CS-562 Programming Assignment 3

Deadline: 30/11/2020

Create a folder including all the .scala files you used and the (.pdf or .docx) report for the answers. Send an email to hy562@csd.uoc.gr (**not the mailing list!**) with subject:

Assign3_StudentIdNumber

Description

The goal of this exercise is to train, test and evaluate a machine-learning pipeline for unsupervised anomaly detection in streaming manner using the Isolation Forest outlier detector (https://medium.com/@hyunsukim_9320/isolation-forest-step-by-step-341b82923168) on the Apache Spark framework (https://spark.apache.org).

The pipeline will be applied on the Mulcross dataset for the task of anomaly detection. This dataset is available for download in a CSV format at OpenML (https://www.openml.org/d/40897). Please pay attention on the characteristics that differentiate an inlier from outlier, because you need that knowledge to describe and explain the behavior of the detector on your data.

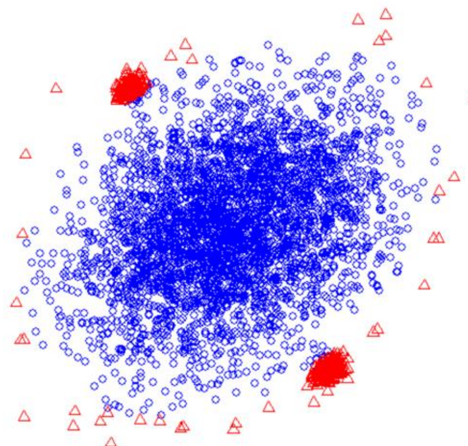

Figure 1 – The Mulcross dataset in two dimensional subspace

In Fig. 1. We illustrate the data points of the Mulcross dataset in a randomly selected 2D subspace. The red triangles represent the true outliers and the blue circles represent the normal data points. We can observe three clusters; a big normal cluster in the center and two small

anomaly clusters on the sides. Moreover, we observe some sparse outliers around the normal cluster.

Background

We assume that you are all familiar with the Apache Spark framework from previous assignments. In this assignment we are going to focus on two Spark modules; Structured Streaming and MLLib, as well as an example using both of these modules. Finally, a third-party implementation of Isolation Forest is going to be used as a detector (https://github.com/linkedin/isolation-forest#building-the-library).

A. Structured Streaming

Streaming applications become more and more challenging and it is getting difficult to implement with the current state of distributed streaming engines.

A Spark DStream (Discretized Stream) is the basic abstraction of Spark Streaming. DStream is a continuous stream of data. It receives input from various sources like Kafka, Flume, Kinesis, or TCP sockets. It can also be a data stream generated by transforming the input stream. At its core, DStream is a continuous stream of RDD (Spark abstraction). Every RDD in DStream contains data from the certain interval.

What are the problems of a DStream?

- Processing with event-time; dealing with late data.
- Interoperate streaming with batch and interactive.
- Reasoning about end-to-end guarantees.

Apache Spark 2.0 adds the first version of a new higher-level API, Structured Streaming, for building continuous applications. The main goal is to facilitate building end-to-end streaming applications, which integrate with storage, serving systems, and batch jobs in a consistent and fault-tolerant way. The last benefit of Structured Streaming is that the API is very easy to use — it is simply Spark's DataFrame and Dataset API. Users have to simply describe the query they want to run, the input and output directions, and optionally a few more details. The system then runs their query incrementally, maintaining enough state to recover from failure, keep the results consistent in external storage, etc.

What is the Programming Model?

Conceptually, Structured Streaming treats all the data arriving as an *unbounded input table*. Each new item in the stream is like a row appended to the input table. We won't actually retain all the input, but our results will be equivalent to having all of it and running a batch job.

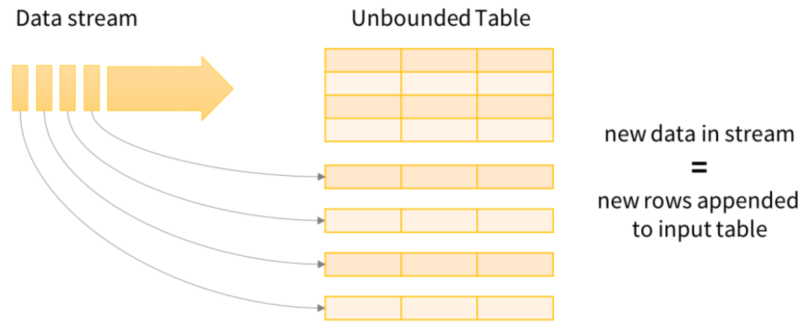

Figure 2 – An Unbounded Table in which new rows are appended over time

A query on the input will generate the Result Table. In every trigger interval (say, every one second), new rows get appended to the Input Table, which eventually updates the Result Table. Whenever the result table gets updated, we want to write the changed result rows to an external sink.

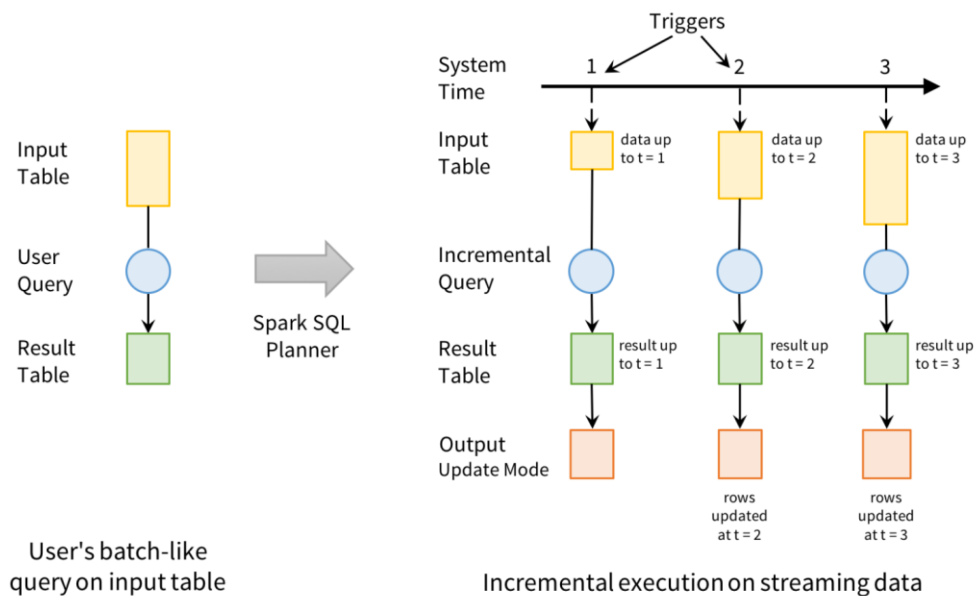

Figure 3 – Structured Streaming Processing Model

In Fig. 3, we can see that users' express queries using a batch API and then spark incrementalizes them to run on streams.

The last part of the model is output modes. Each time the result table is updated, the developer wants to write the changes to an external system, such as S3, HDFS, or a database. We usually want to write output incrementally.

What are the benefits over different Streaming Engines?

To show what's unique about Structured Streaming, the next table compares it with several other systems.

Property	Structured Streaming	Spark Streaming	Apache Storm	Apache Flink	Kafka Streams	Google Dataflow
Streaming API	incrementalize batch queries	integrates with batch	separate from batch	separate from batch	separate from batch	integrates with batch
Prefix Integrity Guarantee	✔	✔	✘	✘	✘	✘
Internal Processing	exactly once	exactly once	at least once	exactly once	at least once	exactly once
Transactional Sources/Sinks	✔	some	some	some	✘	✘
Interactive Queries	✔	✔	✘	✘	✘	✘
Joins with Static Data	✔	✔	✘	✘	✘	✘

Therefore, *Structured Streaming* is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. Note that Structured Streaming is still ALPHA in Spark 2.0 and the APIs are still experimental.

B. MLLib

The goal of MLLib is to make practical machine learning (ML) scalable and easy. Besides new algorithms and performance improvements that we have seen in each release, a great deal of time and effort has been spent on making MLlib *easy*. Similar to Spark Core, MLlib provides APIs in three languages: Python, Java, and Scala, along with user guide and example code, to ease the learning curve for users coming from different backgrounds. In Apache Spark 1.2, Databricks, jointly with AMPLab, UC Berkeley, continues this effort by introducing a pipeline API to MLlib for easy creation and tuning of practical ML pipelines.

A pipeline consists of a sequence of stages. There are two basic types of pipeline stages: Transformer and Estimator. A Transformer takes a dataset as input and produces an augmented dataset as output. E.g., a tokenizer is a Transformer that transforms a dataset with text into a dataset with tokenized words. An Estimator must be first fit on the input dataset to produce a model, which is a Transformer that transforms the input dataset. E.g., random forest is an Estimator that trains on a dataset with labels and features and produces a random forest model.

Creating a pipeline is easy: simply declare its stages, configure their parameters, and chain them in a pipeline object.

C. Structured Streaming and MLLib

In this section we introduce an example from Databricks that shows how to train an MLLib pipeline to produce a PipelineModel that can be applied to transform a streaming DataFrame. They also identify a few points to keep in mind when creating a pipeline that's meant to be used for transforming streaming data. Find the example here:

https://docs.databricks.com/applications/machine-learning/mllib/mllib-pipelines-and-stuctured-streaming.html

Note: you have to be able to follow the above example in order to give a solution for the following assignment.

D. Isolation Forest

The Isolation Forest algorithm [https://ieeexplore.ieee.org/abstract/document/4781136] is a type of unsupervised outlier detection that leverages the fact that outliers are "few and different," meaning that they are fewer in number and have unusual feature values compared to the inlier class. The underlying innovation was to use a randomly-generated binary tree structure to non-parametrically capture the multi-dimensional feature distribution of the training dataset.

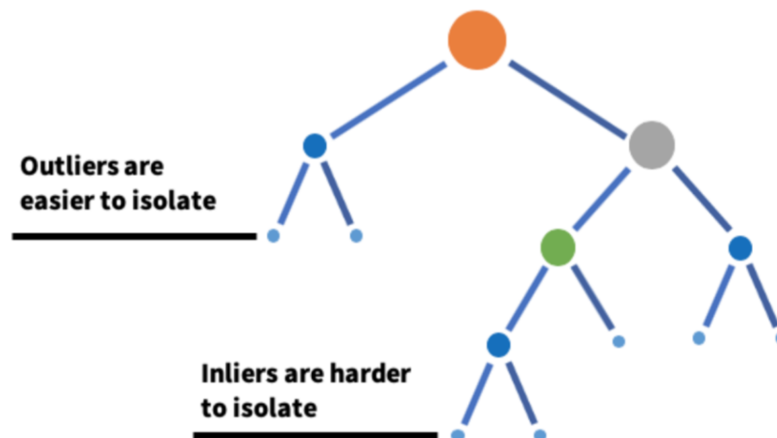

Figure 4 - An example Isolation Tree

The intuition is that outliers—due to being few and different—are easier to isolate in leaf nodes and thus require fewer random splits to achieve this, on average. The result is a shorter expected path length from the root node to the leaf node for outlier data points. The outlier score for a particular instance is a function of the path length from the root node to the leaf node and the total number of training instances used to build the tree.

For more information about the training and anomaly score phase of isolation forest, please take a look on this https://engineering.linkedin.com/blog/2019/isolation-forest tutorial in Linked-in engineering.

Assignment Tasks

In the Fig. 5, we present the required tasks of this assignment. First load the Mulcross dataset in a parquet format. Then split dataset into train/test set. Use train set to fit a model pipeline. Transform the test set into a stream of windows. Use the stream to incrementally score the anomalies through the model pipeline. Evaluate the model pipeline incrementally and print results to console.

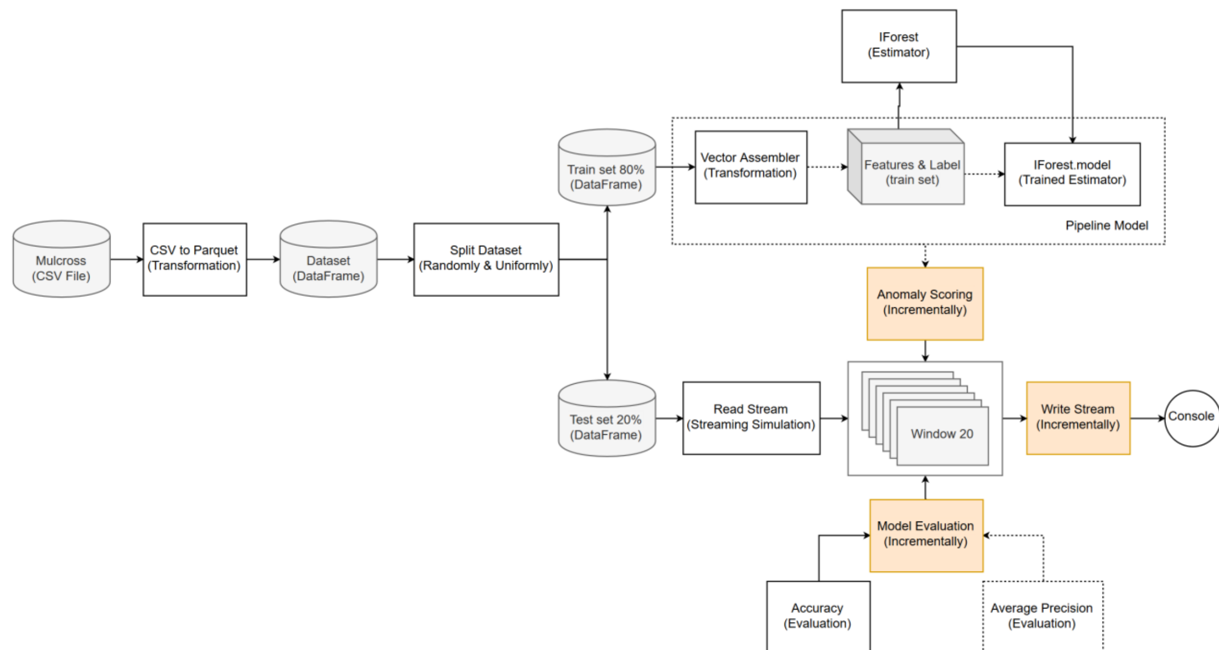

Figure 5 – The Architecture of the Required tasks of this Assignment

In the next sections we analyze all that required tasks that needs to be implemented.

Include External Jar

Download the IForest https://github.com/linkedin/isolation-forest#building-the-library open source Scala implementation and follow the instructions to create the jar file. You can include the path to the generated jar file of the IForest using the :require (before :load) command.

Load Dataset (5 points)

Load the Mulcross dataset using the following tasks:

- Create the schema of the Mulcross dataset using the StructType class.

- Build the Mulcross DataFrame by loading the **Mulcross.csv** from a local directory using the above schema.
- Write the above DataFrame into a local directory in a Parquet format.

Table 1 – The schema of the Mulcross dataset

Field Name	Field Type
v1	Double
v2	Double
v3	Double
v4	Double
label	Integer

Note: For validation purposes you can print the schema of the dataset using the *printSchema()* method.

Setup Pipeline (10 points)

Build the stages of a pipeline that is going to be used for the task of anomaly detection using only the features of the dataset. To this direction we need two stages: Transformation and Estimation.

A. Transformation Stage

Initialize a new *VectorAssembler* object by setting as input columns the array of the features field names **{v1, v2, v3, v4}** and output column the name "**features**".

B. Estimation Stage

Initialize a new *IsolationForest* object by setting the following hyper-parameters:

Table 2 – The required hyper-parameters of the Isolation Forest Estimator

Method	Argument
setNumEstimators	100
setBootstrap	false
setMaxSamples	256
setMaxFeatures	1.0

setFeaturesCol	features
setPredictionCol	predictedLabel
setScoreCol	outlierScore
setContamination	0.1
setRandomSeed	1

For more information about the description of the hyper-parameters of the Isolation Forest model, please take a look on the following subsection: https://github.com/linkedin/isolation-forest#model-parameters.

Extra Bonus: Change the split criterion of Isolation Forest to Random angle instead of vertical lines as it is introduced on the paper Extended Isolation Forest. (10 points)

Split Dataset into Train/Test set (5 points)

Split randomly and uniformly the Mulcross DataFrame into 80% of train and 20% of test set, using the *randomSplit()* method.

Note: This method will result an array of two DataFrames; train and test.

Train Phase (10 points)

First initialize a new *Pipeline* object by setting the transformation and estimation stages. Then build the (pipeline) model by applying the *fit()* method on the initialized pipeline using only the train set.

Load Stream Data (5 points)

We can simulate a stream by saving the test set of data and then using Spark to read it as a stream:

- Write the test set in a parquet format at a local path, using **20** repartitions and **overwrite** mode.
- Read the test set from that path as a Stream using the *readStream()* method, the *Mulcross schema* and **1 max files per trigger** as an option.

Note: Each of the 20 repartitions correspond to a tumbling window (batch and no duplicated set) of data points. The window size = number of test data / number of repartitions.

Anomaly Score Phase (10 points)

Calculate the anomaly score of each data point in the stream, using the *transform()* method on the trained pipeline model. From the resulted DataFrame, select only the "**OutlierScore**" and "**label**" fields.

Evaluation Phase (10 points)

Compute the accuracy of the pipeline and the baseline models over windows, using the resulted DataFrame of the previous phase. The available spark implementations of several evaluation metrics can be found https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html.

Note: The accuracy of the baseline model is computed as the total number of true outliers over the total number of data points.

Extra Bonus: compute the Average Precision (AP) of the model pipeline per window. **(10 points)**

Display Results (5 points)

Display on the console the accuracy (and AP) results of each window incrementally, using the writeStream() method on the resulted DataFrame of the previous phase.

Table 3 – Example of an expected result to be displayed on console per window

Baseline Accuracy	Pipeline Accuracy	Pipeline AP
0.81	0.76	0.10

Note: The *Pipeline AP* is optional, because the computation of it, is added as an extra bonus in the previous phase.

Critical Analysis (40 points)

Use your background knowledge as well as your implemented pipeline model to answer to the following four questions:

1. Did you observe a variation on the performance of the model pipeline over the windows? How do you explain this from the perspective of the dataset and the outlier detector? **(10 points)**
2. What are the data characteristics that makes the model pipeline to result higher performance on some windows? **(10 points)**

3. Will the performance of the model pipeline be affected by decreasing the number of trees to 10 (default rest parameters) and the number of max samples to 100 (default rest parameters) and why? **(10 points)**
4. The Isolation Forest detector does not have a model update phase. You are required to suggest your own update mechanism that can be applied to the tree-based architecture of that detector. What is the need of an update phase over windows from the perspective of performance? **(10 points)**

Note: By using the term performance we refer to the accuracy and optionally the AP. You can take some ideas of update mechanisms on an ensemble tree-based unsupervised outlier detector on the following two papers: HST and RRCF.

Have Fun!