Big Data Analytics (SOEN 498/691)
Laboratory sessions

Tristan Glatard
Department of Computer Science and Software Engineering
Concordia University, Montreal
tristan.glatard@concordia.ca

March 14, 2017

Contents

Part I
MLlib

1 Introduction

MLlib (Machine-Learning library) is a library in Apache Spark that contains implementations of several algorithms seen in the lecture. In this session, we will run through basic examples related to clustering, recommendation systems and the mining of frequent itemsets. These examples may help you start with your project. Most of the material is taken from the MLlib guide and the Pyspark documentation. You are encouraged to further explore these references should you need more details about MLlib. For installation instructions and an introduction to Apache Spark, please refer to the previous lab session. Warning: MLlib is still under active development and backward compatibility even between minor versions is not ensured. Here we use version 2.1.0.

2 The DataFrame API

This section is adapted from the Spark DataFrame Guide. In the introduction session we presented how Resilient Distributed Datasets (RDDs) are used to define parallel operations on datasets. MLlib uses another kind of data objects, called DataFrame. A DataFrame consists of named columns. It is conceptually equivalent to a table in a database.

Before using the DataFrame API, a Spark session must be created:

```
>>> spark = SparkSession.builder.appName("Lab session").getOrCreate()
```

A DataFrame can be created from an RDD or from a file. For instance, here is how to read a DataFrame from a libSVM file (in the remainder of this document, it is assumed that environment variable `SPARK_HOME` points to your Spark installation):

```
>>> import os
>>> spark_home = os.environ['SPARK_HOME']
>>> dataset = spark.read.format("libsvm").load(\
    "file://"+os.path.join(spark_home,"data/mllib/sample_kmeans_data.txt"))
```

DataFrame objects can then be inspected and queried as a database relation:

```
>>> dataset.show()
>>> dataset.select("label").show()
>>> dataset.filter(dataset["label"]==1).show()
```

See the PySpark documentation for more details.

3 Clustering

MLlib includes an enhanced version of kmeans called kmeans||. The library is quite straightforward to use, but you should note that KMeans are implemented as an MLlib Estimator that fits a model to the dataset. Parameters of the `KMeans` estimator are listed as follows:

```
>>> from pyspark.ml.clustering import KMeans
>>> KMeans().explainParams()
```

Having loaded the dataset as in the previous section, the KMeans estimator is used as follows:

```
>>> kmeans = KMeans(k=2)
>>> model=kmeans.fit(dataset)
```

The resulting model essentially contains the cluster centroids:

```
>>> print model.clusterCenters()
```

Finally, a model can be applied to a dataset as follows:

```
>>> classified_dataset=model.transform(dataset)
>>> classified_dataset.show()
```

The complete example is available here:

(Link to file)

See also example in Spark repository.

4 Collaborative filtering

MLlib implements latent-factor-based collaborative filtering using the alternate least-square method. Using MLlib, you can implement a recommendation system with a few lines of code only. The example below, extracted from the Spark documentation, loads a sample of the MovieLens dataset, parses it as DataFrame Rows and creates a DataFrame from these Rows:

```
>>> from pyspark.sql import Row
>>> example_file_path=os.path.join(os.environ['SPARK_HOME'],\
    "data/mllib/als/sample_movielens_ratings.txt")
>>> lines = spark.read.text("file://"+example_file_path).rdd
>>> parts = lines.map(lambda row: row.value.split("::"))
>>> ratingsRDD = parts.map(lambda p: Row(userId=int(p[0]), movieId=int(p[1]),
                                 rating=float(p[2]), timestamp=long(p[3])))
>>> ratings = spark.createDataFrame(ratingsRDD)
```

For evaluation purposes, the DataFrame is then split in a training and a testing (evaluation) set:

```
>>> (training, test) = ratings.randomSplit([0.8, 0.2])
```

The recommendation model is then built as follows:

```
>>> from pyspark.ml.recommendation import ALS
>>> als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating")
>>> model = als.fit(training)
```

And finally its RMSE can be evaluated on the test data:

```
>>> from pyspark.ml.evaluation import RegressionEvaluator
>>> predictions = model.transform(test)
>>> evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
                            predictionCol="prediction")
>>> rmse = evaluator.evaluate(predictions)
```

The complete example is available in the Spark documentation.

5 Frequent itemsets mining

MLlib contains a few algorithms for frequent itemsets mining, however, they are currently available only in the RDD-based API which has entered maintenance mode since Spark 2.0. In the following code snippets, note the imports from `pyspark.mllib` while `pyspark.ml` is used in the DataFrame API. Besides, not all the algorithms have Python bindings.

Here is how to load and parse test data:

```
>>> example_file_path=os.path.join(os.environ['SPARK_HOME'],"data/mllib/sample_fpgrowth.txt")
>>> data = sc.textFile("file://"+example_file_path)
>>> transactions = data.map(lambda line: line.strip().split(' '))
```

The FP-growth algorithm is available in Python:

```
>>> from pyspark.mllib.fpm import FPGrowth
>>> model = FPGrowth.train(transactions,minSupport=0.2, numPartitions=10)
>>> result = model.freqItemsets().collect()
>>> for fi in result:
... print(fi)
```

The complete example is available in the Spark documentation.