Name Entities Detection

Elena Sesoldi
elena.sesoldi@stud.unifi.it

Giulia Pellegrini
giulia.pellegrini1@stud.unifi.it

Abstract

A named entity is a real-world object, such as persons, locations, organizations, products, that can be denoted with a proper name. It can be abstract or have a physical existence.
In this paper we present some different implementations of an algorithm that extraxts named entities from tweet messages and computes foreach different category of considered entity the corresponding histogram of occurences. In particular we develope this work in two different languages: Java and C++ implementing a sequential version and a parallel one. For both parallel versions we use two specific programming technologies that improvement the parallelism: OpenMp for C++ and Apache Hadoop for Java. We employ the same datasets of tweets for testing the variants, but we have to use two different libraries for the extraction of the named entities so we can not compare the results obtained with the Java implementation against C++.

1. Introduction

Named Entity Recognition (NER), also known as entity identification, entity chunking and entity extraction, is a sub-task of *Information Extraction* (IE), that finds and categorizes specific entities in bodies of texts. NER is used in many fields in a *Artificial Intelligence* (AI) including *Natural Language Processing* (NLP) and *Machine Learning* (ML). The use of function models based on grammar or statiscal models allows to Named Entity Recognition to seek out targed informations; in fact NER recognizes entities first as one of many categories such as peaple, location, organizations, expressions, percentages and monetary values. Then using an information extraction utility the named entitys related information are extracted and according to the purpose of Information Extraction is built a machine-readable document from it other tools can further process to extract meaning.

2. Dataset

To test this work we dowload the datasets from a repository[1]. In this repository there are three different dataset. The first is composed by four subsets of tweets; one public timeline subset and four subsets of targeted tweets revolves around economic recession, Australian Bushfires and gas explosion in Bozeman (Montana). The second one is used in [3] which is relatively small in size of tweets but rich in number of NE. The last dataset is the training set provided for the Microposts 2014 challenge [1]. Specifically the first contains 1603 tweets, the second one has only 162 tweets and the last is the biggest with 2339 tweets. All of those are files with *.xml* extension.

3. Implementation

We work out this plan in two different programming languages: *C++* and *Java*. For the correspoding parallel versions we do not use the explicit threads of those languages but for the *C++* we have applied the *OpenMp* API and for the second *Apache Hadoop Framework*.
The final result of our work is the production of histograms: one for each category of entities, in particular we focus in the detection of three different entity types: PERSONS, LOCATIONS and ORGANIZATIONS.

[1] http://github.com/badiehm/TwitterNEED

The first operation to obtain the purpose is to upload *.xml* files from the *GitHub* repository and creates new ones with just the <*TweetText*> tags. In fact the *GitHub* datasets contain not only tweet messages but also some information related to these, as the tweet identity and the tweet entity, placed in other different tags. These new files become the input of two specifc libraries, one for *C++* and the other for *Java*, that extract the named entities.
The results subdivided by entity names show the number of distinct entities identified for each category and for each element how many times it is present in the datasets. Each result is shown in decreasing order so that is easy to see which are the most frequent real-word entities mentioned.

3.1. Results structure

Each entity detected and its number of occurrences are considered as a key-value couple and inserted in a map, so that at the end we obtain three different maps for each category.

4. *C++* Implementation

In this section we explain about our *C++* implementation choices.

4.1. Sequential Implementation

At the beginning the program creates an array of string that contains the tweet texts that are extracted from some files with *.xml* extension. Each of these texts includes only the <*TweetText*> tags of the *.xml* datasets downloaded from *GitHub*. Calling the *tokenize_file* method the main program creates a string vector for each dataset, breaking up the tweet texts into pieces that represent a single token like a word or a Alphanumeric symbol. The program uses the named entity extractor model *ner_model.dat* provided from *MITIE* library[2]

[2]MITIE is a library and tools for information extraction, and it is free (even for commercial use). The dowload version includes tools for performing named entity extraction and binary relation detection as well as tools for training custom extractors and relation detectors. (https://github.com/mit-nlp/MITIE)

and iterates on each element of the vectors that corresponds to a token of a tweet file, using the model of the library on which the predict method is called.
The *predict* method returns two useful vector of size equal to the number of entities found. The first vector contains pairs of doubles: the first represents the start of the corresponding entity in the vector of tokens, the second one the end of the same. The other vector contains doubles, that represent the category of the corresponding entity. The algorithm has to product a resultant map: the key is the real-word object to which it is concatenated its category, in this way it is possible to distinguish the same entity once indentified as person and once identified as location or organization. The value is the number of times that the object with its category has been found. Before the key is created, the string containing the entity is modified: it is converted to lowercase, deprived of all non-alphanumeric characters and multiple spaces, so that a key has the structure *"named entity _CATEGORY"*. As soon as the key is built the program checks if it is already contained in the resultant map, in this case its values is increasing by one, otherwise the key is insert into the map and its value is set to one. Obtained the map, the program produces the three final maps subdividing the resulting map into the three tags. So we have a person map, a location map and an organization map, that are sorted in order of the decreasing values (number of the object occurences). At last the algorithm creates a file that contains the three sorted map and prints on screen the ten most frequency elements of each map, drawing an histogram of asterisks. The program builds also two other files to save some information about the entities found and the histograms already printed on the console.

4.2. Parallel Implementation

For the parallel variant we use *OpenMP* (*Open specifications for Multi Processing*), a parallel shared memory programming model. OpenMP is a paradigm different from *Java Thread* or *Pthread*, in fact those lasts are explicit thread, it

means that the structure of sequential code needs to be changed to work on more threads. The feature of *OpenMP* is to be based on implict thread, that hiding details and keeping the sequential structure; in fact we have only to add compiler directives. It is the compiler's task to manage the opening and closing of threads; the disadvantages of this model is the expense of expressiveness and less flexibility. *OpenMp* can be used in *Fortran*, *C* and *C++*. In *C++* the commands for the compiler are of the following form: ♯ *pragma omp [parallel]*. To perform the parallelism we use the *Reduction Pattern*.
After creating the tokens as in the sequential version, the parallel part of the program starts with the directive ♯ *pragma omp parallel*. Then with the directive ♯ *pragma omp for* a for loop is executed in parallel and distributed among threads already existing in the parallel region. This directive has one clause: *schedule(static, 1)*. The schedule clause avails to control the distribution of the loop between threads. In this case it divides the cycle into pre-defined blocks of given size (one) and the blocks are statically assigned to the threads sequentially following the identifier of the individual thread. Once the available threads are finished, the blocks are reassigned from the first thread. In the loop, a partial result map is defined and created for each thread. After the loop it is defined a critical section with the clause ♯ *pragma omp critical* where each thread merges its partial result in a final common one.
As soon as the total result is completed, the program comes to the generation of the three histograms and the result files as in the sequential version.

5. Experiments

We test our program with the datasets already described repeating them several times, through setting a variabile named *numRepetition*. In this way in each execution we use a number of datasets equal to three (number of real datasets) multiplied by this variable.
We tried more parallel version with different number of threads using one of the few library functions of *OpenMp* that is *omp_set_num_threads(int num_threads)*, to see how execution times changes.

Table 1. Table that reports the execution times

Rep	Seq	Omp1	Omp2	Omp3	Omp4
1	66533	66727	63783	44663	44529
2	133256	134112	72604	84968	66612
3	199776	200994	135204	127669	100748
4	266349	267882	147127	142258	123365
5	336083	334772	211378	184322	164915

In Table 1 we show the execution times of sequential version and parallel ones. In the first column there is the number of the repetitions of the datasets. The other columns report the execution times in milliseconds of sequential, in the second one, and parallel implementation, in the last ones.
For the parallel version the number of thread varies from one to four. For example, in the third column, named *Omp1*, we report the times of the *OpenMP* version with just one thread. Once the times have been detected we compute the relative speedups with the formula:

$$Speedup = \frac{Time_{seq}}{Time_{par}}$$

The speedup is a number that measures the relative performance of two systems processing the same problem. In our case the two systems are the sequential and the *OpenMp* implementation.

Table 2. Table that reports the speedups

Rep	Sp1	Sp2	Sp3	Sp4
1	0.9970	1.0431	1.4896	1.4941
2	0.9936	1.8353	1.5683	2.0004
3	0.9939	1.4775	1.5647	1.9829
4	0.9942	1.8103	1.8722	2.1598
5	1.0039	1.5899	1.9861	2.2198

In Table 2 we show the speedups of the different parallel versions. In the first column there is

the number of the repetitions of the datasets. The other columns report the speedups.
For example, in the second column, named *Sp1*, we have the ratio between the sequential time and the OpenMP time with one thread.
We report only the results of the tests made with the *schedule* clause of the *omp for* directive setted to *static* with *block_size* defined to one.
With the four thread version used with the biggest number of repetitions (fifteen total datasets), we tried also with other setting of the *schedule* clause, as *dynamic* and *auto*, and with different *block_size* dimension, from one to four. However we obtained the highest speedups with the setting *schedule(static,1)*, so we decided not to show them all. The differences between the execution times with *dynamic* and *static* scheduling are in the order of 10 seconds. We also tried to see if the insertion of the *nowait* clause could have brought us a further improvement of the speedups, but the execution times remain about the same. These experiments have been done using four *Intel Core i7-5500U Processor* with a clock frequency of 2.40GHz; regarding the specific software we work with *Ubuntu* 17.10 at 64 bit.

6. *Java* Implementation

In this section we explain about our *Java* implementation choices.

Apache Hadoop

Apache Hadoop is a framework conceived to write applications easily that process large amounts of data in parallel on big size's clusters (made up of thousands of nodes) ensuring high reliability and avaibility. To guarantee these features, *Hadoop* uses macro-system including *HDFS*, a distribute file system, designed specifically to store huge amounts of data.

MapReduce

MapReduce is a framework for creating applications that can process large amounts of data in parallel basing on the concept of *functional programming*, so that there is not sharing data between threads but these are passed as parameters or return values. The *Map* function transforms input data into a series of key-value pairs. The *Reduce* function collects the key-value pairs produced by the map function. Couples are sorted by key and values with the same key are grouped together. On these couples the *Reducer* performs a summary operation.

6.1. Name Entities with Hadoop

The task of the *MapReduce* framework is counting the number of named entities in a collection of *.xml* files containg tweet messages. The *Mapper*, in its *map* methods, processes line by line the input files, looking for the entities. To do this, it uses the *LingPipe* library[3]. In particular it use the model *ne-en-news-muc6.AbstractCharLmRescoringChunker*, provided from the library and passed to the *Mapper* by the context, to build a *Chunker* able to return a chunking of entities given a character sequence, that in this case is the line text. Once obtained the chunk set, it loops on it creating the key-value pairs. The key is a string of the entity combined with its category as already describe in the *C++* implementation, and the value is an integer always setting at one. The *Reduce* in its *reduce* method takes all the key-value pairs for a given real-word object and adds up the values, generating a single key-value pair for each entity, that is written on the context. The *Reducer* is called once for each key. The whole process of *MapReduce* is shown in the Fig 1. For serialization on HDFS, the key and value type are *Text* and *IntWritable*, which are *Hadoop* classes that implement *Comparable* and *Writable*. The *Driver* is a class with a main method that tells *Hadoop* how to run *Mapper* and *Reducer*. It instantiates a *Configuration* object and a *Job*

[3] *LingPipe* is toolkit for processing text using computational linguistics. LingPipe is used to do tasks like find the names of people, organizations or locations, automatically classify Twitter search results into categories and suggest correct spellings of queries.(http://alias-i.com/lingpipe/demos/tutorial/read-me.html)

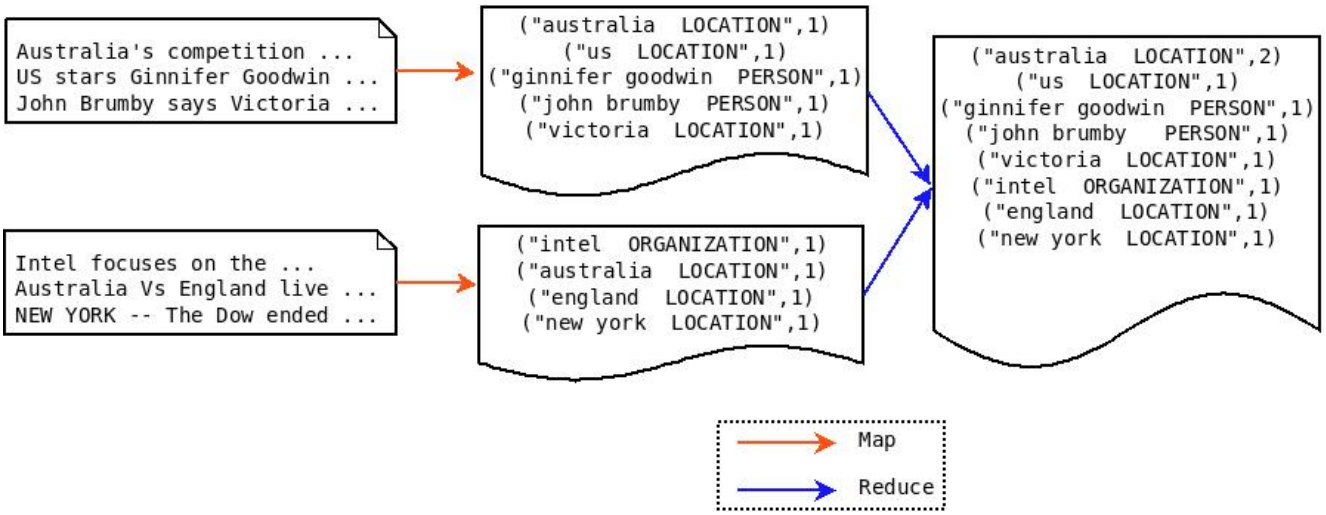

Figure 1. MapReduce scheme

object, to which is passed the *Configuration*. Using the *Job* we set the name of *Mapper* and the *Reducer* classes, set output key and value classes and specify where to find input data and to write output data. In this case there is no need to set the input key and value types, because *Hadoop* assumes by default that we are processing text files.

Input

In the *main* method there is the generation phase of *Mapper* inputs: *GitHub* datasets are loaded and saved as strings that, using the *LingPipe* library, are processed for the extraction of the tweet messages. This is possible because these are inserted in a tag, so they can be identified through a regular expression.
Once found all the tweet messages, examining the whole dataset set, they are saved in new *.xml* files in a directory which is becoming the *Mapper* input.

Output

The program, if successful, generates an output directory where the *Reducer* writes its output to its own file *part-r-00000*, where 00000 is the *Reducer ID*. Once created this file the main program proceeds with creating the three *TreeMap*, Histograms and files as in *C++* versions.
We tried to increase *Reducer* number but the execution time remains unchanged because most of the work is made during the *Map* phase.

7. Results

In the figure(Fig **??**) we present the histograms of both the implementations pointing out the most common named entities detected by our algorithms. The occurence numbers refer to the *GitHub* datasets. As can be seen from the histograms, the use of two distinct libraries for NED produces different results.

8. Conclusion

In this article we propose three different implementations of the creation of histograms that represent the frequence of named entity in tweets. We explain two version written in *C++*, sequential and parallel using the shared memory *OpenMP* API, and one in *Java* using the distributed memory *Apache Hadoop* tool. For the extraction of named entities we benefit of a library for each programming language, in this way the histograms build by our algorithms are different because theese have distinct NER methodologies, although both are based on *Machine Learning*.
We test our programs to see the benefits of the parallelization. For *C++* we can evaluate the par-

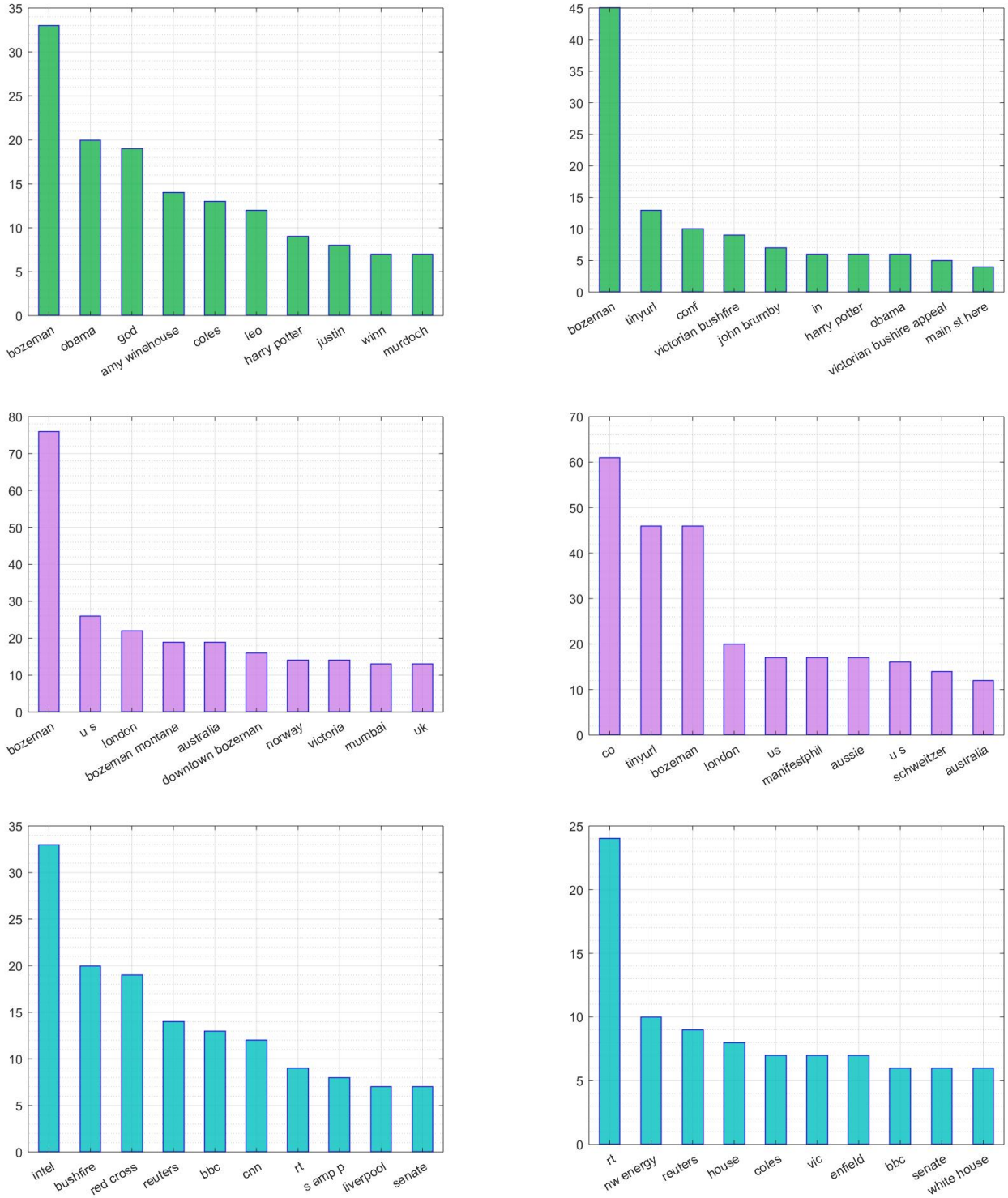

Figure 2. We compare the results of our algorithm obtained with the two libraries. In the left column there are those obtained with *MITIE* on the right there are those detected with *LingPipe*. The green histograms refer to Persons, the pink refer to Locations and the blue represent the Organizations.

allelism computing the speed up, and we obtained the best value equals to 2.2198 with four threads and fifteen total datasets.
Besides this task (counting the occurence of NE) is suitable to the use of the *MapReduce Pattern*, that is nothing but the central operation of *Hadoop*.

References

[1] A. V. M. R. M. S. A. E. Cano Basave, G. Rizzo and A.-S.Dadzie. Sense of microposts (microposts2014) named entity extraction and linking challenge. *In Proceedings of Microposts2014*, pages 54–60, 2014.

[2] badiehm. Twitterneed, 2014.

[3] M. Habib and M. van Keulen. *Unsupervised improvement of named entity extraction in short informal context using disambiguation clues*, pages 1–10. CEUR Workshop Proceedings. CEUR-WS.org, 10 2012.

[4] B. Locke and J. Martin. *Named entity recognition: Adapting to microblogging.* PhD thesis, University of Colorado., 2009.