Map reduce lab :

In the documentation you will see that there are more methods that can be defined to do things before and after the map and reduce steps and also an additional step called combiner that allows to combine results from the map step before being sent to the reduce step.

Suppose we want to count all the words of a huge set of documents that we have in a database or in a filesystem. A possible solution is to define a map-reduce program where the mapper divides all the documents into words and the reducer sums all occurrences of all the words.

The script StreamDocs.py is able to stream to the standard output all the documents from an index of a ElasticSearch database.

Those documents can be pipelined to a mrjob program to do the counting of words…The following map-reduce program counts the number of occurrences of each word in a stream of documents

To cluster the documents :

Expectation; compute similarity between all examples and the prototypes (..?) and return the closest prototype for each example

M

Script for frequencies of words in documents and compute max and min of frequency

Documents.txt is frequency of words etc,

Vocabulary.txt is words selected

A prototypes.txt file will be generated with the format: CLASSN: token1+freq token2+freq ... tokenn+freq 3 Because we are representing a document by its tokens, a prototype (the center of the cluster) is just the tokens of all the documents and their normalized frequency in the cluster (count word/num examples in the cluster).

Notice also that the values for the tokens in the prototypes, differently from the documents, are real numbers, not 0 or 1, and that we are comparing prototypes with documents.

This is how to run :

$ python StreamDocs.py --index news | python MRWordCount.py -r local

In the example ,

Initial prototype : documents for the mapper

One line of the doc doc1:word1, word2,word3…

COMPARE DOC WITH INPUT PROTOTYPE

IN PROTOTYPE

SIMILARITY BERTWEEN

CHAQUE MAPPER FAITT UNE LIGNE CAD UN DOCUMENT1

ON PARCOURT PROTOTYPES ITEMS ? ON FAIT LA SIMILARITé , et if (sim<min sim) on remplace

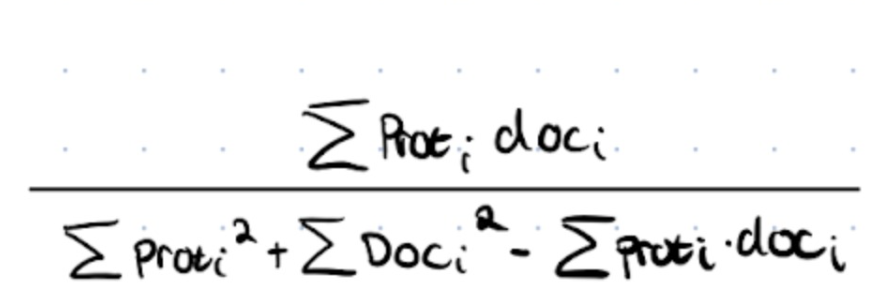
Mapper va renvoyer QUOI EN OUTPUT

WE HAVE TO COMPAREE EVERY DOC TO THE PROTOTYPE ? AND PUT THE MOST SIMILAR DOCUMENTS WITH A prototype ,?

Prototype or cluster id,(doc, words)

Dans reducer , on va calculer la fréquence de chaque mot avec le truc de if() +1 sinon =1 , puis on va calculer la moeynne en faisant la fréquence/total des mots, on va initialiser liste et faire freq[mot]=1 ou…

For jaccard distance :



Prot(i)=(frequencies of words in the prototype) and not 1,0 but doc(i)=1 or 0 if the word is present or not

"""

.. module:: MRKmeansDef

MRKmeansDef

\*\*\*\*\*\*\*\*\*\*\*\*\*

:Description: MRKmeansDef

:Authors: bejar

:Version:

:Created on: 17/07/2017 7:42

"""

from mrjob.job import MRJob

from mrjob.step import MRStep

\_\_author\_\_ = 'bejar'

class MRKmeansStep(MRJob):

    prototypes = {}

    def jaccard(self, prot, doc):

        """

        Compute here the Jaccard similarity between  a prototype and a document

        prot should be a list of pairs (word, probability)

        doc should be a list of words

        Words must be alphabeticaly ordered

        The result should be always a value in the range [0,1]

        """

        return 1

    def configure\_args(self):

        """

        Additional configuration flag to get the prototypes files

        :return:

        """

        super(MRKmeansStep, self).configure\_args()

        self.add\_file\_arg('--prot')

    def load\_data(self):

        """

        Loads the current cluster prototypes

        cluster1: lol+100, edf+234

        :return:

        there is a file for the prototypes

        """

        f = open(self.options.prot, 'r')

        for line in f:

            #we loop threw each line because each line is a different cluster

            cluster, words = line.split(':')

            #split cluster1(prototype1) and the words of this prototype1

            cp = []

            for word in words.split():

                #loop all the words in the prototype1 and append in the list the word  in position 0(for example :have)+frequency of this word in position 1

                cp.append((word.split('+')[0], float(word.split('+')[1])))

                #in the list prototypes, we will have key:cluster1 , and the cp is  lol+100, edf+234

                #prototypes look like :  cluster1: lol+100, edf+234,....

            self.prototypes[cluster] = cp

    def assign\_prototype(self, \_, line):

        """

        This is the mapper it should compute the closest prototype to a document

        Words should be sorted alphabetically in the prototypes and the documents

        This function has to return at list of pairs (prototype\_id, document words)

        You can add also more elements to the value element, for example the document\_id

        #WHAT DOES THIS FUNCTION RETURN ?? (prototype\_id, document words)??

        """

        #in input, each line is the document : doc1:word1,word2,word3....

        #

        # Each line is a string docid:wor1 word2 ... wordn

        doc, words = line.split(':')

        lwords = words.split()

        #we calculate here distance jaccard between each line (each document) in the input and the prototypes in the prototypes list

        #if the distance is inferior to min\_dist (we fix a min dist before)

        #min\_dist become hthe new dist jaccard

        # Return pair key, value

        #yield cluster and point = [document,words]

        yield None, None

    def aggregate\_prototype(self, key, values):

        """

        input is cluster and all the documents it has assigned

        #WHERE DO WE GET THE INPUT FROM ?DO WE GET THE UNPU FROM THE MMAPPER?

        Outputs should be at least a pair (cluster, new prototype)

        It should receive a list with all the words of the documents assigned for a cluster

        The value for each word has to be the frequency of the word divided by the number

        of documents assigned to the cluster

        Words are ordered alphabetically but you will have to use an efficient structure to

        compute the frequency of each word

        :param key:

        :param values:

        :return:

        """

        #INPUT :(cluster id,documents of this cluster)

        #for a cluster : receive list of all words of the documents in this cluster

        #value of each word=frequency of word/number of documents in the cluster

        #efficient

        #OUTPUT:(cluster id,new prototype/centroid)

        #values are going to be the pairs [document, words] resulting from the mapper in the output cluster and point = [document,words]

        #we create a dictionnary for the word and its frequency

        freq={}

        #iterate over each values (input) 'doc,words)

        #we have to calculate the frequency of words in the document (values)

        #iterate over the words in the list and append the mean of this word

        #calculate the mean by :frequency of word divided by total number of documents

        # output is (cluster,new prototype)

        yield None, None

    def steps(self):

        return [MRStep(mapper\_init=self.load\_data, mapper=self.assign\_prototype,

                       reducer=self.aggregate\_prototype)

            ]

if \_\_name\_\_ == '\_\_main\_\_':

    MRKmeansStep.run()