ChatGPT

PySpark Learning Examples by Day

This companion document provides hands-on code examples aligned with the **7-Day Comprehensive Plan to Learn PySpark**. Each day introduces new features and concepts; the examples illustrate how to implement them in practice. You can run these snippets in a local PySpark environment or a managed service such as Databricks. Comments within the code explain what each step does and why it is important.

Day 0 – Pandas vs PySpark

Goal: See how PySpark enables distributed processing compared to pandas on a single machine.

```
import pandas as pd
from pyspark.sql import SparkSession
import pyspark.sql.functions as F

# Example dataset: list of log entries as dictionaries
data = [
    {"date": "2025-01-01", "user_id": 1, "url": "/home"},
    {"date": "2025-01-01", "user_id": 2, "url": "/about"},
    {"date": "2025-01-02", "user_id": 1, "url": "/contact"},
    # ... imagine millions of records
]

# Using pandas (single-threaded, limited by memory)
pdf = pd.DataFrame(data)
pandas_metrics = pdf.groupby("date").agg(
    unique_users=("user_id", pd.Series.nunique),
    page_views=("url", "count")
).reset_index()
print("Pandas metrics:\n", pandas_metrics)

# Using PySpark (distributed across a cluster)
spark =
SparkSession.builder.master("local[*]").appName("pandas_vs_pyspark").getOrCreate()
sdf = spark.createDataFrame(data)
pyspark_metrics = sdf.groupBy("date").agg(
    F.countDistinct("user_id").alias("unique_users"),
    F.count("url").alias("page_views")
)
pyspark_metrics.show()
```

```
# The PySpark version can scale to terabytes of data on many nodes, whereas
pandas operates in memory on a single machine.
```

Day 1 – Installation, Session and RDDs

Goal: Install PySpark, start a session, create an RDD and perform basic transformations.

```
from pyspark.sql import SparkSession

# Start a Spark session; in production use appropriate master (e.g., yarn, k8s).
spark = (
    SparkSession.builder
    .appName("learn_pyspark")
    .master("local[*]")
    .getOrCreate()
)

# Create an RDD from a Python list
data = [
    ("apple", 1), ("banana", 2), ("orange", 3),
    ("apple", 4), ("banana", 5)
]
rdd = spark.sparkContext.parallelize(data)

# Transformation: map values to double the numbers
doubled = rdd.map(lambda x: (x[0], x[1] * 2))

# Action: collect results to the driver (not recommended for very large RDDs)
print(doubled.collect())

# Fault tolerance test (conceptual): If a node fails during computation,
# Spark recomputes the lost partitions using the lineage of transformations.
```

Day 2 – RDD Transformations and Actions

Goal: Practice common RDD operations and understand fault tolerance.

```
from pyspark.sql import SparkSession

spark = (
    SparkSession.builder
```

```
    .appName("word_count")
    .getOrCreate()
)

# Read a text file into an RDD (assume `book.txt` exists in your filesystem)
text_rdd = spark.sparkContext.textFile("/path/to/book.txt")

# Clean and count words using flatMap, map and reduceByKey
word_counts = (
    text_rdd
    .flatMap(lambda line: line.split(" "))
    .map(lambda word: (word.strip().lower(), 1))
    .reduceByKey(lambda a, b: a + b)
)

# Top 10 most frequent words
for word, count in word_counts.takeOrdered(10, key=lambda x: -x[1]):
    print(word, count)

# Save the full word count results to HDFS or local filesystem (could be tens of
gigabytes)
word_counts.saveAsTextFile("/tmp/word_counts")

# If a worker fails mid-job, Spark uses the lineage graph of transformations to
recompute missing partitions.
```

Day 3 – Working with DataFrames and SQL

Goal: Create DataFrames, run SQL queries, and join tables.

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F

spark = SparkSession.builder.appName("dataframe_examples").getOrCreate()

# Read data from CSV with header and infer schema
transactions = spark.read.csv(
    "/path/to/transactions.csv", header=True, inferSchema=True
)
customers = spark.read.csv(
    "/path/to/customers.csv", header=True, inferSchema=True
)

# Inspect schema and sample data
```

```
transactions.printSchema()
customers.show(5)

# Filter: find transactions over a threshold
high_value = transactions.filter(F.col("amount") > 1000)
high_value.show()

# Group by and aggregate
customer_spend = (
    transactions.groupBy("customer_id")
    .agg(F.sum("amount").alias("total_spent"),
F.count("amount").alias("num_transactions"))
    .orderBy(F.desc("total_spent"))
)
customer_spend.show()

# Register DataFrames as temporary views for SQL
transactions.createOrReplaceTempView("transactions")
customers.createOrReplaceTempView("customers")

sql_result = spark.sql(
    """
    SELECT c.customer_id, c.name, SUM(t.amount) AS total_spent
    FROM customers c
    JOIN transactions t ON c.customer_id = t.customer_id
    GROUP BY c.customer_id, c.name
    ORDER BY total_spent DESC
    """
)
sql_result.show()
```

Day 4 – Structured Streaming Example

Goal: Build a simple streaming application that counts words from a socket.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

spark = SparkSession.builder.appName("streaming_word_count").getOrCreate()

# Create streaming DataFrame representing text from a socket connection
lines = (
    spark.readStream
    .format("socket")
```

```
    .option("host", "localhost")
    .option("port", 9999)
    .load()
)

words = lines.select(explode(split(lines.value, " ")).alias("word"))
word_counts = words.groupBy("word").count()

# Start running the query to the console
query = (
    word_counts.writeStream
    .outputMode("complete")
    .format("console")
    .option("truncate", False)
    .start()
)

query.awaitTermination()  # Listen indefinitely until manually stopped

# To test locally, open a terminal and run: `nc -lk 9999` then type text lines
to see counts update.
```

Day 5 – Machine Learning Pipeline

Goal: Create a classification model using MLlib and the pipeline API.

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator

spark = SparkSession.builder.appName("ml_pipeline").getOrCreate()

# Load a binary classification dataset (e.g., titanic.csv) with a label column
and numeric features
df = spark.read.csv("/path/to/titanic.csv", header=True, inferSchema=True)

# Basic preprocessing: drop rows with missing values
df_clean = df.dropna()

# Index the label column if it is string
label_indexer = StringIndexer(inputCol="Survived", outputCol="label")
```

```
# Assemble numeric features into a single vector column
feature_cols = ["Pclass", "Age", "SibSp", "Parch", "Fare"]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

# Define classifier
lr = LogisticRegression(featuresCol="features", labelCol="label")

# Build pipeline and split data
pipeline = Pipeline(stages=[label_indexer, assembler, lr])
train, test = df_clean.randomSplit([0.7, 0.3], seed=42)

# Train model
model = pipeline.fit(train)

# Evaluate
predictions = model.transform(test)
evaluator = BinaryClassificationEvaluator(labelCol="label")
auc = evaluator.evaluate(predictions)
print(f"AUC: {auc}")

# Show some predictions
predictions.select("Survived", "probability", "prediction").show(10,
truncate=False)
```

Day 6 – Performance Tuning and UDF Comparison

Goal: Compare a built-in Spark function to a Python UDF, and use caching and broadcasting.

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import IntegerType
import time

spark = SparkSession.builder.appName("performance_udf").getOrCreate()

# Create a DataFrame with one million numbers
n = 1_000_000
df = spark.range(0, n).withColumnRenamed("id", "value")

# Built-in function: square using pow()
df_builtin = df.withColumn("square", F.pow(F.col("value"), 2))

# Python UDF (slower)
def square_py(x: int) -> int:
```

```
    return x * x

square_udf = F.udf(square_py, IntegerType())
df_udf = df.withColumn("square", square_udf(F.col("value")))

# Cache DataFrames to memory to avoid recomputation
df_builtin.cache()
df_udf.cache()

# Time the operations
start = time.time()
df_builtin.count()
print("Built-in function took", time.time() - start)

start = time.time()
df_udf.count()
print("Python UDF took", time.time() - start)

# Broadcast join example
small_df = spark.createDataFrame([(1, 'A'), (2, 'B'), (3, 'C')], ["key",
"category"])
large_df = spark.range(0, n).withColumn("key", (F.col("value") % 3) + 1)

joined = large_df.join(F.broadcast(small_df), on="key")
joined.show(5)

# The built-in function is generally faster because Spark can optimize it.
# UDFs require data to cross the Python-JVM boundary, adding serialization
overhead.
```

Day 7 – Pandas API on Spark and Advanced Integration

Goal: Use the pandas API on Spark, read/write data, and interact with external systems.

```
import pandas as pd
from pyspark.sql import SparkSession
import pyspark.pandas as ps  # Available in PySpark ≥ 3.2 as pandas API on Spark

spark = SparkSession.builder.appName("pandas_on_spark").getOrCreate()

# Create a pandas DataFrame and convert to pandas-on-Spark
pdf = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
psdf = ps.from_pandas(pdf)
```

```
# Perform pandas operations (these run on Spark under the hood)
psdf["C"] = psdf["A"] + psdf["B"]
print(psdf)

# Convert back to pandas (only safe for small datasets)
pdf_back = psdf.to_pandas()
print(pdf_back)

# Read and write Parquet files
df = spark.createDataFrame(
    [(1, "Alice"), (2, "Bob"), (3, "Cathy")], ["id", "name"]
)
df.write.parquet("/tmp/names.parquet", mode="overwrite")
df2 = spark.read.parquet("/tmp/names.parquet")
df2.show()

# Connect to a JDBC source (e.g., PostgreSQL)
jdbc_df = spark.read.format("jdbc").option("url", "jdbc:postgresql://host:5432/
db") \
    .option("dbtable", "public.sales") \
    .option("user", "username") \
    .option("password", "password").load()
jdbc_df.show(5)

# Submit a job to a cluster (conceptual):
# spark-submit --master yarn --deploy-mode cluster --executor-memory 4G
my_script.py
```

Using These Examples

These examples are designed to be incremental: start by running Day 0 to understand why PySpark matters, set up your environment in Day 1, master RDDs in Day 2, graduate to DataFrames and SQL in Day 3, explore streaming in Day 4, build a machine-learning pipeline in Day 5, optimize performance and understand UDF costs in Day 6, and finally discover the pandas API on Spark and other integrations in Day 7.

Throughout, refer back to the **7-Day Comprehensive Plan** for context, goals and references. Experiment with your own data sets and cluster configurations to deepen your understanding.
