Algoritmos para Big Data

2022/23

Data Stream Processing

This lecture is about processing a stream of data. We will rely on the structure streaming library of Apache Spark.

Structured streaming

A key aspect of structured streaming is to acquire/send data from a streaming data producer/consumer. That is, from a streaming source/sink.

Apache Spark provides methods to read/write from/to a stream, accordingly to some formats we may select from. Of course, some kind of configuration is required.

Firstly, there are the usual file-based formats like json, parquet, csv, text, and so on. Also, we can use socket connections to get/send text data from/to TCP servers, and more importantly, we can rely on functionalities of advanced message systems like Apache Kafka, which will play a sort of buffering role.

Secondly, we have to set an output mode, which defines how the results will be delivered. For instance, to see all data every time, only updates, or just the new records.

Further details can be found in https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Problem formulation

This exercise builds upon the credit card fraud detection prediction notebook we have worked upon in a previous lecture about classification. Recall that the main goal at the time was to create a ML binary classification model to detect frauds based on the dataset available in https://www.kaggle.com/datasets/ealtman2019/credit-card-transactions . The trained and validated SVM model was saved for further use, as well as the data used for the purpose.

This time around we will use the ML model that has been created but now we will deal with a stream of transactions that are expected to be processed, like it would be in in a real-time scenario. Hence, we will simulate such scenario, mostly relying on Spark's Structured Streaming.

The functional requirements for the Spark program we are going to create are as follows:

1. To load a ML model previously built.
2. To process credit card transactions held in a simulated data stream, by applying the ML model.
3. To explore the results obtained.

Also, in order to speed up some processing, we will use some files that were computed in advance.

In [1]:
```
# If we need to install some packages, e.g. matplotlib

# ! pip3 install matplotlib
# ! pip3 install seaborn
```

In [2]:
```python
# Some imports

import os
import time

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
```

Useful visualization functions

Some functions that we can use to plot data but as Python dataframes.

Disclaimer: these functions are broadly distributed among users. Further adjustments are needed and/or advisable. Feel free to use your own plotting functions.

In [3]:
```python
def plotBar(df, xcol, ycol, huecol=None):
    sns.barplot(data=df, x=xcol, y=ycol, hue=huecol)
```

In [4]:
```
def plotCorrelationMatrix(df, annot=False):
    # compute the correlation matrix
    corr = df.corr()

    # generate a mask for the upper triangle
    mask = np.triu(np.ones_like(corr, dtype=bool))

    # set up the matplotlib figure
    f, ax = plt.subplots(figsize=(11, 9))

    # generate a custom diverging colormap
    cmap = sns.diverging_palette(230, 20, as_cmap=True)
    #cmap='coolwarm'

    # draw the heatmap with the mask and correct aspect ratio
    sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0, annot=annot,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

Spark setup

In [5]:
```
# Some Spark related imports we will use hereafter

import sys

from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import *

from pyspark.ml import PipelineModel
```

In [6]:
```
# Build a SparkSession instance if one does not exist. Notice that we can only have one per JVM

spark = SparkSession\
    .builder\
    .appName("Streaming")\
    .getOrCreate()
```

```
23/05/02 17:17:03 WARN SparkSession: Using an existing Spark session; only runtime SQL configurations will take
effect.
```

Data collection

In [7]:
```
# ! pwd & ls -la
```

Recalling from the previous classification notebook ... Regarding data: df_cards.write.mode("overwrite").parquet("cards") df_users.write.mode("overwrite").parquet("users") df_transactions.write.mode("overwrite").parquet("credit_card_transactions") small_df_transactions.write.mode("overwrite").parquet("small-credit_card_transactions") As for the model: pipeline = Pipeline(stages=[string_indexer, ohe_encoder, vec_assembler, lsvc]) pipeline_model = pipeline.fit(df_train) df_prediction = pipeline_model.transform(df_test) pipeline.save("pipeline-LinearSVM") pipeline_model.save("model-LinearSVM") Notice that we are assuming the model has been created so it is going to be used as is (we did not save the train/test datasets anyway)

Loading the Data Stream

As we have no real time scenario in place, we will simulate a data stream by creating a built-in `rate` source to generate events at 1-second intervals, and join those 'ticks' with data from our downloaded dataset. This results in a regular stream of sample values.

Alternatively, for pratical applications, we could have used an Apache Kafka source or even a file source. Apache Kafka was the best solution for that matter.

In [8]:
```
rate_source = spark.readStream.format("rate").load()
```

In [9]:
```
rate_source.
```

```
root
 |-- timestamp: timestamp (nullable = true)
 |-- value: long (nullable = true)
```

In [10]:
```
# Read data

df_transactions = spark.read.parquet("credit_card_transactions")
#df_transactions = spark.read.parquet("small_credit_card_transactions")
```

In [11]:
```
# Check data

df_transactions.
```

```
root
 |-- User: integer (nullable = true)
 |-- Card: integer (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Month: integer (nullable = true)
 |-- Day: integer (nullable = true)
 |-- Time: timestamp (nullable = true)
 |-- Use Chip: string (nullable = true)
 |-- Merchant Name: long (nullable = true)
 |-- Merchant City: string (nullable = true)
 |-- Merchant State: string (nullable = true)
 |-- Zip: double (nullable = true)
 |-- MCC: integer (nullable = true)
 |-- Is Fraud?: string (nullable = true)
 |-- Correct Amount: float (nullable = true)

-RECORD 0------------------------------
 User           | 0
 Card           | 0
 Year           | 2002
 Month          | 9
 Day            | 1
 Time           | 2023-05-02 06:21:00
 Use Chip       | Swipe Transaction
 Merchant Name  | 3527213246127876953
 Merchant City  | La Verne
 Merchant State | CA
 Zip            | 91750.0
 MCC            | 5300
 Is Fraud?      | No
 Correct Amount | 134.09
only showing top 1 row
```

```
Out[11]: 24386900
```

# We may use data sorted by time # df_transactions = df_transactions.sort('Time', ascending=True)
# Just to remember ... number of transactions by year-month-day df_plot = (df_transactions .groupby(['Year', 'Month', 'Day']) .count() .withColumn('Year-Month-Day', F.concat('Year',F.lit('_'),'Month',F.lit('_'),'Day')) .sort('Year-Month-Day', ascending=True) .toPandas()) plotBar(df_plot, 'Year-Month-Day', 'count') plt.title('Number of transactions by year-month-day') plt.show()

df_plot.head()

Creating a continuous data stream

Circularly replaying the data as long as the process is running. It will be a simulated streaming version of the data.

In [12]:
```
# Because we have added two derived columns while creating the model, but only after storing data,
# let us add them here

df_transactions = ( df_transactions
                .withColumn("Hour", F.hour(F.col('Time')))
                .withColumn("Min", F.minute(F.col('Time')))
            )
```

# Useful function to add a column with a sequential ID def dfZipWithIndex (df, col_name="RowID"): new_schema = df.withColumn(col_name, F.lit(1)).schema zipped_rdd = df.rdd.zipWithIndex().map(lambda row,row_id: ([row_id+1] + list(row))) return spark.createDataFrame(zipped_rdd, new_schema)
# Add an id to data transactions ... it takes too much time df_transactions = dfZipWithIndex(df_transactions, 'Transaction_Id')
# As monotonically_increasing_id() does not work with data stream, a work-around could be: (df_transactions .withColumn('Transaction_Id', F.monotonically_increasing_id()) .write.mode("overwrite") .parquet("credit_card_transactions_stream"))

In [13]:
```
# To speed up things, let us read a data stream computed in advance (see codes above)

df_transactions = spark.read.parquet("credit_card_transactions_stream")
```

In [14]:
```
df_transactions.printSchema()
```

```
root
 |-- User: integer (nullable = true)
 |-- Card: integer (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Month: integer (nullable = true)
 |-- Day: integer (nullable = true)
 |-- Time: timestamp (nullable = true)
 |-- Use Chip: string (nullable = true)
 |-- Merchant Name: long (nullable = true)
 |-- Merchant City: string (nullable = true)
 |-- Merchant State: string (nullable = true)
 |-- Zip: double (nullable = true)
 |-- MCC: integer (nullable = true)
 |-- Is Fraud?: string (nullable = true)
 |-- Correct Amount: float (nullable = true)
 |-- Hour: integer (nullable = true)
 |-- Min: integer (nullable = true)
 |-- Transaction_Id: long (nullable = true)
```

In [15]:
```
df_transactions.show(2, vertical=True)
```

```
-RECORD 0------------------------------
 User           | 0
 Card           | 0
 Year           | 2002
 Month          | 9
 Day            | 1
 Time           | 2023-05-02 06:21:00
 Use Chip       | Swipe Transaction
 Merchant Name  | 3527213246127876953
 Merchant City  | La Verne
 Merchant State | CA
 Zip            | 91750.0
 MCC            | 5300
 Is Fraud?      | No
 Correct Amount | 134.09
 Hour           | 6
 Min            | 21
 Transaction_Id | 0
-RECORD 1------------------------------
 User           | 0
 Card           | 0
 Year           | 2002
 Month          | 9
 Day            | 1
 Time           | 2023-05-02 06:42:00
 Use Chip       | Swipe Transaction
 Merchant Name  | -727612092139916043
 Merchant City  | Monterey Park
 Merchant State | CA
 Zip            | 91754.0
 MCC            | 5411
 Is Fraud?      | No
 Correct Amount | 38.48
 Hour           | 6
 Min            | 42
 Transaction_Id | 1
only showing top 2 rows
```

In [16]:
```
# Circularly replay the data

data_stream = ( rate_source
                   .select(F.expr(f'value % {transactions_count}').alias('Transaction_Id'), 'timestamp')
                   .join(df_transactions, 'Transaction_Id')
              )
```

In [17]:
```
data_stream.printSchema()
```

```
root
 |-- Transaction_Id: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- User: integer (nullable = true)
 |-- Card: integer (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Month: integer (nullable = true)
 |-- Day: integer (nullable = true)
 |-- Time: timestamp (nullable = true)
 |-- Use Chip: string (nullable = true)
 |-- Merchant Name: long (nullable = true)
 |-- Merchant City: string (nullable = true)
 |-- Merchant State: string (nullable = true)
 |-- Zip: double (nullable = true)
 |-- MCC: integer (nullable = true)
 |-- Is Fraud?: string (nullable = true)
 |-- Correct Amount: float (nullable = true)
 |-- Hour: integer (nullable = true)
 |-- Min: integer (nullable = true)
```

In [18]:
```
cols_to_check = data_stream.columns
```

Model deployment

Loading the binary classification model

```
In [19]: # Read the ML model via pipeline api (not the simple pipeline)

         persisted_model = PipelineModel.load
```

```
In [20]: # Check the model

         persisted_model.stages
```

```
Out[20]: [StringIndexerModel: uid=StringIndexer_e1064637f5a0, handleInvalid=skip, numInputCols=2, numOutputCols=2,
          OneHotEncoderModel: uid=OneHotEncoder_6a398697f374, dropLast=true, handleInvalid=error, numInputCols=2, numOutp
         utCols=2,
          VectorAssembler_f41b5139a541,
          LinearSVCModel: uid=LinearSVC_912312635f03, numClasses=2, numFeatures=11838]
```

Streaming data transformer

Let us set the operation to be applied to the stream.

```
In [21]: # ML model directly applied to the streaming dataframe using `transform`

         prediction_stream =
```

```
In [22]: prediction_stream.printSchema()
```

```
root
 |-- Transaction_Id: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- User: integer (nullable = true)
 |-- Card: integer (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Month: integer (nullable = true)
 |-- Day: integer (nullable = true)
 |-- Time: timestamp (nullable = true)
 |-- Use Chip: string (nullable = true)
 |-- Merchant Name: long (nullable = true)
 |-- Merchant City: string (nullable = true)
 |-- Merchant State: string (nullable = true)
 |-- Zip: double (nullable = true)
 |-- MCC: integer (nullable = true)
 |-- Is Fraud?: string (nullable = true)
 |-- Correct Amount: float (nullable = true)
 |-- Hour: integer (nullable = true)
 |-- Min: integer (nullable = true)
 |-- Use Chip Index: double (nullable = false)
 |-- Merchant City Index: double (nullable = false)
 |-- Use Chip OHE: vector (nullable = true)
 |-- Merchant City OHE: vector (nullable = true)
 |-- features: vector (nullable = true)
 |-- rawPrediction: vector (nullable = true)
 |-- prediction: double (nullable = false)
```

Consuming predictions

The final step is to do something with the prediction data. For the time being, we are going to limit this step to just querying the data.

For real-world application, we can offer this kind of service to other applications.

Maybe in the form of an HTTP-based API or through pub/sub messaging interactions.

```
In [23]: cols_to_check.append('prediction')
         cols_to_check
```

```
Out[23]: ['Transaction_Id',
          'timestamp',
          'User',
          'Card',
          'Year',
          'Month',
          'Day',
          'Time',
          'Use Chip',
          'Merchant Name',
          'Merchant City',
          'Merchant State',
          'Zip',
          'MCC',
          'Is Fraud?',
          'Correct Amount',
          'Hour',
          'Min',
          'prediction']
```

```
In [ ]: # Just in case we want to start a table containing results but from scratch

        spark.sql("drop table if exists cardtransactionstable")
```

In [ ]:
```
# In case we want to store in an in-memory table (the sink).
# The query name will be the table name

# After executing the code, the streaming computation will start in the background

query_1 = ( prediction_stream
                        .select(cols_to_check)
                        .writeStream
                        .queryName("cardtransactionstable")
                        .outputMode("append")  # append, update
                        .format("memory")
                        .start()
                   )
```

In [ ]:
```
# Setup an aggregation by day concerning the number of frauds detected
#
# We leave this as exercise

# fraud_count = ...

# query_2 = ...
```

In [26]:
```
# Some extra checks

spark.streams.active[0].isActive
```

Out[26]:
```
True
```

In [27]:
```
query_1.status
```

Out[27]:
```
{'message': 'Processing new data',
 'isDataAvailable': True,
 'isTriggerActive': True}
```

In [ ]:
```
query_1.lastProgress
```

Exploring results

```
In [ ]: # Show the tables we hold

        spark.sql("show tables").
```

```
In [ ]: # Interactively query in-memory table

        spark.sql("select * from cardtransactionstable").show(vertical=True, truncate=False)
```

```
In [34]: # Interactively another query in-memory table

         spark.sql("select count(*) from cardtransactionstable").show()
```

```
[Stage 58:>                  (0 + 8) / 9][Stage 60:>                 (0 + 0) / 8]
+---------+
|count(1)|
+---------+
|      89|
+---------+

[Stage 58:===================================================>      (8 + 1) / 9]
23/05/02 17:19:25 WARN DAGScheduler: Broadcasting large task binary with size 1498.9 KiB
[Stage 64:>                                                         (0 + 8) / 9]
```

```
In [ ]: # Interactively another query in-memory table

        # spark.sql ...
```

Visual analysis ... we leave it as an exercise!

Stopping the process

In [ ]:
```
# We can turn off the query now and eventually set up a different one

query_1.stop()
```

In [ ]:
```
# Notice that in a production environment, we have to establish
# that the query is awaiting termination so to prevent the driver
# process from termination when the stream is ative

# query_1.awaitTermination()

# query_2.awaitTermination()
```

Additional exercise

Once this exercise is completed, create a new notebook with similar implementation but using a different streaming setup. Specifically, also relying on the messaging system Apache Kafka.

References

- Learning Spark - Lightning-Fast Data Analytics, 2nd Ed. J. Damji, B. Wenig, T. Das, and D. Lee. O'Reilly, 2020
- Stream Processing with Apache Spark. G. Maas and F. Garillot. O'Reilly, 2019
- Spark: The Definitive Guide - Big Data Processing Made Simple, 1st Ed. B. Chambers and M. Zaharia. O'Reilly, 2018

In [ ]:
```
```