/\*\*

\* Description.

\* Connect to MongoDB Client.

\* Do basic CRUD operations with MongoDB.

\*\*/

const config = require('./config');

const MongoClient = require('mongodb').MongoClient;

const configDB = config.dbConfig;

// Database configuration.

const uri = "mongodb+srv://"

+ configDB.user + ":"

+ configDB.password + "@"

+ configDB.server

+ configDB.database

+ "?retryWrites=true&w=majority";

/\*\*

\* Update MongoDB with incomming logs.

\* @param {JSON} message - Incomming messages to be updated to DB.

\* @param {String} collectionInterface - Connect to specific collection of the DB.

\* @param {Boolean} stream - If incomming messages are from a kafka stream or not.

\*\*/

async function updateMongoDB(message, collectionInterface = configDB.collectionStream, stream=true){

if (stream == true){

console.log(

'kafka ',

JSON.parse(message.value)

);

var logs\_ = JSON.parse(message.value)

} else {

console.log(

'Ml\_Prediction ',

message

);

logs\_ = message

}

let client = await MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });

client.connect(err => {

if (err) throw err;

const collection = client.db(configDB.database).collection(collectionInterface);

collection.insertOne(logs\_, function(err, res) {

if (err) throw err;

console.log("documents inserted." + res.insertedCount);

});

});

client.close();

}

/\*\*

\* Get information about the Database collection.

\* @return {integer} size of the database collection.

\* @return {JSON} return first three logs of the database collection.

\*\*/

async function getDBInfo(){

let client = await MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });

client.connect(err => {

if (err) throw err;

const collection = client.db(configDB.database).collection(configDB.collectionStream);

collection.find({},{ projection: { \_id: 0, Date: 1, Open: 1, High: 1, Low: 1, Close:1, Volume:1 }

}).toArray(function(err, result) {

if (err) throw err;

console.log("Size of DB: "+result.length);

console.log("First three objects of the Array: ");

console.log(result.slice(0,3));

client.close();

});

});

}

module.exports = { updateMongoDB, MongoClient, uri, getDBInfo}

const config = {

"kafkaConfig":{

KafkaHost:'localhost:9092',

KafkaTopic1: 'StockMarketAnalysis',

KafkaTopic2: 'StockMarketPredictions',

},

"dbConfig":{

user: 'VictorBasu',

password: 'V1cTOR',

server: 'clustera1.cvglu.mongodb.net/',

database: 'StockLogsDB',

collectionStream: 'StockData',

collectionML: 'mlPrediction',

},

"trainConfig":{

trainSize: 85,

epoch: 50,

batchSize: 32,

modelDir: 'models',

},

"type": "Open"

};

module.exports = config;

/\*\*

\* Description.

\* Consumer - consumer of the Kafka Stream.

\* Consume messages from the stream and update them to MongoDB.

\*\*/

const kafka = require('kafka-node');

const config = require('./config');

const db = require('./InstantiateDB');

const configKafka = config.kafkaConfig;

const Consumer = kafka.Consumer;

const client = new kafka.KafkaClient({idleConnection: 24 \* 60 \* 60 \* 1000, kafkaHost: configKafka.KafkaHost});

let consumer = new Consumer(

client,

[{topic: configKafka.KafkaTopic1, partition: 0 }],

{

autoCommit: true,

fetchMaxWaitMs: 1000,

fetchMaxBytes: 1024 \* 1024,

encoding: 'utf8',

// fromOffset: false

}

);

consumer.on('message', async function(message){

// Storing or updating consumed stream messages to MongoDB.

db.updateMongoDB(message)

});

consumer.on('error', function(error) {

// handle error

console.log('error', error);

});

const kafka = require('kafka-node');

const config = require('./config');

const client = new kafka.KafkaClient({kafkaHost: config.KafkaHost});

const topicToCreate = [{

topic: config.KafkaTopic1,

partitions: 1,

replicationFactor: 3,

},

{

topic: config.KafkaTopic2,

partitions: 1,

replicationFactor: 3,

}

]

client.createTopics(topicToCreate, (error, result) => {

// result is an array of any errors if a given topic could not be created

console.log(result, 'topic created successfully');

})

/\*\*

\* Description.

\* Consume logs from 2nd topic of the stream pipeline.

\* Tfjs model predict output in real-time.

\*\*/

const kafka = require('kafka-node');

const config = require('./config');

const db = require('./InstantiateDB');

const pre\_process = require('./pre\_process');

const model = require('./tf\_model');

const configKafka = config.kafkaConfig;

const Consumer = kafka.Consumer;

var data\_ = [];

let cnt = 0;

var type = config.type;

const client = new kafka.KafkaClient({idleConnection: 24 \* 60 \* 60 \* 1000, kafkaHost: configKafka.KafkaHost});

let consumer = new Consumer(

client,

[{topic: configKafka.KafkaTopic2, partition: 0 }],

{

autoCommit: true,

fetchMaxWaitMs: 1000,

fetchMaxBytes: 1024 \* 1024,

encoding: 'utf8',

// fromOffset: false

}

);

consumer.on('message', async function(message){

console.log(message.value)

cnt = cnt+1;

// process data from incomming stream

if (cnt != 8){

data\_.push(JSON.parse(message.value));

} else {

var scaled = pre\_process.transform(data\_,type);

const prediction = model.processModel([scaled]);

// Predict output.

prediction.then(function(result) {

var predicted\_stock\_price = pre\_process.inverse\_transform(result.arraySync());

console.log("Prediction",type, predicted\_stock\_price)

});

cnt=0;

data\_ = [];

}

});

consumer.on('error', function(error) {

// handle error

console.log('error', error);

});

/\*\*

\* Description.

\* Preprocess data with MinMaxScalar algorithm.

\* X\_std = (X - X.min()) / (X.max() - X.min())

\* X\_scaled = X\_std \* (max - min) + min

\*\*/

const fs = require('fs');

var X\_min = 0;

var X\_max = 0;

var min\_ = 0;

var max\_ = 1;

/\*\*

\* Write the minimum and maximum to be used for later scaling to a file.

\* Use the values for future transformation of data before model prediction.

\*\*/

function write\_util(){

const fs = require('fs')

// Data which will write in a file.

let data = {"X\_max":X\_max, "X\_min":X\_min, "max\_":max\_, "min\_":min\_};

fs.writeFileSync('utils.json', JSON.stringify(data));

}

/\*\*

\* fit data for preprocessing.

\* @param {Array} X - input data.

\* @param {integer} min - minimum value of the feature range.

\* @param {integer} max - maximum value of the feature range.

\* @return {Array} X\_scaled - Final scaled array fitted within Feature Range.

\*\*/

function fit(X, min=0, max=1){

X\_max = Math.max.apply(null,X)

X\_min = Math.min.apply(null,X)

min\_ = min;

max\_ = max;

var X\_minArr = X.map(function(values){

return values - X\_min

});

// X\_std = (X - X.min()) / (X.max() - X.min())

var X\_std = X\_minArr.map(function(values){

return values / (X\_max - X\_min)

});

// X\_scaled = X\_std \* (max - min) + min

var X\_scaled = X\_std.map(function(values){

return values\*(max - min) + min

});

return X\_scaled

}

/\*\*

\* Fit to data, then transform it.

\* @param {Array} result - array of objects.

\* @param {String} attribute - proprety of the JSON object to be accessed.

\* @return {Array} train\_scaled - Final scaled array fitted within Feature Range.

\*\*/

function fit\_transform(result, attribute){

var data = null;

try{

data = result.map(value => value[attribute]);

} catch (error) {

console.log("attribute undefined.");

}

var train\_scaled = fit(data);

write\_util();

return train\_scaled;

}

/\*\*

\* Scale features of X according to feature\_range.

\* @param {Array} result - array of objects.

\* @param {String} attribute - proprety of the JSON object to be accessed.

\* @return {Array} X\_scaled - Final scaled array fitted within Feature Range.

\*\*/

function transform(result, attribute){

var data = null;

let fit = require('./utils.json');

var data = null;

try{

data = result.map(value => value[attribute]);

} catch (error) {

console.log("attribute undefined.");

}

console.log(fit.X\_max,fit.X\_min)

var X\_minArr = data.map(function(values){

return values - fit.X\_min

});

var X\_std = X\_minArr.map(function(values){

return values / (fit.X\_max - fit.X\_min)

});

var X\_scaled = X\_std.map(function(values){

return values\*(fit.max\_ - fit.min\_) + fit.min\_

});

return X\_scaled

}

/\*\*

\* Undo the scaling of X according to feature\_range.

\* @param {Array} inp - Scaled array according to feature\_range.

\* @param {integer} min - minimum value of the feature range.

\* @param {integer} max - maximum value of the feature range.

\* @return {Array} X\_ - Inverse Scaled Array.

\*\*/

function inverse\_transform(inp, min=0, max=1){

let fit = require('./utils.json');

var X = inp.map(function(values){

return (values - min)/ (max - min)

});

var X\_ = X.map(function(values){

return values \* (fit.X\_max - fit.X\_min) + fit.X\_min

});

return X\_

}

module.exports = { fit\_transform, transform, inverse\_transform }

/\*\*

\* Description.

\* Producer - Producer of the Kafka Stream.

\* Pipeline logs from source to two Topics.

\*\*/

const Kafka = require('kafka-node');

const config = require('./config');

const fs = require('fs');

const parse = require('csv-parse');

var async = require('async');

var path = require('path');

const configKafka = config.kafkaConfig;

const Producer = Kafka.Producer;

const client = new Kafka.KafkaClient({kafkaHost: configKafka.KafkaHost});

var producer = new Producer(client, {requireAcks: 1, partitionerType: 2});

var KeyedMessage = Kafka.KeyedMessage;

var km = new KeyedMessage('key', 'message');

const parentDir = './Datasets/';

const averageDelay = 3000; // in miliseconds

const spreadInDelay = 2000; // in miliseconds

var ProducerReady = false ;

var stockArray;

producer.on('ready', async function () {

console.log("Producer is ready");

ProducerReady = true;

});

producer.on('error', function (err) {

console.error("Problem with producing Kafka message "+err);

})

var parser = parse({delimiter: ','}, function (err, data){

stockArray = data;

handleStock(1);

});

// Read data from a data-source.

fs.createReadStream(parentDir.concat('HPQ.csv')).pipe(parser);

/\*\*

\* Read data from a file and sent it for streaming.

\* @param {integer} dataCount - maintains the row count.

\*\*/

function handleStock(dataCount){

var line = stockArray[dataCount];

var stock = { "Date":line[0]

, "Open":parseFloat(line[1])

, "High":parseFloat(line[2])

, "Low":parseFloat(line[3])

, "Close":parseFloat(line[4])

, "Volume":parseFloat(line[6])

};

stockMarketMessages(stock)

// Adds delay after passing each log to the pipeline.

var delay = averageDelay + (Math.random() -0.5) \* spreadInDelay;

setTimeout(handleStock.bind(null,dataCount+1), delay);

}

/\*\*

\* Pipeline processed logs into 2 topics.

\* @param {JSON} data - processed logs.

\*\*/

function stockMarketMessages(data){

KeyedMessage = Kafka.KeyedMessage;

KM = new KeyedMessage(data.code, JSON.stringify(data));

// Pipeline logs into two topics.

payloadToKafkaTopic = [

{ topic: configKafka.KafkaTopic1, partition: 0, messages: KM },

{ topic: configKafka.KafkaTopic2, partition: 0, messages: KM },

];

if(ProducerReady){

producer.send(payloadToKafkaTopic, function (err, data) {

console.log(data);

});

} else {

console.error("sorry, Producer is not ready yet, failed to produce message to Kafka.");

}

}

const ml = require('./tf\_train');

const ml\_val = require('./tf\_validate')

const http = require('http');

const socketio = require('socket.io');

const TIMEOUT\_BETWEEN\_EPOCHS\_MS = 500;

const PORT = 8001;

// util function to sleep for a given ms

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

// Main function to start server, perform model training, and emit stats via the socket connection

async function run() {

const port = process.env.PORT || PORT;

const server = http.createServer();

const io = socketio(server);

server.listen(port, () => {

console.log(` > Running socket on port: ${port}`);

});

io.on('connection', (socket) => {

socket.on('Train', async () => {

io.emit('Model Performance', await ml.train());

});

socket.on('Validate', async () => {

io.emit('Model Performance', await ml\_val.val());

});

});

ml.train();

io.emit('trainingComplete', true);

}

run();

/\*\*

\* Description.

\* Developing model Architecture.

\* Compiling and fitting the model with training data.

\* Save and Load model.

\*\*/

const tf = require('@tensorflow/tfjs-node');

const config = require('./config');

const configTrain = config.trainConfig;

/\*\*

\* create tfjs model architecture.

\* @param {tensor} input\_ - input tensor.

\* @return {tensor} model - Model architecture.

\*\*/

function createModel(input\_){

const model = tf.sequential();

model.add(tf.layers.reshape({inputShape:[input\_[0].length],

targetShape: [input\_[0].length, 1]}));

model.add(tf.layers.lstm({units: 50, returnSequences: true}));

model.add(tf.layers.dropout(0.20));

model.add(tf.layers.lstm({units: 50, returnSequences: true}));

model.add(tf.layers.dropout(0.25));

model.add(tf.layers.lstm({units: 50, returnSequences: true}));

model.add(tf.layers.dropout(0.20));

model.add(tf.layers.lstm({units: 50}));

model.add(tf.layers.dropout(0.25));

model.add(tf.layers.dense({units: 1}));

return model;

}

/\*\*

\* compute loss and error at the end of every batch.

\* @param {integer} batch - batch number.

\* @param {dict} logs - computed loss and error estimated by loss functions.

\*\*/

function onBatchEnd(batch, logs) {

console.log({"loss":logs.loss, "mse": logs.mse, "mae": logs.mae});

}

/\*\*

\* Save model weight.

\* @param {tensor} model - trained model.

\*\*/

async function saveModel(model){

const savedModel = await model.save('file://'+ configTrain.modelDir);

console.log("Model weights saved.");

}

/\*\*

\* train model.

\*@param {tensor} model - tfjs model to be trained.

\*@param {Array} X - model input.

\*@param {Array} y - target to prediction.

\*\*/

async function train(model, X, y){

// prepare the model for training

model.compile({

optimizer: tf.train.adam(),

loss: 'meanSquaredError',

metrics: ['mse','mae'],

});

// train model

await model.fit(

tf.tensor(X),

tf.tensor(y),

{

epochs: configTrain.epoch,

batchSize: configTrain.batchSize,

callbacks: {onBatchEnd}

}

);

// save model

saveModel(model);

}

/\*\*

\* Load Model weight and predict output.

\* @param {Array} test\_data - input data for model to predict.

\* @return {Array} - Predicted output.

\*\*/

async function processModel(test\_data){

const model = await tf.loadLayersModel('file://'+configTrain.modelDir+'/model.json');

return model.predict(tf.tensor(test\_data))

}

module.exports = { createModel, onBatchEnd, train, saveModel, processModel}

/\*\*

\* Description.

\* Train the tensorflow.js model.

\* Save the trained model.

\*\*/

const config = require('./config');

const db = require('./InstantiateDB');

const pre\_process = require('./pre\_process');

const model = require('./tf\_model');

const configDB = config.dbConfig;

const mlConfig = config.trainConfig;

const timesteps = 7;

const type = config.type;

var i;

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

/\*\*

\* train the tfjs model with 80% data of the database.

\* @return {tfjs} save the trained model in a directory.

\*\*/

async function train(){

let client = new db.MongoClient(db.uri, { useNewUrlParser: true, useUnifiedTopology: true });

client.connect(err => {

if (err) throw err;

const collection = client.db(configDB.database).collection(configDB.collectionStream);

// Get all data from the database.

collection.find({},{ projection: { \_id: 0, Date: 1, Open: 1, High: 1, Low: 1, Close:1, Volume:1 }

}).toArray(async function(err, result) {

if (err) throw err;

var X\_train=[];

var y\_train=[];

// Using 80% of the total data for training the tfjs model.

result = result.slice(0, parseInt((mlConfig.trainSize/100) \* result.length));

// Preprocess data with MinMaxScalar.

var train\_scaled = pre\_process.fit\_transform(result,type);

for (i=timesteps; i<train\_scaled.length; i++){

X\_train.push(train\_scaled.slice(i-timesteps,i));

y\_train.push(train\_scaled[i]);

}

// Create the tfjs model.

const model\_ = model.createModel(X\_train);

model\_.weights.forEach(w => {

console.log(w.name, w.shape);

});

// Train the model and save it.

model.train(model\_, X\_train, y\_train);

sleep(2000);

client.close();

});

});

}

train();

module.exports = { train }

/\*\*

\* Description.

\* Validate the tensorflow.js model performance.

\* Update the database with model performance.

\*\*/

const config = require('./config');

const db = require('./InstantiateDB');

const pre\_process = require('./pre\_process');

const model = require('./tf\_model');

const train = config.trainConfig;

const configDB = config.dbConfig;

const timesteps = 7;

const type = config.type;

/\*\*

\* Validate the ml model perfromance with 20% data of the database.

\* Update database with model performance.

\*\*/

async function val(){

let client = new db.MongoClient(db.uri, { useNewUrlParser: true, useUnifiedTopology: true });

client.connect(err => {

if (err) throw err;

const collection = client.db(configDB.database).collection(configDB.collectionStream);

collection.find({},{ projection: { \_id: 0, Date: 1, Open: 1, High: 1, Low: 1, Close:1, Volume:1 }

}).toArray(async function(err, result) {

var X\_test=[];

// Using 20% of the total data for training the tfjs model.

var test\_data = result.slice(parseInt((train.trainSize/100) \* result.length), result.length);

var test\_data\_inp = result.slice(result.length - test\_data.length - timesteps,

result.length);

// Preprocess data with MinMaxScalar.

var test\_scaled = pre\_process.transform(test\_data\_inp,type);

for (i=timesteps; i<test\_data.length + timesteps; i++){

X\_test.push(test\_scaled.slice(i-timesteps,i));

}

// Get model prediction.

const prediction = model.processModel(X\_test);

prediction.then(function(result) {

// Inverse scale the predicted values to original value.

var predicted\_stock\_price = pre\_process.inverse\_transform(result.arraySync());

for(var i=0; i<test\_data.length; i++){

data\_ = {"date": test\_data[i].Date,

"Prediction": predicted\_stock\_price[i],

"real": test\_data[i][type],

"type": type

}

// Update database with model performance.

db.updateMongoDB(data\_, configDB.collectionML, false)

}

})

client.close();

});

});

}

val();