Apache Spark Software Technology Evaluation Project

Jose Juan Pena Gomez and Enrique Vilchez Campillejo

November 29, 2019

Abstract

The software technology which will be discussed in this report is **Apache Spark**, an unified tool for analysing and processing data. In addition, **PySpark** is going to be used to write the code in **iPython Notebook**.

The purpose of this report is to check the different ways of processing big data using **Distributed Machine Learning** techniques, working with clusters, to see which way of **clustering** is more efficient.

The dataset taken for this project contains taxi travels with information about source location, target location, amount of passengers and taxi fare. This **large dataset** is a proficient approach to work with Spark in big data computing purposes.

There will be three Distributed Machine Learning models presented, in two approaches, one way for **Native Clustering**, which means one core acting as a cluster and the other approach is **Local Clustering** which means as many clusters as cores the processor has.

The models used are **Linear Regression**, **Decision Tree** and **Random Forest** generating several results. Basically, the results acknowledge the efficiency to work with Local Clustering rather than working with Native Clustering.

Another improvement would be to work with **Remote Clustering** in a network with several machines, but it wasn't possible to implement it in the end. Remote clustering is more efficient because of the quantity of cores used to parallelize the tasks.

1 Introduction

The main purpose of the project is to use a unified analytics engine (Apache Spark) to compare how clustering works with machine learning and how the technology works itself. As we worked before in projects of Machine Learning, we think it would be interesting to work with distributed machine-learning framework.

Approximately in 2008, the basis for creating this technology appeared, Hive and HBase, which are two tools of the Hadoop ecosystem. Spark was founded at UC Berkeley in 2009, it was created practically from a Google paper and since then it evolved, going through the mapreduce processes.

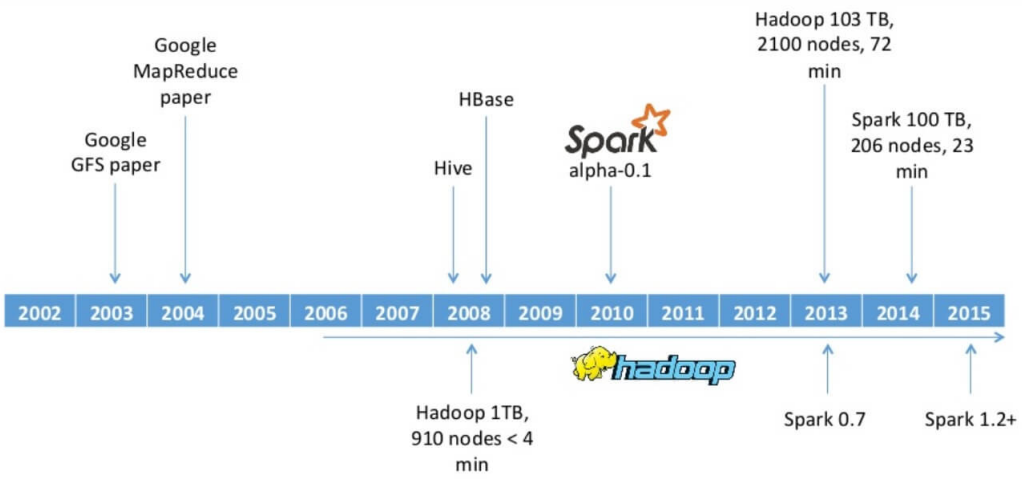

Figure 1: Apache Spark Development Timeline

Comparing this technology with others, for example with sklearn, the most well-known library for machine learning porposes, we are satisfied with our results. The sklearn library is not even able to load big datasets, but Apache Spark is. In addition, we are satisfied with the clustering management which is approximately 60 % faster than without clustering management.

This rest of this report is organised as follows:

- Section 2 gives an overview of the key concepts and the architecture.
- Section 3 gives a view of the prototype and its implementation
- Section 4 gives an explanation of the test-bed environment used and the results of the experiments
- Section 5 gives a conclusion about the main concepts that is observed from this technology.

2 The Software Technology

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since then. It is based on a data computing and analytics system based on Hadoop Map Reduce.

The official web defines Apache Spark as an unified analytics engine for large-scale data processing.[2]

Apache Spark has as its architectural foundation the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way.[6] The Dataframe API was released as an abstraction on top of the RDD, followed by the Dataset API. In Spark 1.x, the RDD was the primary application programming interface (API), but as of Spark

2.x use of the Dataset API is encouraged [1] even though the RDD API is not deprecated The RDD technology still underlies the Dataset API.

It works in memory, which results in a much faster processing speed, it allows you to work on disk with large amount of data. In this way if we have a very large file or a quantity of information that does not fit in memory, the tool allows to store part in disk, which makes lose speed. This means that we have to try to find the balance between what is stored in memory and what is stored in disk, to have a satisfying speed and at the same time make the total cost possibly the lowest, as the memory is always much more expensive than the disk.

Spark and its RDDs were developed in 2012 in response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory.[5]

Spark facilitates the implementation of both iterative algorithms, which visit their data set multiple times in a loop, and interactive/exploratory data analysis, i.e., the repeated database-style querying of data. The latency of such applications may be reduced by several orders of magnitude compared to Apache Hadoop MapReduce implementation Among the class of iterative algorithms are the training algorithms for machine learning systems, which formed the initial impetus for developing Apache Spark.

Apache Spark requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone (native Spark cluster, where you can launch a cluster either manually or use the launch scripts provided by the install package. It is also possible to run these daemons on a single machine for testing), Hadoop YARN, Apache Mesos or Kubernetes. For distributed storage, Spark can interface with a wide variety, including Alluxio, Hadoop Distributed File System (HDFS), MapR File System (MapR-FS), Cassandra, OpenStack Swift, Amazon S3, Kudu, Lustre file system, or a custom solution can be implemented. Spark also supports a pseudo-distributed local mode, usually used only for development or testing purposes, where distributed storage is not required and the local file system can be used instead; in such a scenario, Spark is run on a single machine with one executor per CPU core.[4]

The friendly APIs, which Apache Spark provides, natively supports Java, Scala, R, and Python, giving you a variety of languages for building your applications. These APIs make it easy for developers, because they hide the complexity of distributed processing behind simple, high-level operators that lowers the amount of code required.

It allows real time processing, with a module called Spark Streaming, which combined with Spark SQL will allows to process the data in the real time. As we are injecting the data, we can transform them and turn them to a final result. Resilient Distributed Dataset (RDD), it uses the lazy evaluation, which means that all the transformations that we are carrying out on the RDD, are not resolved, but are stored in a directed acyclic graph (DAG), and when we execute an action, when the tool does not have more option than to execute all the transformations, that's when they are executed. This is a double-edged sword, as it has an advantage and a disadvantage. The advantage is that you gain speed by not making transformations continuously, but only when necessary. The disadvantage is that if

some transformation raises some kind of exception, it will not be detected until the action is executed, so it is more difficult to program or debug something.

2.1 Spark Architecture

The main components that make up the framework are the following ones:

Spark Core : It is the base or set of libraries where the rest of modules are supported, is the core of the framework. It is also a shelter to API, that contains the backbone of Spark, that is RDDs (resilient distributed datasets). The basic functionality of Spark is present in Spark Core like:

1. Memory management
2. Fault recovery
3. Interaction with the storage system.
4. It is in charge of essential I/O functionalities like:
   - Programming and observing the role of Spark cluster
   - Task dispatching
   - Fault recovery
   - It overcomes the snag of MapReduce by using in-memory computation.

Spark SQL : It is the module for the processing of structured and semi-structured data. With this module we will be able to transform and perform operations on RDD or dataframes. It is designed exclusively for data processing.

Spark Streaming : It is the one that allows ingesting the data in real time. If we have a source, for example Kafka or Twitter, with this module we can ingest data from that source and dump them to a destination. Between the ingestion of data and its subsequent dump, we can have a series of transformations.

Spark MLLib : It is a very complete library that contains numerous Machine Learning algorithms, both clustering, classification, regression, and so on. It allows us, in a friendly way, to use Machine Learning algorithms.

Spark Graph : It allows the processing of graphs (DAG). It does not allow to paint graphs, but it allows to create operations with graphs, with their nodes and edges, and to make operations.

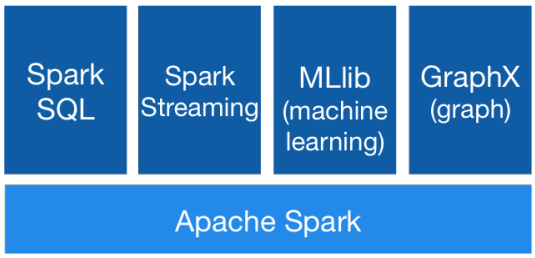

Figure 2: Components of Apache Spark
Spark

2.2 MLlib

MLIB is part of Spark APIs which is interoperable with Python NumPy as well as with R-libraries. Implemented with Spark it is possible to use any type of data from any source or from the Hadoop platform such as HDFS, HBase, related database data sources or local data sources such as text documents. Spark excels in iterative calculation processes allowing the processes written in the MLlib libraries to be executed quickly allowing their use and above all at an industrial level.
MLlib provides many types of algorithms, as well as numerous useful functions. ML includes classification algorithms, regression, decision trees, recommendation algorithms, grouping. Among the most used utilities can include the characteristics of transformations, standardization and normalization, statistical functions and linear algebra.

2.3 Clustering

Apache Spark can be configured to run on distributed mode on the cluster as a master node or slave node. Its master/slave architecture is composed by many main daemons and a cluster manager. T schedules and divides resource in the host machine which forms the cluster.

- Master Daemon (Master/Driver Process)
- Worker Daemon (Slave Process)
- Cluster Manager. Schedules and divides resources in the host machine which forms the cluster

As we can see in the figure 3 .

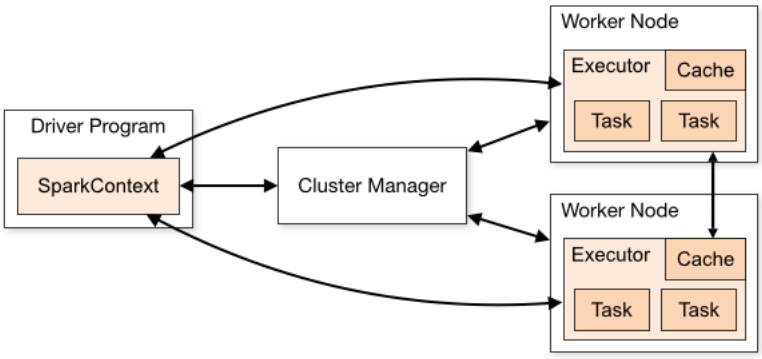

Figure 3: Cluster management of Apache Spark

3 Demonstrator Prototype

3.1 High-Level View

As any other machine learning project, this prototype learns how to predict the results of new data through the results of a given dataset with the results already set. As a consequence, we designed a prototype with a big input dataset of 6 Gb to test the technology. Implicitly the library is prepared for machine learning, the one used here, which is splitting, parallelizing and clustering when we use the data structure of this library call RDD.

While processing, we are able to test the performance of Apache Spark and its component for machine learning, called MLLib. We check the differences in performance using the time spending in every training of a model because it has to compute a big amount of data, as consequence we can test this technology in terms about clustering and parallelizing huge amount of data and compare the results between native clustering and local clustering, by native we mean that we use the component by default with only one core for clustering and by local we mean to use the component with as many core as the machine has. There is a third one, called Remote Clustering, but it will not be discussed in this report.

3.2 Implementation

The following link contains the URL of the GitHub Repository of our project:

`https://github.com/envilk/TaxisFareMLlibPyspark`

The project is implemented in a Jupyter Notebook, in consequence the code is commented already between lines of code. That's why, the report follows with all the code of the prototype.

Taxi

November 29, 2019

1 Apache Spark - Taxi Fare

1.1 Set Up Libraries

There are several libraries imported according to what is needed to build up the models.

[1]:
```
from pyspark import SparkContext
import pyspark as pyspark
from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler
from pyspark.ml.linalg import Vectors

from pyspark.ml.regression import LinearRegression
from pyspark.sql.types import FloatType
import pandas as pd

from datetime import datetime
```

2 Native Clustering

The APIs for Machine Learning and DataBases purposes will be used. Native Clustering means using libraries that implicitly parallelize the data and fault tolerance. It's important to take it into account because later it's used to solve the same problem but with another way of clustering.

2.1 Set a Spark Session

The purpose of this method is to create an interface between the Jupyter Notebook and the Spark-shell.

[2]:
```
spark = SparkSession.builder.appName('taxis_fare').getOrCreate()
```

2.2 Load Dataset

Loading the data set with the spark method.

[3]:
```
# Load training data
df = spark.read.format("csv").option("header", "true").load("data/train.csv")
```

2.3 Preparing Data

Computing the LinearRegression model

[4]:
```
lr = LinearRegression()
```

It's crutial to see the type of the attributes to check if it's needed to process some categorical attributes, but not in this case.

[5]:
```
df.dtypes
```

[5]:
```
[('key', 'string'),
 ('fare_amount', 'string'),
 ('pickup_datetime', 'string'),
 ('pickup_longitude', 'string'),
 ('pickup_latitude', 'string'),
 ('dropoff_longitude', 'string'),
 ('dropoff_latitude', 'string'),
 ('passenger_count', 'string')]
```

It's necessary to cast all the attributes to float because it's one of the datatypes that Spark works with, because Spark doesn't work with strings.

[6]:
```
df = df.select(df['fare_amount'].cast("float").alias('fare_amount'),
               df['pickup_longitude'].cast("float").alias('pickup_longitude'),
               df['pickup_latitude'].cast("float").alias('pickup_latitude'),
               df['dropoff_longitude'].cast("float").alias('dropoff_longitude'),
               df['dropoff_latitude'].cast("float").alias('dropoff_latitude'),
               df['passenger_count'].cast("float").alias('passenger_count'))
```

Selection of the most important features to work with.

[7]:
```
df = df.selectExpr("fare_amount as label",'pickup_longitude','pickup_latitude',
                   'dropoff_longitude','dropoff_latitude','passenger_count')
```

Checking the correctness of the features, by taking a look at the current data.

[ ]:
```
df.show()
```

Gathering all the features into one colum called "features" because Spark needs to work with all the features in one column.

[9]:
```
#colum features
vecAssembler = VectorAssembler(inputCols=["pickup_longitude", "pickup_latitude",
                                          "dropoff_longitude",␣
 ↪"dropoff_latitude",
                                          "passenger_count"],␣
 ↪outputCol="features")
new_df = vecAssembler.transform(df)
new_df.count()
```

[9]:
```
55423856
```

It's vital to delete null rows because Spark doesn't accept them.

[ ]:
```
#Delete null rows
new_df = vecAssembler.setHandleInvalid("skip").transform(df)
```

```
new_df.show()
```

2.4 Train Model

We use a Linear Regression algorithms to train the model. It's necessary to measure the time the model spends to train the data, which is one of the main purposes of this project.

[11]:
```
# Fit the model
start_time = datetime.now()

lrModel = lr.fit(new_df.select('label','features'))

time_elapsed = datetime.now() - start_time
print('TIME OF LINEAR REGRESSION TRAINING (hh:mm:ss.ms) {}'.
 ↪format(time_elapsed))
```

```
TIME OF LINEAR REGRESSION TRAINING (hh:mm:ss.ms) 0:08:17.698225
```

2.5 Measures

2.5.1 Root Mean Squared Error

[12]:
```
#Summarize the model over the training set and print out some metrics
trainingSummary = lrModel.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
```

```
RMSE: 20.710237
```

[13]:
```
spark.stop()
```

3 Decision tree regression

Computing Decision Tree with Local Clustering.

[14]:
```
from pyspark.ml import Pipeline
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

spark = SparkSession.builder.appName('taxis_fare').getOrCreate()

# Load training data
df = spark.read.format("csv").option("header", "true").load("data/train.csv")

df = df.select(df['fare_amount'].cast("float").alias('fare_amount'),
               df['pickup_longitude'].cast("float").alias('pickup_longitude'),
               df['pickup_latitude'].cast("float").alias('pickup_latitude'),
```

```
                df['dropoff_longitude'].cast("float").alias('dropoff_longitude'),
                df['dropoff_latitude'].cast("float").alias('dropoff_latitude'),
                df['passenger_count'].cast("float").alias('passenger_count'))

df = df.selectExpr("fare_amount as label",'pickup_longitude','pickup_latitude',
                   'dropoff_longitude','dropoff_latitude','passenger_count')

new_df = vecAssembler.setHandleInvalid("skip").transform(df)

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = new_df.randomSplit([0.7, 0.3])

# Train a DecisionTree model.
dt = DecisionTreeRegressor()

start_time = datetime.now()

# Train model.  This also runs the indexer.
model = dt.fit(trainingData)

time_elapsed = datetime.now() - start_time
print('TIME OF DECISION TREE REGRESSION TRAINING (hh:mm:ss.ms) {}'.
 ↪format(time_elapsed))

# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("prediction", "label", "features").show(5)

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
TIME OF DECISION TREE REGRESSION TRAINING (hh:mm:ss.ms) 0:20:05.413627
+------------------+------+--------------------+
|        prediction| label|            features|
+------------------+------+--------------------+
| 34.02801889571103|-29.87|[-73.863159179687...|
|11.850275013996376| -20.0|       (5,[4],[5.0])|
| 8.896463517049114|  -6.5|[-73.984352111816...|
| 8.896463517049114|  -6.0|[-73.987518310546...|
| 8.896463517049114|  -3.0|[-73.995063781738...|
+------------------+------+--------------------+
only showing top 5 rows
```

```
Root Mean Squared Error (RMSE) on test data = 23.9768
```

4 Random forest regression

Computing Random Forest Regression with Local Clustering.

[15]:
```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator

spark = SparkSession.builder.appName('taxis_fare').getOrCreate()

# Load training data
df = spark.read.format("csv").option("header", "true").load("data/train.csv")

df = df.select(df['fare_amount'].cast("float").alias('fare_amount'),
               df['pickup_longitude'].cast("float").alias('pickup_longitude'),
               df['pickup_latitude'].cast("float").alias('pickup_latitude'),
               df['dropoff_longitude'].cast("float").alias('dropoff_longitude'),
               df['dropoff_latitude'].cast("float").alias('dropoff_latitude'),
               df['passenger_count'].cast("float").alias('passenger_count'))

df = df.selectExpr("fare_amount as label",'pickup_longitude','pickup_latitude',
                   'dropoff_longitude','dropoff_latitude','passenger_count')

new_df = vecAssembler.setHandleInvalid("skip").transform(df)

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = new_df.randomSplit([0.7, 0.3])

# Train a RandomForest model.
rf = RandomForestRegressor()

# Train model.
start_time = datetime.now()

model = rf.fit(trainingData)

time_elapsed = datetime.now() - start_time
print('TIME OF RANDOM FOREST TRAINING (hh:mm:ss.ms) {}'.format(time_elapsed))

# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("prediction", "label", "features").show(5)
```

```
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
TIME OF RANDOM FOREST TRAINING (hh:mm:ss.ms) 0:35:53.671012
+------------------+------+--------------------+
|        prediction| label|            features|
+------------------+------+--------------------+
|18.195694139170584| -44.9|[-73.871116638183...|
|30.771499981119245|-29.87|[-73.863159179687...|
|13.290360749502483| -18.1|[-73.958274841308...|
| 8.933319739165913| -10.1|[-73.983306884765...|
| 8.933319739165913|  -6.5|[-73.984352111816...|
+------------------+------+--------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 24.8237
```

5 With Local Cluster

Spark implicitly split his own data structures to parallelize them into the clusters, so we set a master to manage the others slaves and then we run as many slaves as logical cores our own computer has.

The local[*] string is a special string that denots you're using a local cluster, which is another way of saying you're running in single-machine mode with several cores working as clusters. The * tells Spark to create as many worker threads as logical cores on your machine.

[16]:
```
spark = SparkSession.builder.master('local[*]').
 ↪appName('taxi_fare_local_cluster').config('spark.driver.memory', '8g').
 ↪getOrCreate()
```

5.0.1 Preparing data

Preparing the dataset as before to be trained in a local cluster.

[17]:
```
# Load training data
df = spark.read.csv('data/train.csv', header=True,
                    inferSchema=True, nullValue=' ')

lr = LinearRegression()

df = df.select(df['fare_amount'].cast("float").alias('fare_amount'),
               df['pickup_longitude'].cast("float").alias('pickup_longitude'),
               df['pickup_latitude'].cast("float").alias('pickup_latitude'),
               df['dropoff_longitude'].cast("float").alias('dropoff_longitude'),
```

```
                df['dropoff_latitude'].cast("float").alias('dropoff_latitude'),
                df['passenger_count'].cast("float").alias('passenger_count'))


df = df.selectExpr("fare_amount as label",'pickup_longitude','pickup_latitude',
                   'dropoff_longitude','dropoff_latitude','passenger_count')

#colum features
vecAssembler = VectorAssembler(inputCols=["pickup_longitude", "pickup_latitude",
                                          "dropoff_longitude",␣
 ↪"dropoff_latitude",
                                          "passenger_count"],␣
 ↪outputCol="features")

#Delete null rows
df_local_clust = vecAssembler.setHandleInvalid("skip").transform(df)
```

5.1 Training Linear Regression

[18]:
```
# Fit the model
start_time = datetime.now()

lrModel = lr.fit(df_local_clust.select('label','features'))

time_elapsed = datetime.now() - start_time
print('TIME OF LINEAR REGRESSION TRAINING (hh:mm:ss.ms) {}'.
 ↪format(time_elapsed))
```

```
TIME OF LINEAR REGRESSION TRAINING (hh:mm:ss.ms) 0:07:36.215578
```

5.2 Measures

5.2.1 Root Mean Squared Error

[19]:
```
#Summarize the model over the training set and print out some metrics
trainingSummary = lrModel.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
```

```
RMSE: 20.710237
```

6 Decision tree regression local cluster

[20]:
```
from pyspark.mllib.tree import DecisionTree, DecisionTreeModel
from pyspark.mllib.util import MLUtils
```

```
spark = SparkSession.builder.master('local[*]').
 ↪appName('taxi_fare_local_cluster').config('spark.driver.memory', '8g').
 ↪getOrCreate()

# Load training data
df = spark.read.csv('data/train.csv', header=True,
                    inferSchema=True, nullValue=' ')

df = df.select(df['fare_amount'].cast("float").alias('fare_amount'),
               df['pickup_longitude'].cast("float").alias('pickup_longitude'),
               df['pickup_latitude'].cast("float").alias('pickup_latitude'),
               df['dropoff_longitude'].cast("float").alias('dropoff_longitude'),
               df['dropoff_latitude'].cast("float").alias('dropoff_latitude'),
               df['passenger_count'].cast("float").alias('passenger_count'))

df = df.selectExpr("fare_amount as label",'pickup_longitude','pickup_latitude',
                   'dropoff_longitude','dropoff_latitude','passenger_count')

#colum features
vecAssembler = VectorAssembler(inputCols=["pickup_longitude", "pickup_latitude",
                                          "dropoff_longitude",␣
 ↪"dropoff_latitude",
                                          "passenger_count"],␣
 ↪outputCol="features")

#Delete null rows
df_local_clust = vecAssembler.setHandleInvalid("skip").transform(df)

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = df_local_clust.randomSplit([0.7, 0.3])

# Train a DecisionTree model.
dt = DecisionTreeRegressor()

start_time = datetime.now()

# Train model.  This also runs the indexer.
model = dt.fit(trainingData.select('label','features'))

time_elapsed = datetime.now() - start_time
print('TIME OF DECISION TREE ON LOCAL CLUSTER TRAINING (hh:mm:ss.ms) {}'.
 ↪format(time_elapsed))

# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
```

```
predictions.select("prediction", "label", "features").show(5)

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
TIME OF DECISION TREE ON LOCAL CLUSTER TRAINING (hh:mm:ss.ms) 0:13:05.248696
+------------------+-----+--------------------+
|        prediction|label|            features|
+------------------+-----+--------------------+
|  8.93128770001487|-45.0|[-73.980125427246...|
|13.792550010457832|-18.1|[-73.958274841308...|
|  8.93128770001487|-10.1|[-73.983306884765...|
|  8.93128770001487| -6.5|[-73.984352111816...|
|  8.93128770001487| -5.3|[-73.984802246093...|
+------------------+-----+--------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 24.0373
```

7 Random forest regression local cluster

[21]:
```
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils

spark = SparkSession.builder.master('local[*]').
 ↪appName('taxi_fare_local_cluster').config('spark.driver.memory', '8g').
 ↪getOrCreate()

# Load training data
df = spark.read.csv('data/train.csv', header=True,
                    inferSchema=True, nullValue=' ')

df = df.select(df['fare_amount'].cast("float").alias('fare_amount'),
               df['pickup_longitude'].cast("float").alias('pickup_longitude'),
               df['pickup_latitude'].cast("float").alias('pickup_latitude'),
               df['dropoff_longitude'].cast("float").alias('dropoff_longitude'),
               df['dropoff_latitude'].cast("float").alias('dropoff_latitude'),
               df['passenger_count'].cast("float").alias('passenger_count'))


df = df.selectExpr("fare_amount as label",'pickup_longitude','pickup_latitude',
                   'dropoff_longitude','dropoff_latitude','passenger_count')
```

```
#colum features
vecAssembler = VectorAssembler(inputCols=["pickup_longitude", "pickup_latitude",
                                          "dropoff_longitude",␣
 ↪"dropoff_latitude",
                                          "passenger_count"],␣
 ↪outputCol="features")

#Delete null rows
df_local_clust = vecAssembler.setHandleInvalid("skip").transform(df)

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = df_local_clust.randomSplit([0.7, 0.3])

# Train a RandomForest model.
#  Note: Use larger numTrees in practice.
#  Setting featureSubsetStrategy="auto" lets the algorithm choose.
rf = RandomForestRegressor()

start_time = datetime.now()

# Train model.  This also runs the indexer.
model = rf.fit(trainingData.select('label','features'))

time_elapsed = datetime.now() - start_time
print('TIME OF RANDOM FOREST ON LOCAL CLUSTER TRAINING (hh:mm:ss.ms) {}'.
 ↪format(time_elapsed))

# Make predictions.
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("prediction", "label", "features").show(5)

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
TIME OF RANDOM FOREST ON LOCAL CLUSTER TRAINING (hh:mm:ss.ms) 0:26:28.543485
+------------------+-----+--------------------+
|        prediction|label|            features|
+------------------+-----+--------------------+
| 8.887903540270049|-45.0|[-73.980125427246...|
|13.359459177697568|-18.1|[-73.958274841308...|
| 8.887903540270049|-10.1|[-73.983306884765...|
|11.238500888096038| -4.5|[-74.006141662597...|
```

```
| 8.934682006302216| -3.0|[-74.004997253417...|
+------------------+-----+--------------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data = 6.8643
```

7.1 Conclusion

As we can observe the Root Mean Squared Error is the same because we use the same algorithm in the same dataset, something relevant to be able to compare them with different approaches. As we see the time is 58% faster while working with the cluster.

4 Test-bed Environment and Experimental Results

4.1 Jupyter Notebooks

The software used to experiment with the data was Jupyter Notebook (to write python code) and PySpark (to run that python code into Spark). Jupyter Notebook (formerly IPython Notebook) is an open-source web application that lets you create and share documents containing live code, equations, visualizations, and narrative text.

Also, it is a widely used application in the field of Data Science to create and share documents including data cleansing and transformation, numerical simulation, statistical modeling, data visualization, automatic learning and much more.
It allows you to edit and run notebook documents through any web browser of your choice, and it can run on a local desktop that does not require Internet access, or it can be installed on a remote server and accessed through the Internet. We can also run Jupyter Notebook without any installation.

4.2 PySpark

PySpark is a python API for spark released by the Apache Spark community to support Python with Spark. Using PySpark, one can easily integrate and work with RDD in Python programming language as well.

Numerous features make PySpark an magnificent framework when it comes to working with huge datasets. Whether it is to perform computations on large data sets or to just analyze them, Data engineers are turning to this tool. What follows, are some of the mentioned features.

The following features are the key features of PySpark:

Real-time computations : Because of the in-memory processing in PySpark framework, it shows low latency.

Polyglot : PySpark framework is compatible with various languages like Scala, Java, Python, and R, which makes it one of the most preferable frameworks for processing huge datasets.

Caching and disk persistence : PySpark framework provides powerful caching and very good disk persistence.

Fast processing : PySpark framework is a lot faster than other traditional frameworks for big data processing.

Works well with RDD : Python programming language is dynamically typed which is helpful when working with RDD.

4.3 About the dataset Taxi Problem

[3]

The goal of this dataset is predicting the fare amount (inclusive of tolls) for a taxi ride in New York City given the pickup and dropoff locations. While you can get a basic estimate based on just the distance between the two points, this will result in an RMSE of $5-8$, depending on the model

used. Our challenge is to achieve it by using distributed Machine Learning techniques.

This dataset was chosen due to containing 6 Gb of data, which is an appropriate amount of data to test our results.

4.4 Experiments

There have been six experiments done (Each of them is a model testing), to try Linear regression, Decision tree regression and Random forest regression in Native Clustering and the rest to try Linear regression, Decision tree regression and Random forest regression in Local Cluster mode.

Mode	Machine Learning Model	Time-1	RMSE-1	Time-2	RMSE-2	Time-3	RMSE-3
Native Cluster	Linear Regression	4:04 min	20.7	11:47 min	20.7	8:17 min	20.7
Native Cluster	Decision Tree	8:37 min	23.96	20:22 min	23.96	20:05 min	23.96
Native Cluster	Random Forest	11:50 min	23.96	35:37 min	24.82	35:53 min	24.82
Local Cluster	Linear Regression	2:33 min	20.7	7:40 min	20.7	7:36 min	20.7
Local Cluster	Decision Tree	5:30 min	16.49	11:35 min	24.03	13:05 min	24.03
Local Cluster	Random Forest	8:44 min	23.96	18:30 min	23.96	**26:28 min**	**6.86**

Table 1: Selected experimental results on the training models.

There have been two approaches tested. One with Local Clustering, which the machine invests less time computing machine learning models than with Native Clustering. Both were trained using the same models respectively to make well-done comparasions.

Clearly, Linear Regression is the fastest model, and Random Forest is highly computing consuming. That's why it is more time spending model, but that's also why is the most accurate one.

Besides, it might be more beneficial to use Clustering with several machines, but it wasn't possible to experiment with that in this case. We didn't achieve to connect our computers to make a Remote Cluster with them by SSH because this protocol requires a lot of set-ups that we could have done properly. The interesting part is that if one machine fails, another can pick up the workload. In this way the system wouldnt never have an issue when the machines are processing data and suddenly a failure occurs. In addition, the work is could be more parallelized than with Local Clustering.

RMSE is the Root Mean Square Error of the machine learning models, it's a variable to measure the error that could produce that model. It's necessary to remark that as lower is the RMSE better is the accuracy of the model .In most of the cases, the RMSE doesn't change as we explain before because we are using the same algorithms with the same dataset(excellent for make comparisons depending on the mode of clustering). Except for one case in the third experiment with Random Forest in Local clustering, which is an excellent result, but the differences with the previous ones make it a wrong choice to compare with the rest. The same happened too in the first sample in the local clustering of the Decision Tree Regression and there is no remarkable information that could help with the comparisons.

5 Conclusion

Apache Spark has been designed in a way that it can scale up from one to thousands of computer nodes. Moreover, the simplicity of APIs in Spark and its processing speed makes Spark an excellent choice framework among data analysis projects.

For the data exploration, Spark uses SQL shell. MLlib supports data analysis and Machine Learning. It lets handling the problem with large data size.

Hence Spark provides a simple way to parallelize the applications across Clusters and hides the complexity of distributed systems programming, network developer, and fault tolerance.

References

[1] Douglas Eadline. *Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem*. Addison-Wesley Professional, 2015.

[2] Apache Software Foundation. Apache spark official site.

[3] Kaggle. New york city taxi fare prediction.

[4] Yandong Wang, Robin Goldstone, Weikuan Yu, and Teng Wang. Characterization and optimization of memory-resident mapreduce on hpc systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 799–808. IEEE, 2014.

[5] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets.