Preprocessing the Data in Apache Spark

Steps in preprocessing

- Deploy data to the cluster
- Creating building RDD
- Verifying the data thru. sampling
- Cleaning data: For example
  - Converting the datatype
  - Filing missing values
- Other steps: integration, reduction, transformation, discretization

The Data Set

A	B	C	D	E	F	G	H	I	J	K	L
id_1	id_2	cmp_fnan	cmp_fnan	cmp_lnan	cmp_lnan	cmp_sex	cmp_bd	cmp_bm	cmp_by	cmp_plz	is_match
37291	53113	0.833333	?	1	?	1	1	1	1	0	TRUE
39086	47614	1	?	1	?	1	1	1	1	1	TRUE
70031	70237	1	?	1	?	1	1	1	1	1	TRUE
84795	97439	1	?	1	?	1	1	1	1	1	TRUE
36950	42116	1	?	1	1	1	1	1	1	1	TRUE
42413	48491	1	?	1	?	1	1	1	1	1	TRUE
25965	64753	1	?	1	?	1	1	1	1	1	TRUE
49451	90407	1	?	1	?	1	1	1	1	0	TRUE
39932	40902	1	?	1	?	1	1	1	1	1	TRUE
46626	47940	1	?	1	?	1	1	1	1	1	TRUE
48948	98379	1	?	1	?	1	1	1	1	1	TRUE
4767	4826	1	?	1	?	1	1	1	1	1	TRUE
45463	69659	1	?	1	?	1	1	1	1	1	TRUE
11367	13169	1	?	1	?	1	1	1	1	1	TRUE
10782	89636	1	?	1	?	1	0	1	1	1	TRUE
26206	39147	1	?	1	?	1	1	1	1	1	TRUE
16662	27083	1	1	1	?	1	1	1	1	1	TRUE
18823	30204	1	1	1	?	1	1	1	1	1	TRUE
38545	85418	1	?	1	?	1	1	1	1	1	TRUE
28963	39172	1	?	1	?	1	1	1	1	1	TRUE
38918	44578	1	?	1	?	1	1	1	1	1	TRUE
23499	40452	1	?	1	?	1	1	1	1	1	TRUE
46469	94155	1	?	1	?	1	1	1	1	1	TRUE
25816	29809	1	?	1	?	1	1	1	1	1	TRUE
28777	39482	1	?	1	?	1	1	1	1	1	TRUE
18914	18916	1	?	1	?	1	1	1	1	1	TRUE
15105	16971	1	?	1	?	1	1	1	1	1	TRUE
75313	76006	1	?	1	?	1	1	1	1	1	TRUE
29624	90371	1	1	1	?	1	1	1	1	1	TRUE
30127	30732	1	?	1	?	1	1	1	1	1	TRUE

Deploy the data to the cluster

- Distributed computing requires the file distributed across the cluster
- Transfer the local data files to hdfs
  - $ hdfs dfs –mkdir linkage
  - $ hdfs dfs –put block_*.csv linkage

Creating

- Create a RDD (Resilient Distributed Dataset) from text file
  - val rawblocks = sc.textFile(“hdfs:///user/yxie2/linkage2”)
- Create RDD from external databases
  - val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
  - val test_enc_orc = hiveContext.sql("*select * from test_enc_orc*")
- Spark is a lazy execution

Sampling data

- Sample:
  - $val head = rawblocks.first()
  - $val top10 = rawblocks.take(10)

- View data:
  - Printing on to client console
    - $head.foreach(println)

The Data Set

A	B	C	D	E	F	G	H	I	J	K	L
id_1	id_2	cmp_fnan	cmp_fnan	cmp_lnan	cmp_lnan	cmp_sex	cmp_bd	cmp_bm	cmp_by	cmp_plz	is_match
37291	53113	0.833333	?	1	?	1	1	1	1	0	TRUE
39086	47614	1	?	1	?	1	1	1	1	1	TRUE
70031	70237	1	?	1	?	1	1	1	1	1	TRUE
84795	97439	1	?	1	?	1	1	1	1	1	TRUE
36950	42116	1	?	1	1	1	1	1	1	1	TRUE
42413	48491	1	?	1	?	1	1	1	1	1	TRUE
25965	64753	1	?	1	?	1	1	1	1	1	TRUE
49451	90407	1	?	1	?	1	1	1	1	0	TRUE
39932	40902	1	?	1	?	1	1	1	1	1	TRUE
46626	47940	1	?	1	?	1	1	1	1	1	TRUE
48948	98379	1	?	1	?	1	1	1	1	1	TRUE
4767	4826	1	?	1	?	1	1	1	1	1	TRUE
45463	69659	1	?	1	?	1	1	1	1	1	TRUE
11367	13169	1	?	1	?	1	1	1	1	1	TRUE
10782	89636	1	?	1	?	1	0	1	1	1	TRUE
26206	39147	1	?	1	?	1	1	1	1	1	TRUE
16662	27083	1	1	1	?	1	1	1	1	1	TRUE
18823	30204	1	1	1	?	1	1	1	1	1	TRUE
38545	85418	1	?	1	?	1	1	1	1	1	TRUE
28963	39172	1	?	1	?	1	1	1	1	1	TRUE
38918	44578	1	?	1	?	1	1	1	1	1	TRUE
23499	40452	1	?	1	?	1	1	1	1	1	TRUE
46469	94155	1	?	1	?	1	1	1	1	1	TRUE
25816	29809	1	?	1	?	1	1	1	1	1	TRUE
28777	39482	1	?	1	?	1	1	1	1	1	TRUE
18914	18916	1	?	1	?	1	1	1	1	1	TRUE
15105	16971	1	?	1	?	1	1	1	1	1	TRUE
75313	76006	1	?	1	?	1	1	1	1	1	TRUE
29624	90371	1	1	1	?	1	1	1	1	1	TRUE
30127	30732	1	?	1	?	1	1	1	1	1	TRUE

- Pre-Process the data – Part II: Structuring Data with Tuples and Case Classes
  - The records in the head array are all strings of comma-separated fields
  - To make it a bit easier to analyze this data, we will need to parse these strings into a structured format that converts the different fields into the correct data type

```
def parse(line: String) ={
   val pieces = line.split(‘,’)
   val id1 = pieces(0).toInt
   val id2 = pieces(1).toInt
   val scores = pieces.slice(2,11).map(_.toDouble)
   val matched = pieces(11).toBoolean
   (id1, id2, scores, matched)
}
val lines= sc.textFile("hdfs://...")
val flines = lines.map(line => parse(line) )
```

- Needs to redefine the toDouble method

```
def toDouble(s: String) = {
    if (“?”.equals(s)) Double.NaN else s.toDouble
}
```

```
def parse(line: String) ={
    val pieces = line.split(‘,’)
    val id1 = pieces(0).toInt
    val id2 = pieces(1).toInt
    val scores = pieces.slice(2,11).map(toDouble)
    val matched = pieces(11).toBoolean
    (id1, id2, scores, matched)
}

val flines = lines.map(line => parse(line) )
val filtered_data = flines.filter(tup => tup(4) > 0)
```

Transformation

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

Action

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.