942354192]|
|Alarms            |Fort Mason|4.883333333333334 |0            |0.0
|[0.980690821117373,0.019309178882626954] |
|Alarms            |Fort Mason|5.266666666666667 |0            |0.0
|[0.980037387363936,0.019962612636064025] |
|Alarms            |Fort Mason|4.35              |0            |0.0
|[0.9805633327841002,0.01943666721589976] |
```

```
top10.explain()
```

```
S LAST], partitionOrderCount=0)
                    +- PhotonGroupingAgg(keys=[Call Type#15921], functi
ons=[avg(response_delay_min#19359), count(1)])
                       +- PhotonShuffleExchangeSource
                          +- PhotonShuffleMapStage REPARTITION_BY_NUM,
[id=#16701]
                             +- PhotonShuffleExchangeSink hashpartition
ing(Call Type#15921, 16)
                                +- PhotonProject [Call Type#15921, roun
```

```
d((cast((cast(gettimestamp(On Scene DtTm#15928, M/d/yyyy h:mm:ss a, Time
stampType, try_to_timestamp, Some(Etc/UTC), true) as bigint) - cast(gett
imestamp(Response DtTm#15927, M/d/yyyy h:mm:ss a, TimestampType, try_to_
timestamp, Some(Etc/UTC), true) as bigint)) as double) / 60.0), 2) AS re
sponse_delay_min#19359]
                                    +- PhotonFilter (((isnotnull(City#15
934) AND isnotnull(Call Type#15921)) AND (City#15934 = San Francisco)) A
ND (round((cast((cast(gettimestamp(On Scene DtTm#15928, M/d/yyyy h:mm:ss
a, TimestampType, try_to_timestamp, Some(Etc/UTC), true) as bigint) - ca
st(gettimestamp(Response DtTm#15927, M/d/yyyy h:mm:ss a, TimestampType,
try_to_timestamp, Some(Etc/UTC), true) as bigint)) as double) / 60.0),
2) > 0.0))
```

```
top10.limit(10).show()
```

```
+--------------------+------------------+-----------+
|           Call Type|     avg_delay_min|count_calls|
+--------------------+------------------+-----------+
|Mutual Aid / Assi...| 86.18238095238097|         21|
|   High Angle Rescue|11.198544061302687|        261|
|Watercraft in Dis...|11.052735849056605|        106|
|        Water Rescue| 8.801310844464568|       4417|
|Train / Rail Inci...| 8.619045801526717|        262|
|  Suspicious Package| 7.456808510638298|         47|
|         Marine Fire| 7.300263157894739|         38|
|    Medical Incident|6.5862889263119415|     921003|
|               Other| 5.812180554645021|      15253|
|   Train / Rail Fire| 5.754761904761905|         21|
+--------------------+------------------+-----------+
```

```
from pyspark.sql.functions import col
import time

df_src = top10
```

```
start = time.time()
_ = df_src.count()                  # first action (full recompute)
t1 = time.time() - start

start = time.time()
_ = df_src.count()                  # second action (recomputes
again)
t2 = time.time() - start

print(f"Baseline (no materialization) - first count:  {t1:.2f}s")
print(f"Baseline (no materialization) - second count: {t2:.2f}s")

# localCheckpoint()
start = time.time()
df_ckpt = df_src.localCheckpoint(eager=True)  # materialize now
t3 = time.time() - start
print(f"\nMaterialize with localCheckpoint(eager=True):
{t3:.2f}s")

# First action after checkpoint
start = time.time()
_ = df_ckpt.count()
t4 = time.time() - start

# Second action after checkpoint
start = time.time()
_ = df_ckpt.count()
t5 = time.time() - start

print(f"After checkpoint - first count:  {t4:.2f}s")
print(f"After checkpoint - second count: {t5:.2f}s")

start = time.time()
df_ckpt.show(5, truncate=False)
print(f"show() from checkpoint took: {time.time() - start:.2f}s")
```

df_ckpt: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, avg_delay_min: double ... 1 more field]

df_src: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, avg_delay_min: double ... 1 more field]

```
Baseline (no materialization) - first count:  4.74s
Baseline (no materialization) - second count: 4.12s

Materialize with localCheckpoint(eager=True): 4.99s
After checkpoint - first count:  0.24s
After checkpoint - second count: 0.12s
+-----------------------------------+------------------+-----------+
|Call Type                          |avg_delay_min     |count_calls|
+-----------------------------------+------------------+-----------+
|Mutual Aid / Assist Outside Agency|86.18238095238097 |21         |
|High Angle Rescue                  |11.198544061302687|261        |
|Watercraft in Distress             |11.052735849056605|106        |
|Water Rescue                       |8.801310844464568 |4417       |
|Train / Rail Incident              |8.619045801526717 |262        |
+-----------------------------------+------------------+-----------+
only showing top 5 rows
show() from checkpoint took: 0.22s
```

check_df: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, avg_delay_min: double ... 1 more field]

df_clean: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 5 more fields]

df_filtered: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 2 more fields]

df_result: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, avg_delay_min: double ... 1 more field]

df_with_delay: pyspark.sql.connect.dataframe.DataFrame = [Call Type: string, City: string ... 5 more fields]

```
|Medical Incident                   |6.586288926311867 |921003     |
|Other                              |5.812180554644988 |15253      |
|Train / Rail Fire                  |5.754761904761905 |21         |
+-----------------------------------+------------------+-----------+
only showing top 10 rows

✅ Successfully wrote results to: /Volumes/workspace/default/analytics_v
ol/fire_calls_top10_parquet

✅ Output successfully reloaded:
+-----------------------------------+------------------+-----------+
|Call Type                          |avg_delay_min     |count_calls|
+-----------------------------------+------------------+-----------+
```