DSSP Data Camp July 10-11, 2015
"Large Scale Opinion Mining with Spark and Python"

PART II: Learning in Parallel

Christos Giatsidis, Apostolos N. Papadopoulos, Michalis Vazirgiannis

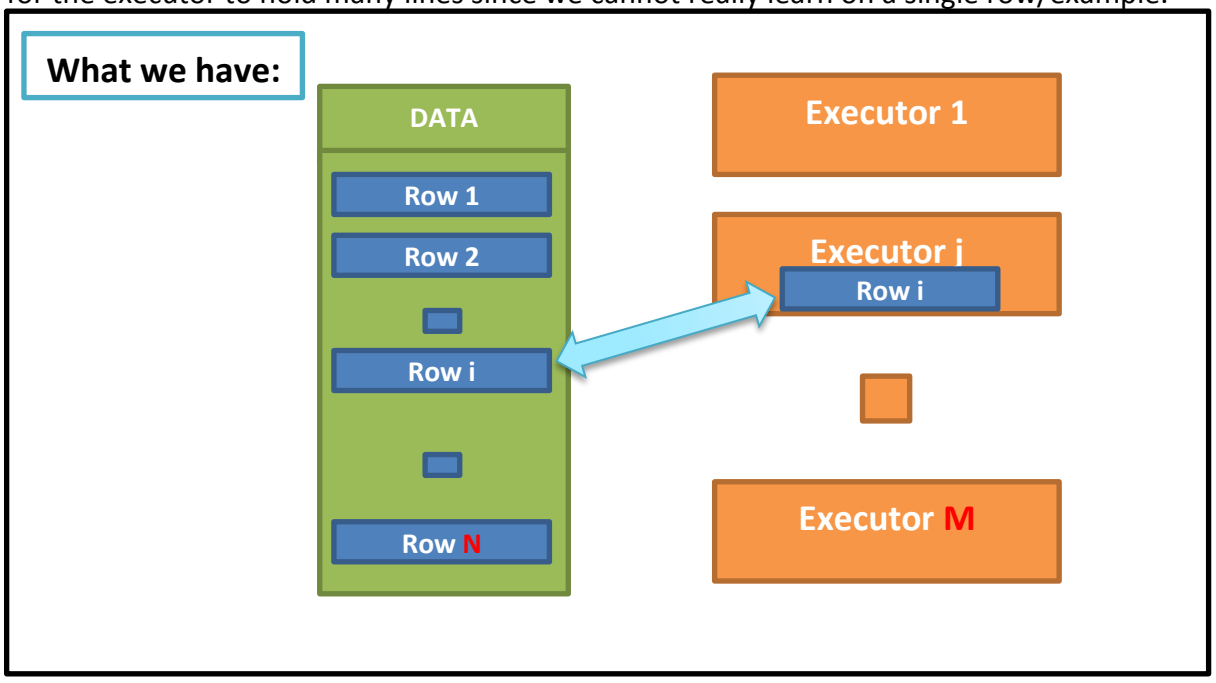
A. INTRODUCTION

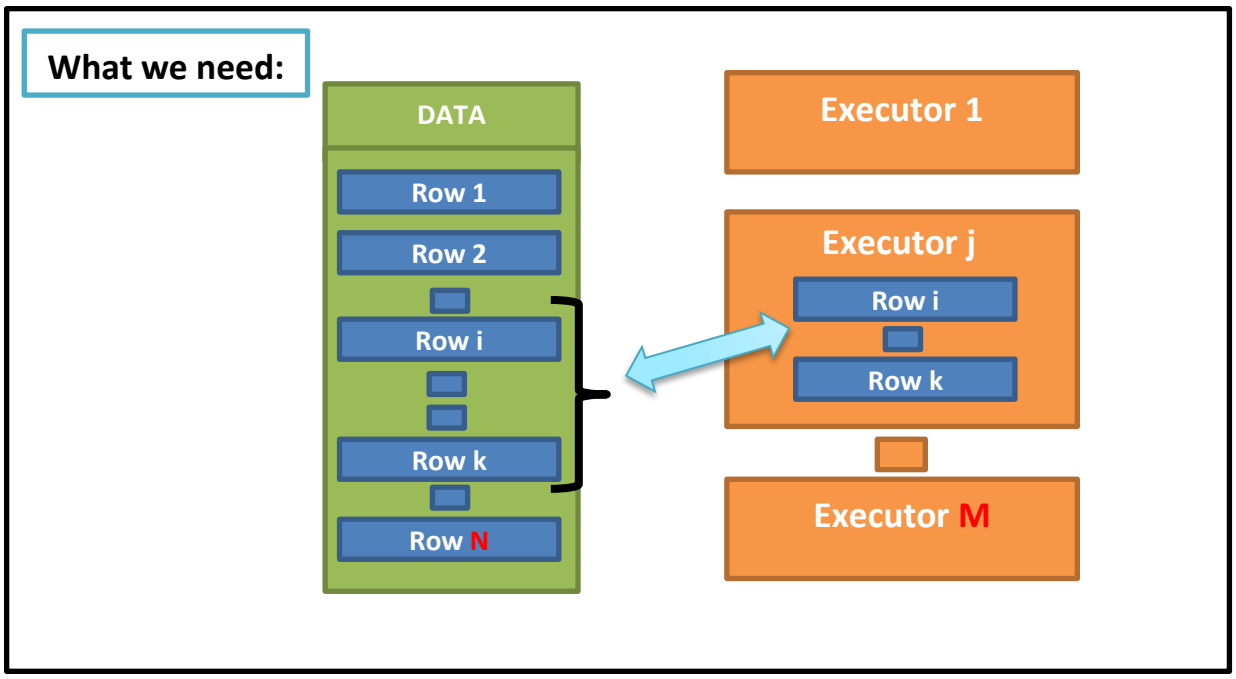
Spark offers, for the time being, a basic set of **Machine Learning algorithms**. While this set is sufficient for initial approaches, one may want to explore other algorithms as well. One solution would be to implement your required algorithm in the Spark distributed environment but this might require unavailable time or expertise. Another solution is to learn in parallel in smaller parts of the data.

We have seen how to use external libraries in Spark which of course are called in each executor separately and are applied on the data the executor holds (these data are of course not the entire distributed dataset but a part of it). In this manner, if we organize our data correctly, we can call other Machine Learning algorithms on parts of the data **in parallel** (e.g., using ML functions from scikit-learn). What we want to get back now is the set of trained models (instead of some statistic). Afterwards, we can use a simple voting scheme to classify new data. The "assumed" functions described here can be found in the **`helpers2.py`** file.

B. DATA PARTITIONING

If we take a look at the scikit-learn functions, we will see that they accept data in matrix forms while, at least on the example we have seen this far, the data are partitioned on a "row basis". This means that each executor at each point in time holds only one row. But what we require is for the executor to hold many lines since we cannot really learn on a single row/example.





We need **to group the rows into logical partitions**. So let's assume an RDD variable called **vectorized_data** which holds the distributed data by row. The first step would be to assign to each row of the **vectorized_data** a partition number. Remember that for the vectorized data each element holds the index of the row and the data of the row in a tuple. Here, we want to create 50 partitions because of memory restrictions in our specific setup and we want a small sized partition.

```
#we assign to each row a value between 0 and 50 (50 not included) randomly
#thus each row now "belongs" to a partition
partitions=vectorized_data.map(lambda x : (np.random.randint(0,50),x))
```

Now that the data are partitioned we could **group them by partition number** but first it would be nice to have a training and testing split in order to evaluate our models.

```
#we filter the partition on the partition key so that we may split to
train and test
train_partition = partitions.filter(lambda x: x[0]<40)
test_partition = partitions.filter(lambda x: x[0]>=40)
```

Since we want to group our data based on the partition number, it would be simple to include the row index information as the last index in the row data. Moreover we can also add the class value by replacing the row index with the value of the class in that index. Here we assume that **bY** is a broadcasted array that holds the class values.

```
#append as a last element the class value to each row
train_partition=train_partition.map(lambda x:
(x[0],np.append(x[1][1],bY.value[x[1][0]])))
#append as a last element the row index
test_partition=test_partition.map(lambda x: (x[0],np.append(x[1][1],x[1][0])))
#notice that we keep the partition number : we will needed later
```

The data now are organized in tuples where the first element is the partition number and the second the row data. Now we can group by the partition index:

```
train_partition = train_partition.groupByKey()
test_partition = test_partition.groupByKey()
```

The last function groups the data based on the first element of a tuple and puts the second element in a list-like structure . We can convert this structure into a matrix with the combination of a numpy and a python function : **`numpy.array(list(structure))`.**

C. LEARNING ON THE PARTITIONED DATA

Now that we have partitioned the data, we can call **any learning function** upon them. In scikit a learning function is part of an object which holds the model and is trained on data. Let's assume we have a function which:

- creates such an object
- trains it on data
- returns the trained model

And let's call this function **trainClassifier**. This function also requires the number of columns as information which we can broadcast. Training multiple classifiers in parallel becomes a very simple task:

```
part_classifiers=train_partition.map(trainClassifier).collect()
# now that we have our classifier trained we have to broadcast them so
# that during classification all the executors can access all the models
#the number of classifiers is the same as the number of the partitions
in the train_data
classifiers=[]
for x in part_classifiers:
            classifiers.append(x)

models=sc.broadcast(classifiers)
```

Let's also assume a function **evaluateByMatrixPartitioned** which applies all the models on the test data. The test data are also organized in matrices and each executor will get one such matrix. What we want to do is a simple voting scheme where we classify within each executor with all the classification models. The combined vote for each class will determine the final classification. In order to make sure that we can evaluate the classification, the function **evaluateByMatrixPartitioned** should return a tuple with two arrays: the row index and the prediction.

We can get the results of the voted classification as such :

```
voted_results=test_partition.map(evaluateByMatrixPartitioned).collect()
```

The voted results are now in a list of the aforementioned array of two tuples. So we can evaluate the accuracy as such:

```
correct=0
all=0
for x in voted_results:
     row_i=x[0]
     result=x[1]
     for i,v in enumerate(row_i):
          all=all+1
          if bY.value[v]==result[i]:
               correct=correct+1

accuracy=float(correct)/float(all)
```

C1. Are the Partitioned Matrices too Small?

In our example the resources might be a little limited for large sample in the partitioned data. In scikit-learn there is a special function for some classifiers that allows us to update a classifier with new data (called **partial_fit**). In this case, we can update the classifier by randomly reassigning the already trained models. We can call the **shuffle** function on the **classifiers** variable before broadcasting them. Since the models use the same number of partitions for the train data, we can use the partition number of the partition to select only one classifier. Let's assume a function that is similar to the training function but, instead of creating a new model, it takes one from the broadcasted variable based on the partition number. We call this function **updateClassifier** we can easily retrain our models as such :

```
classifiers = train_partition.map(updateClassifier).collect()
#do not forget to re-broadcast the classifiers so that we may use the updated ones
```

C2. What about the Unlabeled Data?

You might not want to partition in matrices the unlabeled data. For this reason, we could assume a function that applies the voting scheme by row in the original vector form of the data. We call this function **evaluateByRow**. If we try to apply this function, we will eventually notice that it takes a much longer time to classify the data. This is because it is much more efficient to apply the scikit evaluation function in a matrix than it is to apply it in a row by row manner (for small enough data the function is already optimized).

RESOURCES

http://spark.apache.org/docs/latest/api/python/

https://spark.apache.org/docs/latest/programming-guide.html

https://spark.apache.org/examples.html

https://www.youtube.com/watch?v=xc7Lc8RA8wE