Big Data Analytics Project 2015
"Opinion Mining with Spark"

Michalis Vazirgiannis, Apostolos N. Papadopoulos, Christos Giatsidis

A. INTRODUCTION

The area of *text analytics* (or *text mining*) includes techniques from multitude scientific areas (A.I., statistics, linguistics), and it has a wide range of applications (security, marketing, information retrieval). One of them is that of *Sentiment Analysis* and *Opinion Mining*. Opinion mining utilizes a subset of text analytics techniques with the goal to categorize opinions/reviews within a pre-specified range of "non-favorable to favorable" rankings. While part of the review forms might be in a structured format (with multiple options to select as answers), all forms contain a section to write down in "free text" personal notes, observations and comments.

This unstructured text is the focus of text analytics (and opinion mining). While structured data are "ready" for analysis and evaluation, unstructured text requires transformation in order to uncover the underlying information. The transformation of unstructured text into a structured set of data is not a straight forward task and text analytics offers a wide variety of tools to tackle with the idioms, ambiguities and irregularities of natural language.

In this data camp, we will tackle the problem of Opinion Mining in **movie reviews** with a basic set of techniques used in text classification. Recall that through the course, we covered some important issues regarding Opinion Mining, by using a small dataset. In this task, we are going to use a much bigger dataset and we have the option to utilize a cluster of machines running Apache Spark, in order to parallelize the work.

B. TASK OVERVIEW

B1. Data Description

We are given two sets of movie reviews:

1. A set of 25,000 documents that contain **labeled reviews** either as positive or negative (50%-50%). This will be used for **TRAINING**.
2. Another set of 25.000 documents containing unlabeled reviews that **we need to assign labels to them.** This set will be used for **TESTING**.

The reviews were taken from a review form that the user required to include a rating for the movie **in the range of 1 to 10**. In this collection, up to 30 reviews were allowed for any movie, as to avoid an "overpopulation" of reviews with similar ratings. In the labeled dataset:

- Negative reviews were considered as such if they had a rating less than or equal to 4.
- Positive reviews were considered as such if they had a rating higher than or equal to 7.

Using this approach, "neutral" reviews are eliminated since they usually introduce noise to the dataset. The task overview is depicted in Figure 1.

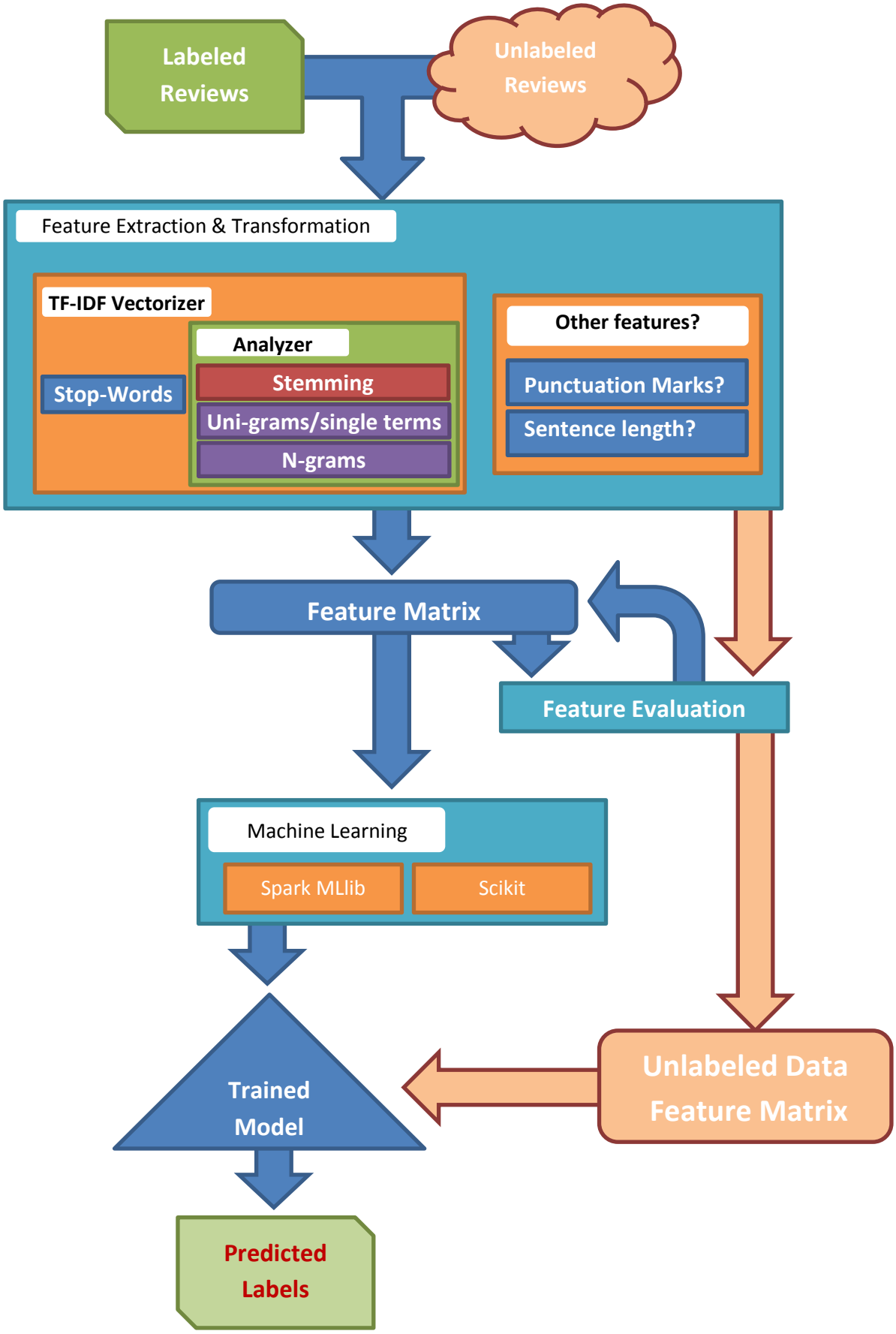


Figure 1. Task outline

B2. Resources

B2.1 The cluster

In order to connect to the cluster you need to **SSH** to *master-bigdata.polytechnique.fr.* The cluster is composed by:

- 16 machines
    - 224 GB RAM
- The master where you connect to and where you submit all jobs

B2.2 Code

You are given the following files:

- **loadfiles.py**: contains two function to load the labeled and unlabeled data into arrays accompanied by :
    - Labeled : two arrays that one contains in each cell one document and the other (array) contains the corresponding class (1,0)
    - Unlabeled: two arrays that one contains in each cell one document and the other contains the name of the corresponding document (without the file extension)
- **main.py:** contains a simple example of loading the training data and performing a simple classification on the test data.

C. RUNNING THE TASK ON THE CLUSTER

Multiple aspects of this project could produce very large matrices (in compressed in non-compressed format). Moreover some individual tasks might take too long on a single machine. Thus, we strongly recommend data and task distribution .Here, we explain how to utilize PySpark.

C1. How to Run PySpark Interactively

Starting a command line interface reserves resources even if we don't execute anything, but it is handy to test unfinished ideas.

```
pyspark--master yarn –conf spark.ui.port=6660
```

When starting a command line interface you have access to the spark context with a predefined variable **sc:**

```
>>> temp=range(10000)                    # local variable : list with 10000 entries

>>> temp_rdd=sc.parallelize(temp)        #variable that refers to the distributed
                                         #version of temp
```

C2. How to Use Spark-Submit with Python

If we have a more "complete" code (e.g., code.py) we may send the code to run on the cluster using **spark-submit**:

```
spark-submit --master yarn --conf spark.ui.port=6660 /path_to_code/code.py
```

When submitting code you have to create a variable that will contain the spark context. Assuming you have already passed the master parameters in the "spark-submit" command:

```
# code.py file
from pyspark import SparkContext
sc = SparkContext(appName="Simple App")
```

C3. Parameters

The 'conf' parameter in the examples above sets the port for the pyspark client so all the clients won't have the same port. Other parameters that could be useful (do NOT abuse the parameters):

Parameter	Exaplanation
--master	Where (url location) to find the master. yarn= look at the local yarn configuration
--num-executors	How many executors to reserve. Each executor reserves a predefined amount of vCPUs and RAM
--py-files	Path to local zip file that contains custom code. This code is available to all executors if we want them to use it. If we don't pass the files in a zip the executors will not have access to the code
--driver-memory	How much local memory (max) should the client use (default is 512mb and usually runs out)
--conf	Additional spark/yarn configurations. E.g the port we want to connect to. Another example : memory per executor/worker : spark.executor.memory. Two pass multiple yarn parameters you have to define --conf multiple times. E.g. ***--conf*** *spark.ui.port=6660* ***--conf*** spark.executor.memory=*2g*

D. DISTRIBUTED COMPUTATION

D1. Parallelizing the Data

Assuming we have a local variable with the data and their corresponding class <text, class>: "**data_class"**. We can distribute the variable through the cluster :

```
dcRDD=sc.parallelize(data_class,numSlices=16)
```

D2. Labeled Points

Spark contains a special class LabeledPoint which represents an instance of data by a vector (of features) and the corresponding class (the label). Since this structure requires only the additional information of the class (and the class variable is always relatively small), we can assume a function "**createLabeledPoint**" that converts a document to a vector representation; this function should take also as an input the dictionary of the collection (all the possible words/terms that appear throughout the documents). So, now we have:

```
labeled=dcRDD.map(createLabeledPoint)
```

We can utilize MLlib from Spark to pass the RDD and learn in a distributed manner:

```
# we need to import a Machine Learning Model e.g. NaiveBayes
from  pyspark.mllib.classification import NaiveBayes
# training
model = NaiveBayes.train(labeled)
```

The labeled points have the vector information and the class information per document. As the dictionary may be too big, in the examples on or code we use *SparseVector* which has a compressed version of a "normal" Vector: it stores only the non-zero values in a coordinate form of information (e.g. positions 1 and 2 have a value of 5 and 6).

D3. What about the Unlabeled Data?

For the Unlabelled data, we can assume a function "Predicted" that takes as extra input the trained model and applies it to each vector from the corresponding document. We can assume that we pass to this function a tuple <docid,text> and by accessing the dictionary we can create the vector. This function returns a tuple <docid,classification>. Then in order to get all the predicitons we can call:

```
# test_data contains a list of tuples <docid,text>
id_text=sc.parallelize(test_data)
predictions= id_text.map(Predict).collect()
```

D4. Python Partial

The map function applies any function to each element of an RDD by giving each one as an input to the first parameter of that function. In order to call functions that take multiple parameters we use the *partial* function of python. So assuming that Predict has the following form:

```
def Predict(id_text,dictionary,model) # id_text is a tuple
```

Then we can call Predict as such:

```
#dictb and modelb are the broadcasted variables of the dictionary and the model
predictions=id_text.map(partial(Predict,dictionary=dictb.value,model=modelb.value)).collect()
```

D4. Extra Stuff

You may require to run an function which will not require distribution or you may want to use a custom library that does not work with the distributed environment. **In order not to congest the master server with the computation,** we recommend the following technique to compute a non-distributed computation.

Assume a function "**compute**" which we require to run on a single machine on some "**data**". Then first step will be to broadcast the data to all nodes of the cluster and select only one to run the function. The function "**compute**" will have access to the broadcasted variable. For this example assume we have 4 nodes:

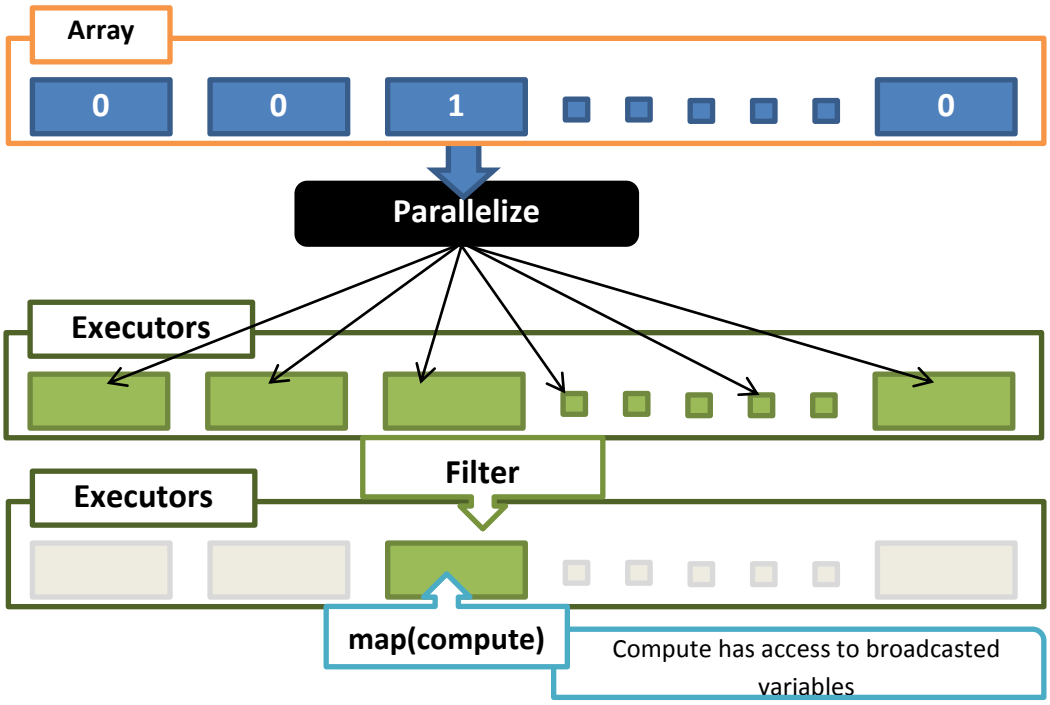
```
datab=sc.broadcast(data)
```

In spark when you apply a filter on an RDD, any code applied after the filter is obviously going to be applied only the filtered data of the RDD. ***That also means that the code is going to be executed only on the machines/executors that contain that data.***

We can create an array the same size as the number of the executors/nodes and have only one value being non-zero. If we distribute this array to as many executors as we have then each executor will get only one value of the array. Afterwards, if we apply a filter for non-zero values, whatever code we call after the filter is going to be executed on the machine that contains that value.

```
ex=np.zeros(4)
rp=randint(0,3)
ex[rp]=1                                    #an array with zeros
                                            except for one
                                            #random cell

tmpRDD=sc.parallelize(ex,numSlices=4).      #filter the array and
filter(lambda x: x!=0).map(compute)         call compute
```



External Libraries for machine learning

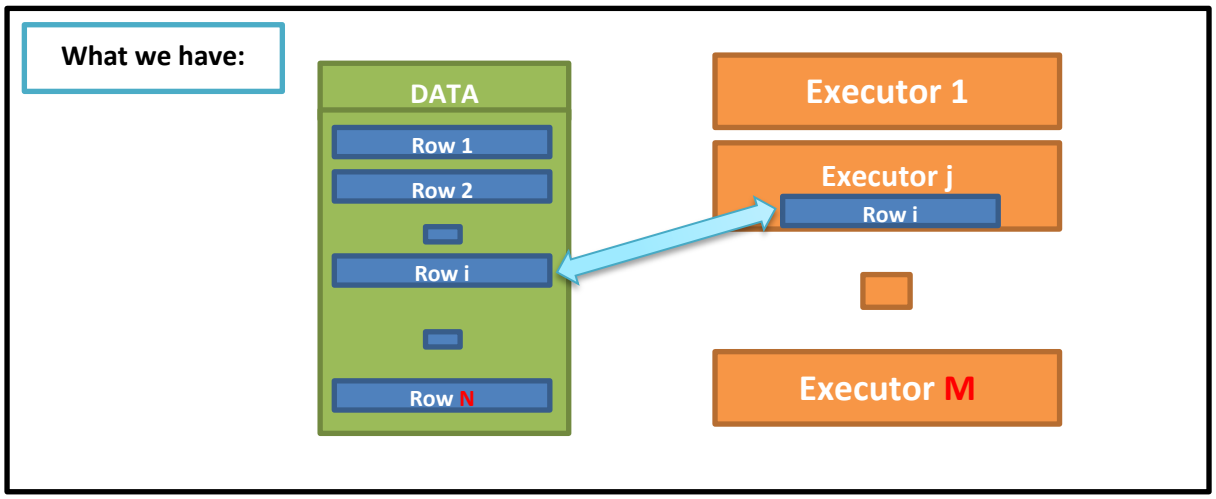
Since in the example we only use the machine learning library of spark, there are many algorithms which we don't take into account. Of course this library is designed over spark; if you want to use external learning libraries in spark you will have to learn models on (parallel) chunks of data and combine them afterwards (e.g. create a voting scheme over the models learned from each chunk). Of course that would mean that you should partition your data so that the parallelized RDD will not distribute one element at a time but many.

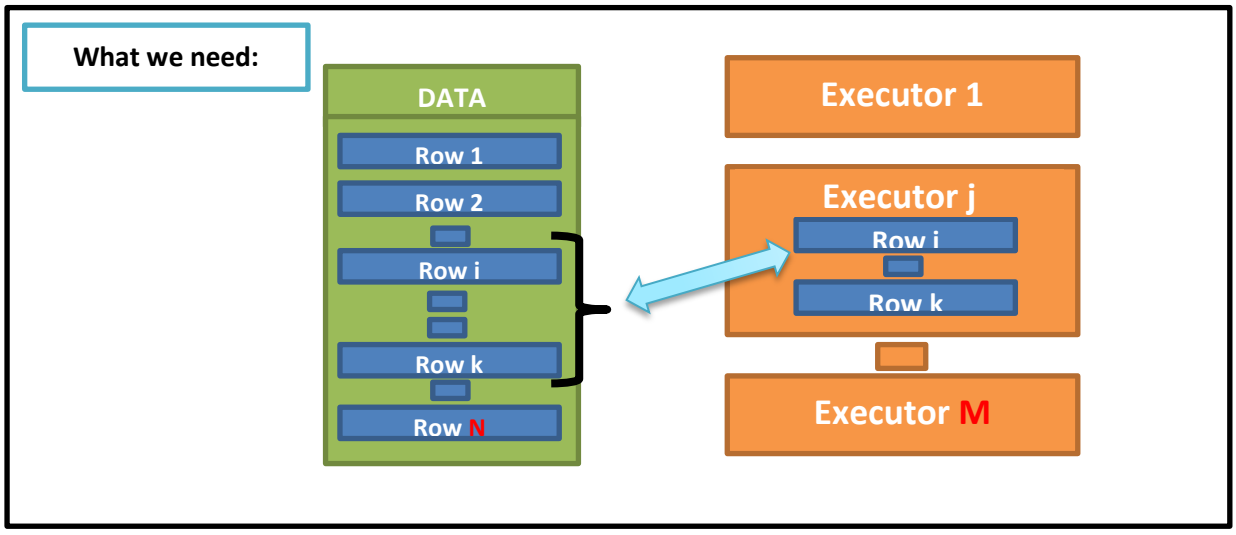
Data partitioning

Partitioning will also assist in the evaluation of your model with creating training/testing partitions for the data that you know their class. Assuming you have an RDD of "single" element i.e. rows you can do the following for 50 partitions:

```
#we assign to each element 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))
```

You can group then the partitions (look at spark documentation for the appropriate function[1]) to groups of "elements".





[1] http://spark.apache.org/docs/latest/api/python/

E. EVALUATION OF RESULTS

As part of this task you are required to form teams and deliver your classification results per team. Each team should have three people. Each team will deliver only one solution.

- The classification results should be in a file where each line has the name of the text and the corresponding class: e.g. for doc 00001.txt if it belongs to the positive class there should be one line:
  00001\t1.0
  the '\t' represents the tab character
- The classification results on the unlabeled data will be evaluated based on accuracy: the ratio of how many was correct from both classes over the total number of unlabeled documents.
- **Each solution, besides the classification results, it should be accompanied by** :
  - A report on all approaches and their respective performance
  - The code that you wrote
- The accuracy of the results as well as the completeness of your solution will determine the evaluation of the members of your team.

Team formation is necessary to cover the multitude of topics under this task and to limit the amount of the required resources for the cluster (the more people working on the task the more resources will be reserved).

When and Where:

You must send an email with your solution to : giatsidis@lix.polytechnique.fr by the 4th of January 2016. One email per team, all names must be listed in the report. If the files for your solution are too big for an email attachment please provide a link (e.g. use dropbox to provide a public link).

F. APPENDIX : Starting Ideas

Stemming & Cleaning

The data have not been cleaned in any manner

Stopword Removal

Some terms cannot contribute in discriminating between documents, simply because they appear in the vast majority of the documents. For example, words like "the", "a", "of", etc can be eliminated from the document collection. Stopword elimination has been proven effective in increasing the accuracy of the results, and you may use it.

Feature Selection

Each document is represented as a vector in the *m*-th dimensional space, where *m* is the number of distinct terms in the document collection (set of reviews). Therefore, each term is considered a *feature*. Some features may be more important than others. In order to increase the accuracy of the results you may select a limited number of features using feature selection technique. This way, you may decrease the number of dimensions by keeping the most important once, eliminating features that make your data noisy.

Using *N*-grams

In many cases, n-grams may help in increase the accuracy of the classification task. The default technique is to use each term as a single "gram". An *n*-gram, is generated by applying a sliding window of size *n* over the sequence of terms of the document. For example, assume that we have the document: "this is a simple text document". The unigrams of this document are simply the words in order, i.e., "this", "is", "a", "simple", "text", "document". The bigrams

or 2-grams of the document are: “this is”, “is a”, “a simple”, “simple text”, “text document”. *N*-grams can be computed in the same way, by collecting words that appear together in the document and shifting the sliding window from left to right. Note that if you use *n*-grams, each document is represented as a vector in the n-gram space and not in the term space. Essentially, each *n*-gram is like a “complex term”.

Different Classification Algorithms

In the running example we have shown how to use a Naïve Bayes classification approach to provide the results. It is up to you to test other algorithms as well and compare them with respect to the accuracy of results.

Learning in Parallel

Consider more than what MLlib provides.