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

PART I: The Main Task

Christos Giatsidis, Apostolos N. Papadopoulos, Michalis Vazirgiannis

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 in the lab of June 26, we covered some important issues regarding Opinion Mining, by using a small part of the movie reviews dataset. In this data camp, we are going to use the full dataset. Also, we are going to utilize the 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.

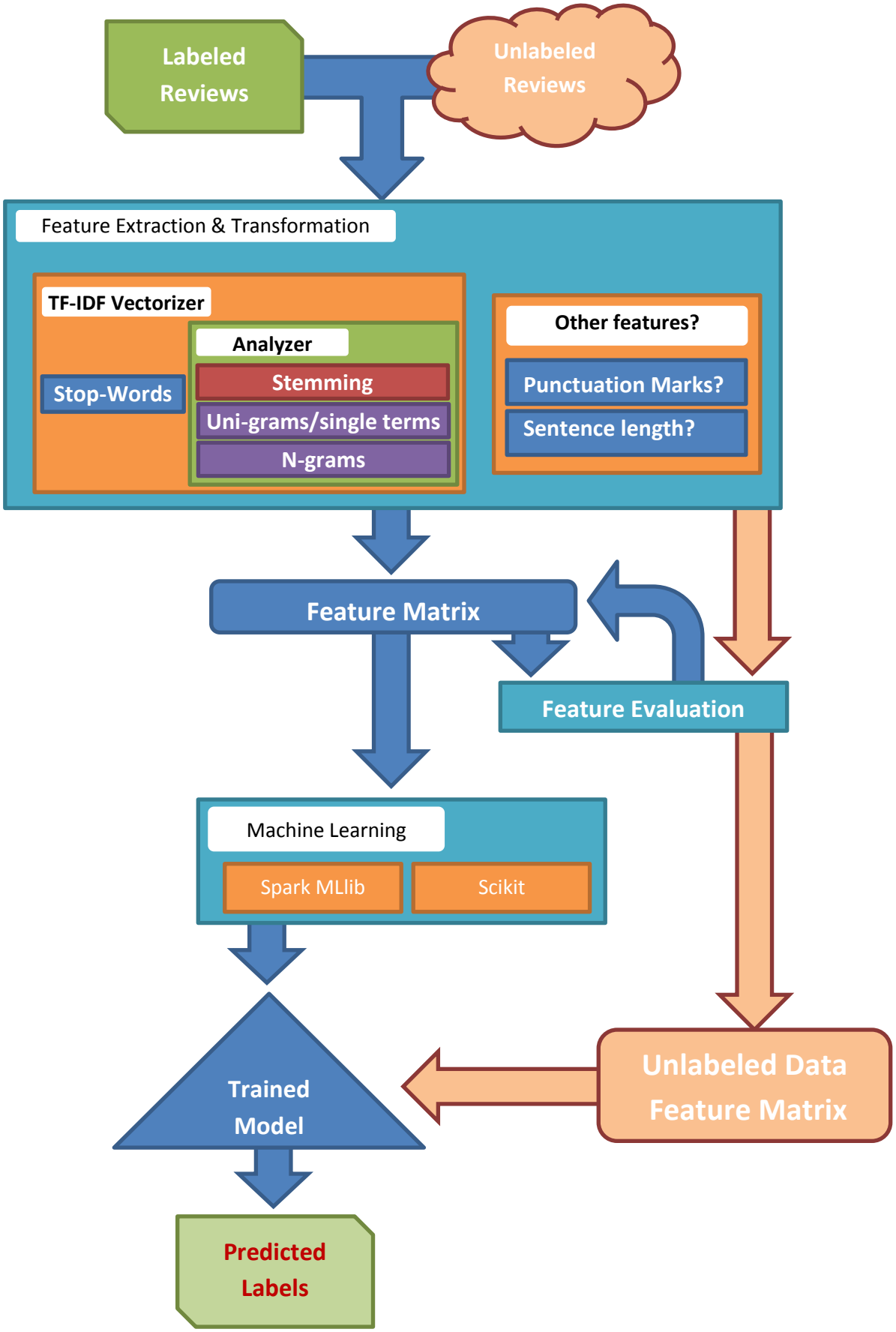


Figure 1. Task outline

B2. Vectorizing the Text Documents

One of the main approaches, to transform any corpus into a set of vectors, is by creating a tf-idf matrix. Here, we will utilize the scikit-learn Class "TfidfVectorizer". This class offers an easy to use interface to convert a set of texts into a TF-IDF matrix. Moreover, it offers the great flexibility of providing your own functions for the various parts of the process. For example, it has the following parameters that can be set with custom functions or parameters:

- **preprocessor** (default :None): We can set a function here that cleans the text from irrelevant or useless stuff. For example the text contains html tags we could set a function here to remove allsuch tags
- **tokenizer** (default :None) : A function to transform words (e.g. stemming)
- **stop_words** (default:None) : we can set up a list of words to ignore from a list. If not specified an automatic process filters words based on thresholding.
- **analyzer :** This parameter takes different types of values. It specifies what types of features we are going to extract. By default we extract single terms. **If this is set the values of the other parameters might be ignored.** Thus we can define our own function here which can take into account stop words and tokenizing. **This function should take as an input a piece of text and return a list of words.** E.g :
    - Without stemming and stop word removal one result of the following text :
        - "the beginning is the end is the beginning"
        - Result = ["the","beginning","is","the","end","is","the","beginning"]. *Duplicates are obviously allowed as the output of this function is used to calculate the tf-idf values.*

We focus here on defining the analyzer and preprocess the text separately in order not to create a confliction between the two. During this data-camp we are providing you with some start-up code to build upon your own procedures. For the transformation steps we are providing you with 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)
- **preprocess.py** : contains a function to clean up irrelevancies from text
- **exctract_terms.py** :contains a set of two functions that split a document into sentences and then into terms and returns the array of terms. **The intention of this file is to be used as the analyzer in the TfidfVectorizer.**

These files are mainly for the transformation step. An example usage can be found at **tfidfTEST.py.** There we use all the aforementioned files to create an initial tf-idf matrix :

- The preprocessing/cleaning up step is not passed as a parameter in the TfidfVectorizer as this might not work along with the analyzer (up until the latest version the two don't seem to work together.

- The only purpose of this code is to test locally the function of transformation
    - The matrix we get as result is not really stored in rows and columns but in a compressed format that DOES NOT store zeros. This is really not appropriate or efficient for the uses we intend to use this matrix. **We estimate the size of the actual matrix in terms of GB.**
        - What is the size of the matrix? Would a real example fit in a single computer?
    - We can easily get a list of all the terms found in the corpus.

NOTE: *Vectorizer needs to be "fitted" on training dataset initially in order to build the dictionary of possible words (features) and then he transforms the data. This can be done in one step (fit_transfrom) and the "fitted" vectorizer can then be applied to new data (only transform). This way the feature space is uniform among different datasets.*

C. RUNNING THE TASK ON THE CLUSTER

The basic process provided already produces a huge matrix that you may not be able to handle in a single machine (in non-compressed format). Even if you utilize parts of the labeled data, in order to maintain as much as possible of the information for the final prediction, you will have to handle the entire dataset. This means that you should run the task on the cluster. The entire tf-idf matrix will not fit on the main memory of a single machine, but it will fit on the (aggregated) main memory of the cluster. Thus, data and task distribution is the only solution in this case. In this camp, we are going to use PySpark which implemented a Python API to the Spark execution engine. Moreover, you may use spark-submit to send a piece of code to Spark. We cover both approaches in the sequel.

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 --num-executors 8--py-files tosend.zip --driver-memory 2g  --
confspark.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 --num-executors 8 --py-files tosend.zip --driver-memory 2g  --
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")
```

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. COMBINING LOCAL AND DISTRIBUTED COMPUTATION

Suppose we want to create TF-IDF Matrix and then use it as an input to a Machine Learning algorithm:

- Spark does not have its' own TfidfVectorizer yet.
- The non-compressed TfIdf Matrix might exceed the RAM limitations of a single Machine
    - We could use a "bigger" machine e.g. The master server?
        - We can't all use it at the same time
        - In a bigger dataset we would need an even "bigger" machine

What we need:

a) To utilize the TfidfVectorizer
    a. Make sure we won't congest the server
b) To convert the compressed matrix to an uncompressed RDD

In the next section all the "assumed" functions are already implemented in the **`helpers.py`** file for convenience. Their actual implementation might include some additional parameters to take into account broadcasted variables.

D1. Running theTf-Idf Vectorizer in a Remote Machine

In order not to congest the master server we would like to put the data and code for the vectorizer in remote machine and also to retrieve the data back in order to convert them in an RDD:

- The data are not really that big (only the non-compressed matrix)
    - We can make them available in all the slaves as local variables (broadcasting) and we could also retrieve the uncompressed matrix
- We want to eliminate the computation on a single machine.

Assume we have a function **"compute"** which can apply a vectorizer to a set of data and return the compressed matrix and the fitted vectorizer.

```
#broadcast local
variables to all
executors
md=sc.broadcast(m)          # m is the initialized vectorizer

datad=sc.broadcast(data)    #data is the locally loaded
                            documents
```

Now all the executors (in the slaves) have access to the data and the vectorizer. But we only need one of them to execute the code. Simple trick:

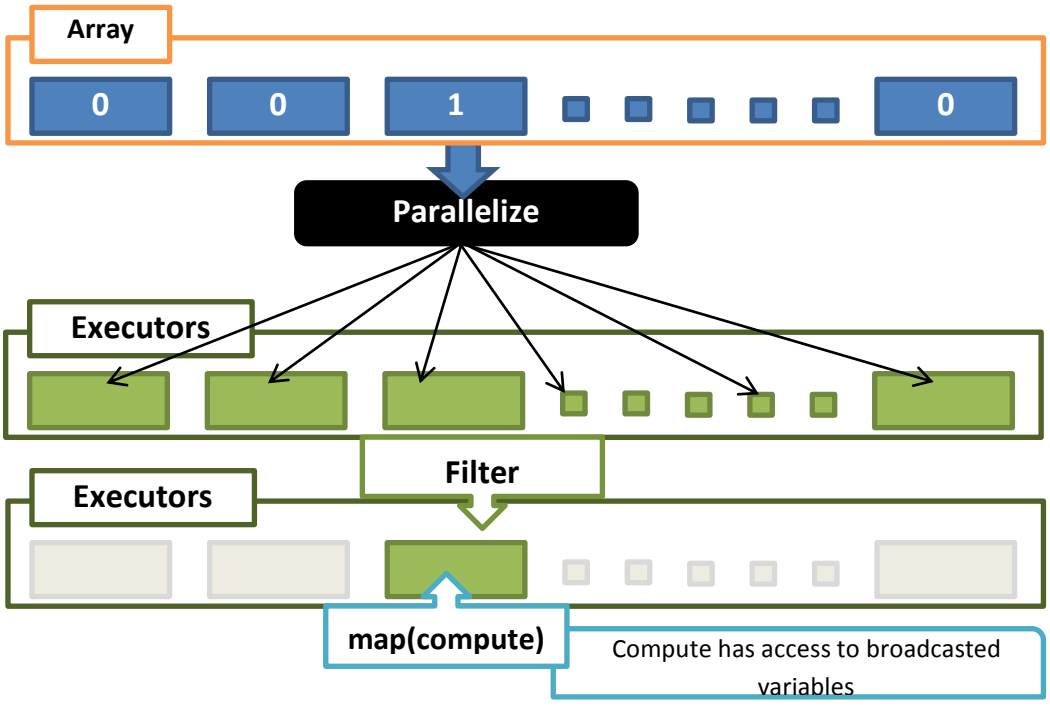
> *In spark when you apply a filter on an RDD, any code applied after the filter is obviously going to be applied only the filtered data of the RDD.**That also means that the code is going to be executed only on the machines/executors that contain that data.***

We can create an array the same size as the number of the executors 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(8)
rp=randint(0,7)
ex[rp]=1                                    #an array with zeros
                                            except for one
                                            #random cell

tmpRDD=sc.parallelize(ex,numSlices=8).      #filter the array and call
filter(lambda x: x!=0).map(compute)         compute
```

Now if we call "collect" on tmpRDD we are going to get back two variables: the compressed tfidf matrix and the "fitted" vectorizer.



D2. Decompressing the Data

There are multiple compressed formats and one of them, which can be easily transmitted and handled in Spark, is the coordinate matrix format. In this format the matrix is stored in three arrays: **[data, rows, columns]**.

The "data" array contains all the non-zero values of the matrix while the "rows" and "columns" arrays contain the coordinates of those values in the matrix. E.g., for the matrix M, **`M[rows[i],columns[i]]=data[i]`**. (i.e., if you want to know in which column, row belongs a value in index i of the data array – data[i] - then you take the row index from **`rows[i]`** and the column index from **`columns[i]`**.

It is much simpler to handle the data if we transpose the three arrays into one matrix so that each row will contain an instance of:

```
[data_value, row_index, column_index]
```

It would be convenient and efficient to convert this data into vectors by row. Assuming we have a function that takes all coordinates by row, then it would be easy to return a decompressed vector. Let's assume this function "**toVector**" that takes as an input the row number and a list of tuples in the following format :

```
[ (data_value,column_index), (data_value,column_index),...]
```

Each element in this list represents the value in the vector and the position in the vector. In this manner we can decompress the data into vectors by aggregating first based on the row index and then passing the data into to the "toVector" function. This function should return a vector that represents the row (and the row number to keep track). Of course all of this has to be a done in a distributed manner and the result should be an RDD as the data might not fit into one machine.
Assume the coordinate data are stored in a local variable **coo**. First we need to separate, in the representation of the data, the information about the row

```
tmp=sc.parallelize(coo).map(lambda x: (x[1],(x[0],x[2]))
# each row was :data, row_index,column_index
#now it is parallelized and each item is : row_index,(data,column_index)
```

Now we need to aggregate the data based on the row_index value. *But how do you aggregate tuples?*
We also need to define an aggregation function. For simplicity we define it quickly as a lambda function.

```
comb = (lambda x,y : np.vstack([np.array(x),np.array(y)]))
#it assume it receives two inputs (in our case two tuples)
#converts them into arrays and stack them one on top of the other
#if they are already arrays and not tuples, the conversion will do nothing
tmp=tmp.aggregateByKey([0,0],comb,comb)
#first parameter : what should the aggregation return if there are no data
#second parameter : which function to use to aggregate per machine/partition
#third parameter : which function to use to aggregate across machines/partitions
#in our case second and third are the same
vectorized_data=tmp.map(toVector)
```

D3. 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 "**toLB**" that can also access the broadcasted class and has similar functionality to the "**toVector**" function. So, in the last line now we have:

```
labeled=tmp.map(toLB)
```

In both cases (Vector,Labeled data), it is advised against calling "collect" on the "RDD" data as the results might be too large. On the other hand, 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)
```

D4. What about the Unlabeled Data?

The data for which we want to predict are required to be in vectorized form similar to the vector of the labeled data. We can assume a function **"computeTest"**whichworks like "compute" but only with a pre-fitted vectorizer (which we broadcast) and it only return back the transformed data. Let's also assume that we have :

- the transformed vectorized data are in a variable called **"vectorized"**
- the names of all the unlabeled documents in a list where the index in the list corresponds to a row in the vectorized data

Then we can perform classification in the following manner:

```
predictions=vectorized.map(lambda
x:(names[int(x[0])],modelb.predict(x[1]))).collect()
#We define a lambda function that takes the vectorized data
#vectorized data : row,vector_of_the_row
#the function returns the name based on the row (x[0])
#and the prediction of the model based on the vector (predict(x[1]))
```

E. PUTTING IT ALL TOGETHER

All the example code above assumed we are working in the pySpark Command Line Interface (CLI). While that offers flexibility, it is hard to keep track of your code. Thus it is sometimes more convenient to organize your code into files. The only issue is that some functionality is not the same when submitting code to Spark:

- **Broadcast variables:** Access to broadcasted variables from custom functions is not seamless. In CLI, we can define a custom function and broadcast variables and the broadcasted variables don't have to be a parameter of the custom function. When submitting code the custom functions must have in their definition any extra parameters that might be required from broadcasted variables. We pass those broadcasted variables to the custom functions with the "**partial**" function from the python "**functools**" library (the code is not as "neat" as it is in the examples above.
- **Including custom code:** When opening the cli we can send custom code in a zip file. This works the same when submitting a task in pySpark as well but if we want the workers/slaves/executors to have access to this files we need to add them to the spark context from the local file before we import the class :
    - *sc.addFile("path to python file")*

There are two additional files that put together the various points covered in the examples above:

- **evaluation.py :** This one contains the code to load the labelled data and create the labeled points. Moreover it displays a simple manner to utilize spark function to split the labelled point into a train and test dataset in order to evaluate the predictive model.
- **classify.py:** This one contains the code that covers the examples above :
    - loading labeled and unlabeled data,
    - transforming and vectorising the data,
    - training a **Naïve Bayes** model on the labeled data,
    - predicting the class of the unlabeled data, and
    - saving the predictions to a file.

The first file is intended to be used as a template to build your own evaluation pipeline by expanding on it. The second is intended to be used as an example to apply the selected process (once the evaluation is over) and save the results in the required format.

D. 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 two persons. Each team will deliver only one solution.

- The classification results should be in the same format as the one produced by the "**classify.py**" code.
- 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.
- 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 data camp 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). In order to increase the accuracy of your classification you may choose to apply several techniques. These techniques are already known to you from previous labs. You may freely reuse any code fragment you find useful. In particular, the techniques you may use to increase the accuracy of the results are summarized as follows:

Stemming

The use of stemming is mandatory! Therefore, you should use it in your solution. For example, we may apply it like this:

```
stemmer = PorterStemmer()
doc = [stemmer.stem(w) for w in doc]
```

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*. Based on what you have learned in previous lectures, 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 the **chi-squared test**. This way, you may decrease the number of dimensions by keeping the most important once, eliminating features that make your data noisy.

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. On way to apply stopword removal is to parameterize the tf-idf vectorizer. Another way to do it is to apply a "preprocessing" and remove any word that is found in the set of stopwords. A similar technique has been followed in the text mining lab of June 26.

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. For example, you may use Random Forest, Logistic Regression, SVM and in general all binary classification algorithms.

Learning in Parallel

This subtask is described in detail in Part II of the data camp. In this subtask, you should exploit the power of the cluster towards learning multiple models concurrently using distributed computation. Then, the best model is selected by a simple voting mechanism. This way, you will be able to test more models and select the one that shows the best accuracy results.

Note that, different combination of techniques may give very different results. Therefore, it is important to test as many combinations as possible, to maximize the accuracy of the results. Don’t be surprised if Naïve Bayes with a good feature selection and stopword removal is better than Logistic Regression without feature selection and stopword removal. Thus, you should spend a significant amount of time in evaluating the classification models you are going to use. For your evaluation, each team should provide the following:

1) the classification of the TEST data,
2) the Python code you have used and
3) short report (at most 2 pages) explaining the procedure followed in order to provide the results. You can prepare your report and send it to us within two days after the completion of the data camp.

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