**KEY POINTS:**

**The docker files are in work-samplewithdocker.rar inside ‘samar’ folder.**

**Problem 1 and 2 are implemented but Problem 2 and 3 might take some time to run as the data is stored parallelly in runtime with each csv being saved seperately.**

**Task 3 is still under development.**

**To run the code in your machine with docker 🡪 first build the docker which is highlighted(see below) then run the docker which is highlighted(further below) in yellow.**

//The following workflow shows the working of the code:

data --->docker --->spark-shell --->schema --->df --->dfwithmovingavg --->dfwithrollingavg --->outputdir --->testdata --->result --->predictions --->mae,mse --->modelPath --->logMessage --->logPath

**To build docker in cmd first:**

**docker build -t spark\_shell:v2 .**

**Creating DockerFile with apache spark dependencies:**

**Create a ‘dockerfile’ in notepad**

FROM openjdk:8-jdk

RUN apt-get update && apt-get install -y curl

# Download and install Apache Spark

RUN curl -O https://downloads.apache.org/spark/spark-3.2.4/spark-3.2.4-bin-hadoop3.2.tgz && \

tar -xvf spark-3.2.4-bin-hadoop3.2.tgz && \

mv spark-3.2.4-bin-hadoop3.2 /spark && \

rm spark-3.2.4-bin-hadoop3.2.tgz

# Set environment variables

ENV SPARK\_HOME=/spark

ENV PATH=$PATH:$SPARK\_HOME/bin

# Start the Spark shell

CMD spark-shell --master local[\*]

COPY ./data /data

**To start the dockerfile with image of spark and dataset from local directory: This command runs the spark hdfs to process the huge amount of data we have in the directory that we have our data saved in.**

docker run -it --volume //c/Users/samar/OneDrive/Documents/samar/data:/data spark\_shell:v1

dir

**OR**

to use the docker file and run it independently: **docker run -it spark\_shell:v2**

**//Problem 1: Raw Data Processing**

**//Objective: Ingest and process raw stock market datasets.**

import org.apache.spark.sql.types.\_

import org.apache.spark.sql.types.{IntegerType,StringType,StructType,StructField}

import org.apache.spark.sql.{Row, SparkSession}

import org.apache.spark.sql.{SparkSession, DataFrame}

//Unit-Test

//val stocks = spark.read.format("csv").option("header","true").load("/data/etfs/AAAU.csv")

//stocks.schema

//stocks.show

**//UNIT TEST**

A picture containing text, screenshot, software, multimedia software

Description automatically generated

// Define the schema for the merged data

val schema = StructType(Seq(

StructField("Symbol", StringType, nullable = true),

StructField("Security Name", StringType, nullable = true),

StructField("Date", StringType, nullable = true),

StructField("Open", FloatType, nullable = true),

StructField("High", FloatType, nullable = true),

StructField("Low", FloatType, nullable = true),

StructField("Close", FloatType, nullable = true),

StructField("Adj Close", FloatType, nullable = true),

StructField("Volume", FloatType, nullable = true)

))

// Function to read CSV files and create a dataframe with the specified schema

def readCSVFiles(folderPath: String, schema: StructType): DataFrame = {

val df = spark.read.format("csv")

.schema(schema)

.option("header", "true")

.load(folderPath)

df

}

// Read CSV files from the etfs folder

val etfsFolderPath = "/data/etfs" // Replace with the actual path to the etfs folder

val etfsDF = readCSVFiles(etfsFolderPath, schema)

// Read CSV files from the stocks folder

val stocksFolderPath = "/data/stocks" // Replace with the actual path to the stocks folder

val stocksDF = readCSVFiles(stocksFolderPath, schema)

// Union the two dataframes to create a single dataframe with merged data

val mergedDF = etfsDF.union(stocksDF)

// Show the resulting dataframe

mergedDF.show()

**//MERGED DF WITH NEW SCHEMA:**

A picture containing electronics, text, screenshot, software

Description automatically generated

**//Tasks:**

**//Calculate the moving average of the trading volume (Volume) of 30 days per each stock and ETF, and retain it in a newly added column vol\_moving\_avg.**

**//Similarly, calculate the rolling median and retain it in a newly added column adj\_close\_rolling\_med.**

**//Retain the resulting dataset into the same format as Problem 1, but in its own stage/directory distinct from the first.**

**//(Bonus) Write unit tests for any relevant logic.**

import org.apache.spark.sql.expressions.Window

import org.apache.spark.sql.functions.\_

// Calculate the moving average of the trading volume (Volume) of 30 days per each stock and ETF

val windowSpec = Window.partitionBy("Symbol").orderBy("Date").rowsBetween(-29, 0)

val mergedDFWithMovingAvg = mergedDF.withColumn("vol\_moving\_avg", avg("Volume").over(windowSpec))

// Calculate the rolling median of the Adj Close per each stock and ETF

val rollingWindowSpec = Window.partitionBy("Symbol").orderBy("Date").rowsBetween(-29, 0)

val mergedDFWithRollingMed = mergedDFWithMovingAvg.withColumn("adj\_close\_rolling\_med", percentile\_approx(col("Adj Close"), lit(0.5), lit(1000)).over(rollingWindowSpec))

// Define the output directory path

val outputPath = "/data/merged\_with\_rolling.csv" // Replace with the desired output path for the merged dataset with rolling statistics

// Write the merged DataFrame with rolling statistics to the output path with a single partition

mergedDFWithRollingMed

.coalesce(10) // Set the number of partitions to 10

.write

.format("csv")

.mode("overwrite")

.option("header", "true")

.save(outputPath)

// View the contents of the output directory

spark.read.format("csv").option("header", "true").load(outputPath).show()

**// Tasks 3: TASK 3 IS STILL UNDER DEVELOPMENT AND MIGHT PRODUCE ERRORS**

**// Integrate the ML training process as a part of the data pipeline.**

**// Save the resulting model to disk.**

**// Persist any training metrics, such as loss and error values as log files.**

import org.apache.spark.ml.regression.RandomForestRegressor

import org.apache.spark.ml.feature.VectorAssembler

import org.apache.spark.ml.{Pipeline, PipelineModel}

import org.apache.spark.ml.evaluation.RegressionEvaluator

// Define the feature columns

val featureColumns = Array("vol\_moving\_avg", "adj\_close\_rolling\_med")

val targetColumn = "Volume"

// Create a VectorAssembler to combine the feature columns into a vector column

val assembler = new VectorAssembler()

.setInputCols(featureColumns)

.setOutputCol("features")

// Split the data into training and test sets

val Array(trainingData, testData) = mergedDFWithRollingMed.randomSplit(Array(0.8, 0.2), seed = 42)

// Create a RandomForestRegressor model

val rf = new RandomForestRegressor()

.setLabelCol(targetColumn)

.setFeaturesCol("features")

.setNumTrees(100)

.setSeed(42)

// Create a pipeline with the VectorAssembler and RandomForestRegressor

val pipeline = new Pipeline().setStages(Array(assembler, rf))

// Train the pipeline on the training data

val model = pipeline.fit(trainingData)

// Make predictions on the test data

val predictions = model.transform(testData)

// Calculate evaluation metrics (e.g., mean absolute error, mean squared error)

val evaluator = new RegressionEvaluator()

.setLabelCol(targetColumn)

.setPredictionCol("prediction")

val mae = evaluator.setMetricName("mae").evaluate(predictions)

val mse = evaluator.setMetricName("mse").evaluate(predictions)

// Save the model to disk

val modelOutputPath = "/data/model" // Replace with the desired output path for the model

model.write.overwrite().save(modelOutputPath)

// Persist training metrics to log files

val logPath = "/data/logs" // Replace with the desired output path for the log files

val logData = s"MAE: $mae\nMSE: $mse"

sc.parallelize(Seq(logData)).coalesce(10).saveAsTextFile(logPath)

**References:**

[**https://yuchen52.medium.com/getting-started-with-docker-scala-sbt-d91f8ac22f5f**](https://yuchen52.medium.com/getting-started-with-docker-scala-sbt-d91f8ac22f5f)

[**https://medium.com/@kale.miller96/how-to-mount-your-current-working-directory-to-your-docker-container-in-windows-74e47fa104d7**](https://medium.com/@kale.miller96/how-to-mount-your-current-working-directory-to-your-docker-container-in-windows-74e47fa104d7)

[**https://alvinalexander.com/scala/scala-execute-exec-external-system-commands-in-scala/**](https://alvinalexander.com/scala/scala-execute-exec-external-system-commands-in-scala/)

**ChatGPT was used to produce code for the tasks provided by using generating small batches of code and configuring it as required, most of ChatGPT was used after creating schema.**